

Überblick

Verteilte Dateisysteme

Dateisysteme

Apache Hadoop

Hadoop Distributed File System (HDFS)

Container-Betriebssystemvirtualisierung

Motivation

Docker

Einführung

Architektur

Arbeitsablauf

Aufgabe 3

Übersicht

Java API for RESTful Services (JAX-RS)

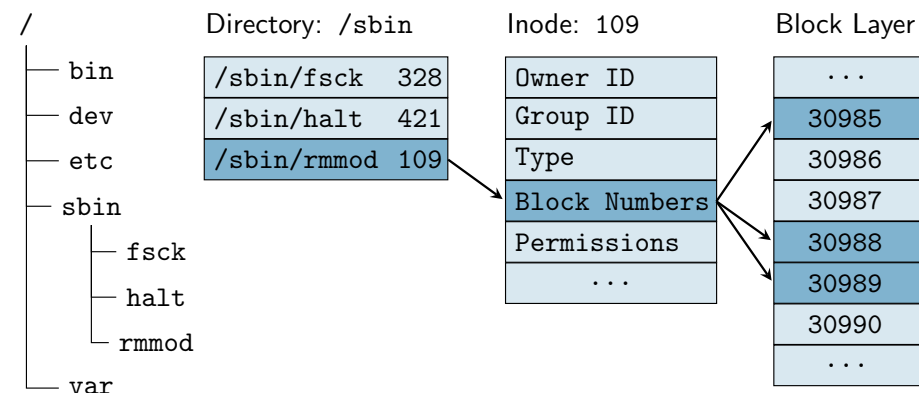
Hinweise



Dateisysteme

Lokale Dateisysteme

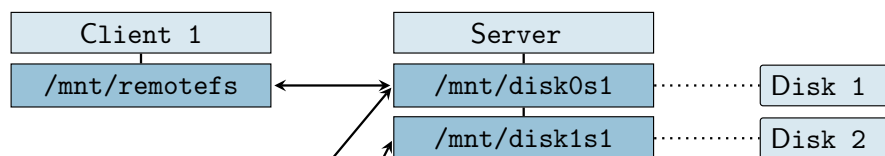
- Logische Schnittstelle des Betriebssystems für Zugriff auf persistente Daten durch Anwendungen und Benutzer
- Adressierung von Daten auf physikalischen Datenträgern
- Beispiele: FAT{,32}, Ext{3,4}, Btrfs



Dateisysteme

Netzwerk-Dateisysteme

- Zugriff auf entfernte, persistente Daten über Rechnergrenzen hinweg
- Für gewöhnlich werden Netzwerk-Dateisysteme in den Namensraum lokaler Dateisysteme eingebunden
- Beispiele: Andrew File System (AFS), Network File System (NFS), Samba



Probleme:

- Fehleranfälligkeit (z. B. Ausfall von Netzwerkverbindungen)
- Flaschenhalsbildung durch Ungleichgewicht (Anzahl Clients/Server)



Dateisysteme

Verteilte Dateisysteme

- Trennung von Belangen (engl. *separation of concerns*)
 - Indizierung
 - Datenverwaltung
- Transparenzen
- Replikation der Daten für höhere Ausfallsicherheit → Einhaltung von Dienstgüterevereinbarung (engl. *Service-Level-Agreement*, kurz: SLA)
- Auflösung von Konflikten zwischen Clients
- Beispiele: Ceph, Google File System, Hadoop Distributed File System

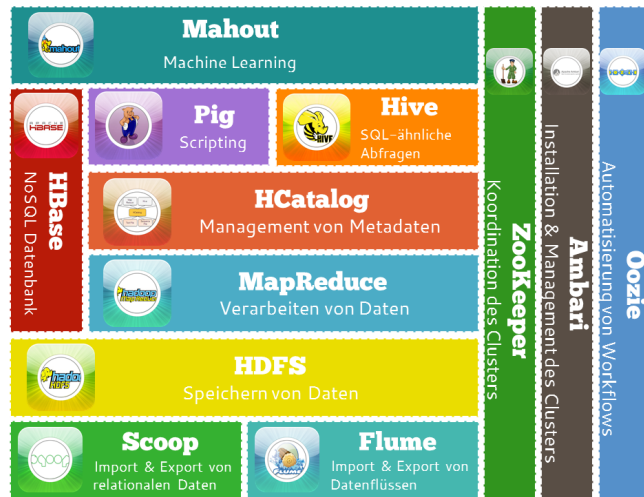
Literatur

- Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler **The Hadoop distributed file system**
Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10), pages 1-10, 2010.



Apache Hadoop: Überblick

- Framework für skalierbare, verteilte Datenverarbeitung
 - Basiskomponenten: Hadoop Distributed File System, Hadoop MapReduce
 - Zusätzliche Komponenten (Auszug): HBase, Pig, ZooKeeper

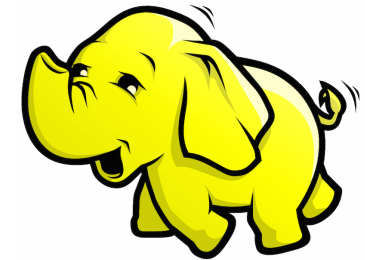


Quelle der Illustration: <https://blog.codecentric.de/2013/08/einfuehrung-in-hadoop-die-wichtigsten-komponenten-von-hadoop-teil-3-von-5/>

Hadoop Distributed File System (HDFS)

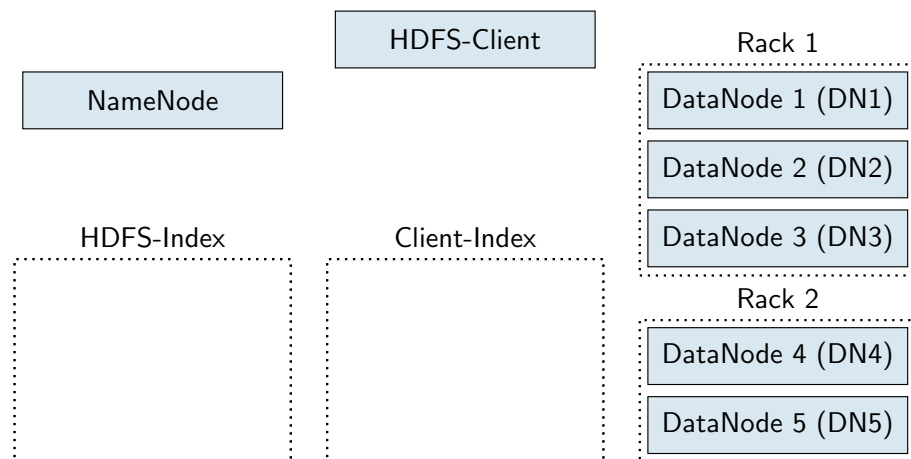
- Architektur
 - HDFS-Client
 - NameNode → Namensraum (Index, Metadaten)
 - DataNode → Blockreplikate (Blockdaten + Metadaten)

- Konzepte
 - Write-once, read-many (WORM)
 - Replikation
 - Datenlokalität („rack-aware“)



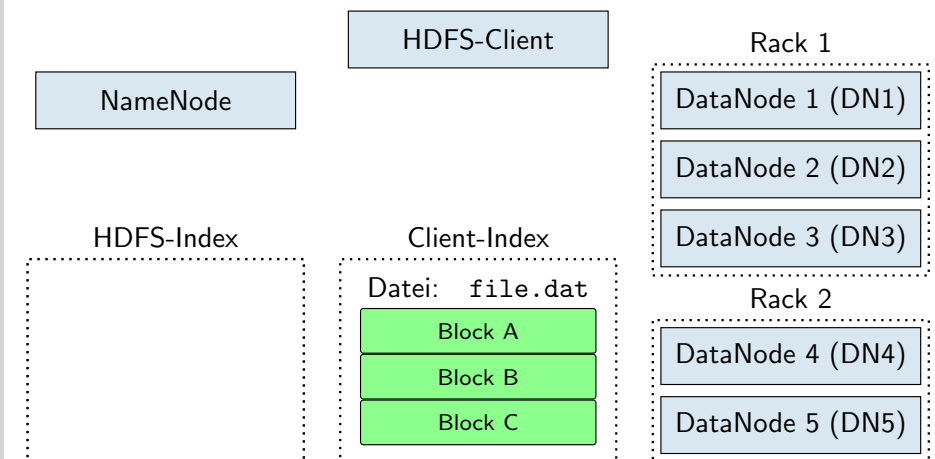
- Literatur
 - Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler
The Hadoop distributed file system
Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10), pages 1–10, 2010.

Hadoop Distributed File System (HDFS)



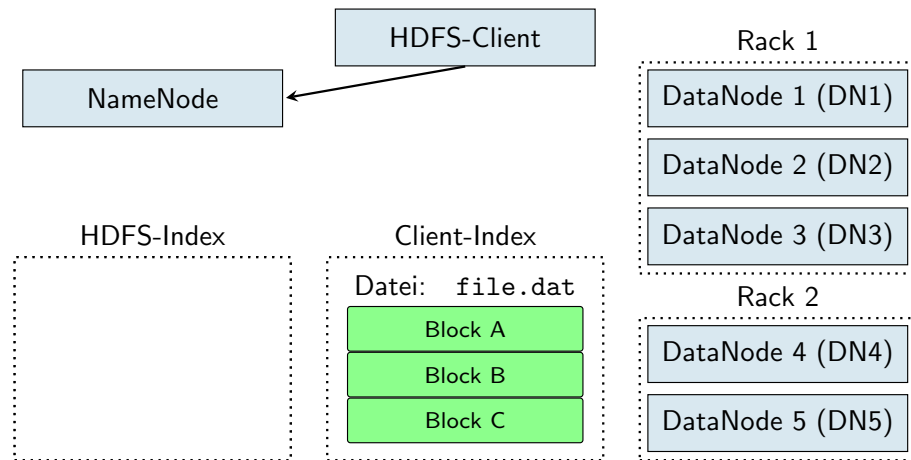
- System-Konfiguration
 - 1x HDFS-Client
 - 1x NameNode
 - 5x DataNodes (Rack 1: DN1–3, Rack 2: DN4–5)

Hadoop Distributed File System (HDFS) — Schreiben



- HDFS-Client legt die aus drei Blöcken (Block A, B und C) bestehende Datei file.dat im HDFS an

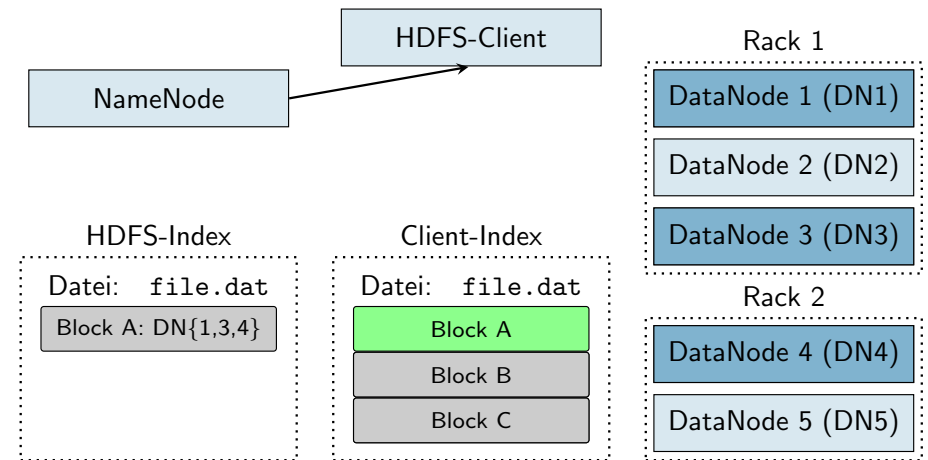
Hadoop Distributed File System (HDFS) — Schreiben



1. HDFS-Client → NameNode:
Anforderung einer sog. Miete (engl. *lease*) für das Schreiben der Datei `file.dat`



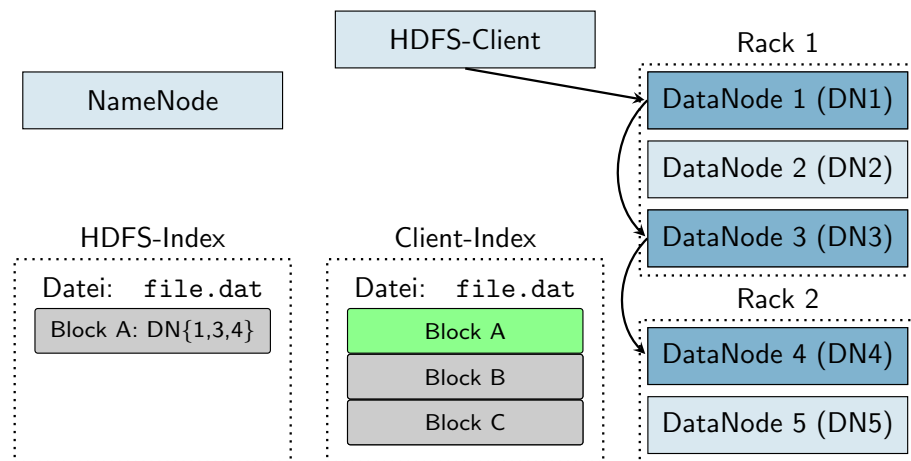
Hadoop Distributed File System (HDFS) — Schreiben



2. NameNode → HDFS-Client:
Erteilung der Miete, Erzeugung einer Block-ID für den ersten Block (Block A), Zuteilung der Replikate (DN1, DN3 und DN4)



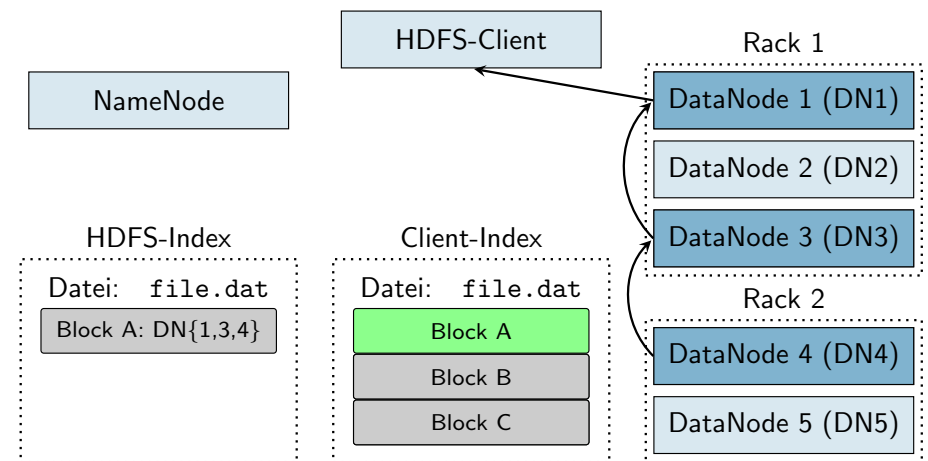
Hadoop Distributed File System (HDFS) — Schreiben



3. „Daten-Pipeline“ zur Vorbereitung der Schreiboperationen von Block A:
HDFS-Client — DN1 — DN3 — DN4



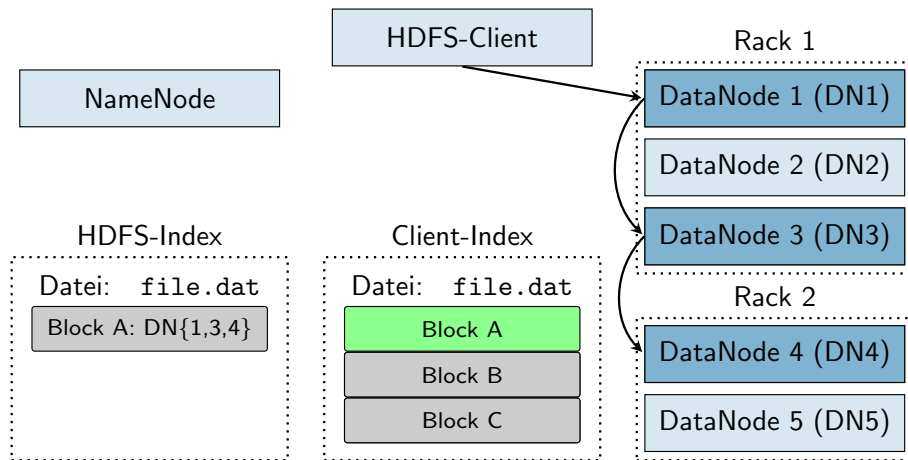
Hadoop Distributed File System (HDFS) — Schreiben



3. „Daten-Pipeline“ zur Vorbereitung der Schreiboperationen von Block A:
HDFS-Client — DN1 — DN3 — DN4



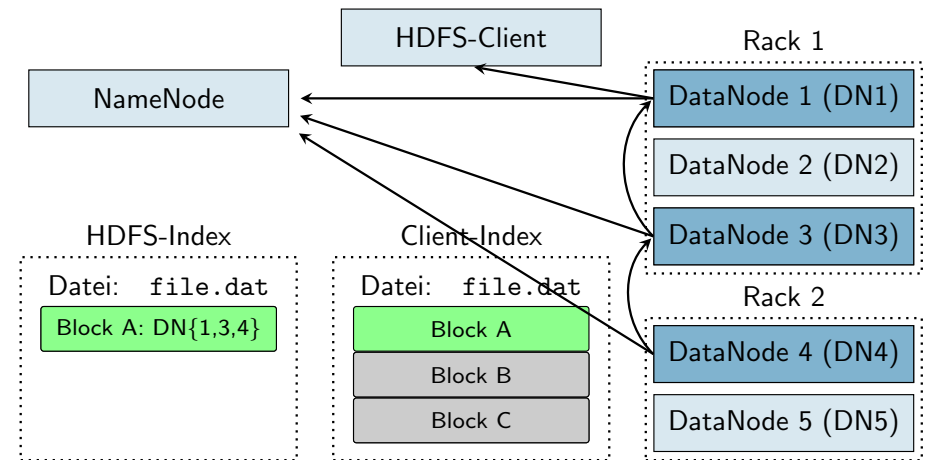
Hadoop Distributed File System (HDFS) — Schreiben



4. Durchführung der Schreiboperationen:
 HDFS-Client sendet Block A an DN1
 DN1 sendet empfangenen Block A an DN3
 DN3 sendet empfangenen Block A an DN4



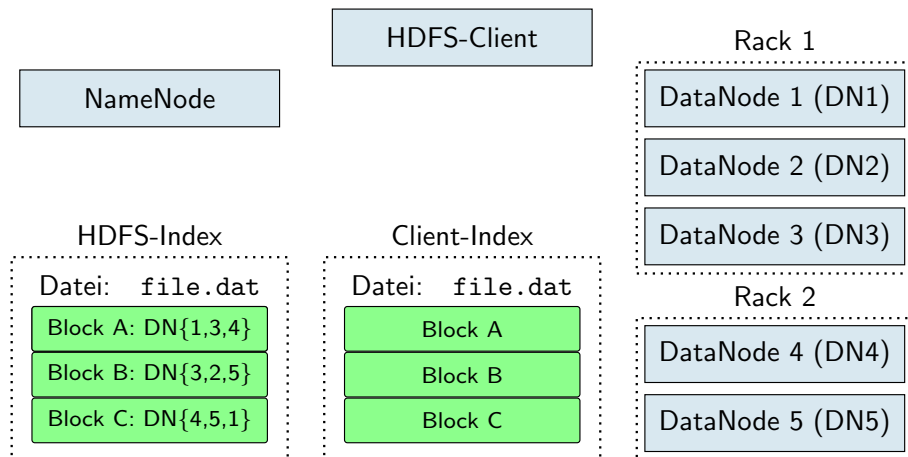
Hadoop Distributed File System (HDFS) — Schreiben



5. Bestätigung der Schreiboperationen:
 Jede DataNode bestätigt das erfolgreiche Schreiben von Block A gegenüber dem NameNode *und* entlang der Pipeline (Abbau)



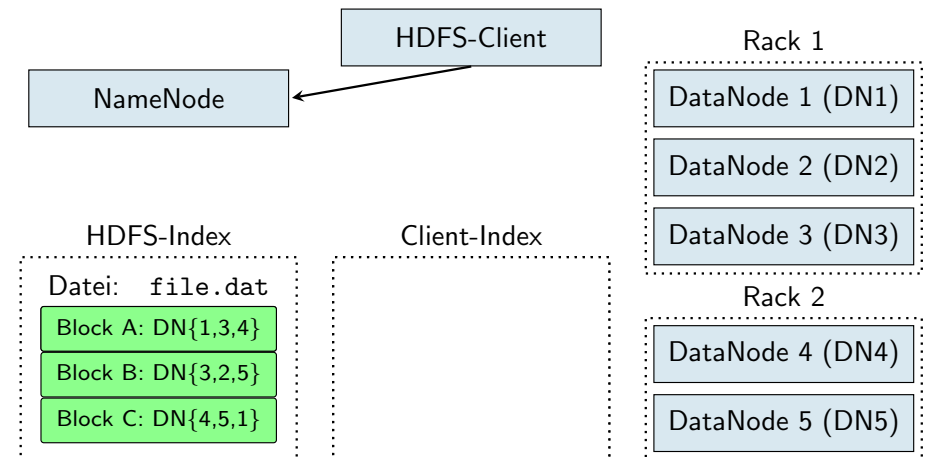
Hadoop Distributed File System (HDFS) — Schreiben



- HDFS-Client → DataNodes:
 Analog werden die restlichen Blöcke der Datei vom HDFS-Client an die durch den NameNode zugeordneten DataNodes verschickt

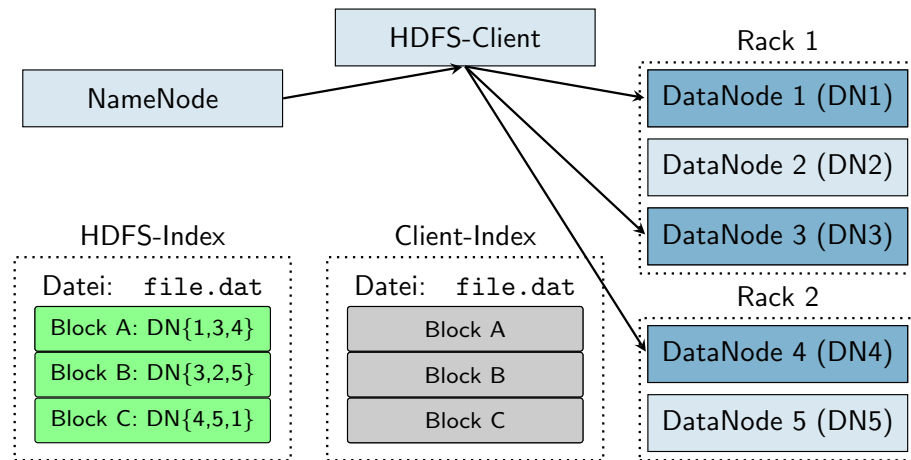


Hadoop Distributed File System (HDFS) — Lesen

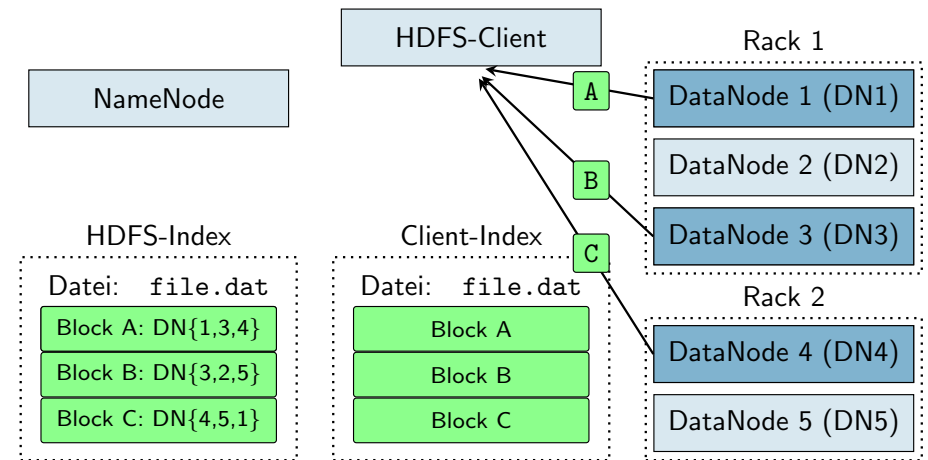


1. HDFS-Client → NameNode:
 Anforderung der DataNodes-Liste: Alle DataNodes, die Blöcke der zu lesenden Datei file.dat speichern





2. NameNode → HDFS-Client, HDFS-Client → DataNodes:
Client erhält DataNodes-Liste und wählt den ersten DataNode für jeden der Datenblöcke

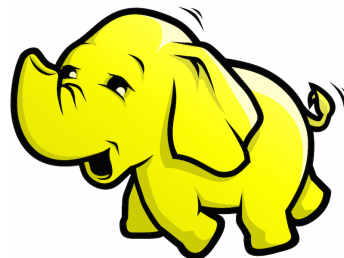


3. DataNodes → HDFS-Client:
HDFS-Client liest die Blöcke sequentiell, DataNodes senden die angeforderten Blöcke an den HDFS-Client



Hadoop Distributed File System (HDFS)

- (Weitere) HDFS-Details
 - Herzschlag-Nachrichten (engl. heartbeat): DataNodes → NameNode
→ Alle drei Sekunden (Default) ein Herzschlag
 - Herzschlag wird durch TCP-Verbindung realisiert
→ Grundlast bei sehr großen Clustern
 - Block-Report (jeder zehnte Herzschlag): NameNode generiert Metadaten aus den Block-Reports
→ Replikation
 - NameNode
→ *Die Sollbruchstelle des Systems?*
- Informationen und Links
 - [Apache Hadoop: HDFS Architecture](#)
 - [Shvachko et al.: The Hadoop distributed file system](#)



Überblick

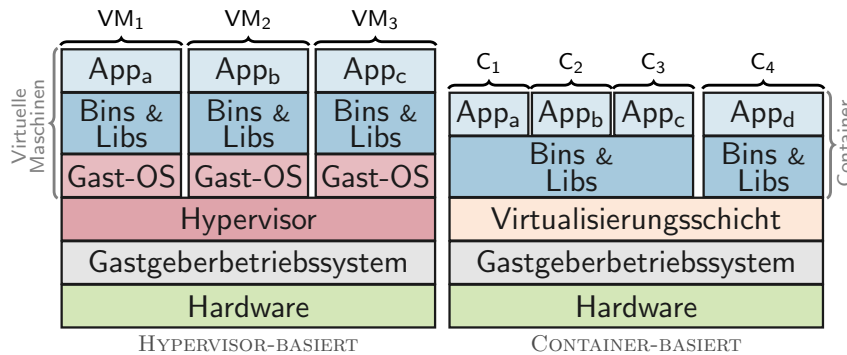
Verteilte Dateisysteme
Dateisysteme
Apache Hadoop
Hadoop Distributed File System (HDFS)

Container-Betriebssystemvirtualisierung
Motivation
Docker
Einführung
Architektur
Arbeitsablauf

Aufgabe 3
Übersicht
Java API for RESTful Services (JAX-RS)
Hinweise



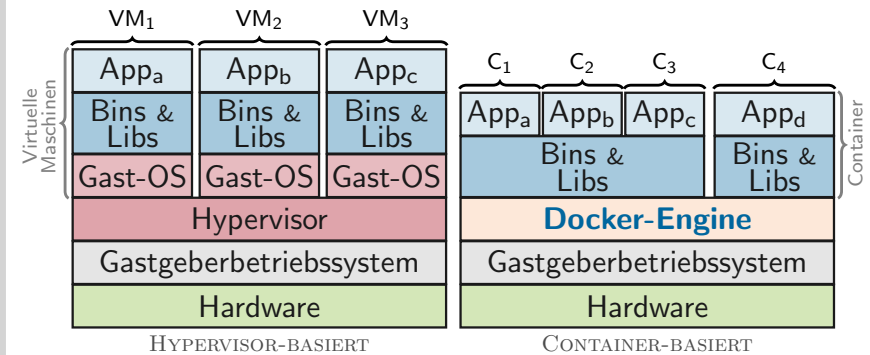
Virtualisierungsformen im Vergleich



- Hypervisor-basierte Virtualisierung (Vollvirtualisierung)
 - Stärken liegen in der Isolation unabhängiger virtueller Maschinen
 - Erlaubt Virtualisierung von kompletten Betriebssystemen
- Container-basierte Virtualisierung
 - Leichtgewichtig: Hypervisor entfällt, kleinere Abbilder
 - Benötigt unter Umständen angepassten Betriebssystem-Kernel



Virtualisierungsformen im Vergleich



- Container-Betriebssystemvirtualisierungsstellvertreter
 - {Free,Open,Net}BSD: FreeBSD Jail, sysjail
 - Windows: iCore Virtual Accounts, Sandboxie
 - Linux: OpenVZ, Linux-VServer
- Im Rahmen dieser Übung betrachtet: **Docker**



Docker



Video: „What is Docker?“

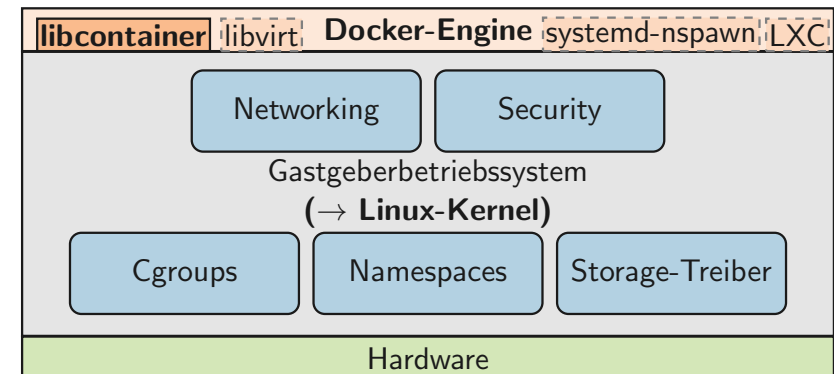
Kurzvortrag von Docker-Erfinder Solomon Hykes

(Kopie: /proj/i4mw/pub/aufgabe3/What_is_Docker.mp4, Dauer: 7:15 Min.)



Docker-Architektur

Überblick



- Docker setzt auf bereits existierenden Linux-Komponenten auf
- Dominierende Komponenten
 - Ressourcenverwaltung: Control Groups
 - Namensräume
 - (Union-)Dateisysteme

} **libcontainer**



- Control Groups (cgroups) ermöglichen das Steuern und Analysieren des Ressourcenverbrauchs bestimmter Benutzer und Prozesse
- Durch Control Groups steuerbare Ressourcen
 - Speicher (RAM, Swap-Speicher)
 - CPU
 - Disk-I/O
- Funktionsweise
 - cgroups-Dateisystem mit Pseudoverzeichnissen und -dateien
 - Prozesse werden mittels Schreiben ihrer PID in passende Kontrolldatei zu einer Control Group hinzugefügt
 - Auflösen einer Control Group entspricht dem Entfernen des korrespondierenden Pseudoverzeichnisses



Paul Menage et al.
CGROUPS

<https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>, 2014.



- Namensräume werden zur Isolation von Anwendungen auf unterschiedlichen Ebenen herangezogen
- **Dateisysteme**
 - Jedes Dateisystem benötigt eigenen Einhängpunkt, welcher einen neuen Namensraum aufspannt
 - Union-Dateisysteme (mit Docker *noch* verwendbar: aufs) erlauben Verschmelzen von Verzeichnissen aus eigenständigen Dateisystemen
- **Prozesse**
 - Hierarchische Struktur mit einem PID-Namensraum pro Ebene
 - Pro PID-Namensraum eigener *init*-ähnlicher Wurzelprozess
 - Isolation: Prozesse können keinen Einfluss auf andere Prozesse in unterschiedlichen Namensräumen nehmen
- **Netzwerke**
 - Eigene Netzwerk-Interfaces zwischen Host und einzelnen Containern
 - Jeweils eigene Routing-Tabellen und iptables-Ketten/Regeln

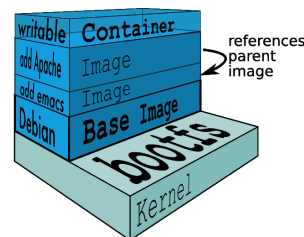


Dockerizing: Anwendung → Container

- Unterscheidung
 - Docker-Abbild: Software-Basis zum Instanzieren von Docker-Containern
 - Docker-Container: Instanziiertes Docker-Abbild in Ausführung

- Inhalt eines Docker-Containers

- Dateisystem
- Systembibliotheken
- Shell(s)
- Binärdateien



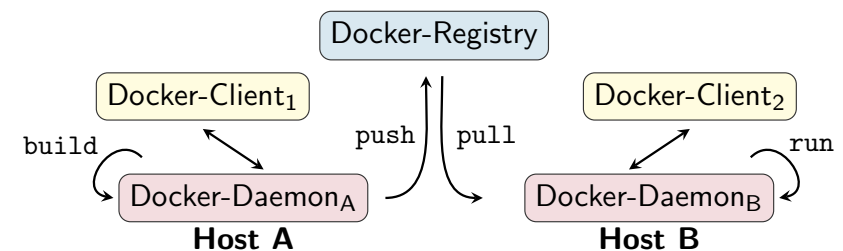
Quelle der Illustration: <https://docs.docker.com/terms/layer/>

- **Dockerizing:** „Verfrachten“ einer Anwendung in einen Container
 - Instanzieren eines Containers erfolgt über das Aufrufen einer darin befindlichen Anwendung
 - Container an interne Anwendungsprozesse gebunden → Sobald letzte Anwendung terminiert ist, beendet sich auch die Container-Instanz



Docker-Arbeitsablauf

- Git-orientierter Arbeitsablauf
 - Ähnliche Befehlsstruktur (z. B. pull, commit, push)
 - Git Hub ↔ Docker Hub
- Typischer Arbeitsablauf
 - 1) Docker-Abbilder bauen (build)
 - 2) Ausliefern: Abbilder in Registry ein- und auschecken (push/pull)
 - 3) Docker-Container instanzieren und zur Ausführung bringen (run)



Docker-Registries und -Repositories

- Von Docker, Inc. bereitgestellte Registry: **Docker Hub**
 - Cloud-Service zur Verwaltung von Docker-Abbildern bzw. -Anwendungen
 - Registrieren bzw. Anlegen eines Benutzerkontos notwendig
 - Anzahl kostenloser, **öffentlicher** Repositories nicht begrenzt
 - Nur ein privates Repository kostenlos
- **Private Registry** (hier: I4-Docker-Registry)
 - Ermöglicht das Verwalten garantiert nicht-öffentlicher Repositories
 - Unabhängigkeit von Verfügbarkeit einer öffentlichen Registry
- Authentifizierung gegenüber der (privaten) Docker-Registry
 - An-/Abmelden an/von (optional spezifiziertem) Docker-Registry-Server

```
$ docker login [<OPTIONS>] [<REGISTRY-HOSTNAME>]
$ [...] // Registry-zugreifende Befehle ausführen, siehe nächste Folie
$ docker logout [<REGISTRY-HOSTNAME>]
```

- **Achtung:** Weglassen eines Registry-Hostnames impliziert Verwendung der **Docker-Hub-Registry** bei nachfolgenden push- oder pull-Befehlen.
→ (I4-Docker-Registry-Hostname: faui42.cs.fau.de:8082)

Umgang mit der Registry

- Abbild aus Repository herunterladen und direkt verwenden/verändern
 - 1) Vorgefertigtes Abbild aus Repository auschecken

```
$ docker pull <NAME>[:<TAG>]
```

Hinweis: TAG nur optional, wenn Image mit Default-Tag (= latest) existiert.
 - 2) Container starten (mehr ab Folie 4–22); darin evtl. Änderungen vornehmen

```
$ docker run -t -i <NAME>[:<TAG>] <COMMAND>
```

Mit /bin/bash als COMMAND können im Container über die Shell beliebige Programme via Paket-Manager installiert werden, z. B. apt-get -yq install vim.
- Optionale Schritte (nur falls Änderungen erfolgt sind, die erhalten bleiben sollen)
 - 3) Änderungen persistent machen und Abbild (lokal!) erzeugen

```
$ docker commit <CONTAINER-ID> <NAME>[:<TAG>]
```
 - 4) Abbild publizieren bzw. in Registry einspielen

```
$ docker push <NAME>[:<TAG>]
```

Hinweis: Da pull und push keinen Registry-Hostnamen vorsehen, müssen die Abbilder bei eigenen Registries über den <NAME>-Parameter getaggt sein.

- <NAME> besteht aus {Abbild,Benutzer}name und Registry-Hostname
- Beispiel: \$ docker push faui42.cs.fau.de:8082/user/myimage:test

Docker-Abbilder bauen

Skriptbasiert (1)

- In der Praxis: **Dockerfiles**
 - Rezepte zum skriptbasierten Bauen eines Abbilds
 - Zeilenweises Abarbeiten der darin befindlichen Instruktionen
- Vordefinierte, voneinander unabhängige **Docker-Instruktionen**
 - FROM <IMAGE>[:<TAG>] → Basisabbild auswählen (obligatorisch)
 - EXPOSE <PORT> [<PORT>...] → Container-übergreifende Port-Freigabe
 - RUN <COMMAND> → Ausführen eines Befehls (in *Shell-Form*)
 - CMD [<EXE>, [<PARAM-1>], ...] → Ausführung bei Container-Start
 - ENTRYPOINT [<EXE>, [<PARAM-1>], ...] → Container-Einstiegspunkt setzen
 - Nur ein Einstiegspunkt (= Befehl) pro Container möglich
 - Container-Aufruf führt zwangsläufig zu Aufruf des entsprechenden Befehls
 - Parameter (bei Container-Start) und Argumente nachfolgender RUN/CMD-Befehle werden als zusätzliche Parameter an <EXE>-Binärdatei übergeben
 - COPY <SRCs> <DST> → Dateien/Verz. ins Container-Dateisystem kopieren
 - ... [→ vollständige Referenz: <https://docs.docker.com/reference/builder/>]

Docker-Abbilder bauen

Skriptbasiert (2)

- Vorgehen
 - Datei Dockerfile anlegen und mit Docker-Instruktionen befüllen
 - Build-Prozess starten mit Kontext unter PATH, URL oder stdin (-)

```
$ docker build -t <NAME>[:<TAG>] <PATH | URL | - >
```
- Beispiel-Dockerfile (Anm.: mwps.jar liegt im aktuellen Arbeitsverzeichnis, d. h. PATH=.)
Aufruf: \$ docker build -t faui42.cs.fau.de:8082/gruppe0/mwcc-image .

```
1 FROM faui42.cs.fau.de:8082/gruppe0/javaimage
2 EXPOSE 18084
3 RUN useradd -m -g users -s /bin/bash mwcc
4 WORKDIR /opt/mwcc
5 RUN mkdir logdir && chown mwcc:users logdir
6 COPY mwps.jar /opt/mwcc/
7 USER mwcc
8 ENTRYPOINT ["java", "-cp", "mwps.jar:lib/*", "mw.printer.MWPrintServer"]
9 CMD ["-logdir", logdir]
```

 - 1) Eigenes Abbild javaimage als Ausgangsbasis heranziehen; 2) Port 18084 freigeben
 - 3) Benutzer mwcc erstellen, diesen zur Gruppe users hinzufügen und Shell setzen
 - 4) Basisverzeichnis setzen (/opt/mwcc und lib-Unterverzeichnis existieren bereits)
 - 5) Log-Verzeichnis erstellen, Benutzerrechte setzen und 6) JAR-Datei hineinkopieren
 - 7) Ausführenden Benutzer und 8) Einstiegspunkt setzen; 9) Java aufrufen

- Besonderheiten von Docker-Abbildern
 - Jeder Befehl im Dockerfile erzeugt ein neues Zwischenabbild
 - Basis- und Zwischenabbilder können gestapelt werden
 - Differenzbildung erlaubt Wiederverwendung zur Platz- und Zeitersparnis

- **Lokal** vorliegende Docker-Abbilder anzeigen (inkl. Image-IDs):

```
$ docker images
REPOSITORY          TAG         IMAGE ID      CREATED       VIRTUAL SIZE
<none>              latest     7fd98daef919 2 days ago   369.8 MB
fau142.cs.fau.de:8082/ubuntu latest     5506de2b643b 11 days ago  197.8 MB
```

- Repository: Zum Gruppieren verwandter Abbilder
- Tag: Zur Unterscheidung und Versionierung verwandter Abbilder
- Image-ID: Zur Adressierung eines Abbilds bei weiteren Befehlen

Hinweis: Beim Erstellen eines Abbilds mit bereits existierendem Tag wird das Abbild nicht gelöscht, sondern mit <none>-Tag versehen aufgehoben (siehe 1. Eintrag in Ausgabe).

- Nur lokale Abbilder können über die Kommandozeile gelöscht werden

```
$ docker rmi [<OPTIONS>] <IMAGE> [<IMAGE>...] # IMAGE := z. B. Image-ID
```



- Docker-Container im Hintergrund mittels `-d(etached)`-Flag starten

```
$ docker run -d [<OPTIONS>] <IMAGE> [<COMMAND> + [ARG...]]
```

→ Für IMAGE kann NAME[:TAG] (vgl. Folie 4–18) oder die Image-ID eingesetzt werden.

- Laufende Container und insbesondere deren **Container-IDs** anzeigen

```
$ docker ps -a
CONTAINER ID  IMAGE          COMMAND          CREATED          ...
ba554f163f63  eg_pgql:latest "bash"          33 seconds ago  ...
345b60f9a4c5  eg_pgql:latest "/usr/lib/postgresql" 7 minutes ago  ...
5496bd5d89d9  debian:latest  "bash"          46 hours ago    ...
... STATUS          PORTS          NAMES
... Up 32 seconds    5432/tcp       sad_lumiere
... Up 7 minutes     0.0.0.0:49155->5432/tcp pg_test
... Exited (0) 46 hours ago          hungry_brattain
```

→ `-a`-Flag, um auch beendete Container und deren Exit-Status anzuzeigen

- Weitere Operationen auf Containern

- Entfernen/Beenden → `docker rm [OPTIONS] <CONTAINER-IDs...>`

- Attachen → `docker attach --sig-proxy=false <CONTAINER-IDs...>`

Hinweis: `--sig-proxy=false` nötig, um mit `Ctrl-c` detachen zu können



- Möglichkeiten der Container-Analyse

- Logs ($\hat{=}$ Ausgaben auf `stderr` und `stdout`) eines Containers anzeigen

```
$ docker logs [<OPTIONS>] <CONTAINER-ID>
```

- Container-Metainformationen (Konfiguration, Zustand, ...) anzeigen

```
$ docker inspect <CONTAINER-ID>
```

- Laufende Prozesse innerhalb eines Containers auflisten

```
$ docker top <CONTAINER-ID>
```

- Jegliche Veränderungen am Container-Dateisystem anzeigen

```
$ docker diff <CONTAINER-ID>
```

- Es existieren eine Reihe von Container-Zuständen bzw. -Events

- Start/Wiederanlauf: `create`, `start`, `restart`, `unpause`
- Stopp/Unterbrechung: `destroy`, `die`, `kill`, `pause`, `stop`

→ **Sämtliche** Events am Docker-Server anzeigen: `$ docker events`



- **Netzwerk-Ports** (Publish-Parameter)

- Jeder Container besitzt eigenes, internes Netzwerk
- EXPOSE-Instruktion im Dockerfile gibt Ports nur zwischen Containern frei
- Für Zugriff von außen, interne Ports explizit auf die des Host abbilden

- Automatisch: zufällig gewählter Port (Bereich: 49153–65535) auf Host-Seite

```
$ docker run -P ...
```

- Manuell, um Host- und Container-Port exakt festzulegen

```
$ docker run -p <HOST-PORT>:<CONTAINER-PORT> ...
```

- **Container-Linking** (Link-Parameter)

- Container direkt mittels sicherem Tunnel miteinander verbinden
- Anwendungsfallabhängiger Vorteil: Zugriff kann nur noch durch andere(n) Container und nicht über das umliegende Netzwerk erfolgen

- Befehl, um einen Ziel- mit einem Quell-Container zu verbinden

```
$ docker run --name <SRC-NAME> --link <DST-NAME>:<LINK-ALIAS> ...
```

- Legt entsprechende Umgebungsvariablen und `/etc/hosts`-Einträge an



Überblick

Verteilte Dateisysteme

Dateisysteme

Apache Hadoop

Hadoop Distributed File System (HDFS)

Container-Betriebssystemvirtualisierung

Motivation

Docker

Einführung

Architektur

Arbeitsablauf

Aufgabe 3

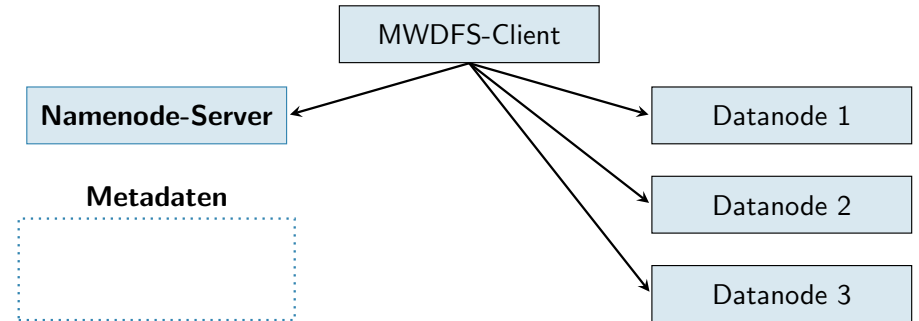
Übersicht

Java API for RESTful Services (JAX-RS)

Hinweise



Übersicht

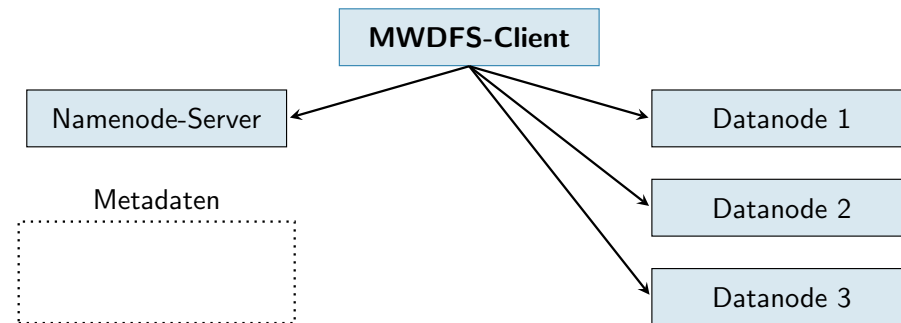


■ Namenode-Server

- Metadaten
- Datei-Operationen (Anlegen, Anzeigen, Löschen)
- Leases für Schreibzugriffe
- Verzeichnisstruktur (optional für 5,0 ECTS)



Übersicht

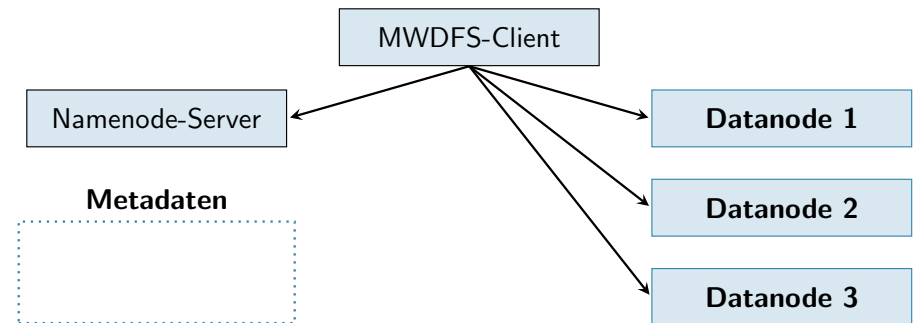


■ MWDFS-Client

- Datenzugriff
- Datei-Operationen (Anlegen, Anzeigen, Löschen)
- Verzeichnisstruktur (optional für 5,0 ECTS)



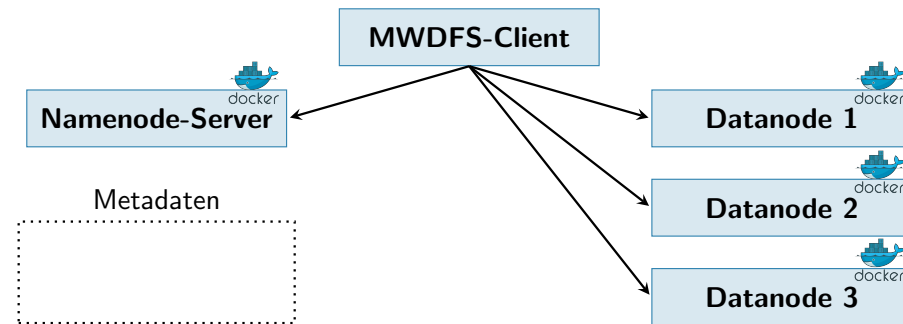
Übersicht



■ Replikation (optional für 5,0 ECTS)

- Datenblöcke redundant auf mehreren Datanodes speichern
- Erweiterung der serverseitigen **Metadaten**





- **Docker und OpenStack**
 - Docker-Images erstellen
 - Betrieb von Namenode-Server und drei Datanodes als **Docker-Container** → OpenStack-Cloud
 - Zugriff auf das System über MWDFS-Client → CIP-Pool

Java-Annotationen (Package: javax.ws.rs)

- Annotationen zum Festlegen von
 - Pfade zu Ressourcen (→ URL)
 - Aktionen (HTTP-Methoden), die auf den Ressourcen ausgeführt werden
 - Repräsentationstypen in Anfrage- und Antwortnachrichten einer Methode
- Mit Ausnahme von OPTIONS und CONNECT existieren für jede HTTP-Operation korrespondierende Annotationen (z. B. @GET)

- Einfaches URI-Matching

```
@Path("/printsrv")
public class MWPrintServer {
    @GET
    @Path("/listall")
    public Response getPrinters() {
        return Response.ok(printers).build(); // printers := MWPrinterList
    }
    // (-> siehe Folie 5-32)
```

- Mögliche (gültige) Anfrage:
http://localhost:18084/printsrv/listall

JAX-RS: Implementierung der Server-Seite

Erweiterte Java-Annotationen

- **URI-Path-Templates**
 - Verwendung von regulären Ausdrücken in @Path möglich
 - Einbetten von Variablen mit „{“ und „}“; Referenzierung über @PathParam
 - URI-Query-Parameter (?<param>=<value>, siehe Beispiel unten): @QueryParam
- Festlegen von Nachrichten-Repräsentationen durch MIME-Typ(en)
 - @Produces: Repräsentation(en) der Antwort zum Client
 - Mögliche MIME-Typen:
 - byte[] → „application/octet-stream“
 - JAXB → „application/xml“
 - ...
- **Beispiel**

```
@POST
@Produces("application/xml")
@Path("/settings{printer:/{.*}?.}")
public String setSettings(@PathParam("printer") String printer,
    @DefaultValue("white") @QueryParam("bgcolor") String color) {
```

→ Mögliche gültige und ungültige Anfragen

- http://localhost:18084/printsrv/settings?bgcolor=yellow ✓
- http://localhost:18084/printsrv/settings/room42/printer1 ✓
- http://localhost:18084/printserver/printer2 ✗ (→ Fehler 404)

JAX-RS: Implementierung der Server-Seite

Antwortnachrichtengenerierung und Fehlerpropagierung

- **Antwortnachrichten**
 - Standard-Java-Rückgabetypen möglich (z. B. siehe vorheriges Beispiel)
 - Rückgabe inkl. Metadaten über Response-Objekt
 - Fehlerfreier Fall: Response.ok(<Object>).build()
 - Leere Antwort: Response.noContent().build()
- **Fehlerpropagierung** über WebApplicationException, um korrekten HTTP-Status-Code zurückzugeben (Default: 500)

```
@PUT
@Path("/addroom/{room}")
@Produces("text/plain")
public Response addRoom(@PathParam("room") String roomName,
    byte[] putData) {
    if (rooms.contains(roomName)) // rooms := static Set<String>
        throw new WebApplicationException("Room already exists.", 409);
    rooms.add(roomName); // HTTP-Status-Code 409 := Conflict
    return Response.ok(roomName + " added.").build();
}
```

JAX-RS: Implementierung der Server-Seite

Server starten

■ Beispiel

```
import java.net.URI;
import javax.ws.rs.core.UriBuilder;
import org.glassfish.jersey.jdkhttp.JdkHttpServerFactory;
import org.glassfish.jersey.server.ResourceConfig;

public static void main(String[] args) {
    URI baseUri = UriBuilder.fromUri("http://[::]/").port(18084).build();
    ResourceConfig config = new ResourceConfig(MWPrintServer.class);
    JdkHttpServerFactory.createHttpServer(baseUri, config);
}
```

■ Server erzeugt neue Instanz der Klasse bei jedem Aufruf

- Default-Konstruktor notwendig
- static für Variablen notwendig

■ Erforderliche JAR-Dateien für den Jersey-HTTP-Server (u. a.) liegen in /proj/i4mw/pub/aufgabe3/jaxrs-ri-2.13/ bereit



JAX-RS-Client: Anfragen und Fehlerbehandlung

■ Senden von Anfragen mit Hilfe von WebTarget-Objekten, z. B.

```
WebTarget ps = ClientBuilder.newClient()
    .target("http://localhost:18084/printsrv");
```

■ WebTarget-Klasse bietet Methoden zur Steuerung von Anfragen

■ path()-Methode

```
// Fuege Raum hinzu (vgl. addRoom() auf Folie 5-29)
ps.path("addroom").path(room).request("text/plain")
    .put(Entity.entity(data, "application/octet-stream")).close();
```

■ queryParams()-Methode

```
// Hintergrundfarbe auf Gelb setzen (vgl. setSettings() auf Folie 5-28)
Response r = ps.path("settings").queryParams("bgcolor", "yellow")
    .request().post(Entity.text(""));
```

→ Benötigt zugehörigen, mit @QueryParam annotierten Parameter auf Server-Seite (vgl. Parameterliste der setSettings()-Methode auf Folie 4-28)

■ Überprüfen auf Fehlerfall

```
if (r.getStatus() != Status.OK.getStatusCode()) // != 200 OK
    System.err.println("Bad status: " + r.getStatusInfo());
r.close();
```



JAX-RS-Client: Antwortobjekte

■ Antwortobjekt deserialisieren

- readEntity()-Aufruf am Response-Objekt

- Beispiel: `String statusMsg = r.readEntity(String.class);`

■ Vollständige API-Dokumentation

<https://jax-rs-spec.java.net/nonav/2.0-rev-a/apidocs/index.html>

■ JAXB- $\{Des,S\}$ erialisierung mit annotierten Java-Klassen

■ Beispiel

- Definition einer annotierten Java-Klasse

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class MWPrinterList {
    public List<String> printers;
    public int activeJobs;
}
```

- Verwendung (hier: Holen von Druckerstatusinformationen)

```
MWPrinterList mwpl = ps.path("listall").request("application/xml").get(MWPrinterList.class);
```

- Zur Kennzeichnung von Variablen, die nicht serialisiert werden sollen:

@XmlTransient-Annotation voranstellen



Effektives HTTP-Debugging

Hinweise

■ cURL-Kommandozeilen-Tool kann zum Debuggen verwendet werden

■ Wichtige Parameter [Referenz: <http://curl.haxx.se/docs/https scripting.html>]

- Ausgabe des vollständigen Nachrichtenaustauschs: `-verbose (-v)`
- Explizites Festlegen der HTTP-Methode: `-X {POST,GET,HEAD,...}`
- Modifizieren des Headers: z. B. `--header "Accept: text/plain"`

```
$ curl -v -X PUT http://localhost:18084/printsrv/addroom/room42
[...]
* Connected to localhost (:::1) port 18084 (#0)
> PUT /printsrv/addroom/room42 HTTP/1.1
> User-Agent: curl/7.26.0
> Host: localhost:18084
> Accept: /*/*
>
[...]
< HTTP/1.1 200 OK
< Content-type: text/plain
< Content-length: 24
< Date: Wed, 05 Nov 2014 12:16:46 GMT
<
room42 added.
```



- **Hilfsskripte** liegen in OpenStack-VM bereit unter `/usr/local/bin`
- **Verfügbare Skripte**

- Löschen aller {gestoppten, ungetaggtten} Docker-Container

```
$ docker-rm-{stopped, untagged}
```

- Alle Container stoppen und Docker-Daemon neustarten

```
$ docker-full-reset
```

- Alle getaggten Abbilder in die I4-Docker-Registry hochladen

```
$ docker-images-push
```

- I4-Docker-Registry durchsuchen

```
$ docker-registry-search <SEARCH_STRING>
```

