OpenShift Overview

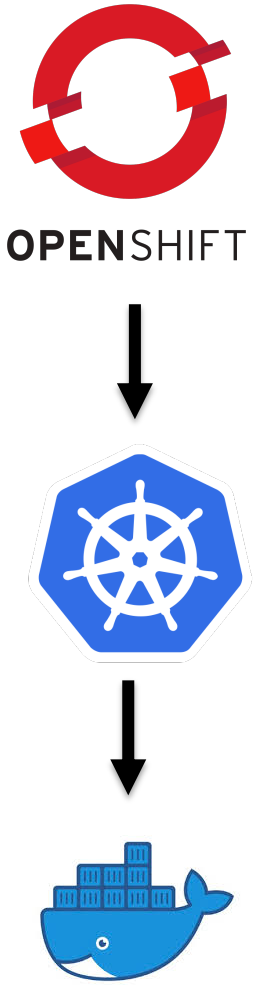
IBM **Developer**

Sudharshan Govindan

sudharshan.govindan@in.ibm.com

@sudhargovindan

Open Source



What are Containers (not Docker)?

Similar to VMs but managed at the **process level**

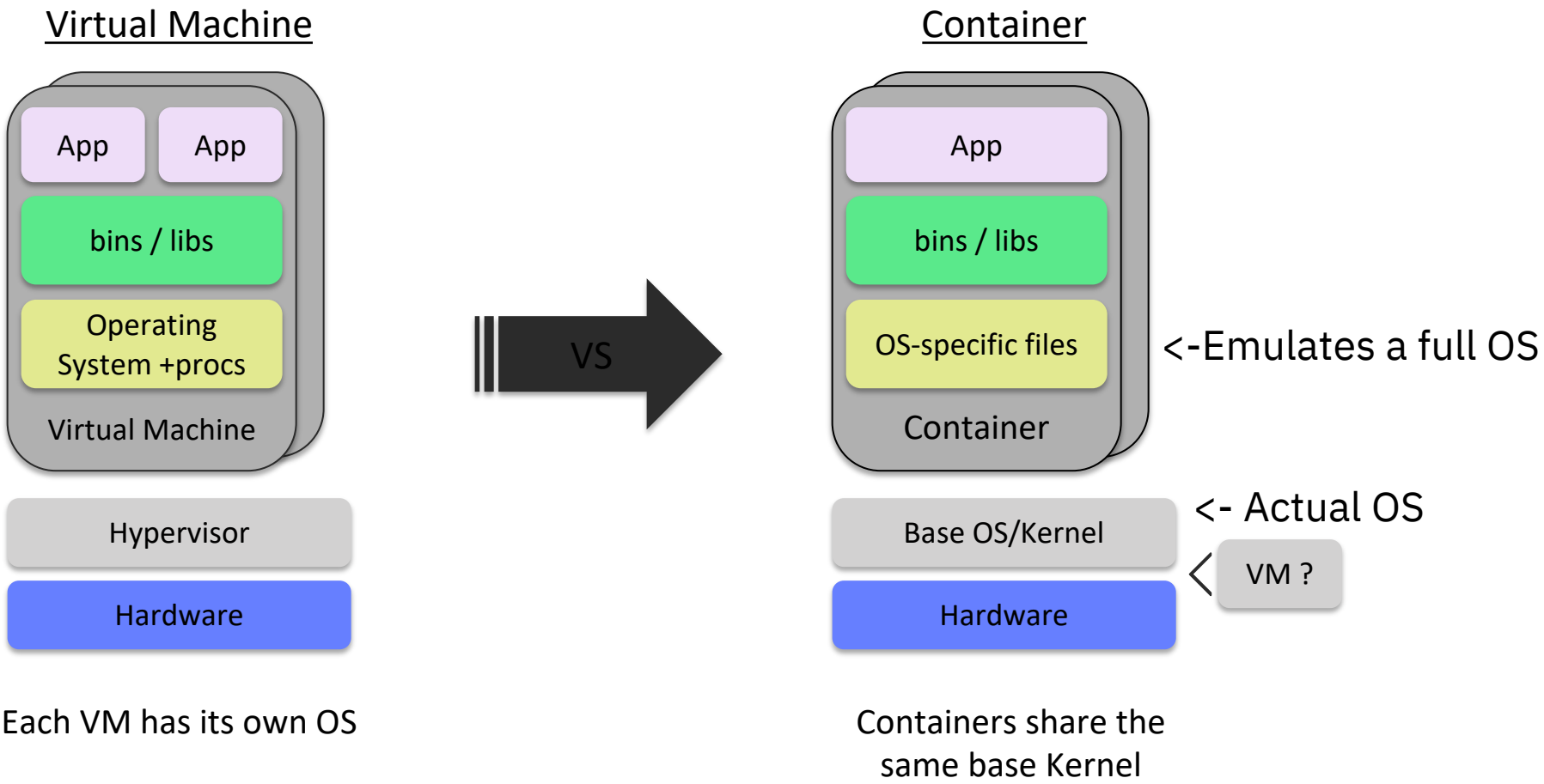
"VM-like” isolated achieved by set of "**namespaces**" (isolated view)

- PID –isolated view of process IDs
- USER- user and group IDs
- UTS - hostname and domain name
- NS - mount points
- NET - Network devices, stacks, ports
- IPC - inter-process communications, message queues

cgroups - controls limits and monitoring of resources

The key statement: **A container is a process(es) running in isolation**

VM vs Container



Each VM has its own OS

Containers share the same base Kernel

What is Docker?

Containers is the technology, Docker is the **tooling** around containers

Without Docker, containers would be **unusable** (for most people)

Docker **simplified** container technology to enable it for the masses

Added value: Lifecycle support, setup file system, etc

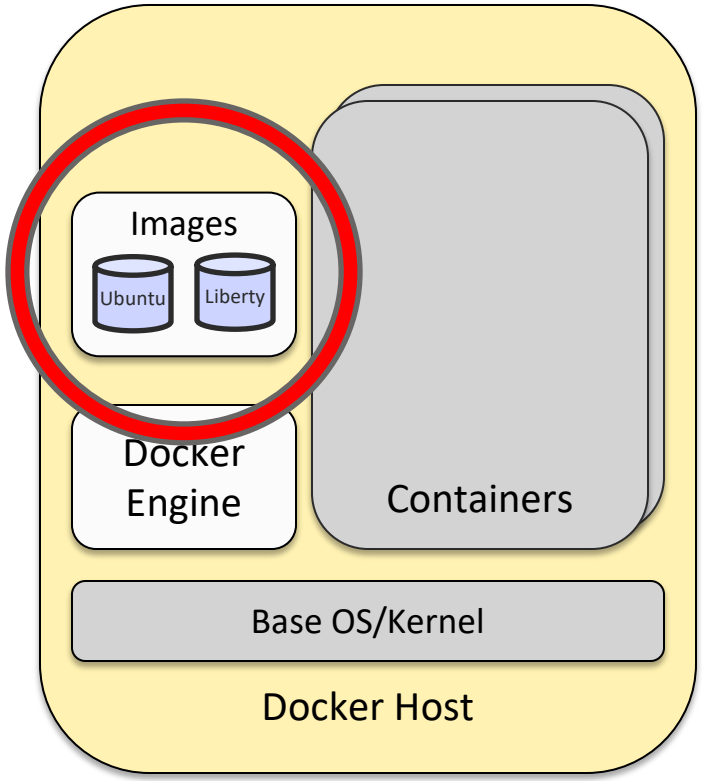
For extra confusion: Docker is also a company, which is different then Docker the technology...

Docker Images

Tar file containing a container's filesystem + metadata

For sharing and redistribution

– Global/public registry for sharing: DockerHub



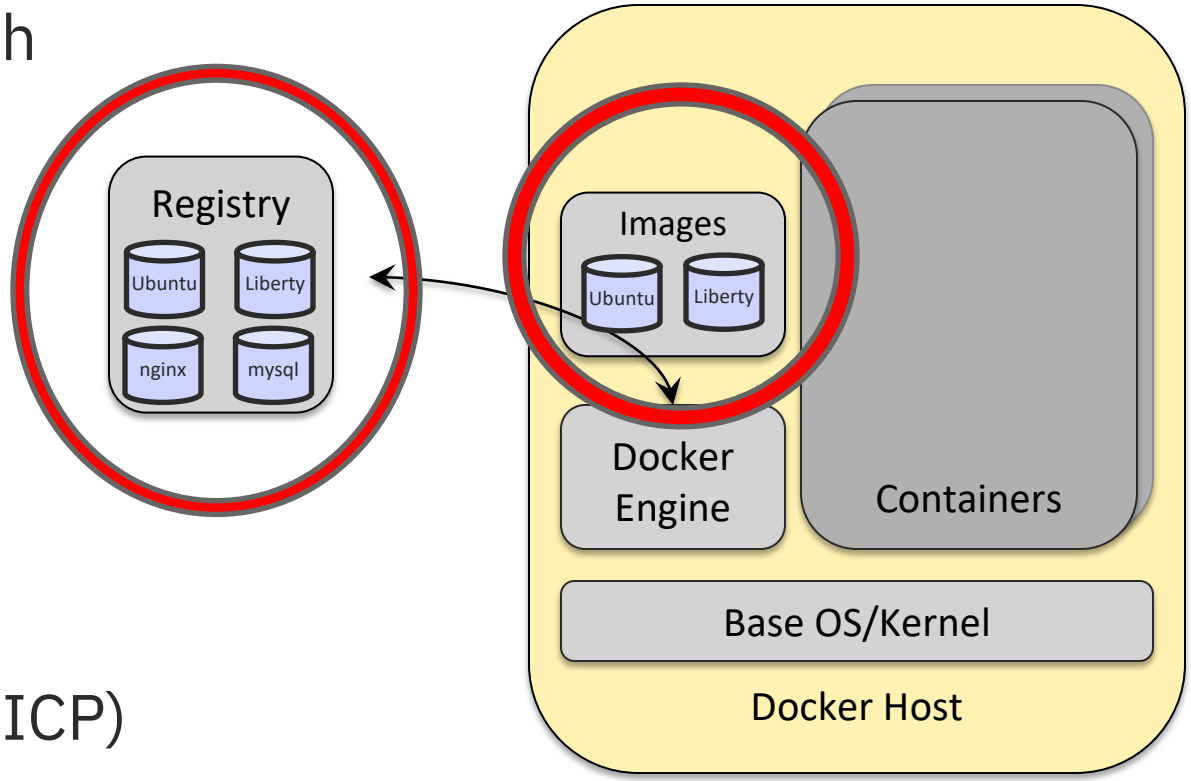
Docker Registry

The central place to share images with friends! (or coworkers)

DockerHub - http://hub.docker.com

- Public registry of Docker Images
- Also useful to find prebuilt images for web servers, databases, etc

Enterprises will want to find a private registry to use (such as one built into ICP)



Build your own image with a Dockerfile!

Step 1) Create Dockerfile to script how you want the image to be built

```
FROM java:8 # This might be an ubuntu or...
COPY *.jar app.jar
CMD java -jar app.jar
```

Layer
Layer
Layer

Step 2) **docker build** to build an image

Step 3) **docker push** to push to registry

Step 4) From another location, **docker pull** to download an image

Shared / Layered / Union Filesystems

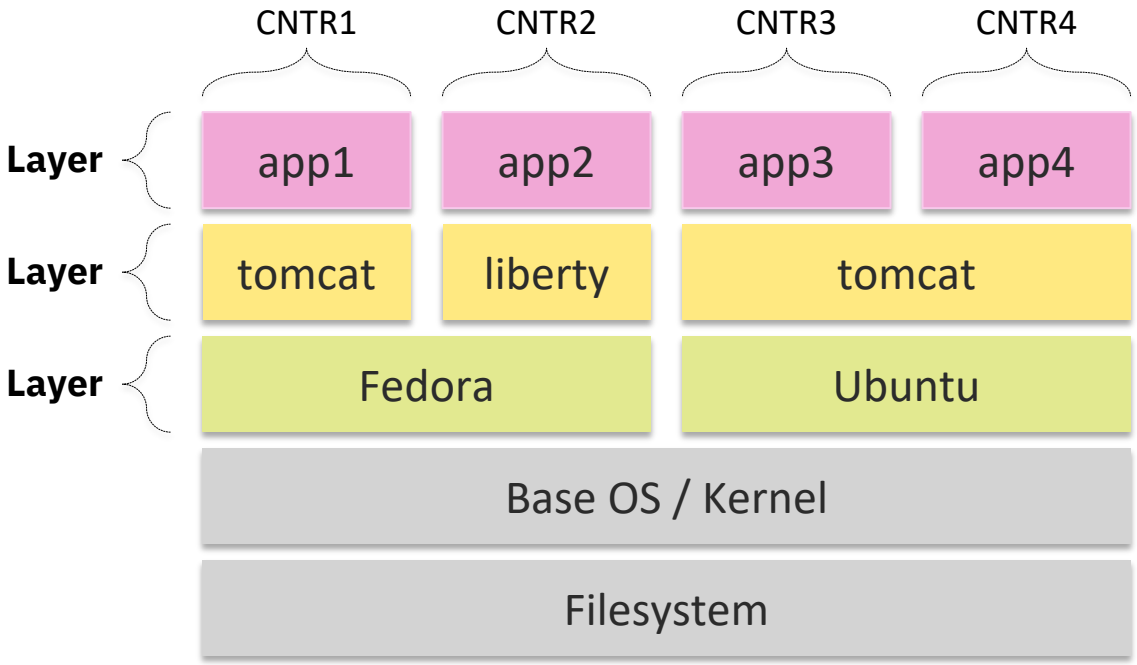
Read-only image layers are shared via Union File System and copy-on-write

At Build Time

- Layers are built on top of each other
- Layers that are already built, are not re-built
- Layers that are already pushed, are not re-pushed

At Run Time

- Containers based off the same image reuse the same layers
- Many containers from the same image, just create the top layer.



In a Traditional Deployment...

Are you testing these on ever commit?

Code (packages archive) ✓

App server ✗

Runtime versions ✗

System libraries and versions ✗

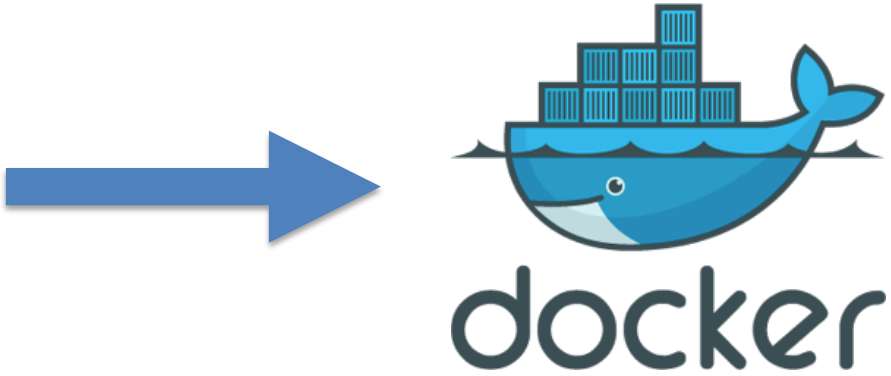
Container = Code + Dependencies

Code (packages archive)

App server

Runtime versions

System libraries and versions

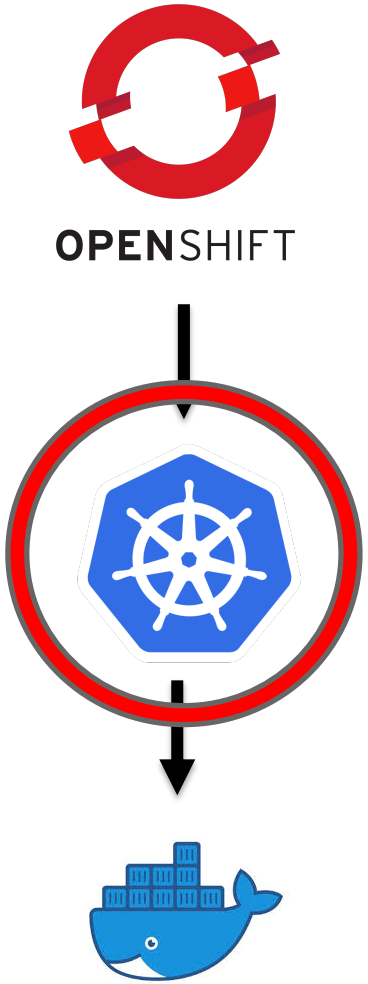


Dependency Matrix from Hell

	Development VM	QA Server	Single Prod Server	Production Cluster	Public cloud	Developer's Laptop	Customer Servers
Static Website	?	?	?	?	?	?	?
Web Frontend	?	?	?	?	?	?	?
Background Workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

Source: https://blog.docker.com/2013/08/paas-present-and-future/

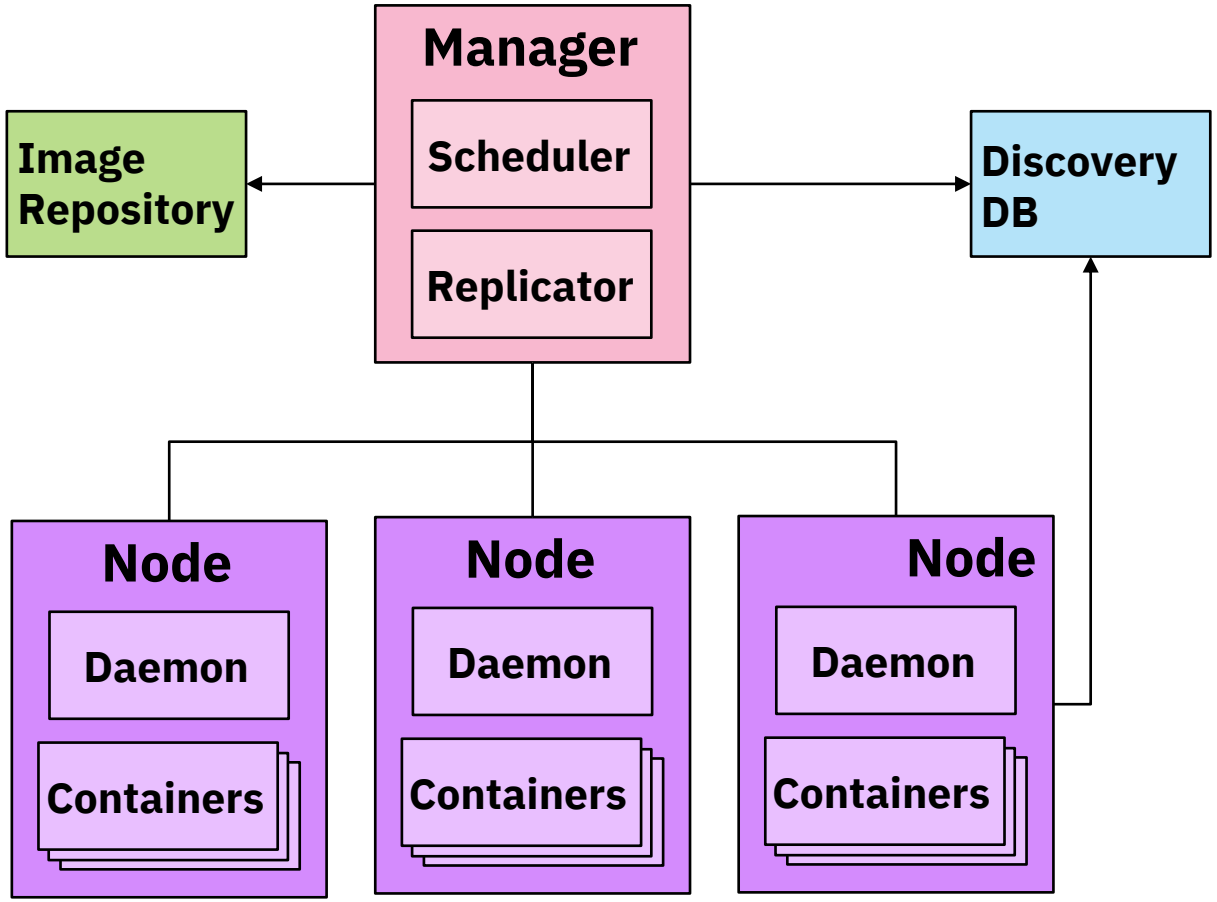
Halfway up the stack



What is container orchestration?

For a distributed set of nodes:

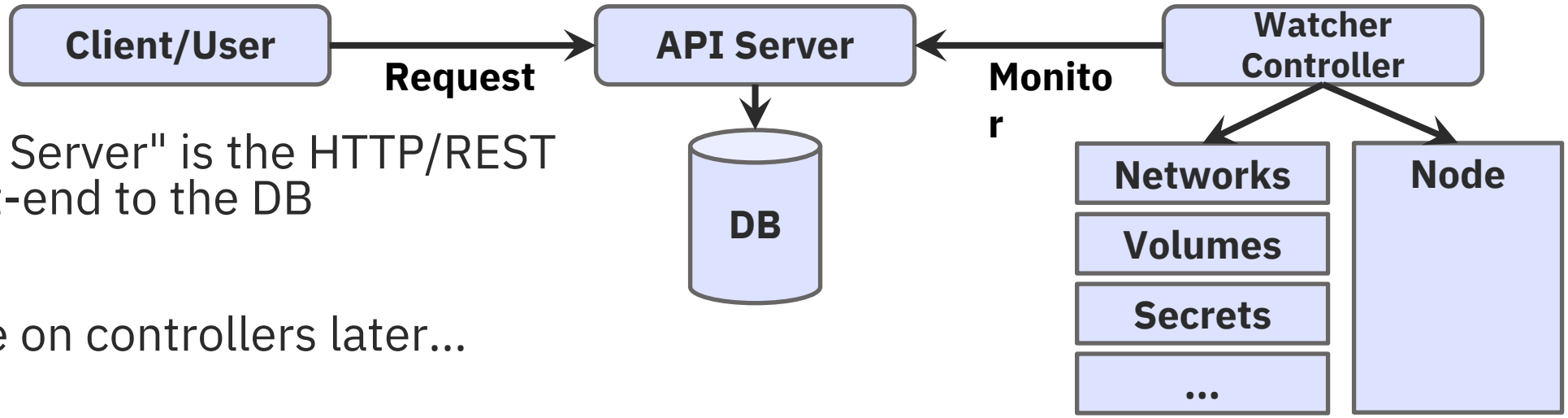
- Scheduling
- Service discovery
- Scaling up/down
- Health management
- Manage cluster resources



Kubernetes Architecture

- At its core, Kubernetes is a database (etcd).
  With "watchers" & "controllers" that react to changes in the DB.
  The controllers are what make it Kubernetes.
  This pluggability and extensibility is part of its "secret sauce".

- DB represents the user's desired state
  - Watchers attempt to make reality match the desired state



"API Server" is the HTTP/REST front-end to the DB

More on controllers later...

Kubernetes Resource Model

A resource for every purpose

- Config Maps
- Daemon Sets
- Deployments
- Events
- Endpoints
- Ingress
- Jobs
- Nodes
- Namespaces
- Pods
- Persistent Volumes
- Replica Sets
- Secrets
- Service Accounts
- Services
- Stateful Sets, and more...

- Kubernetes aims to have the building blocks on which you build a cloud native platform.

- Therefore, the internal resource model is the same as the end user resource model.

<u>Key Resources</u>

- Pod: set of co-located containers
  - Smallest unit of deployment
  - Several types of resources to help manage them
  - Replica Sets, Deployments, Stateful Sets, ...

- Services
  - Define how to expose your app as a DNS entry
  - Query based selector to choose which pods apply

Kubernetes Client

CLI tool to interact with Kubernetes cluster

Platform specific binary available to download

– https://kubernetes.io/docs/tasks/tools/install-kubectl

The user directly manipulates resources via json/yaml

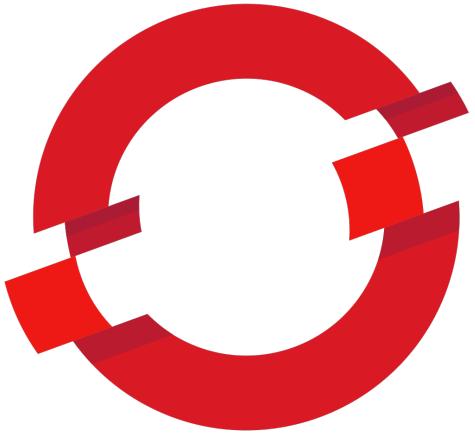
```
$ kubectl (create|get|apply|delete) -f myResource.yaml
```

But Wait? What About Production?

Kubernetes by itself is not enterprise-ready

Kubernetes must integrate with underlying platform to provide infrastructure, storage, etc.

Lacking in operational view + controls, pre-built catalogs

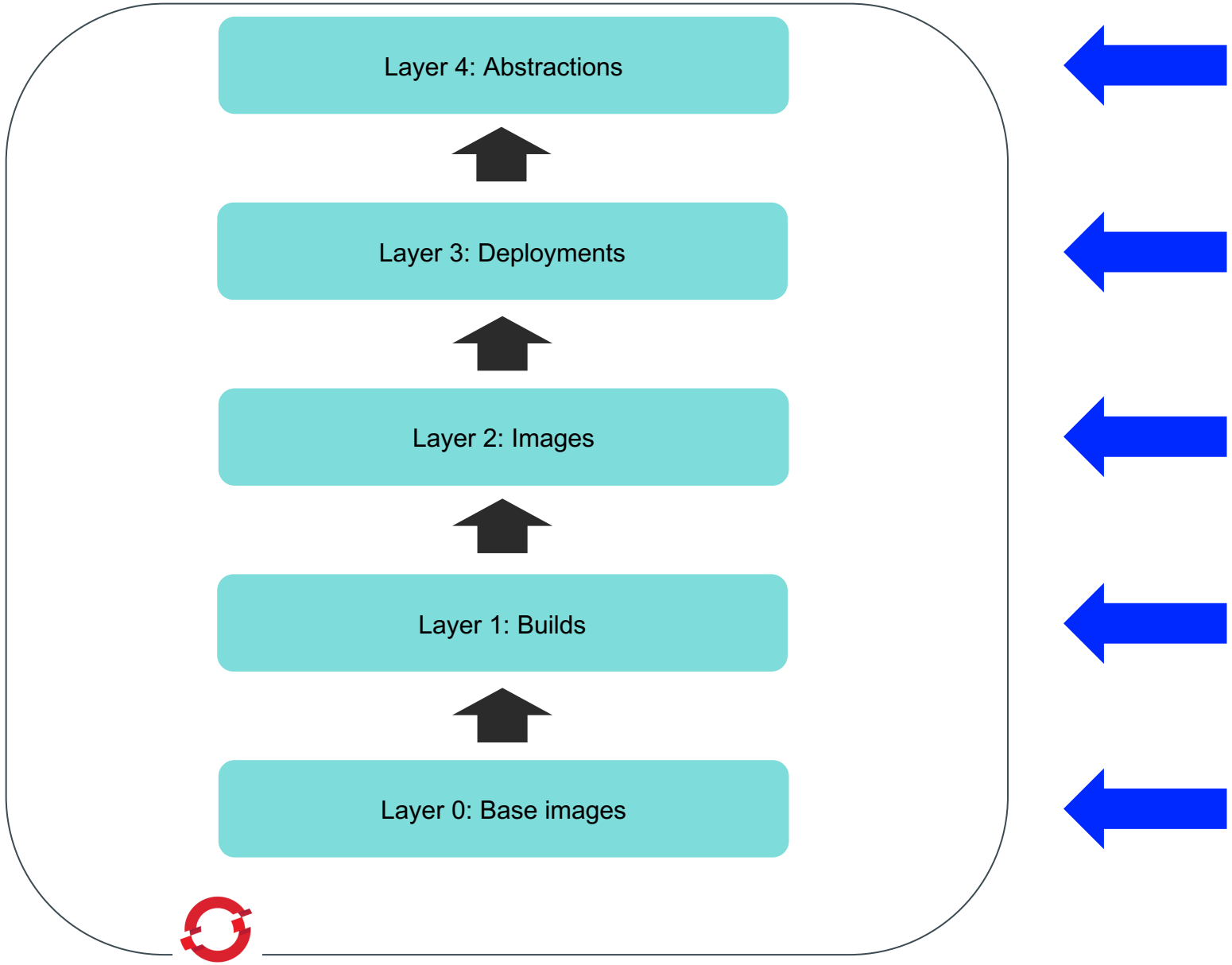


OPENSHIFT

Deploying to OpenShift – S2I and templates

Think of OpenShift deployment as a set of layers

OpenShift Deployment – Developer view



Services, routes, PVCs, secrets etc

Defines what will run in OpenShift

Images created by build layer

Defines the builds required for the app

The base images used by the apps

OPENSHIFT

Layer 0 – Base image examples

- Base operating systems, specific runtimes, databases, application servers and more
  - Available as Image Streams, an enhanced set of metadata about each image

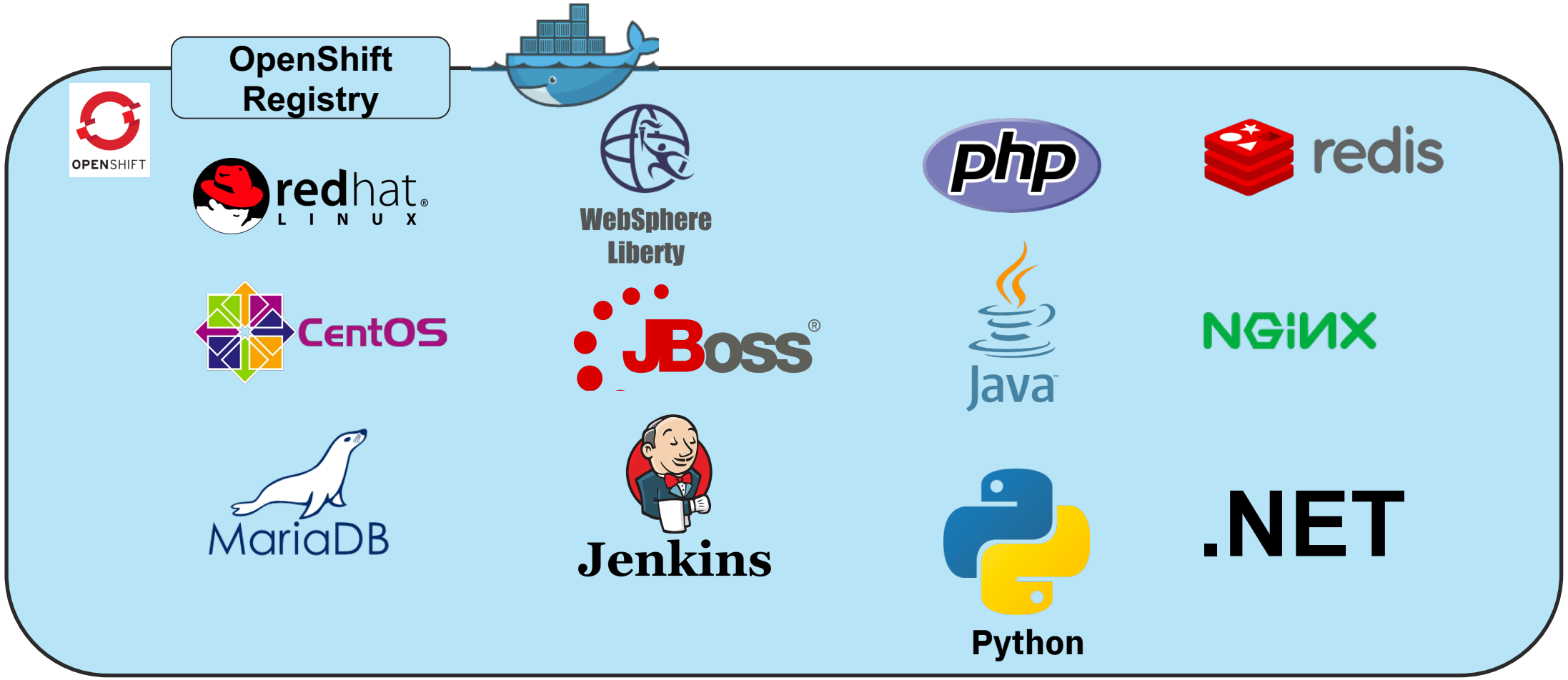
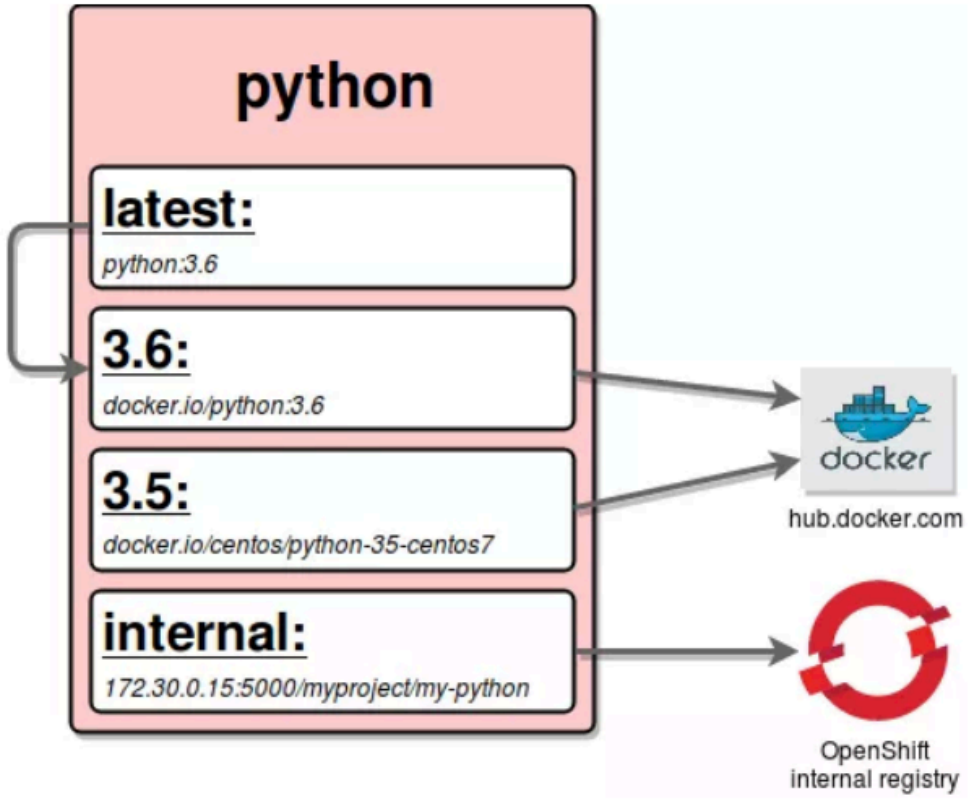


Image Streams

- An image stream represents one or more Docker images identified by tags.
  - Presents a single virtual view of related images
    - Can refer to images from any of the following:
      - Its own image repository in OpenShift's integrated Docker Registry
      - Other image streams
      - Docker image repositories from external registries
    - Image Streams are trigger an event when underlying image is changed (even if tag remains the same)

Image Stream example



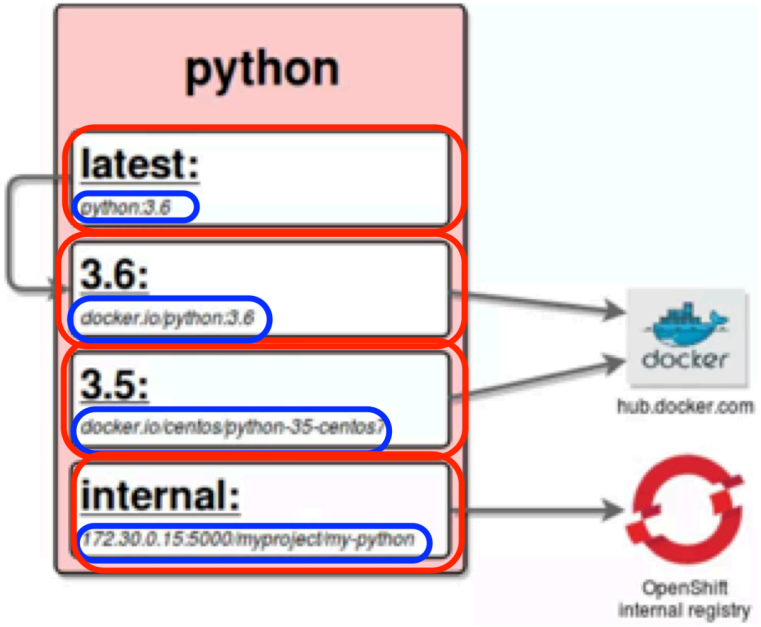
Graphic source: https://blog.openshift.com/image-streams-faq/

Level 1: Builds

- Defined via a **BuildConfig** object
  - A blueprint for a process to transform base images + source code, app binaries, or Dockerfiles to an app image:
  - Key attributes:
    - Input (input to the build process) - e.g. file, directory, GH repo etc
    - Strategy (how to build the app image)
      - Options:
        - ***S2I*** – Use a specialized builder image to generate app image (more later)
        - ***Docker*** - Use a Docker file to generate app image
        - ***Pipeline*** - A Jenkins pipeline that generates the app image from source
        - ***Custom*** – Encapsulate your build process via a custom builder image
    - Output (result of the build process) – typically an ImageStream tag

Level 2: Images

- Defined via an **ImageStream** object
  - An abstraction for working with Docker images inside OpenShift
  - Key attributes:
    - ImageStreamImage (reference to actual image) - typically not used directly
    - ImageStreamTag (reference to a given ImageStream and tag)



Key:

ImageStreamImage

ImageStreamTag

Level 3: Deployments

- Defined via a **DeploymentConfig** object
  - Encapsulates the K8s Deployment and adds deployment strategies and triggers
  - Key attributes:
    - Strategy (how to deploy if the underlying Deployment is already deployed)
      - Options:
        - *Recreate*– Blow away old deployment first and then deploy new one
        - *Rolling (default)* – Zero downtime rollout via K8s *RollingUpdate*
        - *Advanced* - (Note: require routes)
          - *Blue-Green* - running 2 versions of the deployment at the same time and moving traffic from the in-production version (the green version) to the newer version (the blue version).
          - *A/B* – partitioning requests between 2 versions of the deployment at the same time and observing the behavior
    - Triggers (define conditions under which the deployment is automatically triggered)

      e.g Input image updated, config changes

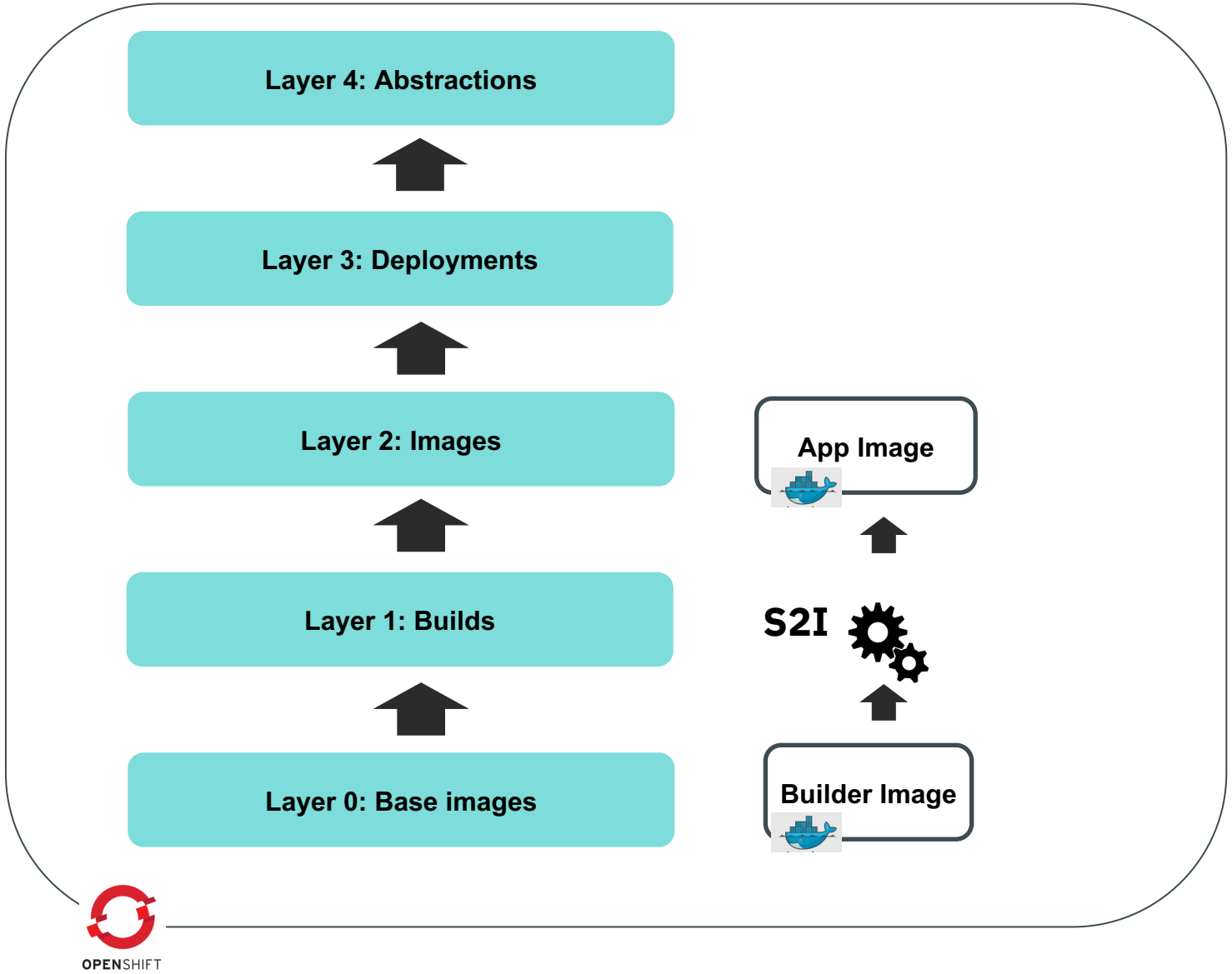
Level 4: Abstractions

- Defines all of the additional resources needed for an app like networking, storage, security etc
  - Frequently used resources: (there are several others)
    - Service – Internal load balancer and router. Directs network traffic to replicated Pods
    - Route – Exposes a Service at a defined host name to allow access from external traffic
    - Persistent Volume Claim – Permanent storage used by apps to persist data after the app has stopped running
    - Secret – Mechanism for holding sensitive data. Decouples sensitive data from the the Pods that use them.

S2I provides a repeatable method to generate application images from source/binary code

S2I Overview

- S2I is a tool that merges source code or binary into an application specific image
  - Uses a builder image
    - OpenShift provides multiple builder images
      - e.g. Node.js, Java
    - Can create and add your own



Anatomy of an S2I Builder Image

Docker file contains S2I specific labels and env vars	`LABEL io.openshift.s2i.scripts-url=image:///usr/local/s2i \`
	`io.s2i.scripts-url=image:///usr/local/s2i \`
	`io.k8s.description=” S2I for Open Liberty Profile" \`
	`io.k8s.display-name="Liberty 19.0.0.4 javaProfile7" \`
	`io.openshift.expose-services="9080/tcp:http" \`
	`io.openshift.tags="runner, builder,liberty" \`
	`io.openshift.s2i.destination="/tmp"`
	
	`ENV STI_SCRIPTS_PATH="/usr/local/s2i" \`
	`S2I_DESTINATION="/tmp"`

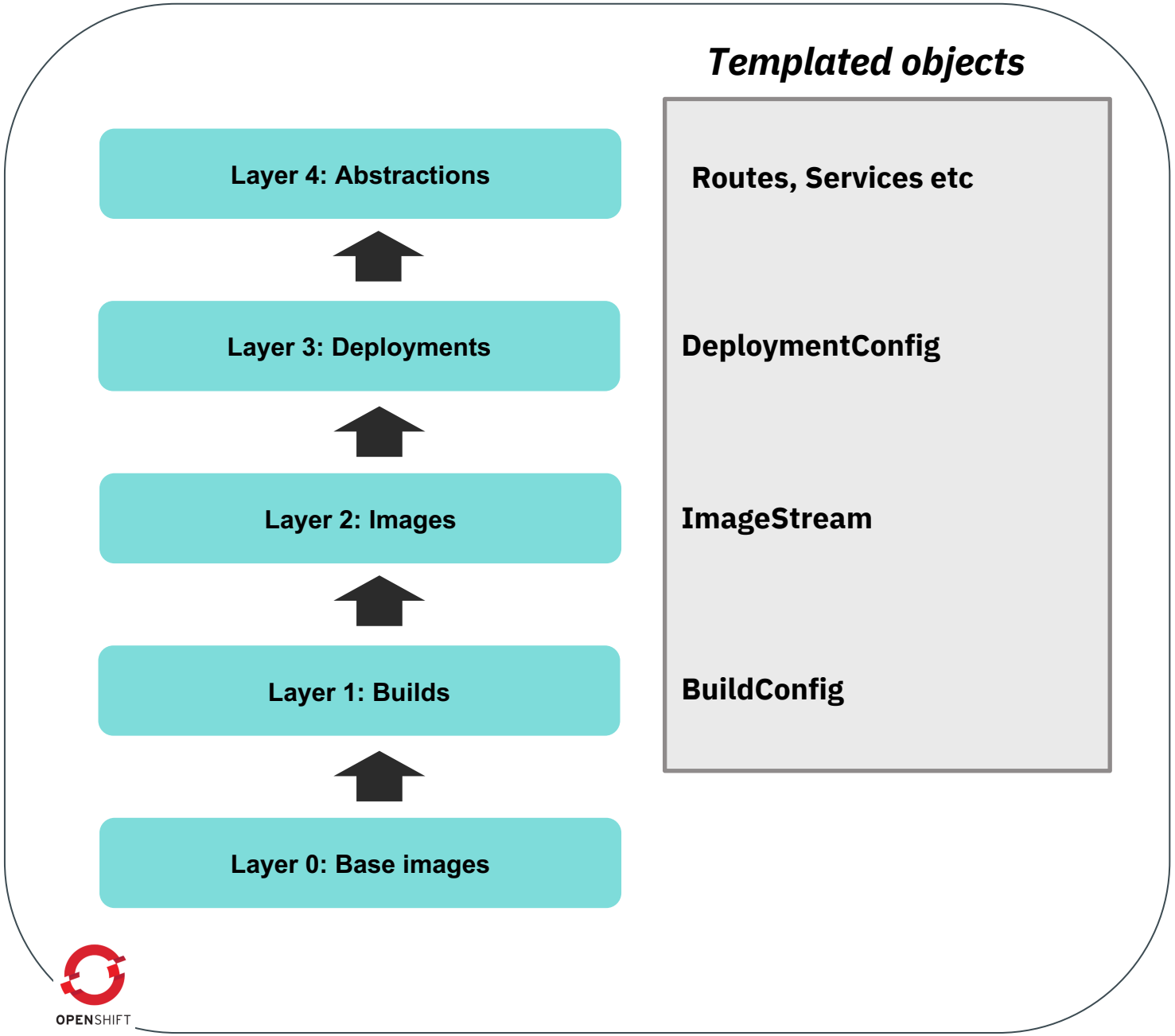
Provides base image for applications generated from the S2I image	`# This S2I image provides a base for building and`
	`# running WebSphere Liberty applications.`
	`FROM websphere-liberty:javaee7`

A series of scripts	**assemble** – Compiles and/or assembles app components from input (required) **run** – Command to run generated app image (required) **usage** – Outputs usage info about the generated image (optional) **save-artifacts** – Saves artifacts to support incremental builds (optional)

Templates provide a parameterized set of objects that can be processed by OpenShift

Templates Overview

- Templates typically contain parameterized objects for the various layers of the deployment process for an app or service
- Once created they are tightly integrated with the Web console and CLI
- Simplifies the deployment of apps or services that require several objects to be created
- OpenShift includes several templates that can be used OOTB
  - e.g. Jenkins, MariaDB etc



Anatomy of a Template

Metadata with name, description	`apiVersion: v1`
	`kind: Template`
	`metadata:`
	`  name: pbw-liberty`
	`  annotations:`
	`  openshift.io/display-name: Plants by WebSphere on Liberty w/MariaDB`
	`  description: Plants by WebSphere on Liberty App using MariaDB as the database`
	`  tags: liberty,websphere`
	`  iconClass: icon-liberty`
	`  openshift.io/provider-display-name: IBM Client Dev Advocacy.`
	`  openshift.io/documentation-url: https://github.com/djccarew/app-modernization-plants-by-websphere-jee6.git`
	`  openshift.io/support-url: https://developer.ibm.com`

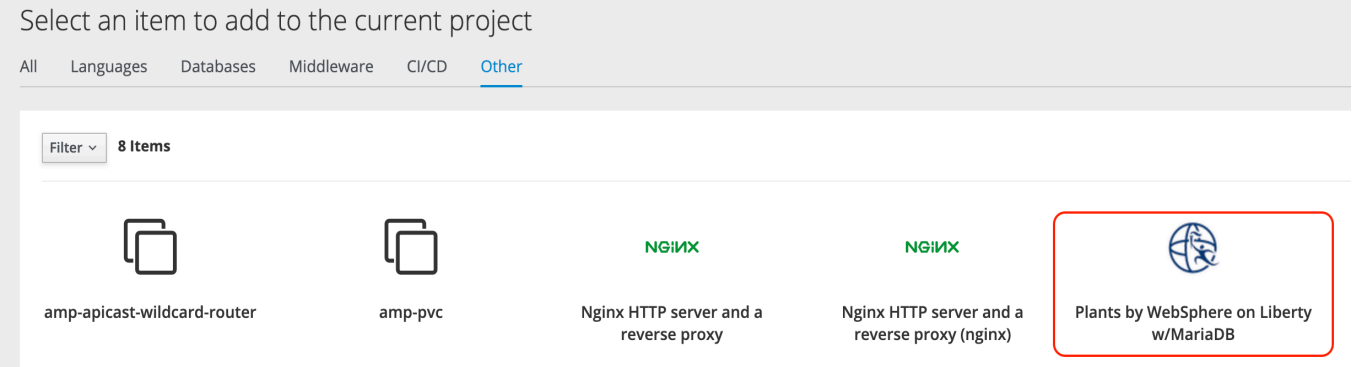
Parameters	`parameters:`
	`- name: APPLICATION_NAME`
	`  displayName: Application name`
	`  description: The name for the application.`
	`  value: pbw-liberty-mariadb`
	`  required: true`
	`...`

Parameterized objects	`- apiVersion: image.openshift.io/v1`
	`  kind: ImageStream`
	`  metadata:`
	`  name: "${APPLICATION_NAME}"`
	`...`

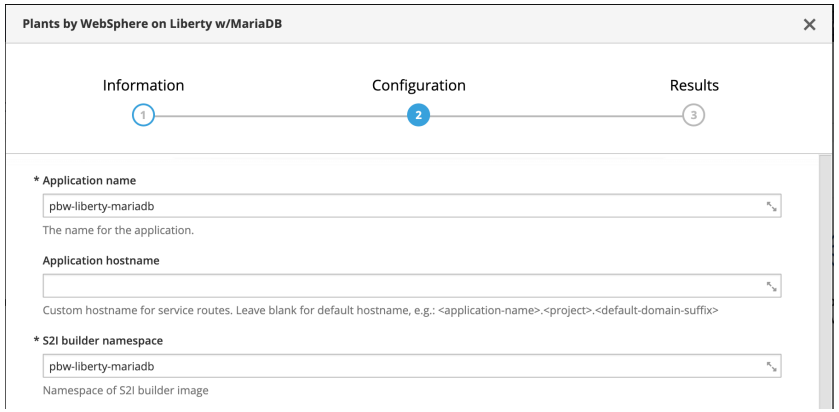
Using a template

From Web console

1 .Select template



2. Apply parameters



From CLI

Apply template and specify parameters by running `oc process` and then piping the result to `oc` **`create`**

```
$ oc process -f pbw-liberty \
 -p APPLICATION_HOSTNAME=foobar.com \
 | oc create -f -
```

Resources

S2I
https://docs.openshift.com/container-platform/3.11/creating_images/s2i.html

OpenShift Templates
https://docs.openshift.com/container-platform/3.11/dev_guide/templates.html