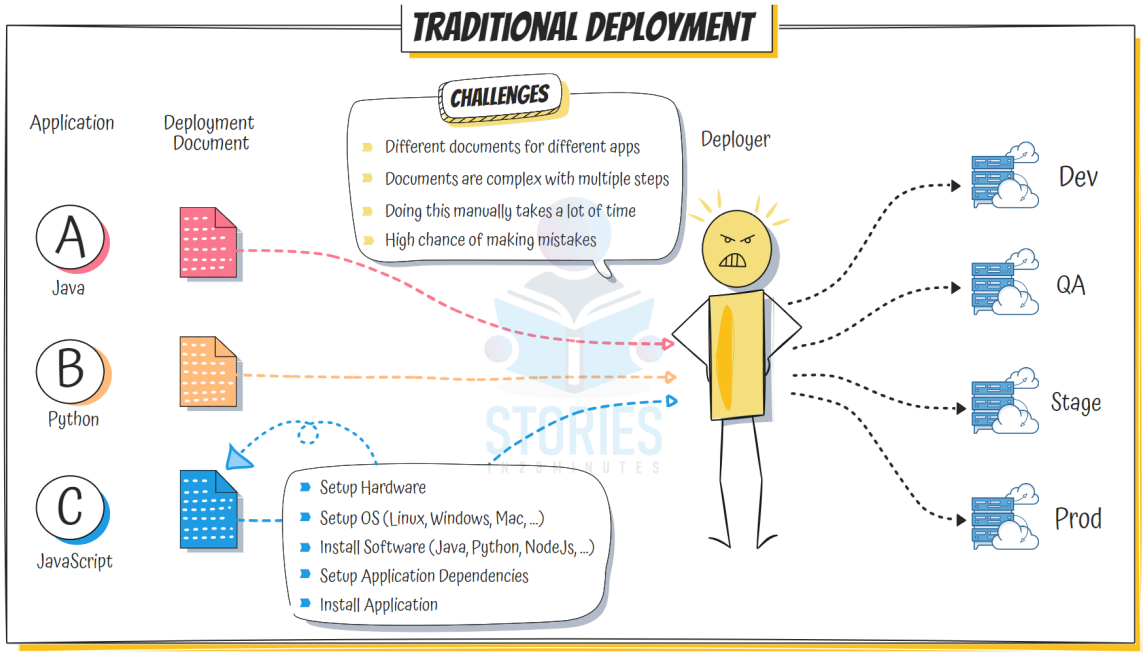
Docker Fundamentals

On this page

What was the Traditional Deployment Approach Before Containers?



- **Documentation-Based Deployment**: A document details the steps for creating environments and deploying software
  - **Hardware Setup**: Preparing physical servers or virtual machines
  - **OS Installation**: Installing operating systems like Linux, Windows, or Mac
  - **Software Installation**: Set up essential software such as Java, Python, or Node.js

- **Dependency Configuration**: Install and configure application dependencies and libraries
- **Application Deployment**: Install and configure the actual application on the system

What are the challenges with Traditional Deployment Process?

- **Multiple setup steps**: Setting up OS, software, and dependencies repeatedly can be tedious
- **Time-consuming process**: Manually performing these steps across multiple environments (Dev, QA, Stage, Prod) takes a lot of effort
- **Lack of automation**: Traditional methods lack automation
- **High chance of mistakes**: Numerous manual steps and separate documents increase the likelihood of errors
- **Different documents**: Each application - depending on its type (Java, Python, JavaScript) - needs different kind of setup instructions

Can you explain the sequence of steps in creating and deploying a Container Image?

- **Build Container Image**: Create a Container image of your application with necessary components
- **Push Image to Container Registry**: Upload your created image to the registry for later use.
- **Deploy it to Your Environment**: Pull the image from the registry and run your application
    - Same approach in Local, DEV, QA, Stage, Production,..

What are the Advantages of using Docker?

- **Standardized Deployment Package**: Same packaging approach irrespective of programming language
- **Standardized Deployment Process**: Run any Container image the same way, regardless of its contents
- **Standardized Operations**: Collect logs, metrics, and traces the same way, regardless of the container
- **Lightweight Alternative to VMs**: Uses fewer resources, making it faster and more efficient

Compare Virtual Machines (VM) and Containers

Feature	VMs	Containers
Architecture	Complete virtualized computers with their own OS, running on a hypervisor	Lightweight, sharing the host OS core, running using Container Engine
Performance	Resource-intensive - Each VM contains full OS	Lightweight, shares host OS
Boot Time	Slower, full OS startup	Faster, shares host OS
Resource Usage	Consumes more CPU, memory, storage	More efficient resource usage
Image Size	Larger, includes a full OS	Smaller, only app and dependencies

Feature	VMs	Containers
Security	Stronger, separate OS for each VM	Shared OS kernel but container engine ensures isolated processes for each container

Explain Important Components in Container Architecture

- **Container Image**: A package representing a specific version of your application, containing everything needed to run it
  - **Complete Package**: Includes the Operating System, Application Runtime, and Application Code & Dependencies
  - **Consistent Deployment**: Ensures your app runs the same way everywhere, from your local machine to cloud servers
- **Container Engine**: The core software that runs and manages containers
  - **Run your Container Images**: Creates containers
- **Container Registry**: Storage location for Container images

    - **Examples**: Docker Hub, Amazon Elastic Container Registry, Google Artifact Registry, Azure Container Registry
    - **Collaboration**: Facilitates easy distribution of container images across teams
    - **Integration**: Works with CI/CD systems for automated building and deployment
    - **Discovery**: Acts as a discovery mechanism for official and community images
- **Container Repository**: A collection of Container images for a specific app

---

Why is Containerization important for DevOps and Microservices?

What is Containerization?

- **Definition**: Package app + dependencies into one container
- **Goal**: Run app the same everywhere – dev, test, prod
- **Tools**: Popular tool is Docker. Alternatives include containerd, runc, ..

Why is it Important in DevOps?

- **Consistent Environments**: Same setup from dev to prod
- **"Works on My Machine" Solved**: Containers avoid environment mismatch

Containerization Simplifies Deployment

- **Faster Setup**: No need to install libraries manually
- **Less Errors**: Fewer issues due to missing dependencies
- **Quick Start**: Run app with one command

Containerization Promotes Automation

- **Automated Image Creation**: Automatically build container images
- **Works with CI/CD**: Integrate with tools like Jenkins or GitHub Actions
- **Supports Orchestration**: Use Kubernetes to manage many containers

Containerization Simplifies Microservices Architectures

- **Create microservice specific container images**: Create different container images for each of the microservices
- **Same Process For Different Technology**: Same process irrespective of technology used in a microservice - Java, Python, Node.js or ..
- **Scale Easily**: Add or remove containers as needed

- **Containers Improve Observability**: Get Logs, Metrics and Traces for different types of applications the same way

Containers are Platform Neutral

- **Same Image Everywhere**: Build once, run anywhere
- **Run Anywhere**: Works on local, server, or cloud
- **Cloud Ready**: Compatible with AWS, Azure, GCP

Containers are Resource Efficient

- **Lightweight**: Host OS is shared
- **Faster Start-up**: Containers launch quickly
- **Higher Density**: Run more containers (than VMs) on same hardware

Previous
Cloud Interview - Diagrams

Next
Building Docker Images

Keep Learning

Our Products

Home
Springboot
Cloud
Roadmaps
Flashcards
Bookshelf