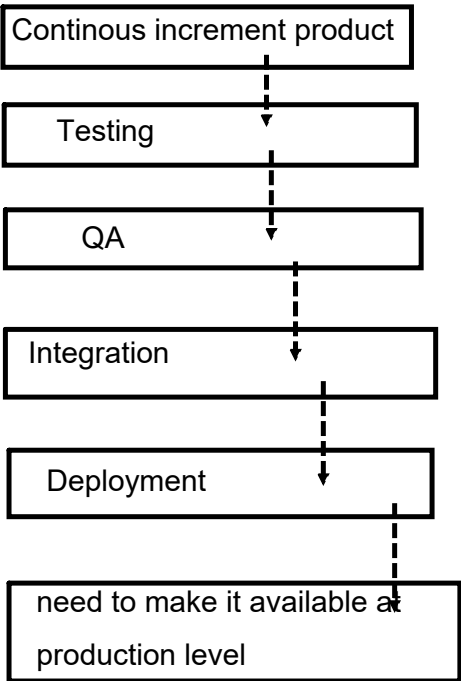


Agile Development : incremental iterative development process

Output:

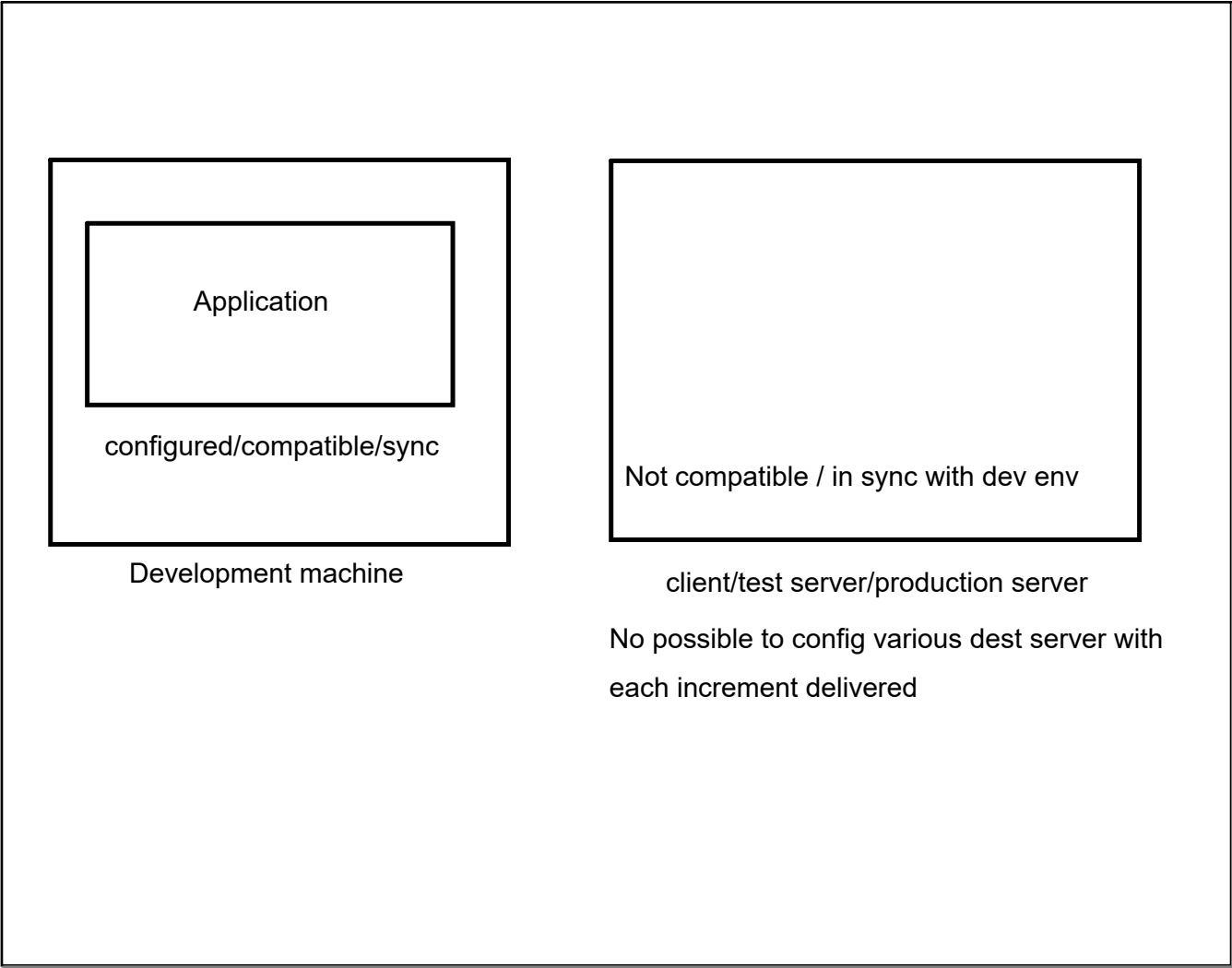


continous and smooth

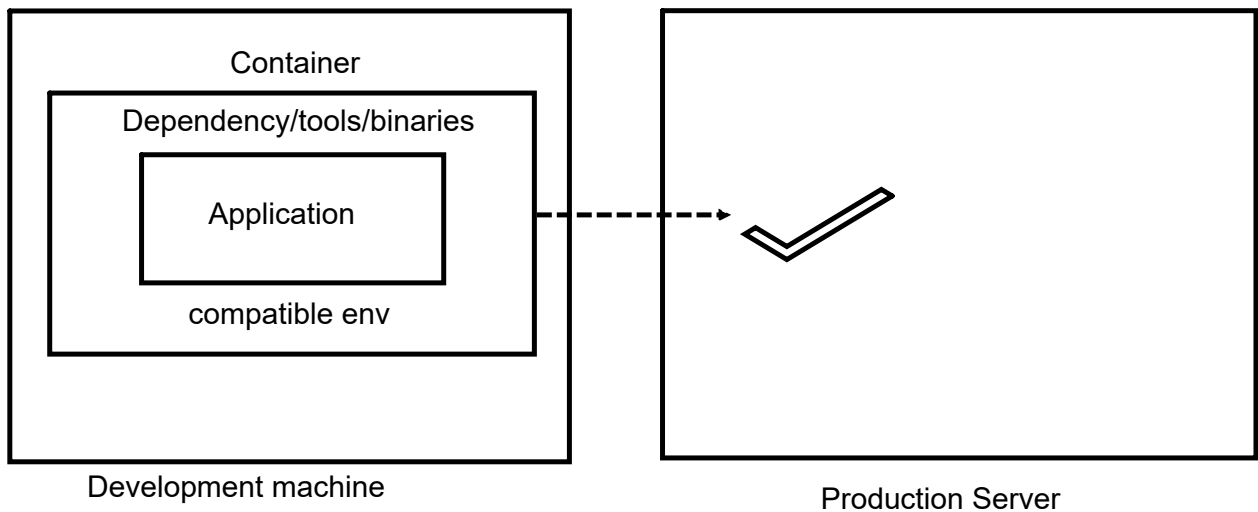
challenges : variation in platform for these different activities

"works on my machine"

Solution : Containers (DOCKER)



Container ship solution



We ship the complete container to destination server : no more concerned about dest config:

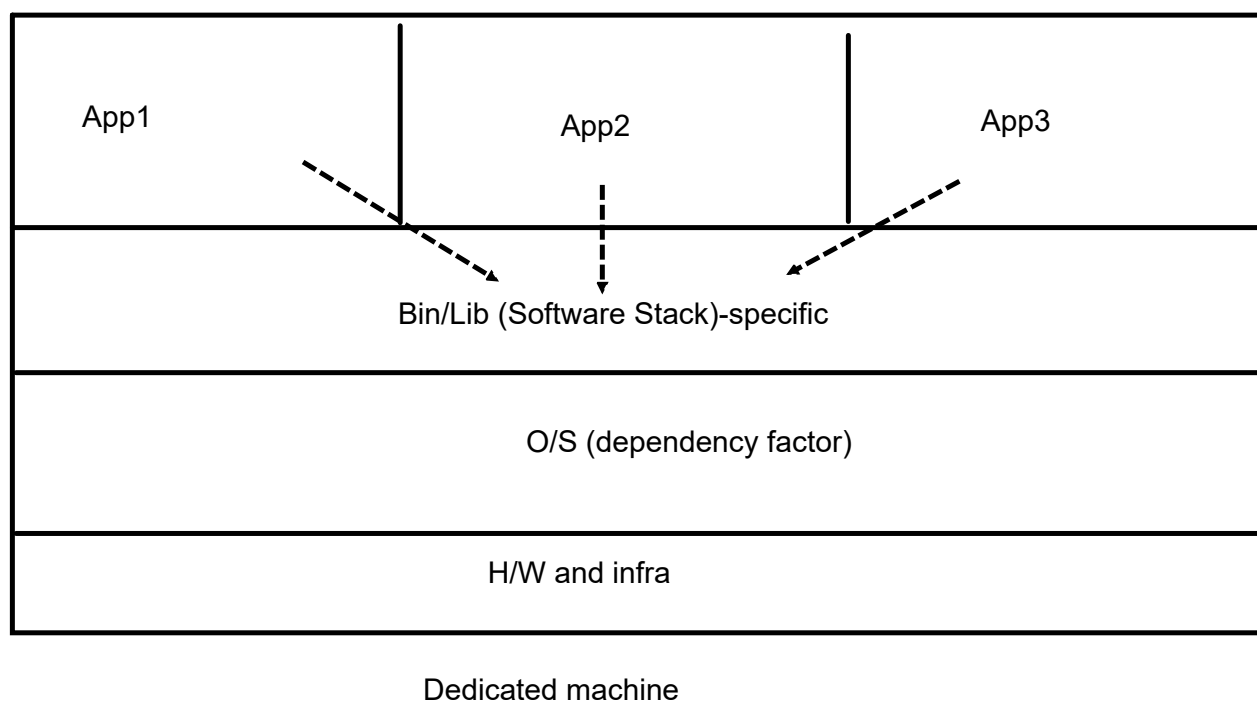
Environment setup :

1. Dedicated machines

Machine specifically configured for a particular environment

#can run a specific type of application

#needs to re-configured continuously as per application demand



#Virtual Machine:

#Multiple env working on same machine

#able to run diff app with diff req on same machine

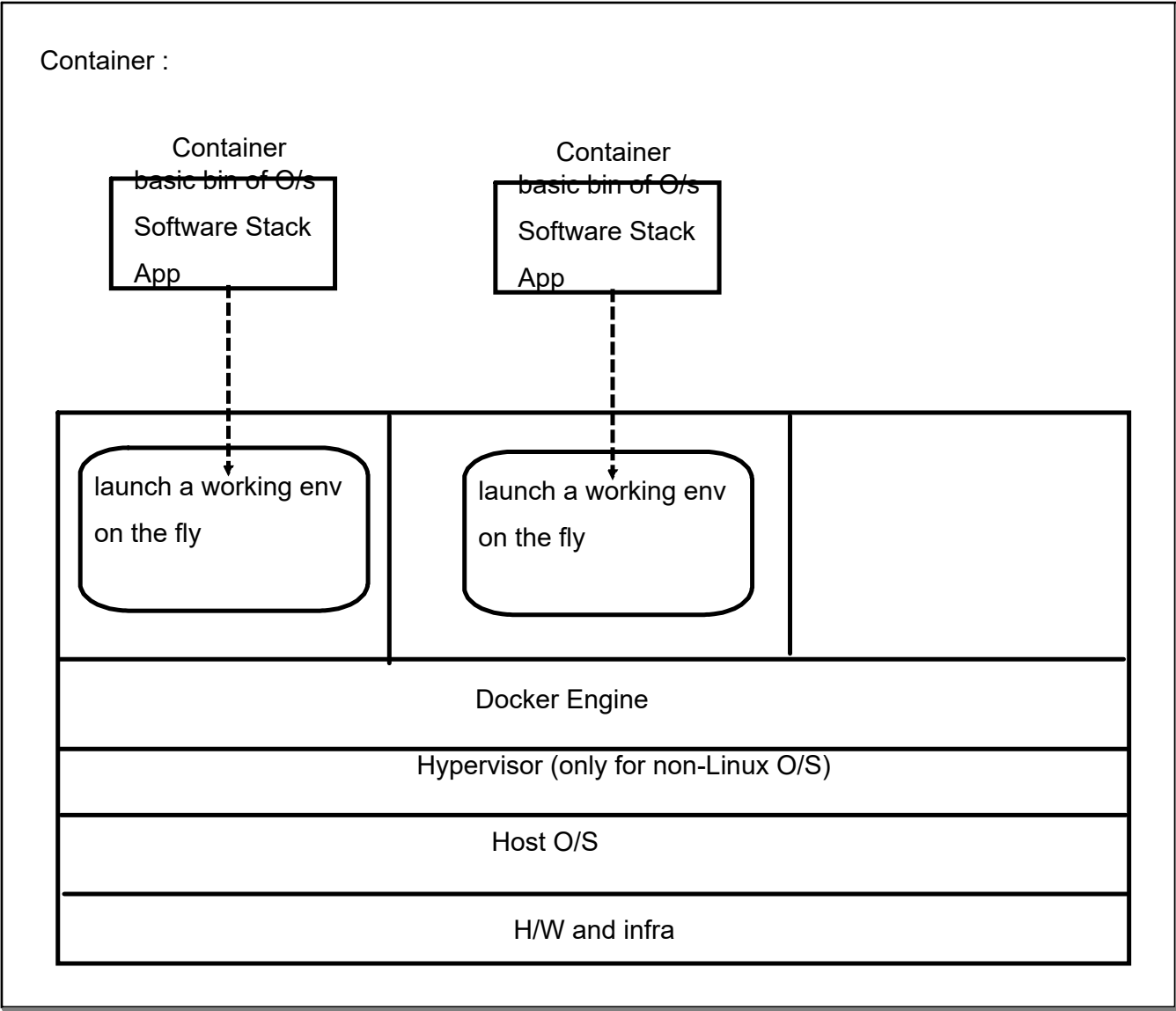
#Allows to have multiple guest O/S working on top of HOST O/S

App1	App2	App3
Software Stack 1	Software Stack 2	Software Stack 3
Guest O/S 1	Guest O/S 2	Guest O/S 3
Hypervisor (allows to plugin other O/S over host O/S)		
Host O/S		
H/W and infra		

Each VM is a heavy weight implementation (eats up lots of system resources)

Dynamic scaling/ spinning up new VM is not easy-time consuming

In case of new requirements / new version : need to reconfigure the VM



Docker(Container) Advantages:

1. Lightweight
2. Containers can quickly spin-ed up as per req (scale-up/scale-down)
3. any modification in app software stack need not be configured in host machine

Environment setup:

download the docker setup (Community Edition) : majorly contains DOCKER ENGINE

#Docker Engine require Linux platform

#if installing on WIN/MAC : Hypervisor (already available with setup)

=>Docker Hub (online repository of containers/docker-images) :

#need to a registration account on docker hub

=>will help to pull/push the docker-images in hub

Windows Powershell (bash terminal) : to run docker command (Docker CLI)

Docker Images:

Docker Container : runtime entity (env)

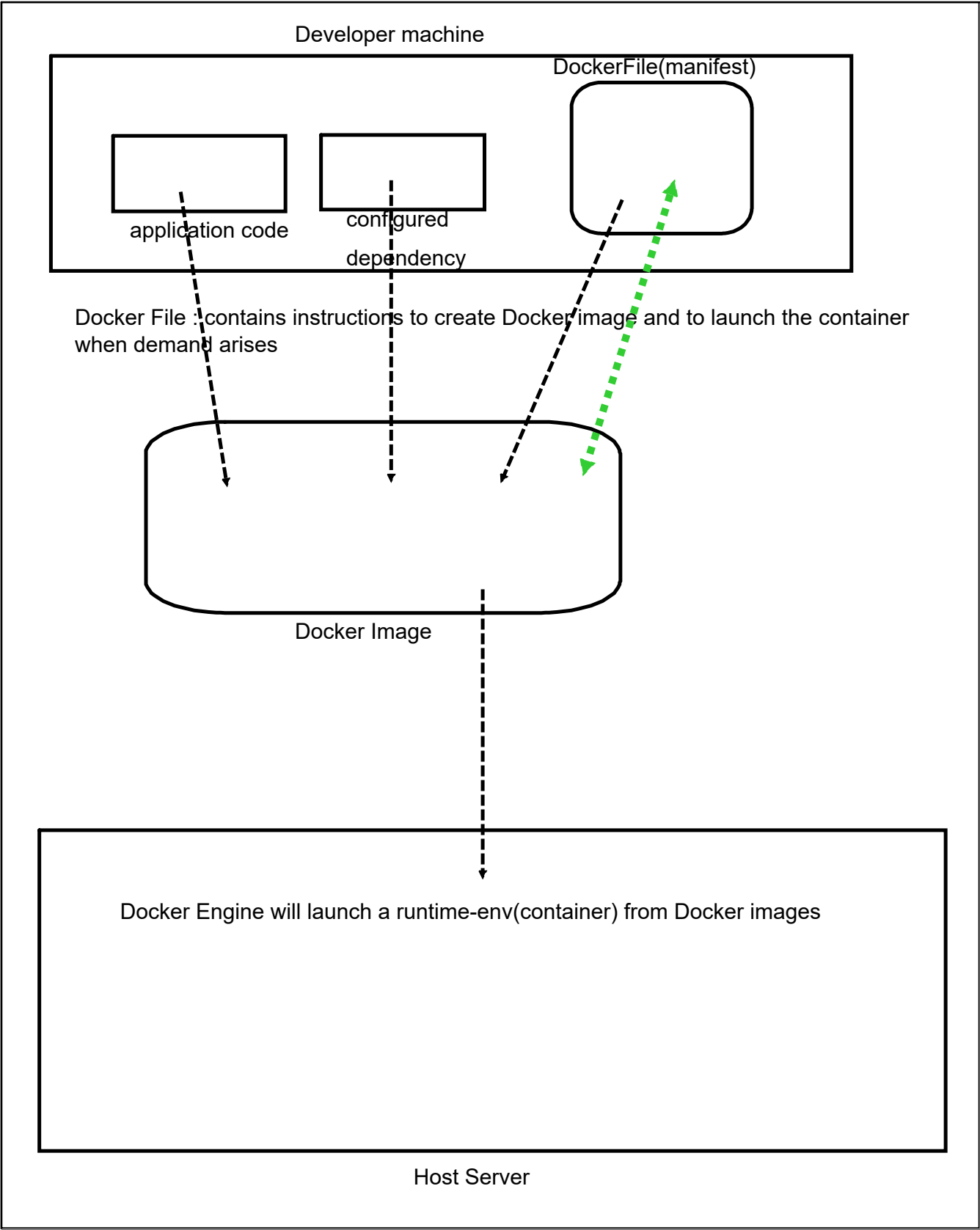
Docker Images : a file containing information to create/launch/spinup a container

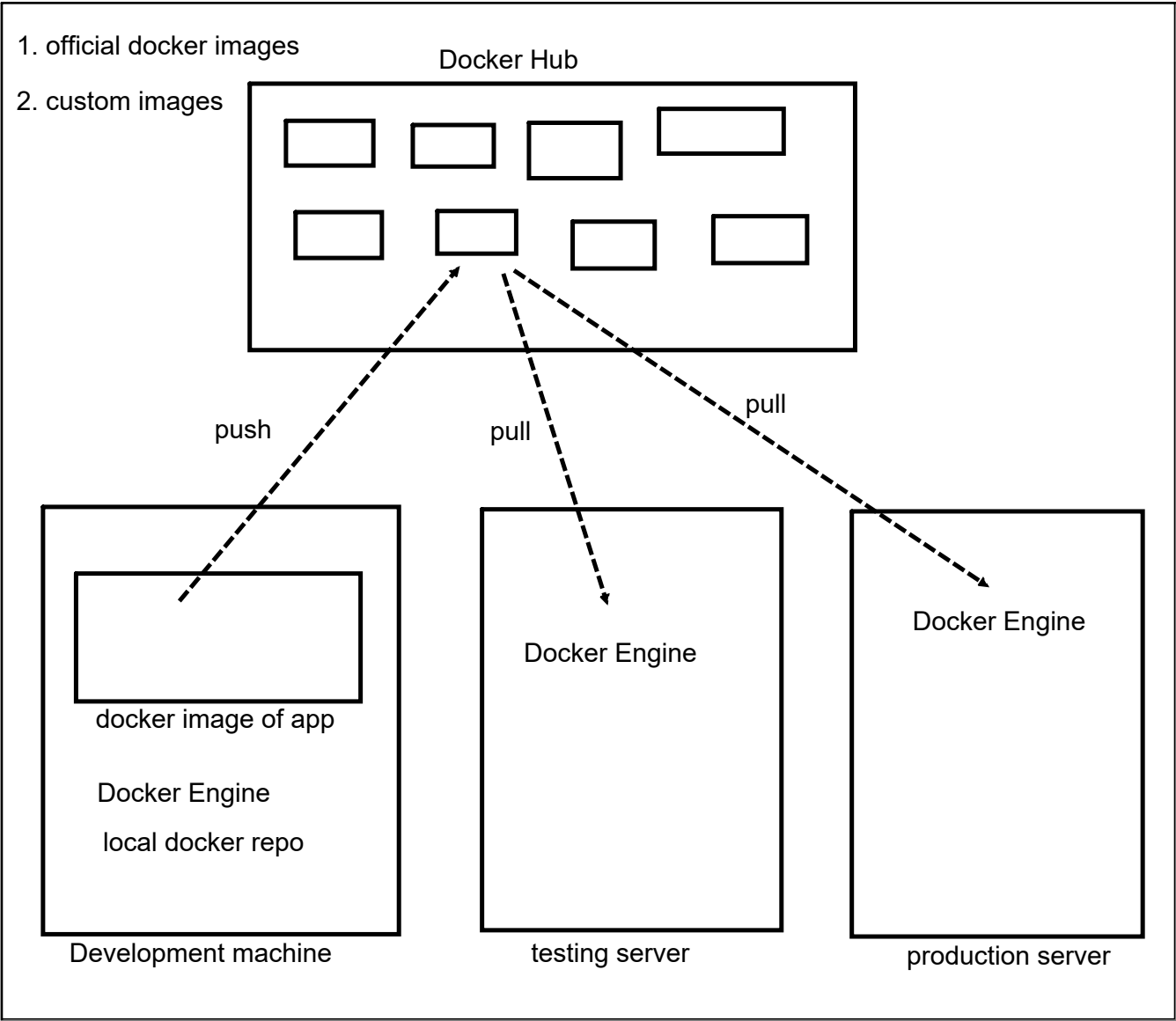
=>Application code

=>File system of base runtime env

=>Tools

=>runtime instructions to be followed when container is launched





Docker Engine also maintains a local docker repository of images

=>docker images

#lists all docker images in local repository

=>docker container ls

Image properties

#Repository : image name

#Tag : additional info/differentiate images

#image id: unique

#created : date/time of creation

#size : size of image file

#getting image from docker hub(pull)

==>docker image pull <image name:tag name>

by default tag name: latest

#launching application of docker image (launching container: preparing runtime env and running the application inside it)

==>docker container run <image-name> (used to both pull image and launch the container)

#container is live only till application is running

#removing image from local repository

==>docker rmi <image-name/image-id>

==>docker rmi -f <image-name/image-id> (forceful removal)

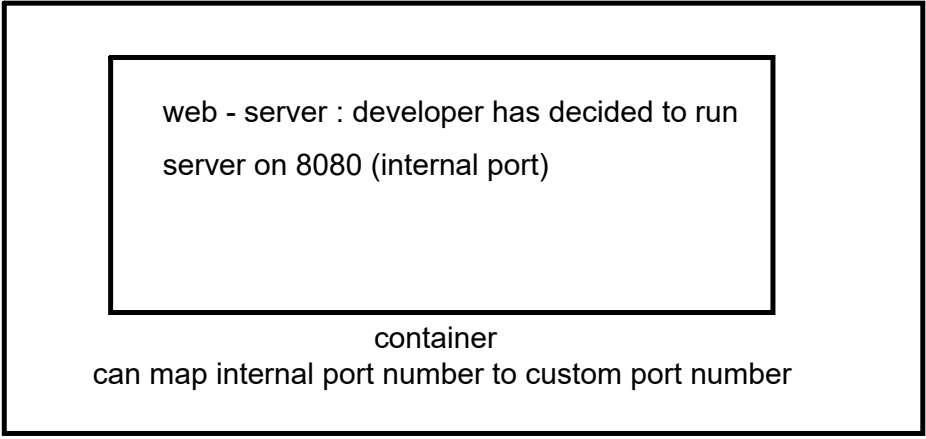
nginx server (official docker image on docker hub)

1. blocking application -server : by default when a container is launched : attached mode
(block powershell terminal)

option : detached mode (-d)

2. server : engage a port as well (-p <internal port>:<custom port>)

==>docker container run -d -p 80:2000 nginx



web - server : developer has decided to run
server on 8080 (internal port)

container
can map internal port number to custom port number

Host machine

=>custom application (HTML page)

Docker image containing complete env dependency (web server):nginx

Manifest file : named as Dockerfile (without any extension)

==>node js application :

Docker command : used inside manifest file...

FROM

LABEL

RUN

COPY/ADD

EXPOSE

ENTRYPOINT/CMD

#build docker image

==> docker build -t <image name> <location>

#stop the container

docker container stop <containerId>

FROM : tells Docker the base (docker image) needed for current images

LABEL : Additional info: version,description,author

RUN : interaction where we specify to install some software, command, scripts

COPY : copy local resources to some container based file-system

ADD : copy the remote res to some container based file-system

EXPOSE : which describes the containers port number on which app will going to run inside the container: this port number does not conflicts with host machine port number

CMD: used to specify command to run at launching of container

#push image to docker hub:

custom-node-app:latest : cannot be pushed directly

rename it : any custom docker image (other than official docker images)

must be associated with user-id

Prepare image to push

==>docker image tag <current image name> <new image name>

(creates alias)

Pushing image on docker hub

==>docker push <image name>

#used to mention dependency on any other docker image

FROM nginx:latest

#nginx has its own file system

#need to specify a particular location as work dir (resource shall be uploaded at that loc)

WORKDIR /usr/share/nginx/html

#copy index.html to working dir (deployment)

COPY index.html index.html

```
#dependency on other docker image
#1. nodejs images (official node js image)
#2. alpine image (lightweight linux env)
#Docker hub contains a clubbed image (nodejs + alpine)

FROM node:9.3-alpine

#EXPOSE internal port number

EXPOSE 3000

#dependency over alpine package tini (req to run nodejs)
RUN apk add --update tini

#need to copy package.json (contains exact nodejs api req ) to alpine file system
COPY package.json package.json

#need to run a command to install those dependencies
RUN npm install

#best practices for NodeJs : clean cache
RUN npm cache clean --force

#Copy all application code from local machine to docker image
COPY . .

##uptil now all command will setup the required bin for application run

##must also contain command to be executed at the time of launching of container (not at
time of creation of docker image)

#tini --node ./bin/www
#CMD tini --node ./bin/www (not going to work)
CMD ["tini","--","node","./bin/www"]
```

```
FROM java:8-jdk-alpine
```

```
COPY ./target/demo-docker-0.0.1-SNAPSHOT.jar /usr/app/
```

```
WORKDIR /usr/app
```

```
RUN sh -c 'touch demo-docker-0.0.1-SNAPSHOT.jar'
```

```
ENTRYPOINT ["java","-jar","demo-docker-0.0.1-SNAPSHOT.jar"]
```

Let's take a look at the commands and fully understand them before proceeding:

FROM – The keyword **FROM** tells Docker to use a given base image as a build base. We have used 'java' with tag '8-jdk-alpine'. Think of a tag as a version. The base image changes from project to project. You can search for images on [docker-hub](#).

COPY - This tells Docker to copy files from the local file-system to a specific folder inside the build image. Here, we copy our .jar file to the build image (Linux image) inside /usr/app.

WORKDIR - The **WORKDIR** instruction sets the working directory for any **RUN**, **CMD**, **ENTRYPOINT**, **COPY** and **ADD** instructions that follow in the Dockerfile. Here we switched the workdir to /usr/app so as we don't have to write the long path again and again.

RUN - This tells Docker to execute a shell command-line within the target system. Here we practically just "touch" our file so that it has its modification time updated (Docker creates all container files in an "unmodified" state by default).

ENTRYPOINT - This allows you to configure a container that will run as an executable. It's where you tell Docker how to run your application. We know we run our spring-boot app as `java -jar <app-name>.jar`, so we put it in an array.

More documentation can be found on the [Dockerfile reference page](#).

Before moving further, we need a Spring Boot .jar file. This file will be used to create the Docker image as mentioned above.

Run the `mvn clean install` command to make sure that it's generated.

Let's build the image using this Dockerfile. To do so, move to the root directory of the application and run this command:

```
$ docker build -t greeting-app .
```

Dockerizing using Maven

In the previous section we wrote a simple Dockerfile and build our application using the native docker build command. However, there are a couple of issues that we may encounter in our projects using this method:

The .jar name - We have to mention the jar name (along with the version) in the file. As our application grows our versions will change and we have to, again and again, update this Dockerfile too.

Using the terminal - We have to manually open a terminal and run Docker commands. It would be nice if we could make it a part of a Maven life-cycle so that we can build images as a part of our CI/CD (Continuous Integration/Continuous Delivery) pipelines.

There are many Maven plugins available that we can use in our pom.xml file that would make our life much easier. The way that this Maven plugin works is that it internally creates the Dockerfile based on the configuration in the pom.xml file and then uses the generated Dockerfile to build the image.

Using this method, there's no need for us to manually update the name, nor run the terminal.

We will be using the fabric8io/docker-maven-plugin.

The plugin should be located in our pom.xml file after the build tag. This will be an optional build plugin using Maven profiles. It's always a good idea to use this via profiles because we want the regular mvn clean install command to work on a developer's machine, which doesn't have Docker installed too:

```
<profiles>
<profile>
  <activation>
    <property>
      <name>docker</name>
    </property>
  </activation>
  <build>
    <plugins>
      <plugin>
        <groupId>io.fabric8</groupId>
        <artifactId>docker-maven-plugin</artifactId>
        <version>0.26.0</version>
        <extensions>true</extensions>
        <configuration>
          <verbose>true</verbose>
          <images>
            <image>
              <name>${project.artifactId}</name>
            </image>
            <build>
              <from>java:8-jdk-alpine</from>
              <entryPoint>
                <exec>
                  <args>java</args>
                  <args>-jar</args>
                  <args>/maven/${project.artifactId}-${project.version}.jar</args>
                </exec>
              </entryPoint>
            </build>
            <assembly>
              <descriptorRef>artifact</descriptorRef>
            </assembly>
          </images>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <executions>
    <execution>
      <id>build</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
</profile>
</profiles>
```

Let's take a closer look at this:

Our profile is named docker - If we have to build the image using Maven, we should run the command with -Ddocker.

The <name> tag - This tag specifies the image name, which is the artifactId - in our case, it's demo-docker.

The <from> tag - This tag specifies the base image of java:8-jdk-alpine.

The <args> tag - This tag is used to specify how the image should run.

Now let's build the image:

```
$ mvn clean install -Ddocker
```

```
FROM tomcat:7-jre8-alpine
```

```
# copy the WAR bundle to tomcat
```

```
COPY /target/Dummy-1.0-SNAPSHOT.war /usr/local/tomcat/webapps/app.war
```

```
# command to run
```

```
CMD ["catalina.sh", "run"]
```

The Application.java file in the example looks like this:

```
package hello;
```

```
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication
```

```
public class Application {
```

```
    public static void main(String[] args) {
```

```
        SpringApplication.run(Application.class, args);
```

```
    }
```

```
}
```

This is a valid SpringBoot application, but NOT a deployable application to Tomcat. To make it deployable, you can can:

redefineApplication to extend SpringBootServletInitializer from Spring framework web support; then

override the configure method:

```
package hello;
```

```
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
import org.springframework.boot.builder.SpringApplicationBuilder;
```

```
import org.springframework.boot.web.support.SpringBootServletInitializer;
```

```
@SpringBootApplication
```

```
public class Application extends SpringBootServletInitializer {
```

```
    @Override
```

```
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
```

```
        return application.sources(Application.class);
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        SpringApplication.run(Application.class, args);
```

```
    }
```

```
}
```