LU10b - Multi-Node Lab mit Vagrant (verteilte Systeme)

Ziel: Du kannst mit **Vagrant** eine **reproduzierbare Multi-VM-Umgebung** (mehrere Nodes) definieren, starten, vernetzen und verwalten – als Praxis-Setup für **verteilte Systeme**.

Übersicht

Begriff	Kurz erklärt
Vagrant	Tool, um **VM-Umgebungen als Code** zu definieren (Vagrantfile) und reproduzierbar zu starten.
VM (Virtual Machine)	Komplettes virtuelles Betriebssystem (Kernel + Userspace) auf einem Host.
Provider	VM-Backend, z.B. **VirtualBox**, VMware, Hyper-V (je nach System).
Box	Vorgefertigtes VM-Image (Basis-OS), z.B. Ubuntu.
Provisioning	Automatische Konfiguration nach dem Boot: Pakete installieren, Services konfigurieren, Dateien kopieren…
Multi-Node	Mehrere VMs als „Cluster“ (z.B. db/api/proxy) inkl. Netzwerk.

Warum Vagrant im Modul „Verteilte Systeme“?

In verteilten Systemen geht es selten nur um „einen Dienst auf einem Rechner“. Typische Themen sind:

- **Mehrere Nodes** (Dienste sind verteilt)
- **Netzwerk & Ports** (Kommunikation zwischen Nodes)
- **Fehlersuche über Maschinen-Grenzen** (Logs, Connectivity, Firewall)
- **Reproduzierbarkeit** (alle Lernenden haben *dieselbe* Ausgangslage)

Vagrant liefert dir genau das: **ein Vagrantfile** → **ein Befehl** → identische Multi-VM-Umgebung.

Merke: Container sind „leicht“ und schnell – aber manchmal brauchst du das **volle OS**: systemd, Kernel-nahe Tools, realistische Netzwerkkonfiguration, Legacy-Stacks. Dafür sind **VMs** top.

Vagrant Architektur in der Praxis

Vagrant ist im Kern ein Orchestrator um diese Teile:

- **Vagrantfile**: „Bauplan“ (Ruby-DSL, aber sehr lesbar)
- **Box**: Basis-System (z.B. Ubuntu)
- **Provider**: Virtualisierung (meist VirtualBox)
- **Communicator**: Kommunikation in die VM (typisch SSH)
- **Synced Folders**: Ordner zwischen Host und VM teilen
- **Networking**: NAT, Port-Forwarding, private/public network
- **Provisioning**: Shell/Ansible/etc. (kommt in LU10b)

Wichtigste Befehle (Workflow)

```
# im Projektordner
vagrant init              # Vagrantfile erstellen (Basis)
vagrant up                # VMs starten/erstellen
vagrant status            # Status ansehen
vagrant ssh <name>        # in eine bestimmte VM einloggen
vagrant halt              # sauber herunterfahren
vagrant reload            # Neustart (Konfigänderungen übernehmen)
vagrant provision         # Provisioning erneut ausführen
vagrant destroy -f        # VMs löschen (Achtung: Daten weg)
```

Typischer Fehler: Änderungen am Vagrantfile werden oft erst nach **reload** oder **destroy/up** vollständig wirksam (je nach Änderung).

Netzwerk-Grundlagen (was du wirklich brauchst)

In Multi-Node Labs sind diese Netzwerkarten zentral:

NAT (Standard)

- VM kommt „raus ins Internet“
- andere VMs erreichen sie nicht zuverlässig über NAT

Port Forwarding

- Host-Port → VM-Port (z.B. localhost:8080 → VM:80)
- gut für „Service im Browser testen“

Beispiel:

```
config.vm.network "forwarded_port", guest: 80, host: 8080
```

Private Network (Host-only / internes Netz)

- Jede VM bekommt eine **fixe interne IP**
- VMs können sich direkt erreichen (perfekt für verteilte Systeme)

Beispiel:

```
config.vm.network "private_network", ip: "192.168.56.11"
```

Didaktik-Tipp: Für verteilte Systeme ist ein **Private Network mit fixen IPs** am besten: reproduzierbar, einfach zu debuggen, klare Topologie.

Beispiel: Multi-Node Setup (db / api / proxy)

Die Idee:

- **db**: Datenbank/Cache
- **api**: Anwendung/Service
- **proxy**: Reverse Proxy / Einstiegspunkt

Vagrantfile (Beispiel)

```
# Vagrantfile
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/jammy64"

  # Gemeinsame Defaults
  config.vm.provider "virtualbox" do |vb|
    vb.cpus = 2
    vb.memory = 2048
  end

  # --- db node ---
  config.vm.define "db" do |db|
    db.vm.hostname = "db"
    db.vm.network "private_network", ip: "192.168.56.11"
    # Optional: DB nach aussen testbar machen (nur falls nötig)
    # db.vm.network "forwarded_port", guest: 5432, host: 15432
  end

  # --- api node ---
  config.vm.define "api" do |api|
    api.vm.hostname = "api"
    api.vm.network "private_network", ip: "192.168.56.12"
    # api.vm.network "forwarded_port", guest: 8000, host: 18000
```

```
  end

  # --- proxy node ---
  config.vm.define "proxy" do |proxy|
    proxy.vm.hostname = "proxy"
    proxy.vm.network "private_network", ip: "192.168.56.13"
    # Reverse proxy nach aussen
    proxy.vm.network "forwarded_port", guest: 80, host: 8080
  end
end
```

Starten und testen

```
vagrant up

# in die proxy VM
vagrant ssh proxy
ip a
ping -c 2 192.168.56.12   # api
ping -c 2 192.168.56.11   # db
```

Synced Folders (Code teilen)

Standardmässig wird der Projektordner oft nach `/vagrant` gemountet.

```
vagrant ssh api
ls -la /vagrant
```

Das ist praktisch, wenn du z.B. „api-code“ auf dem Host bearbeitest, aber in der VM laufen lässt.

Synced Folders können je nach OS/Provider Performance kosten. Für grosse Projekte: bewusst einsetzen.

Debugging-Grundmuster (sehr prüfungsrelevant)

Wenn „verteiltes System“ nicht funktioniert, geh systematisch vor:

Ebene	**Check**
DNS/Hosts	Können die Nodes Namen auflösen? (oder IPs nutzen)
Routing/Netz	Ping / traceroute / `ip a`, `ip r`
Port offen?	`ss -tulpn`, `nc -vz <ip> <port>`

Ebene	Check
Firewall	`ufw status`, iptables/nftables
Service läuft?	`systemctl status <service>`, Logs

Beispiel:

```
# von api -> db Port testen
nc -vz 192.168.56.11 5432

# offene Ports ansehen
ss -tulpn

# systemd Service prüfen
systemctl status nginx --no-pager
journalctl -u nginx -n 50 --no-pager
```

Grenzen & typische Stolpersteine

- **Provider-Probleme** (VirtualBox/Hypervisor nicht aktiv, CPU-VT)
- **Port-Kollisionen** (Host-Port 8080 ist schon belegt)
- **Netzwerk-Konflikte** (Host-only Netz kollidiert mit bestehendem Netz)
- **Performance** (VMs brauchen RAM/CPU – Multi-Node braucht Ressourcen)

Für Klassen: Lieber **kleine VMs** (1 CPU / 1024–2048 MB) und ein „Minimal-Cluster“ als zu viel auf einmal.

M319-LU06, M319-C1E