Packaging

CS 346: Application Development

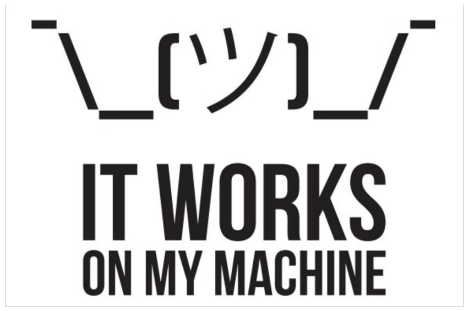
Deployment Challenges

Our goal is to deliver **stable, consistent software** to our customers.

Every version of the OS that we support, every different platform, every conceivable configuration should work as expected.

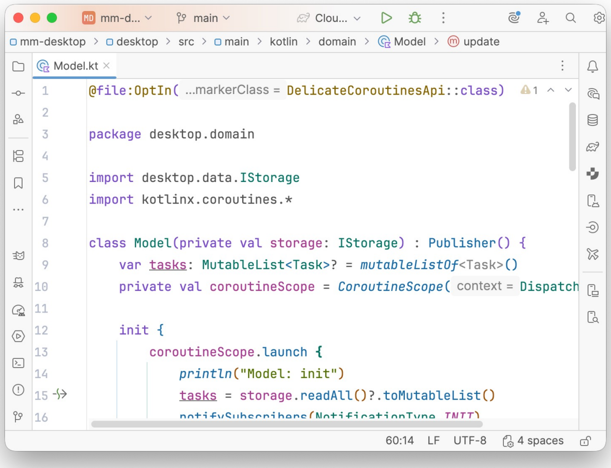
However, if our development/test/deployment environments don't match exactly, our software might not run properly!

- 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".

The Challenge


mm-d...
main
Clou...
mm-desktop › desktop › src › main › kotlin › domain › Model › update
Model.kt
60:14 LF UTF-8 4 spaces

```kotlin
@file:OptIn( ...markerClass = DelicateCoroutinesApi::class)

package desktop.domain

import desktop.data.IStorage
import kotlinx.coroutines.*

class Model(private val storage: IStorage) : Publisher() {
    var tasks: MutableList<Task>? = mutableListOf<Task>()
    private val coroutineScope = CoroutineScope( context = Dispatch

    init {
        coroutineScope.launch {
            println("Model: init")
            tasks = storage.readAll()?.toMutableList()
            notifySubscribers(NotificationType.INIT)
```

What we think of as a single application can be hundreds of files that all need to be installed.

e.g., IntelliJ IDEA includes an executable, libraries, resources, and even text files.

How do we manage all of this?

~/Applications/IntelliJ IDEA Ultimate.app

```
$ tree -L2

.
└── Contents
    ├── _CodeSignature
    ├── bin
    ├── CodeResources
    ├── help
    ├── Info.plist
    ├── jbr
    ├── jdk-shared-indexes
    ├── lib
    ├── license
    ├── MacOS
    ├── modules
    ├── plugins
    └── Resources
```

~/Library/Preferences

```
$ tree -L 1 | grep jetbrains

├── com.jetbrains.intellij.plist
├── com.jetbrains.jbr.java.plist
```

~/Library/Application Support /JetBrains/IntelliJIdea2025.2

```
$ tree -L1

.
├── app-internal-state.db
├── bundled_plugins.txt
├── codestyles
├── colors
├── disabled_plugins.txt
├── disabled_update.txt
├── early-access-registry.txt
├── event-log-metadata
├── extensions
├── idea.key
├── idea.properties
├── idea.vmoptions
├── inspection
├── jdbc-drivers
├── keymaps
├── kotlinNotebook
├── light-edit
├── migration
├── migration_installed_plugins.txt
├── options
├── plugin_AIP.license
├── plugin_PCWMP.license
├── plugins
…
```

Solution 1: Installers

An **installer** is simply an application that installs other software.

Typical actions that an installer can perform:

1. Create the folder structure for your application
    - Install files to the correct location, as dictated by your OS.
    - Set file and directory permissions for your configuration.
2. Install system libraries, register with the OS
    - e.g., update the Windows Registry to handle a new file type.
    - e.g., register a library that your application included.
3. Register with application with the operating system
    - Create an application icon, add it to the Start Menu/Dock.
    - Set initial preferences for your application.

This is very OS specific!

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.
    - You typically need to generate the installer on the same platform you want to support.
3. Android
    - Build > APK file
    - “Correct” application registration may also require code signing and other “hoops”.

Where installers Fail

Sometimes an application won’t run after installation. This can happen if your deployment and development conditions differ.

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` needs to be installed but isn’t bundled with your application.
- The runtime environment might need to be configured in a specific way
  - e.g., AWS_SECRET='KASDJFTG_&JGJMHGF_!@GHHY@’
  - e.g., specific network configuration for service location.

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)

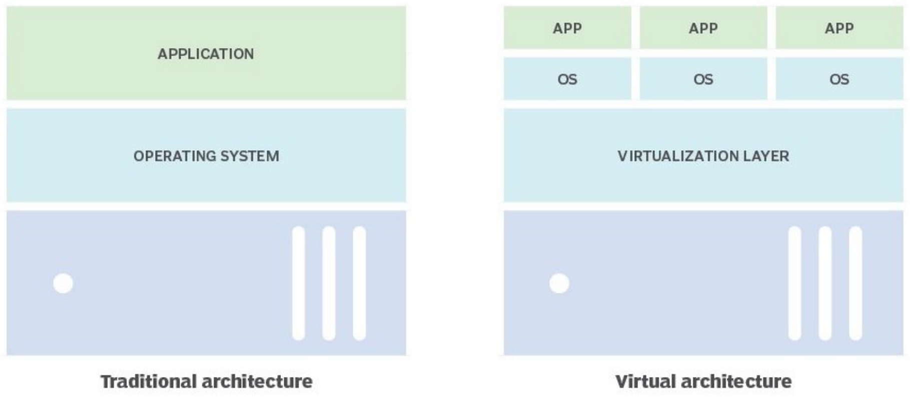


Diagram from TechTarget (2024)

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.

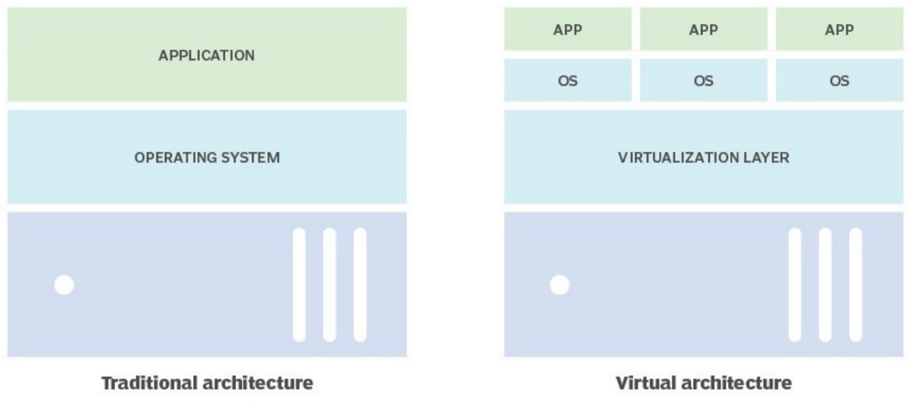


Diagram from TechTarget (2024)

Containerization

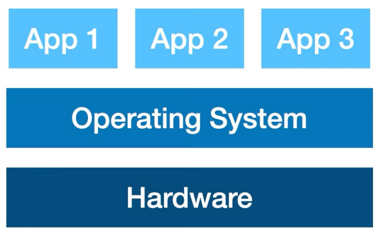


Fig 1. Standalone Hardware

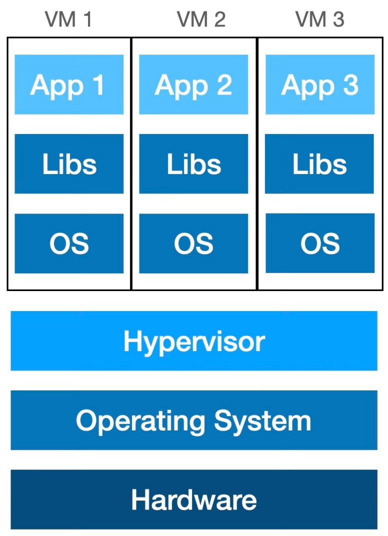


Fig 2. Virtualization

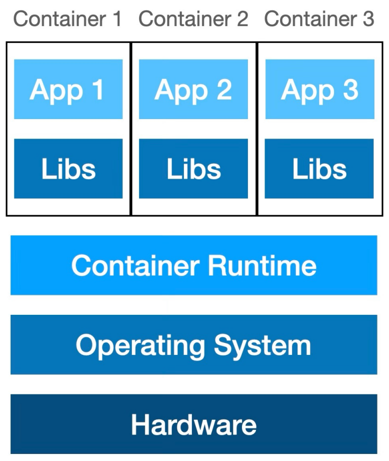


Fig 3. Containerization

One OS that hosts everything

Dedicated virtual machines

Lightweight containers?

Comparison

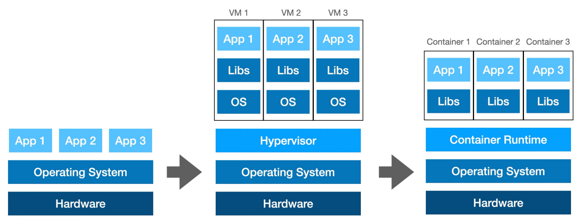


Fig 1. Standalone Hardware Fig 2. Virtualization Fig 3. Containerization

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.

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

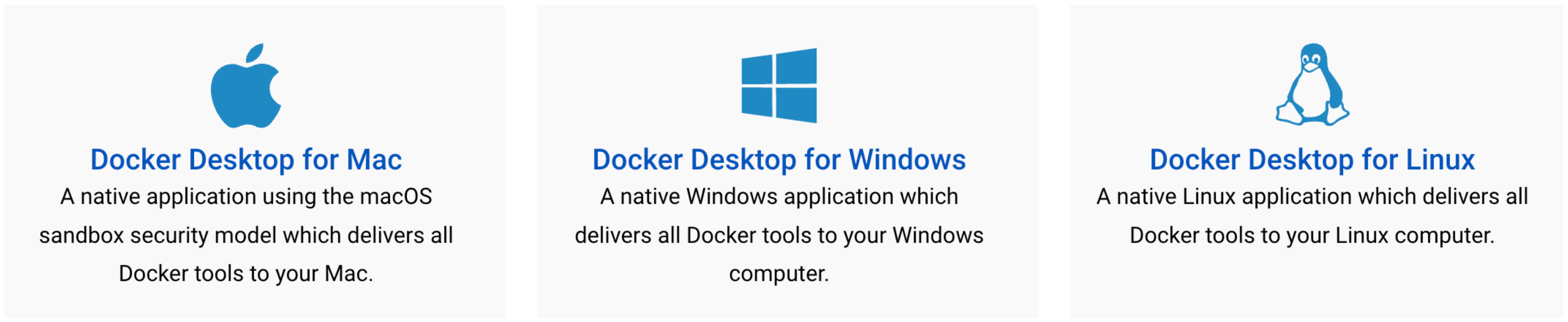
Docker is a **containerization** platform.

- 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


Docker Desktop for Mac
A native application using the macOS sandbox security model which delivers all Docker tools to your Mac.

Docker Desktop for Windows
A native Windows application which delivers all Docker tools to your Windows computer.

Docker Desktop for Linux
A native Linux application which delivers all Docker tools to your Linux computer.

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

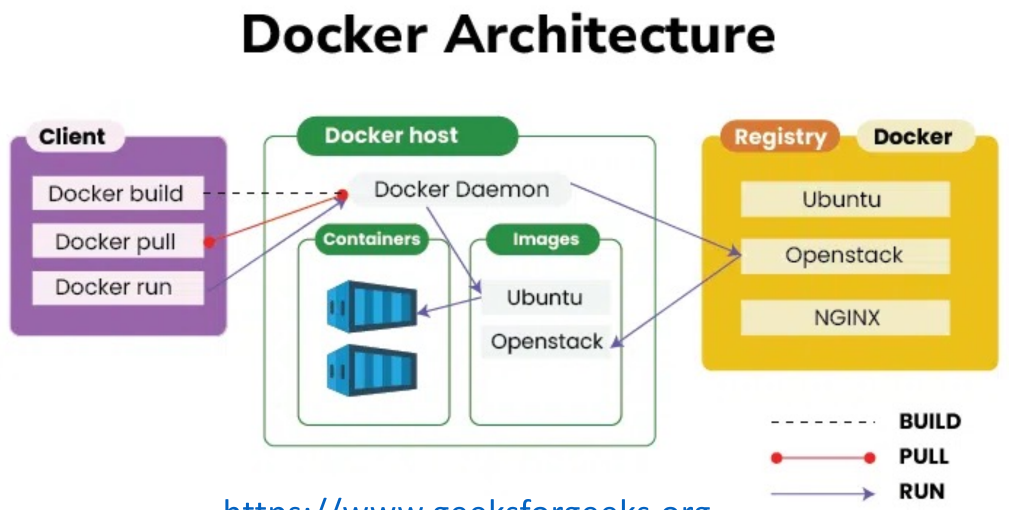
Install Docker from installers on their website or your favorite package manager. e.g., `brew install docker` on macOS.

```
$ 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 be exposed under the /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
```

Mapping port numbers

- 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"]
```

When should I use a container?

- Do not use this for a standalone application.
- If you built a web service – this is a great way to run it locally!
  - You can push your service to DockerHub and just provide instructions for the user (TA) to pull and run the image.