Packaging and Deployment

- Java Packaging for Bare Metal
  - JAR Files. Fat (Uber) JARs
  - JPMS. `jlink`/`jpackage`
  - GraalVM `native-image`
- Linux Packaging for Bare Metal (DEB, RPM)
- Virtualization and Containers 101. Containers: Concept, Implementation, Solutions. Container Orchestration
- Containers and Java: Build. Manual `Dockerfile`, `docker-maven-plugin`, Google Jib
- Containers and Java: Runtime

JAR (**J**ava **AR**chive)

https://docs.oracle.com/en/java/javase/17/docs/specs/jar/jar.html

- **ZIP archive** with **compiled Java classes**, **resources** and **metainformation** (`META-INF/`)
- **Compiled classes** and **class resources** are put into directories corresponding to Java packages.
  - Top-Level class `ru.hse.java.HelloWorld => ru/hse/java/HelloWorld.class`
  - Anonymous, inner, static inner: `ru.hse.HelloWorld.Inner => ru/hse/HelloWorld$Inner.class`
- Most important **metainformation** is the Manifest, `META-INF/MANIFEST.MF`:

```
Manifest-Version: 1.0
Main-Class: ru.hse.MyCoolProjectMain
       <more key-value pairs...>
```

```
java -jar my-cool-project-1.0.jar
```

- `META-INF/` directory also MAY contain:
  - Digital signature files (`*.RSA, *.DSA, SIG-*`)
  - **Service Providers** (`META-INF/services/<fully-qualified name of Service Class Impl>`) *@see* future seminar on DI

Fat (Uber) JARs

https://stackoverflow.com/a/36539885/3438672

- Bring all of your dependencies (transitively) into a single large JAR file
- Three methods:
  - **Unpack all dependency JARs (`maven-assembly-plugin` → `jar-with-dependencies`)**
  - Unpack all dependency JARs, but also rename their packages and merge resources (`maven-shade-plugin`)
    - To avoid conflicts with your library users' dependencies
  - Copy JARs into your JAR and use a special *class loader* to transparently access their classes (`onejar`, `spring-boot-plugin:repackage`)
- Classpath is Linear. Multiple Versions of Artifact on Class Path → JAR Hell
  - `maven-enforcer-plugin` helps by forcing **dependency convergence**. Use it! https://maven.apache.org/enforcer/enforcer-rules/dependencyConvergence.html
  - Explicitly divide your code into modules: JPMS (Java 9+ standard), OSGi (more flexible)

JPMS. `jlink/jpackage`

https://nipafx.dev/java-module-system-tutorial/

- **Java Platform Module System**
  - Dependencies (**`requires`** `[`**`transitive`**`] ...`). Cycles and Split Packages are disallowed
  - Strong Encapsulation. Visibility: **`exports`** `x [`**`to`**` z]`, Reflection Access (**`opens`** `y [`**`to`**` x]`)
  - Inter-module dependencies are assumed to be *mostly* static (determined at app start time)
- Requires you to **cleanly separate your system into modules**, which is **HARD**
- `jlink` image will include only modules used by your app, including JDK modules:

```
jlink --module-path $JAVA_HOME/jmods:mlib # JDK \
      --add-modules com.greetings         # Your Main Module \
      --output greetingsapp
```

https://dev.java/learn/creating-runtime-and-application-images-with-jlink/

- `jpackage` can create native packages for Windows (`.msi`), Mac OS (`.dmg`) and Linux
- **Non-modular apps are supported, too!**

https://dev.java/learn/jpackage/#examples
https://medium.com/azulsystems/9568c5e70ef4

GraalVM `native-image`

- Unlike `jlink` which only links **platform-independent** modules together, `native-image` Ahead-of-Time compiles your code into a **platform-dependent native application**
  - Some limitations, mostly around Reflection usage and static class initialization (**`static final`**s and **`static`** `{}` blocks spawning threads and the like)
  - Popular frameworks are supported out-of-box, less popular can encounter problems
- Emerging frameworks using native-image as the default: Quarkus, Micronaut
- Good fit for Microservices, Serverless, CLIs, Agents, …
  - **Fast start** (low start latency) far more important than **peak throughput**
  - GC and runtime is naive compared to OpenJDK’s Hotspot

Linux Packaging (DEB, RPM)

- General tools for managing components of a Linux distribution
  - Manage *packages*, which provide *files* installed into your filesystem, optionally with *hook* scripts executed on package installation and removal
  - Packages are *versioned* and *depend* on each other (specific version or version range)
  - High-Level Package Managers (*e.g.* `apt`, `yum`, `zypper`): Dependency Resolution, Repository Management
  - Low-Level Package Managers (*e.g.* `dpkg`, `rpm`): Local package DB, Package installation
  - Package authoring tools (*e.g.* `rpmbuild`): Transform specs, source files and scripts into a single coherent package
- Cannot have multiple versions of the same package installed at the same time!
  - Multiple packages, however, *can* provide the same binary
    E.g. `/bin/java` via `update-java-alternatives` (basically, symlinks)

Virtualization

Run 1+ *Guests* (OS w/virtualized hardware) on a single *Host*, managed by a *Hypervisor*

Reasonably fast: hardware (*e.g.,* Intel VT-x) and software-assisted (*Para*virtualization)

Advantages (over bare metal)	Drawbacks

• **Tight Isolation** (CPU, RAM, Network, Storage, …)
• **Better Resource Utilization:** A single host server can manage multiple guest Vms
• **Better Scalability:** both Vertical (allocate more host resources to guest) and Horizontal (spawn more guests)
• **Better Resilience and Disaster Recovery:**
  • VM crashed? Just spawn a new one instead
  • Replicated network block storage
	• **You bring *Everything but the Kitchen Sink*:** Full OS image + all packages + your app (**orders of magnitude** smaller)
	• **Expensive VM setup and teardown** Tens of seconds to minutes, depending on workload and resources available)
	• ***Some* performance degradation** (top → Steal time) Especially in burstable/preemptible VMs

Containers

- Hypervisors virtualize hardware and then run a full-fledged real OS on it
- But modern OSs **already have powerful primitives** for isolation (process, network, storage) and resource constraints!

Container includes all relevant (software and data, but **NOT** OS kernel/image) dependencies for your application/service

…unlike VMs, you cannot *e.g.* run it on a fixed specific kernel version. But this is rarely required for most common apps and services anyway

Containers: Packaging & Delivery

- Linux Containers were mostly intended for **standard packaging and delivery** of software
  - Compare *e.g.* FreeBSD Jails, which were designed **as a security feature** first and foremost
- Docker Images → OCI (container image standard)
  - Optimized for **efficient download and caching**:
    - Layered filesystem
    - Hash-based layer identity (SHA256 digest)
  - Images are **tagged**, typically with version. `latest` tag (convention)
- Container Registries are somewhat similar to Maven artifact repositories, but **content-addressable** via layer hashes
  - Public, *e.g.* DockerHub
  - Private. Cloud offerings from all major cloud providers, including Yandex Cloud

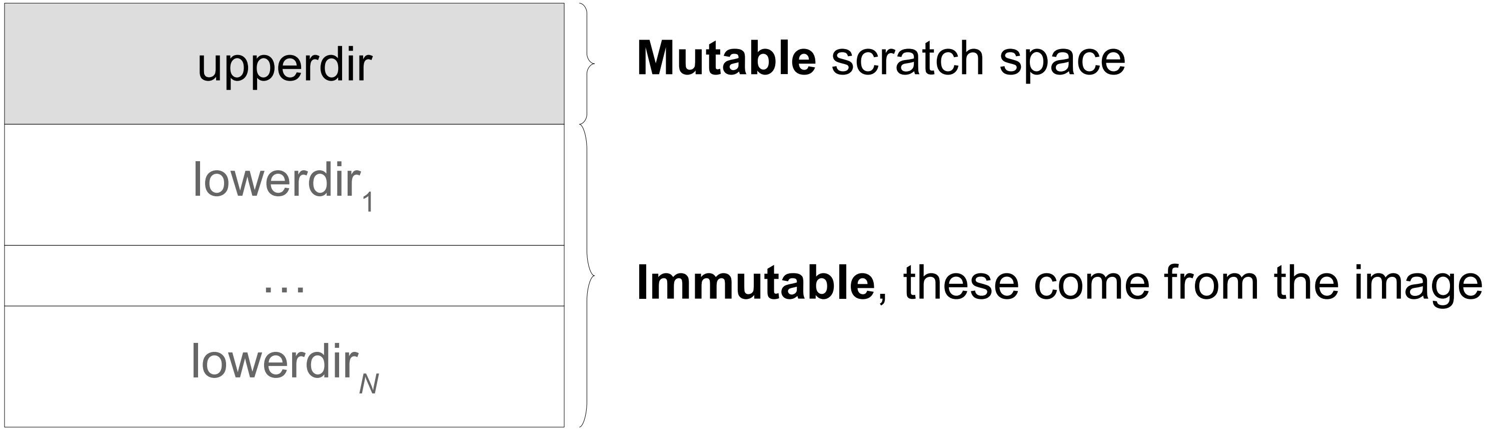
Containers: Isolation → Namespaces

Unit of isolation in Containers is the **Process**

- Namespaces limit **visibility** of sensitive system entities: processes, network interfaces, users, mount points, …
- **Process isolation:** main process in container runs as PID 1, but its PID in the host is different
- **Network isolation:** container has limited access to host network interfaces, can have special *bridge* network interfaces etc.
- **User isolation:** process inside container runs as root (or as some user that you specify)
- **Filesystem isolation:** containers cannot access host storage unless explicitly specified

Containers: Isolation → FS

- `pivot_root` (`chroot` on steroids) to have `/` independent of host root filesystem
- Union Filesystem, *overlayfs*



Containers: Limits and Security

- cgroups (Control Groups): Hierarchical Resource Accounting and Limits
  - Limit CPU core usage
  - Memory usage (RSS, resident set size)
  - I/O (read and write iops)
  - Network bandwidth
  - Process is typically killed or throttled when resource overuse is detected
- Container Security: Principle of least privilege
  - Drop capabilities (`CAP_...`)
  - seccomp-bpf: Selective filtering of syscalls

Container Orchestration

- Containers are much more lightweight that VMs
  - You can have tens and even hundreds of them running on a modest VM!
- Container Orchestration (*E.g.* Kubernetes, Docker Swarm, Nomad, Amazon ECS, …): Managing lots of containers, and Clusters of similar containers
  - Resource Allocator
  - Workload Scheduler
    - Jobs: batch, throughput-oriented workloads
    - Services (stateless and stateful): interactive, latency-oriented workloads
  - Resilience: restarts/retries, container migration
  - Autoscaling
  - Persistent storage management
- Container Management Philosophy: Cattle **NOT** Pets!

Popular Container Solutions

- **Basic Container Management:** ***Docker*** (safer, simpler production alternative is *rkt*)
  - build image, push (publish to registry), pull (download from registry)
  - run image/run command inside image
  - view stdout/stderr of container process/route process logs to syslog and whatever
- **Basic Orchestration:** ***Docker Compose***
- **Advanced Orchestration:** *Kubernetes / OpenShift*, *Amazon ECS*, *Mesosphere DC/OS, Hashicorp Nomad*
- **Container-Optimized** Linux distributions (*Alpine*, *distroless*): Minimal dependencies required to run containers → → Smaller attack surface and better performance
- **Container-First** Operating Systems: CoreOS, AWS Bottlerocket

Containers and Java: Build

- By hand (ew!)
- Fat JAR (maven-assembly-plugin) + io.fabric8:docker-maven-plugin
- Google Jib (Java Image Builder): Layered Images, no fat JARs. @*see* https://phauer.com/2019/no-fat-jar-in-docker-image/

Containers and Java: Manual Build

src/main/docker/Dockerfile

```
FROM bellsoft/liberica-openjdk-alpine-musl:17.0.3-7
RUN apk add strace
COPY echo-server-1.0-SNAPSHOT-jar-with-dependencies.jar /
CMD ["java", "-jar", "/echo-server-1.0-SNAPSHOT-jar-with-dependencies.jar"]
EXPOSE 48484
```

Fat JAR!

Terminal

```
$ mvn clean package && docker build target/ \ # Build directory
  -t cr.yandex/crp024d3b67qde0rnl3r/echo-server:latest \  # Tag
  -f src/main/docker/Dockerfile                  # Path to Dockerfile
```

Containers and Java: Manual Run

Terminal 1

Ports | Docker Image

```
$ docker run --rm -it -p 48484:48484 cr.yandex/crp024d3b67qde0rnl3r/echo-server:latest
```

Run Interactive, with Terminal attached

Cleanup on Termination

Terminal 2

```
$ docker ps -a
CONTAINER ID   IMAGE                                               COMMAND                  CREATED
   STATUS         PORTS                                           NAMES
f53c9e43ab97   cr.yandex/crp024d3b67qde0rnl3r/echo-server:latest   "java -jar /ech…"   3 seconds ago
   Up 1 second    0.0.0.0:48484->48484/tcp, :::48484->48484/tcp   recursing_hawking

$ telnet localhost 48484
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hi
Hi
END
Connection closed by foreign host.
```

docker-maven-plugin

https://github.com/fabric8io/docker-maven-plugin

- Modeled after `maven-assembly-plugin`, but dependencies are packaged into a Docker container instead of a Fat JAR

- `Dockerfile` map to <tags> in `pom.xml`:

```
<from>bellsoft/liberica-openjdk-alpine-musl:17.0.3-7</from>
<assembly><descriptorRef>artifact-with-dependencies</descriptorRef></assembly>
<cmd><exec>
    <arg>java</arg><arg>-jar</arg><arg>maven/${project.build.finalName}.jar</arg>
</exec></cmd>
```

- Needs `docker` binaries for the build (Mac users beware!)

- After build, you have to manually tag built image and push it:

```
$ docker tag echo-server:1.0-SNAPSHOT cr.yandex/crp024d3b67qde0rnl3r/echo-server:latest
$ docker push cr.yandex/crp024d3b67qde0rnl3r/echo-server:latest
```

Google Jib (Java Image Builder)

https://github.com/GoogleContainerTools/jib

- Builds an **Efficient Layered Image**: your code in one layer, dependencies in the other
- Does not need `docker` binaries to run
- Pushes the built image to your Docker repository
- Convention over Configuration:

```
<from>
    <image>bellsoft/liberica-openjdk-alpine-musl:17.0.3-7</image>
</from>
```

Optional!

```
<to>
    <image>cr.yandex/<...>/${project.artifactId}:latest</image>
</to>
```

Tutorial: https://phauer.com/2019/no-fat-jar-in-docker-image/

Containers and Java: Runtime

- Enable namespace and cgroup-related `java` command line switches, e.g.:

  `-XX:+UseContainerSupport` `-XX:InitialRAMPercentage=80 -XX:MaxRAMPercentage=80`
- Getting memory and CPU limits is **REALLY HARD** and needs a lot of fine tuning, even for experts. @*see* https://www.youtube.com/watch?v=kKigibHrV5I
- To properly catch `SIGTERM`, use **`CMD`** not **`RUN`** in `Dockerfile`:

  `ENTRYPOINT` `["java"]` **`CMD`** `["-jar", "/your-app.jar"]`
- A shell script which runs `java`? It **MUST** properly handle `SIGTERM`!
  - Running java with `exec` is the simplest solution
  - In more complex cases, you might need to register a `trap` handler in the script
- You run as PID 1 and need to take care of Zombie processes. `docker run --init`
- App stderr and stdout: `docker logs`
  @*see* https://docs.docker.com/config/containers/logging/