HSLU Hochschule Luzern

Verteilte Systeme und Komponenten

Deployment

Roland Gisler

FH Zentralschweiz

Inhalt

- Deployment Grundlagen
- Deployment Diagramme
- Umfang des Deployments
- Releases und Versionierung
- Technisches Deployment (Anhand von Java)
- Beispiel 1: VSK-Logger-(Server-)Projekt
- Beispiel 2: JavaFX-Projekt (z.B. DemoApp oder LoggerViewer)
- Beispiel 3: Deployment als Container

Lernziele

- Sie kennen die verschiedenen Aspekte die es beim Deployment zu beachten gilt.
- Sie können verstehen einfache Deploymentdiagramme und können diese erstellen.
- Sie kennen das dreistellige Versionsschema nach «semantic versioning» und können es anwenden.
- Sie kennen Sinn und Zweck eines Binär-Repository und können dieses nutzen.
- Wie kennen verschiedene Deployment-Arten von Java und können diese umsetzen.

Deployment Grundlagen

Übersicht: Deployment

- **Verteilung**: Verteilung der Software über Downloads / Datenträger oder SMS (Software Management System) oder z.B. OpenWebStart* (Java), inkl. Dokumentation.
- **Installation**: Kopieren der nötigen Dateien an die vorgesehenen Orte und Registrieren der Anwendung, allenfalls überprüfen, ob das Zielsystem für die Anwendung geeignet ist.
  - Hardwareausstattung, Betriebssystemversion etc.
- **Konfiguration**: Einstellungen der/des Programme(s) auf Benutzer, Netzwerkumgebung, Hardware etc.
- **Organisation**: Planung, ggf. Produktion, Information (Marketing), Schulung (Mitarbeiter*innen), Support bereitstellen

*ex. Java Web Start, nun OSS: https://openwebstart.com/

Wann findet Deployment statt?

- Deployment findet natürlich am Ende des Projektes statt.
- Aber: Aufgrund iterativer und agiler Entwicklungsmodelle soll das Deployment kontinuierlich (und früher) stattfinden!
  - CI/CD(elivery) erfordert auch Continuous Deployment!
- Einzelne Build-/Sprint-/Iterationsergebnisse fortlaufend deployen.
  - z.B. auf interne Testumgebungen oder direkt beim Kunden (Alpha, Beta, Release Candidate etc.).
  - sogar bis in die Produktion mittels einer Build-Pipline!
- Continuous Delivery (CD)
  - Analogie zum Testen, vgl. Continuous Integration!
  - Nutzung von DevOps Technologien, Infrastructure as Code.
- Staging: Deployment auf unterschiedliche Umgebungen!
  - Entwicklung, Test, Integration, Vor-Produktion, Produktion.

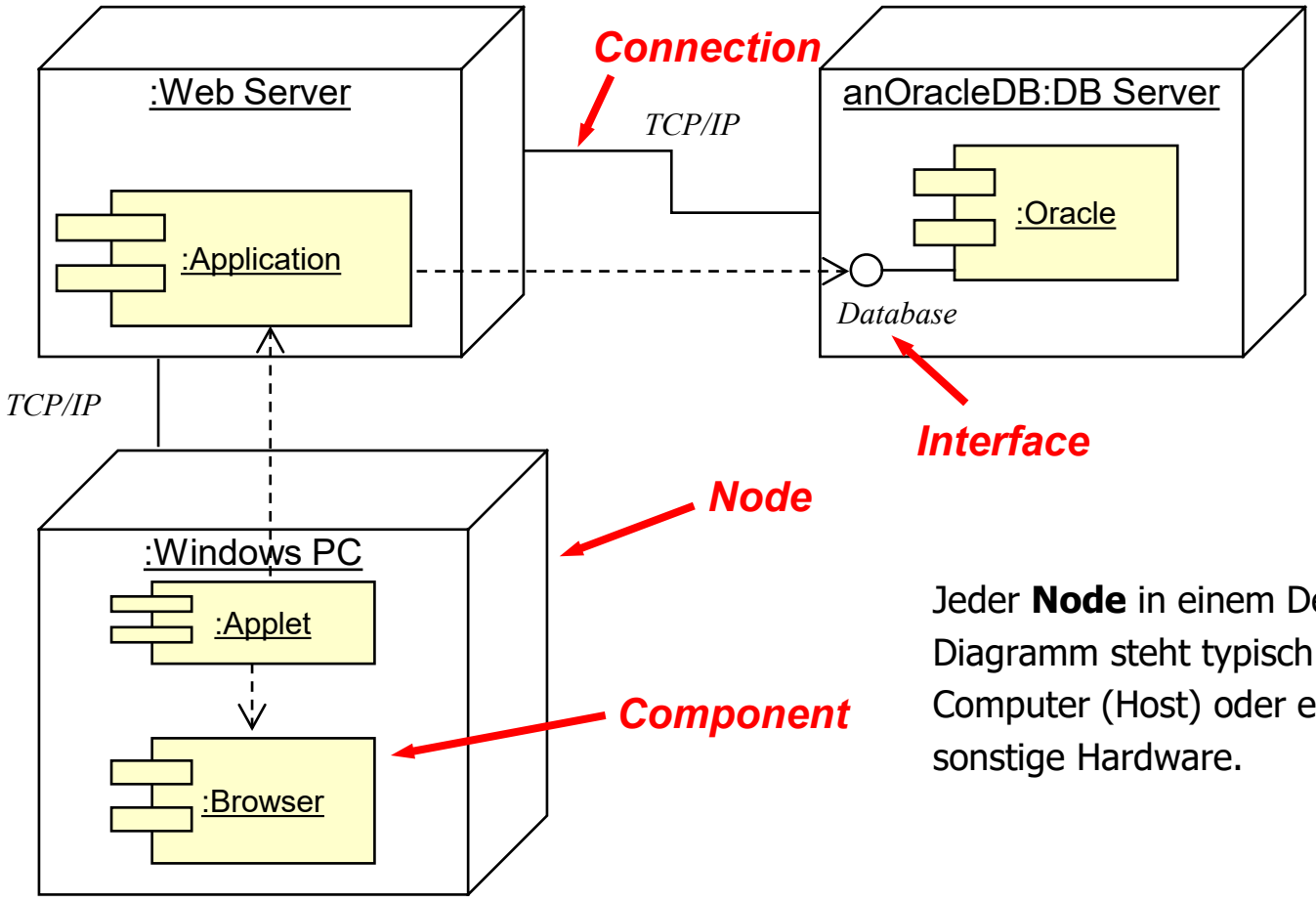
Deployment - Umfang

- Technische Aspekte:
  - Deployment Diagramme (Zuordnung Komponenten / Hardware)
  - Installations- und Deinstallationsprogramme / -skripte
  - Konfiguration (Default~, Beispiel~ etc.)
  - Installationsmedium
  - Repositories (Ablage der Binaries)
- Organisatorische Aspekte:
  - Konfigurationsmanagement: Welchen Komponenten bilden welchen Release (Baseline) oder BOM (bill of material).
  - Installations- und Bedienungsanleitungen
  - Erwartungsmanagement: Welche Funktionalität ist vorhanden?
  - Bereitstellung von Support (intern und extern, Levels)

➔ Deployment-Dokumentation als Quelle der Informationen!

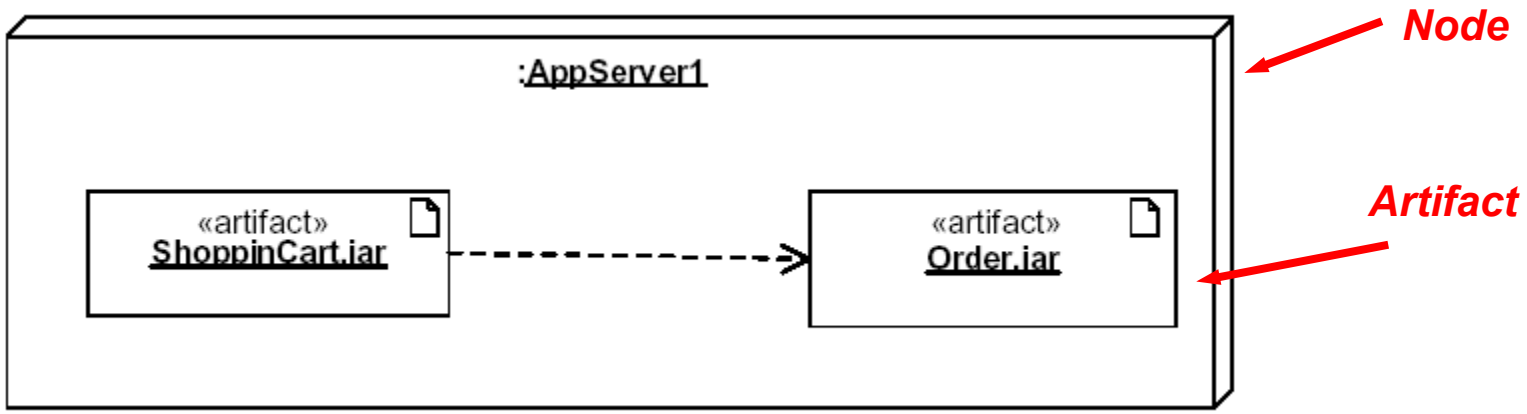
Deployment-Diagramme

UML 1.x Deployment Diagramm



Jeder **Node** in einem Deployment Diagramm steht typisch für einen Computer (Host) oder eine sonstige Hardware.

UML 2 Deployment Diagramm



- Node: Stellt einen Computer (Host) oder eine sonstige Hardware dar (identisch zu UML 1.x).
- Artifact: Stellt ein ausführbares Binary, ein Skript etc. dar welches durch die Installation explizit einem Node zugeordnet wird.
  - Wird in einer Deployment-Spezifikation detaillierter beschrieben.

Deployment - Aspekte

Installation und Deinstallation

- Installation (und Update) einer Software soll möglichst vollständig automatisiert (➔Reproduzierbar) sein.
  - Saubere Deinstallation ist ebenfalls wichtig, nicht vergessen!
  - Vollautomatisierte Verteilung möglich (Software-Management).
  - Potential von Package-Management Systemen.
- Sehr unterschiedliche Kundenbedürfnisse / Zielgruppen:
  - **Endbenutzer*in**: Grafische, interaktive GUI-Installation für Endanwender auf dem Desktop.
  - **Administrator*in**: Möglichst Script-basierte, durch Parametrisierung voll automatisierte Installation auf dem Server für die Administrator*in.
  - **Entwickler*in / Tester*in**: Spezielle Distributionen, entweder manuell (download) oder über (zentrale) Repositories.

Konfiguration von Anwendungen

- Ein klassischer Zielkonflikt!
- Die neu installierte Anwendung soll:
  - 1. möglichst sofort «out-of-the-box» lauffähig sein, …
  - 2. sich an verschiedenste Umsysteme anpassen können, und…
  - 3. trotzdem möglichst einfach aktualisierbar sein.
- Typische Beispiele / Anforderungen:
  - Datenbankanwendung: Lauffähigkeit auf einer individuellen, bestehenden DBMS-Umgebung.
  - Logging / Audit: Einsatz unterschiedlicher Logging- und Überwachungs-Mechanismen/Frameworks.
  - Security: Support verschiedener Authentifizierungs- und Autorisierung-Techniken (z.B. LDAP, Kerberos etc.).

Konfigurationsmanagement

- Mit der Zeit haben verschiedene Kunden unterschiedliche Versionen einer Software (typisch für Individual-Entwicklungen).
  - Welche Version läuft bei welchem Kunden?
  - Welcher Kunde hat welche Lizenzen?
- Verschiedene Kunden haben unterschiedliche Produkte und Versionen der Umsysteme (z.B. Datenbank) und der Hardware (z.B. Einfluss auf Performance)
  - Wer hat welche Konfiguration? Mit welchen Komponenten?
  - Welche Kombinationen sind überhaupt lauffähig?
- Ist ein Update von jeder existierenden Konfiguration möglich?
  - Müssen bestimmte Abfolgen eingehalten werden?
  - Wurden die unterschiedlichen Szenarien auch getestet?

Deploymentdokumentation / Manuals

- **Release Notes** - Erste Quelle über/zu:
  - Neuen Funktionsumfang.
  - Veränderte / zusätzliche Vorbedingungen.
  - Neue / veränderte Datenformate oder Protokolle.

  etc.
- Installationsanleitung
  - Haben sich HW- oder SW-Voraussetzungen geändert?
  - Varianten unterschiedlicher Konfigurationen berücksichtigen.
  - Müssen bestimmte Abfolgen eingehalten werden?
  - Tipp: Installation möglichst automatisieren. ➔ weniger Doku.
- Bedienungsanleitung / User-Manual
- VSK-Logger: KISS, **muss** aber vorhanden sein!

Releases und Versionierung

Releases und Versionierung

- Deployment findet mit einem wohldefinierten Release statt
  - Eindeutige Bezeichnung und Version
    - technische Version ist die eindeutige Identifikation.
    - 'Tagging' im Versionskontrollsystem!
    - 'Marketing-Version' unbedingt trennen, als Ergänzung.
- Bewährt: Dreistellige Version: **x.y.z** (z.B. **7.2.3**) mit Semantik
  ➔ Semantic Versioning http://semver.org/
- Anhand der Version soll möglichst einfach und klar ersichtlich sein was sich prinzipiell verändert hat:
  - Änderungen, Erweiterungen oder Korrekturen
  - Version hat speziell bei technischen Releases (z.B. Frameworks, Libraries, Komponenten) eine sehr wichtige Aussage

Semantic Versioning - Repetition

- **Major**-Release (**X**.x.x): Veränderungen in der API, in der fachlichen Funktion oder in der Konfiguration, welche zu früheren Versionen nicht mehr kompatibel sind und somit Anpassungen notwendig machen
- **Minor**-Release (x.**X**.x): Erweiterungen in der API, der fachlichen Funktion oder der Konfiguration, welche aber vollständig Rückwärtskompatibel sind und (zumindest ohne Nutzung derselben) keine Anpassungen notwendig machen
- **Bugfix/Maintenance**-Release (x.x.**X**): Reine Korrekturen oder Änderungen in der Implementation, voll rückwärtskompatibel, keinerlei neue Funktionen, keine veränderte Funktionen, direkter Einsatz möglich

Alternative: Time-Based Release Versioning

- Man hält einen festen «Takt» an Release-Terminen ein.
- Sinnvollerweise macht die Versionsnummer deutlich, um welchen Zeitpunkt es sich handelt.
- Beispiel Ubuntu-Releases: …, **21.04, 21.09, 22.04, 22.09**, …
  - **`YY.MM`**-Format: Jahr und Monat
  - Zur Eindeutigkeit häufig mit «Buildnummer» ergänzt.
- Einsatz bei «grossen» Produkten und/oder Marketing-Versionen sinnvoll, aber weniger bei Libraries/Komponenten.
  - War Ende des 20. JH sehr beliebt (V2000), Anfang 21. JH eher unbeliebt (**0**x-er Jahre!), und wird jetzt sehr beliebt… ☺
- Empfehlung: Versionieren Sie mit **Vernunft**!
  - Unterscheiden Sie zwischen technischer und Marketing-Version!

Time-Based Release Versioning bei Oracle / Java

- Java-Versionen seit September 2017 im Format:
  $**FEATURE**.$**INTERIM**.$**UPDATE**.$**PATCH**
- Bedeutung der Stellen:
  - **FEATURE**: Inkrementeller Counter, aktuell bei **16**.
  - **INTERIM**: Bleibt derzeit bei **0**.
  - **UPDATE**: Inkrementeller Counter für Updates.
  - **PATCH**: Optionale Patch/Build-Nummer (**17.0.3.1** erstmalig)
- Aktueller Release-Plan:
  - Feature Releases alle 6 Monate (jeweils März und September).
  - Updates jeweils +1/+3 Monate (Apr./Jul. und Okt./Jan.).
  - Alle **zwei** bis **drei** Jahre wird der September-(Feature-)Release als ein **LTS** (Long Time Support) deklariert (Aktuell: `17.0.5`).
  - Nächster LTS für 2023 angekündigt, somit vermutlich 21.0

Versionierung - Release-Notes

- Sauberes Nachführen von allen Änderungen, Erweiterungen und Korrekturen.
  - Nachvollziehbare Entwicklungsgeschichte.
  - Meist manuell nachgeführt, da qualitative Aussage.
  - Evt. unterstützt durch Issue-Tracking-Systeme.
    - Bugzilla, JIRA, Mantis, Trac, Redmine etc.
    - Direkter Bezug auf Change-Request oder Bug.
- Für Entwickler*innen die zentrale Informationsquelle um die Möglichkeit bzw. die Notwendigkeit einer Migration auf eine neue Version (und das damit verbundene Risiko) einschätzen zu können.
  - Wenn Sie selber Release-Notes lesen, sollten Sie auch bereit sein, für Ihr Produkt ein Release-Notes zu schreiben! ☺

Technisches Deployment (Java)

Techniken und Methoden – Beispiel: Java

- Verschiedene Plattformen (OS) und Applikationsarten.
  - Java: «***w**rite **o**nce, **r**un **a**nywhere*» (wora).
  - Einzelplatzanwendung (ein Host) - für Endanwender.
  - Serveranwendung (JEE, Applicationserver) - für Operating.
  - Library/Framework - für Entwickler
- Verteilung vieler einzelner `*.class`-Dateien ist in der Regel nicht praktikabel (aber es gibt Ausnahmen).
- Basiskonzept: Zusammenfassung von `*.class`-Dateien und Ressourcen in unkomprimiertem Zip-Format.
  - **J**ava **AR**chive (jar-Datei) oder **J**ava **MOD**ul (jmod-Datei).
- Erweiterte Konzepte (für Webcontainer und Applicationserver):
  - **WAR** (**W**eb **AR**chive, mit `META-INF/web.xml` etc.)
  - **EAR** (**E**nterprise **AR**chive, mit `META-INF/application.xml`)

Deploymentziele und Projektarten

- Je nach Produkt unterschiedliche Techniken zur Verteilung.
- Applikation / Anwendung für Endbenutzer*in: Verteilung in binärer Form, evt. mit automatischem Installationsprogramm (setup.exe) und Anleitung/Manual, ggf. sogar inkl. JRE.
  - Separierte JAR-Dateien (Komponenten) oder Single-JAR.
- Server-Applikation für Operating: Klassisch meist separierte JAR-Dateien für gezielte Updates, bei Microservices eher Single-JAR.
  - Immer beliebter als Alternative: Container-Deployment.
- Library/Komponente, für Entwickler*innen: Typisch Download, binäres Repository (z.B. Maven-Repo)
  - Mit konsistenter Versionssemantik (z.B. **`x.y.z-SNAPSHOT`**).
  - Inklusive Metadaten in strukturierter Form, für automatisches Dependency-Management (z.B. **`pom.xml`**).

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

- Eine vollständige Applikation
  - kann selber aus **1..n** einzelnen JAR's bestehen.
  - kann zusätzlich **0..n** Dependency-JAR's benötigen.
- JAR's können extern (Name) oder intern (Manifest) versioniert sein.
  - Bei >1 JAR-Dateien ist ein Classpath notwendig!
- Eine Menge von JAR's:
  - Einzelne Komponenten/Libraries z.B. für Bugfixing leicht austauschbar (+) vs. Dynamik des Classpath. (-)
- Einzelnes JAR (Shade-/Uber-/Fat-JAR):
  - Zusammenfassung des Inhaltes mehrerer JAR-Dateien in einem einzigen JAR zwecks einfacherem Deployment.
  - Einfachheit (+) vs. Redundanzen der Dependencies (-)
  - Problematik der Neusignierung und META-INF (Manifest etc.)

Modularisierung seit Java 9 (Project Jigsaw)

Mit Java 9 (September 2017) wurden endlich die langersehnten technische Möglichkeiten zur «echten» Modularisierung mit Java implementiert. Dabei standen drei Ziele im Vordergrund:

- **Reliable Configuration**: Den fehleranfälligen Classpath durch den auf Modul-Abhängigkeiten basierenden Modul-Path ablösen.
- **Strong Encapsulation**: Ein Modul definiert explizit sein öffentliches API. Auf alle restlichen Klassen ist von Ausserhalb kein Zugriff mehr möglich (auch wenn `public`)!
- **Scalable Platform**: Die Java-Plattform selber wurde modularisiert, so dass für Anwendungen individuell angepasste, schlankere Runtime-Images gebaut werden können.

Modularisierung in Java 9 - Umsetzung

- Java-Packages können neu in Modulen zusammengefasst werden.
  - Optionale, zusätzliche Strukturebene in der Dateiablage.
  - Eindeutige Namensgebung nötig (analog zu Packages).
- Pro Modul wird ein **`module-info.java`** definiert. Darin werden explizit die Import, Exports und Abhängigkeiten definiert!
  - Somit wird eine Designverifikation zur Compile-Time möglich!
- Zusätzlich findet beim Start einer Applikation eine Laufzeitprüfung statt, ob alle notwendigen Komponenten vorhanden sind.
  - **`ClassNotFound`** Exeption sollte somit nicht mehr auftreten!
- Das Ende der «JAR-Hell»: Es wurde ein neues Format (**`jmod`**) definiert, der Classpath wird durch den Modul-Path abgelöst.
- Und das alles vollständig rückwärtskompatibel! ☺

Beispiel eines sehr einfachen `modul-info.java`

- Sehr einfaches Beispiel eines **`modul-info.java`**
  - Ist eine «spezielle» Klasse.
  - Hält sich (analog zu **`package-info.java`**) bewusst **nicht** an die Namenskonvention von Java, so dass es nirgends zu Konflikten kommt.

```
module ch.hslu.sort.quicksort {
      exports ch.hslu.sort.quicksort.impl;
      requires ch.hslu.sort.api;
}
```

Verteilung über Binär-Repositories

- Ein **Binär**-Repository
  - hat **nichts** mit einem VCS/SCM (git etc.) zu tun!!
  - strukturierte Ablage von Binaries (JAR, WAR, EAR, JMOD etc.).
  - Primär für Entwicklungsprozess (Dependency-Management).
- Ursprünglich als «Maven-Repository» entstanden (Apache Maven).
  - https://repo1.maven.org/maven2/
  - Basierte ursprünglich auf einer Verzeichnisstruktur im FS.
  - Heute meist als spezielle Serversoftware implementiert, erlaubt z.B. effizientere Suche, Management und Analysen.
- Produkte für on-site-Betrieb (typisch in Organisationen).
  - Beispiele: JFrog Artifactory, Apache Archiva, Sonatype Nexus etc.
- Auch als reine Cloud-Dienste, z.B. integriert in Codehosting-Systeme wie https://gitlab.com/ (Package Repository).

Verteilung für Java: HSLU Nexus Repository

- HSLU EnterpriseLab bietet seit HS17 ein Repository auf Basis von Nexus zur Verfügung.
  - https://repohub.enterpriselab.ch/
- Über das Installierte **`settings.xml`** (Maven-Konfiguration) verwenden Sie dieses Repository nur lesend.
  - Cache/Mirror für externe Repositories ➔ Schneller.
- Ausschliesslich der (Jenkins-)Buildserver verfügt über die notwendigen Rechte um in das Repository zu schreiben.
  - Projekte wurden laufend in das Repository deployed.
- Achtung: Unterschiedliches Verhalten des Deployments und der Dependency-Resolution bei **Releases** oder **`SNAPSHOTS`**!
  - Haben Teams, die den «SNAPSHOT»-Appendix entfernt haben, etwas festgestellt?

Deployment bei Open Source Projekten

- Deployment erfolgt häufig über zwei verschiedene Distributionen:
  - **Binär**-Distribution: Enthält binäre Runtime plus Dokumentation, direkt einsetzbar. Häufig als ZIP-Datei zum Download.
  - **Source**-Distribution: Enthält nur den Quellcode und alle notwendigen Buildartefakte. Heute häufig über VCS-Zugriff.
- Aus der Source-Distribution sollte die Binär-Distribution jederzeit wieder erstellt werden können.
  - Entwicklungswerkzeuge (JDK etc.) vorausgesetzt
- Alle Distributionen werden sauber versioniert
  - Ideal / Empfehlung: ➔Semantic Versioning
- **Hinweis:** Binäre Repositories unterstützen auch die Verteilung von Quellen und JavaDoc(**`*-source.jar`** und **`*-javadoc.jar`**).
  - Quellpacket entspricht **nicht** einer Source-Distribution!

Beispiel 1: VSK-Logger-(Server-)Projekt

VSK-Logger - Deploymentvarianten

- Jede Applikation bzw. Komponente bzw. Library ist ein eigenes (Teil-)Projekt und baut je ein eigenes, versioniertes JAR.
  - Nebenbei: Das wären in Zukunft die (gröbstmöglichen) Module.
- Variante **1**: Start der Applikationen (z.B. der Server) mit Classpath welcher alle notwendigen JAR-Dateien enthält.
  - Variante 1**a**: Argument **`-cp`** beim Start (z.B. über `Shell`-Script).
    - Einfache Erweiterung des Classpath, z.B. für Resourcen.
    - Einfacher Austausch (!) einzelner JAR-Dateien (z.B. Bugfix).
  - Variante 1**b**: Class-Path-Eintrag in `META-INF/MANIFEST.MF` in der JAR-Datei des Anwendung.
    - JAR (Manifest) muss bei Änderung neu gebaut werden.
- Variante **2**: FAT-JAR, welches ALLE Klassen enthält (Shade-JAR).
  - Muss immer komplett neu gebaut werden ➔ hier **nicht** sinnvoll!

VSK-Logger – JAR-Dependencies auflösen (kopieren)

- Das «Maven Dependency Plugin» hilft uns:
  https://maven.apache.org/plugins/maven-dependency-plugin/index.html
- Beispiel: Liste der Abhängigkeiten anzeigen (ohne `test`-Scope):

```
mvn dependency:list -DincludeScope=compile
```

- Resultat (Beispiel für g01-demoapp):

```
The following files have been resolved:
   ch.hslu.vsk22hs.g01:g01-loggercommon:jar:1.0.0-SNAPSHOT:compile -- module g01.loggercommon (auto)
   ch.hslu.vsk22hs.g01:g01-loggercomponent:jar:1.0.0-SNAPSHOT:compile -- module g01.loggercomponent (auto)
   ch.hslu.vsk22hs:loggerinterface:jar:1.0.0:compile -- module loggerinterface (auto)
   ch.hslu.vsk:stringpersistor-api:jar:5.0.4:compile -- module stringpersistor.api (auto)
   ch.hslu.vsk22hs.g01:g01-stringpersistor:jar:1.0.0-SNAPSHOT:compile -- module g01.stringpersistor (auto)
```

- Mit Hilfe dieses Plugins lassen sich die Dependencies auch automatisch aus dem Binär-Repository zusammenkopieren:

```
mvn dependency:copy-dependencies
 -DincludeScope=compile
```

  - Resultat liegt danach im Verzeichnis **`./target/dependency`**

VSK-Logger – Starten der Applikationen – Variante 1

- Ausserhalb der IDE kann eine Applikation mit Hilfe von Apache Maven gestartet werden (also noch mit Hilfe des Build-Tools).
  - «Maven Exec Plugin» hilft hier weiter: https://www.mojohaus.org/exec-maven-plugin/
- Beispiel (für g01-demoapp):

```
mvn exec:java↵
      -D"exec.mainClass=ch.hslu.vsk.demoapp.DemoApp"
```

  - Voraussetzung: Apache Maven ist installiert, der lokale Build war erfolgreich, und alle Buildartefakte liegen somit vor.
- Typisch nur für Entwickler*innen-Arbeitsplätze nutzbar/sinnvoll.
  ➔ Wegen der Abhängigkeit zum Buildtool sicher **keine** Lösung für Endbenutzer*innen.

VSK-Logger – Starten der Applikationen – Variante 2

- Wenn die JAR-Dateien alle vorliegen, kann die Applikation auch direkt (nur über Java) gestartet werden!
- Beispiel (für g01-demoapp):

```
java -cp "./g01-demoapp-1.0.0.jar;./dependency/*"↵
   ch.hslu.vsk.demoapp.DemoApp
```

  - Voraussetzung: Dependencies wurden vorher kopiert; aktuelles Verzeichnis ist **`./target`** (dort befindet sich das Haupt-JAR).
- Befehl am einfachsten in ein Shell-Script (cmd, sh etc.) ablegen!
  - Sieht «rustikal» aus, ist aber einfach, robust und transparent!
- Achtung: Pfadangaben sind plattformspezifisch, unter Unix/Linux ist z.B. das Pfadtrennzeichen ein Doppelpunkt (`:`), bei Windows ein Semikolon (`;`).

Beispiel 2: JavaFX Applikation

Java FX – Modularisierte Architektur

- Java FX ab Version 11 (und höher) nutzt die neuen (seit Java 9) verfügbaren Techniken zur Modularisierung.
- Somit muss man sich bei einer JavaFX-Anwendung «zwingend» mit Modularisierung auseinander setzen, auch wenn die restliche Applikation diese Technik (noch) nicht einsetzt.
- Seit dem beschleunigten Releasing von Java (seit Version 9) herrscht eine relativ grosse Dynamik, und man «rennt» in den Buildsystemen und den IDE's der Realität etwas hinterher.

➔ **Wichtig**: Das ist **kein** Nachteil oder Kritik an Java FX.
Es kommen hier «einfach» ein paar Dinge zusammen, um welche man sich bisher (bei Java) nicht kümmern musste.
- Potential offenbart sich derzeit leider noch nicht so deutlich.

Java FX – Starten einer Applikation mit Maven

- Für Maven existiert ein Plugin von OpenJFX, welches den korrekten Start einer (teil-)modularisierten JavaFX-Applikation automatisiert.
  - **`org.openjfx:javafx-maven-plugin:0.0.8`** (Nov. 2022)
  - Befehle **`mvn javafx:run`** [**`-debug`**] oder **`mvn javafx:jlink`**
- Mittels **`-debug`**-Option von Maven kann man sich den Startbefehl (sinngemäss) herausholen. Beispiel (konzeptionell):

```
java
 --module-path
  javafx-base-17.0.5-win.jar;javafx-base-17.0.5.jar;
  javafx-controls-17.0.5-win.jar;javafx-controls-17.0.5.jar;
  …
 --add-modules javafx.base,javafx.controls,…
 -cp app.jar;log4j-api-2.19.1.jar; …
 ch.hslu.demo.GuiStartClass
```

- Gute Hilfestellung unter https://openjfx.io/openjfx-docs/#maven

Deployment als Container

VSK-Logger – Deployment als FAT-JAR in Container

- Vorteile:
  - Bau des Images und Start-Befehl sehr einfach.
  - Simples Dockerfile zur Konfiguration.
  - FAT-JAR kann auch ausserhalb des Containers sehr einfach genutzt/getestet werden.
- Nachteile:
  - FAT-JAR kann relativ gross werden ➔ Buildprozess (zusammenkopieren) wird langsam.
  - Fetter Layer im Container-Image (kleine Änderung, trotzdem ganzes JAR/Layer neu erstellt)
  - Problematik mit den Manifesten und Konfigurationen.
- Beispiel: vsk-echoserver, Docker-Image mit Fabric8-Plugin erstellt.
  - siehe Struktur des Images – Live-Demo (dive)!

VSK-Logger – Deployment als «layered» Container

- Vorteile:
  - Feingranulare Trennung der Applikation, Dependencies und jeweilige Konfigurationen in getrennten Layern.
  - Unterschiedliche Änderungshäufigkeit (App. versus Deps.) beschleunigt den Build massiv und senkt Ressourcenbedarf.
  - Automatisierung z.B. durch Google JIB-Plugin.
- Nachteile:
  - Bau des Images komplizierter, weil differenzierter.
  - Komplexeres Dockerfile (aber ggf. gar nicht sichtbar).
  - Applikation in dieser Form ohne manuelle Eingriffe nur im/mit Container lauffähig.
- Beispiel: vsk-echoserver, Image mit JIB-Plugin erstellt.
  - siehe Struktur des Images – Live Demo!

Demo’s / Screencast’s

- Java Deployment: Starten von Java-Applikationen ausserhalb der Entwicklungsumgebung:
  EP_40_SC01_JavaDeployment.mp4

Zusammenfassung

- Verschiedene Aspekte des Deployments:
  Technisch (Verteilung, Installation etc.), Organisatorisch.
- Sinnvolle Deployment-Dokumentation:
  Unterschiedliches Zielpublikum berücksichtigen.
- Semantic Versioning
- Deployment-Techniken für/mit Java
- Spezifische Anforderungen für effizientes Deployment in/mit Container-Images.

Fragen?