Packaging & Docker

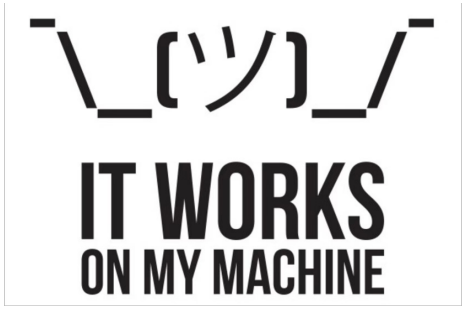
CS 346: Application Development

Deployment Challenges

Our goal is **stable, consistent software** that we can deliver to our customers. Every version of the OS that we support, every different platform, every conceivable configuration should work as expected.

However, if our development / testing / deployment environments don't match exactly, our software might not run properly!

- You might be missing files e.g., a config file that you created manually on your development machine.
- You may have incorrect versions of libraries e.g., different versions of DirectX installed.
- You may have a different versions of the OS.



"Things you should never say to a customer".

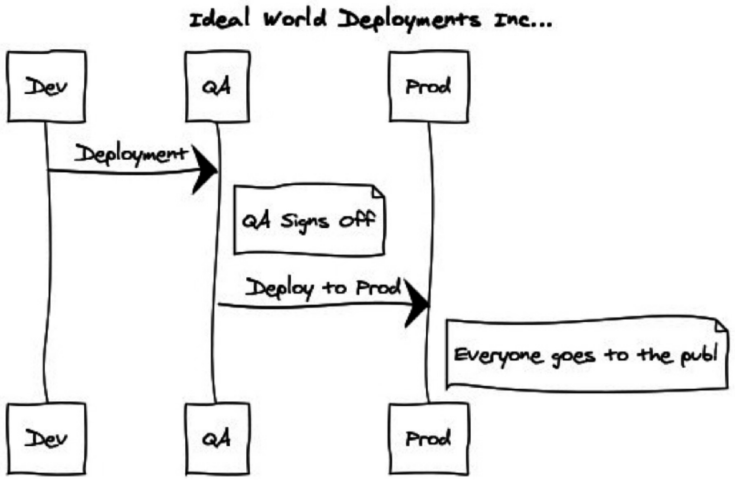
How can we accomplish this?

Multiple environments, that you keep in sync.

- **Development**: system for builds.
- **Testing**: configurations representing each environment that you support (i.e., supported versions of macOS, Windows and so on).
- **Deployment**: a system where you might deploy the software (live, running!)

How do you guarantee that each environment has the correct versions of every piece of software that your application uses or requires?

How do you handle this when you cannot control the deployment environment? e.g., an end-user's home computer?



https://dzone.com/refcardz/deployment-automation-patterns

Installers

Solution 1: Installers

An **installer** is an application that installs other software. e.g., setup on Windows.

Typical actions that they perform:

1. Create a folder for your application e.g., c:\Program Files\MyApp
2. Install your executable program, set permissions, register with the OS.
3. Install system libraries, register with the OS e.g., c:\Windows\System32
4. Setup initial preferences including application icons.

Note that these are very OS specific! Mac, Windows, Linux have different systems.

Gradle Packaging

Gradle can create installers for you.

1. Console applications
   - Tasks > Build > distZip
   - Creates a JAR file, and scripts to execute it (all in a ZIP file).
2. Desktop/JVM applications
   - Tasks > Compose Desktop > packageDistribution
   - Creates a Windows MSI, macOS DMG or Linux DEB.
3. Android
   - Build > APK file (sideload)

Where installers Fail

Installers don’t always work. Sometimes an application won’t run after installation. Often this happens because your operating conditions are different from the conditions under which you developed and tested.

Examples:

- You may have tested on a different version of the operating system than the user has, so your software may work differently on their system.
- Your application might rely on other software to be installed e.g., `sed`.
- The runtime environment might need to be configured in a specific way for your application to run correctly e.g., environment variables holding private keys, like AWS_SECRET='KASDJFTG_&JGJMHGF_!@GHHY@', or specific network configurations.

How do we fix this? *We specify and control the deployment environment*.

Containerization

Controlling the deployment environment.

Solution 2: Virtualization

- Virtualization uses software to create an **abstraction layer** over computer hardware, enabling the division of a single computer's hardware components—such as processors, memory and storage—into multiple virtual machines (VMs).
- Each VM runs its own operating system (OS) and behaves like an independent computer, even though it is running on just a portion of the actual underlying computer hardware. – IBM (2024)

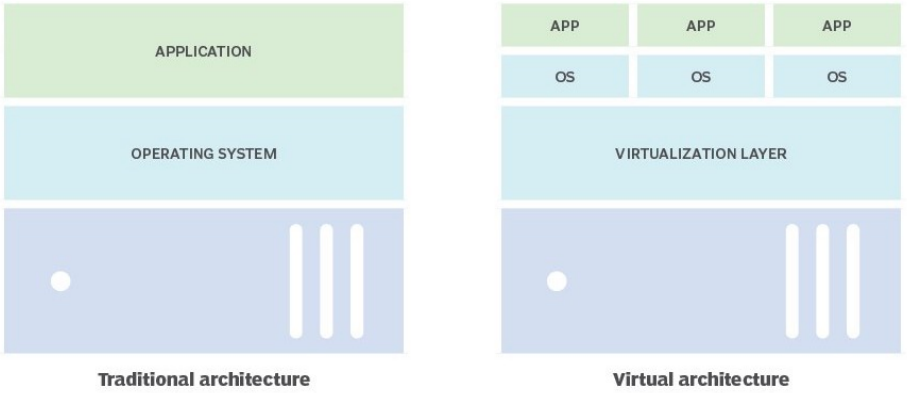


Diagram from TechTarget (2024)

Virtualization is “Heavyweight”

Benefits?

- Resource efficiency: Physical hardware can be shared across multiple operating environments.
- Easier management: Virtual “Machines” can be started up as needed (backed up, moved).
- Control: We can use this to specify the runtime environment!

Downsides?

- Fairly heavyweight. We’re hosting an OS specifically for our application.

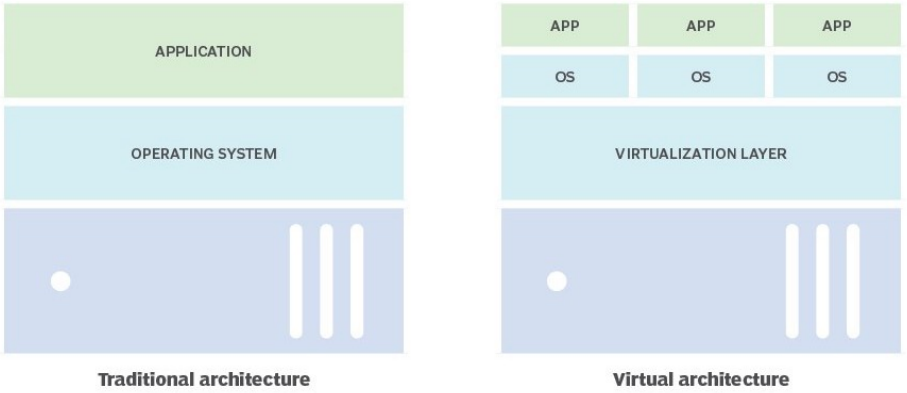


Diagram from TechTarget (2024)

Containerization

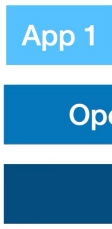


Fig 1. Standalone Hardware

One OS that hosts everything

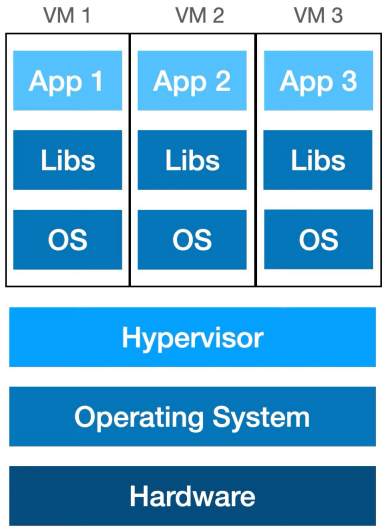


Fig 2. Virtualization

Dedicated virtual machines

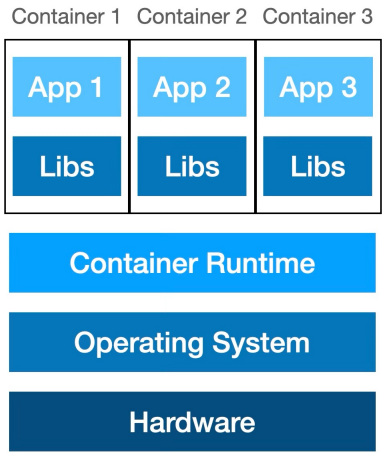


Fig 3. Containerization

Lightweight containers?

Comparison

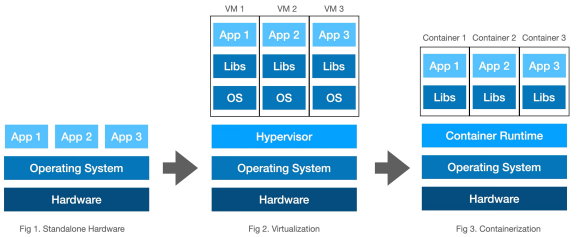


Fig 1. Standalone Hardware Fig 2. Virtualization Fig 3. Containerization

Standalone: Software is installed directly into the host operating system.
- The OS must allocate and manage resources for each application.

Virtualization: Multiple virtual machines, each an abstraction of a physical machine.
- Each virtual machine is running a complete OS, allocated memory, CPU cycles etc.
- Can dictate how physical resources are shared across VMs e.g., split 128 GB RAM.
- Provides isolation of each application into its own OS instance for improved security.

Container: An isolated environment for running an application.
- Containers run on the same underlying host OS; lightweight vs. virtual machines.
- The host OS schedules CPU, resources to the containers not VMs.
- Smaller, easy to start/stop; can be deployed on any physical and virtual machines.

Docker

Simple containerization.

Solution 3: Containerization (Docker)

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications.

https://docs.docker.com/get-started

Docker is a **containerization** platform.

- Create images that bundle your application and its environment together.
- Provides an online hub where you can distribute these images to other people.
- Provides the runtime engine to execute images.

Docker is NOT meant for end-users!

- It's for people like us that need an efficient and consistent way to install servers/services.

Installation

https://docs.docker.com/get-started/get-docker/

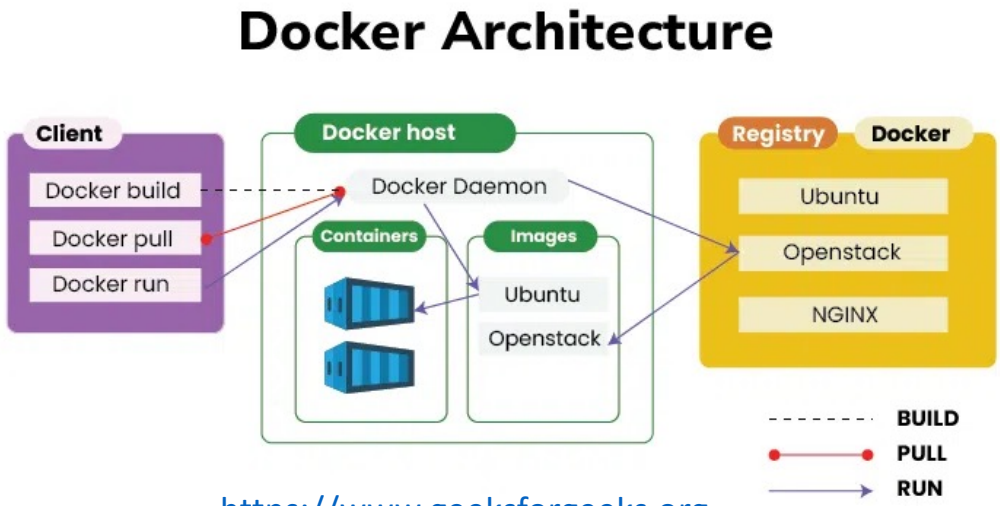
Install Docker from installers on their website or your favorite package manager. e.g., `brew install docker` on macOS.

```
$ docker version

Client: Docker Engine - Community
 Version:           27.3.1
 API version:       1.47
 Go version:        go1.23.1
 Git commit:        ce1223035a
 Built:             Fri Sep 20 11:01:47 2024
 OS/Arch:           darwin/arm64
 Context:           desktop-linux

Server: Docker Desktop 4.34.3 (170107)
Engine:
 Version:           27.2.0
 API version:       1.47 (minimum version 1.24)
 Go version:        go1.21.13
 Git commit:        3ab5c7d
 Built:             Tue Aug 27 14:15:41 2024
 OS/Arch:           linux/arm64
 Experimental:      false
containerd:
 Version:           1.7.20
 GitCommit:         8fc6bcff51318944179630522a095cc9dbf9f353
runc:
 Version:           1.1.13
 GitCommit:         v1.1.13-0-g58aa920
docker-init:
 Version:           0.19.0
 GitCommit:         de40ad0
```

Architecture



https://www.geeksforgeeks.org

- The **Client** is the system on which you are running Docker commands
  The **Docker Host** is a background process that manages Docker runtime.
  - An **Image** is a snapshop of your environment + application at a point in time.
  - A **Container** is a running instance of your Image.
- The **Registry** is an online repository to store images for others to use.

Workflow

- Step 1: Write a program
  - Create a program that can be executed. For us, this will typically be a JAR file that we can run using `java –jar filename.jar`.
- Step 2: Write a Dockerfile
  - Create a configuration file that describes your environment.
- Step 3: Create a Docker Image
  - Create an image which contains your environment (including dependencies) and executable at a point-in-time.
- Step 4: Run your Docker Image

Step 1: Compile your program

Write a complex and useful application (Hello.kt in this example).

```
fun main() {
   println(“Hello Docker!”)
}
```

Compile it to a JAR file, and copy the JAR file to a new/empty directory.

```
$ kotlinc Hello.kt -include-runtime -d Hello.jar
$ java -jar Hello.jar
Hello Docker!

$ mkdir docker
$ cp Hello.jar docker/
```

Step 2: Dockerfile

Create a file named `Dockerfile` in the same directory as your JAR file.

```
# start with this image, it includes a Linux kernel and Java JDK 17
FROM openjdk:17

# import your Hello.jar file, and host in the app subdir.
# at runtime, your filesystem will expose under /app subdir
COPY Hello.jar /app

# set /app as your working directory and `cd` to it
WORKDIR /app

# run the application
CMD java -jar Hello.jar
```

Step 3: Create an Image

Build an image in this directory (which uses the Dockerfile)

```
$ cd docker
$ docker build -t hello-docker .
```

`-t` tells Docker to tag it with a version (defaults to latest).
`hello-docker` is the name that will be assigned to our image.
`.` indicates that it should include the current directory's contents in the image.

Step 4: Run it

Check that it was created

```
$ docker images
$ REPOSITORY   TAG     IMAGE ID     CREATED    SIZE
 hello-docker  latest  a615e715b56d 7 second ago  455MB
```

Run it!

```
$ docker run hello-docker
Hello Docker!
```

Step 5: Publish it (Optional)

1. Create an account on Docker Hub if you haven't already. Login.
2. Create a repository to hold your images.
3. Tag your local image with your username/repository.
4. Push your local image to that repository.

```
$ docker image ls
REPOSITORY    TAG       IMAGE ID      CREATED         SIZE
hello-docker  latest    f81c65fd07d3  3 minutes ago   455MB

$ docker tag f81c65fd07d3 jfavery/cs346

$ docker push jfavery/cs346:latest
```

When to use this?

- Docker containers are extremely common when publishing web services to the cloud! Publish a container and have AWS/Firebase/some service host and run the container.
- You may need to map a port number to direct network traffic from the host machine to the running container. e.g., below.

```
# Dockerfile
FROM openjdk:17
VOLUME /tmp
EXPOSE 8080
ARG JAR_FILE=target/service-docker.jar
ADD ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```