JAVA DEVOPS CLOUD ENTWICKLUNG

DEVOPS & CLOUD COMPUTING IM JAVA UMFELD

DevOps, Cloud Computing, Iaas, PaaS, Container und Micro-Services haben die Welt der Java Entwicklung nachhaltig verändert. Der Kurs stellt einige Entwicklungen und Lösungen speziell für Java Entwickler auf.

CGS – IT Solutions @ 2020/2024
Version 1.0.6

Christian Schaefer
cgs@cgs.at

1 Inhalt

2 Einleitung und Kursüberblick

Dieser Kurs gibt einen Überblick über spezifische Änderungen, Technologien und Herausforderungen für Java Entwickler im Kontext von DevOps und Cloud-basierter Softwareentwicklung. Da diese Neuerungen nicht isoliert bestehen/entstehen wird versucht einen kombinierten Überblick über DevOps, Cloud/IaaS und den damit einhergehenden Software Architektur Änderungen wie Micro-Services zu geben.

2.1 Ziele

- Sie haben einen Überblick über den Aufbau moderner Architekturen und DevOps Konzepten
- Sie können den Aufbau von Cloud Anwendungen beurteilen und kennen die wesentlichen Building Blocks dieser Systeme
- Sie kennen die Funktion von CI/CD und können einfache Abläufe selbst konfigurieren
- Sie können einfache Ansible Automatisierungs-Scripts schreiben und kennen den technologischen Aufbau
- Sie können ein einfaches Micro-Service auf Basis von Quarkus selbst entwickeln.
- Sie können ein einfaches Docker Image erstellen und in der Amazon Cloud deployen.

3 Technologie und Architektur Überblick

Mehrere teilweise unabhängige Entwicklungen führen zu einer veränderten Architektur und Lösungs-Umsetzung für Softwareanwendungen.

Agile Iterative Softwareentwicklung führte auch dazu, dass die von jedem Scrum Team entwickelten Teile autonomer und getrennter voneinander entwickelt, getestet und deployed werden und damit auch agile Micro-Service Architekturen fördern oder und fordern. Siehe: (t2informatik) Conway's Law.

DevOps wiederum versucht die Lücke zwischen Entwicklung und Software-Betrieb zu schließen indem zusätzlich zum klassischen agilen Team auch der Betrieb inkludiert wird.

Sowohl klassisches Cloud Computing ala (AWS_Amazon) als auch Container orientierte Architekturen erfordern einen DevOps orientierten Ansatz. (Wikipedia)

3.1 DevOps

DevOps ist kein genau spezifizierter Begriff, bezeichnet aber die verbesserte und integrierte Zusammenarbeit zwischen Entwicklung und Betrieb.

Dabei werden hier auch die Begriffe CI (Continuous integration) und CD (Continuous delivery) für eine kontinuierliche Test und Integration der Software als auch eine automatische Ausrollung der Software in die Produktion geprägt.

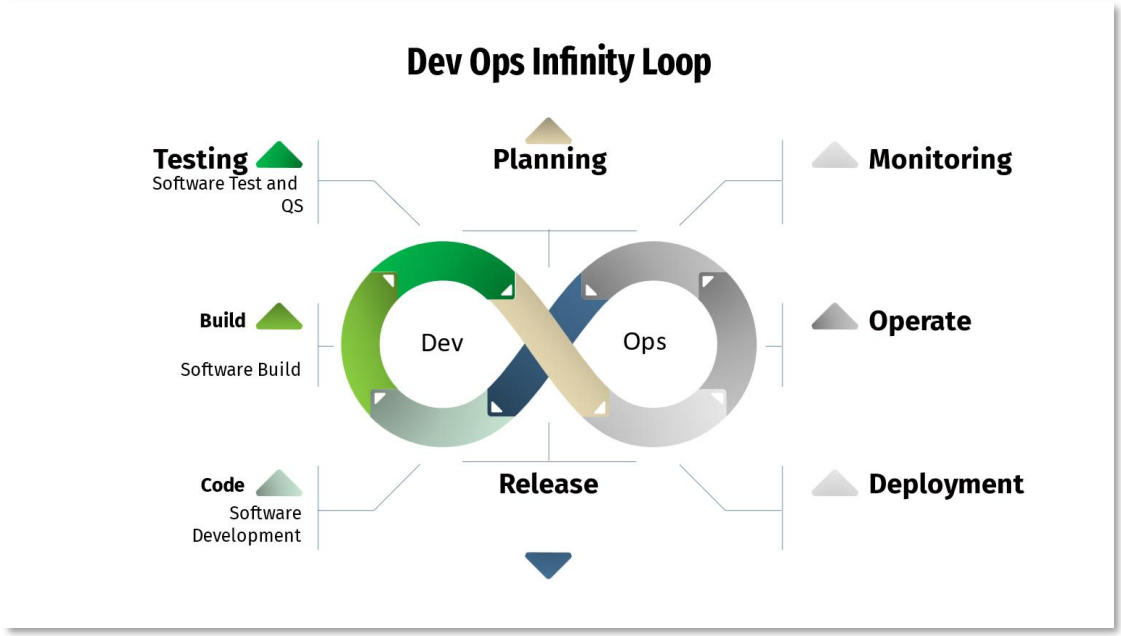


Abbildung 1 DevOps

3.1.1 Dev Ops Pipeline

Der Ablauf und die Steuerung von CI/CD wird auch als „Pipeline“ bezeichnet. Pipeline als Begriff findet sich auch im klassischen Tool „Jenkins“ als Begriff.

Die Pipeline steuert dabei den Ablauf der zu erledigenden Tasks bzw Schritte (Steps). Pipelines können selbst wieder miteinander verbunden und verknüpft werden. Die hier abgebildete vereinfachte Darstellung einer klassischen Java/git Pipeline wird in der Praxis eher in mehrere einzelne Pipelines mit jeweils eigenen Steps unterteilt werden.

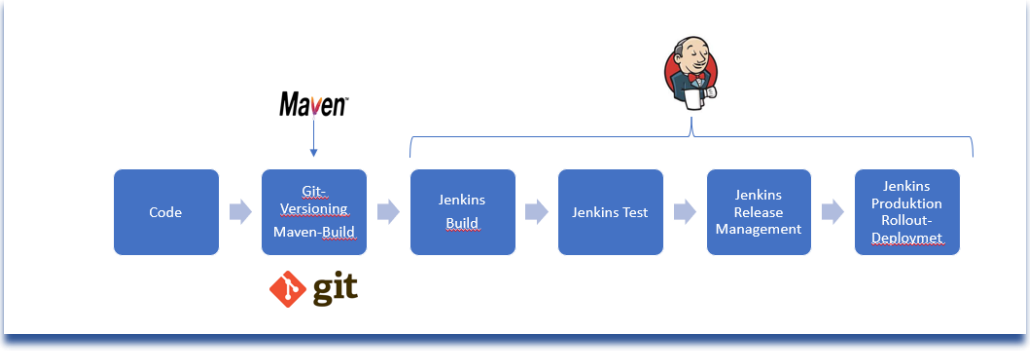


Abbildung 2 Dev Ops Pipeline

3.2 Cloud Computing

Aus der Definition von Wikipedia:

*„**Cloud Computing** (deutsch Rechnerwolke oder Datenwolke) beschreibt ein Modell, das bei Bedarf – meist über das Internet und geräteunabhängig – zeitnah und mit wenig Aufwand geteilte Computerressourcen als Dienstleistung, etwa in Form von Servern, Datenspeicher oder Applikationen, bereitstellt und nach Nutzung abrechnet.“*
https://de.wikipedia.org/wiki/Cloud_Computing

Für (Java) Entwickler/Architekturen ergeben sich daraus unzählige neue Möglichkeiten der Software-Umsetzung und Architektur.

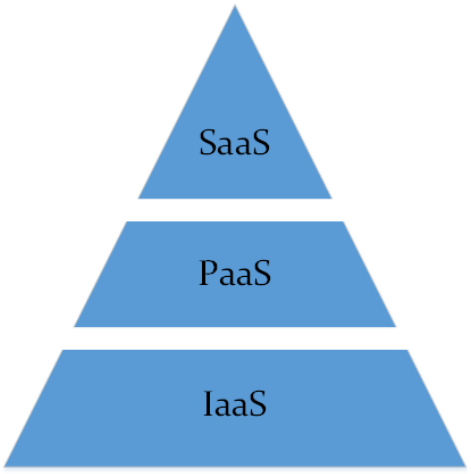


Abbildung 3 Quelle Wikipedia

Wichtige grundlegende Definitionen sind:

3.2.1 On Premise vs Cloud

Als wichtige grundlegende Entscheidung zwischen eigener ge-hosteter „On Premise“ Lösungen oder der Nutzung eines Cloud Providers wie Amazon, Google, Microsoft etc. stellt die Zielumgebung sowohl bei den technischen Möglichkeiten als vor allem auch bei der Konzeption der Sicherheit eine wesentliche Rolle.

Als Lösungskonzepte tendiert man bei In-House On Premises Lösungen eher zu stärkerer Docker bzw Kubernetes Komponentisierung, da das Netzwerk und die Storage Cloud Features oftmals bei InHouse Lösungen nicht in dieser Art und Weise verfügbar sind wie sie das etwa in einer AWS Cloud sind.

3.2.2 IaaS – Definiton

Als erste wichtige Definition stellt IaaS (Infrastructure as a Service) Basisressourcen wie Netzwerk, Speicher und Virtuelle Server zur Verfügung.

3.2.3 PaaS - Definiton

Platform as a Service baut auf IaaS auf und stellt zusätzlich zur Basis Infrastruktur eine Laufzeitumgebung für die entsprechende Platform zur Verfügung. Beispiele dafür wären Google App Engine (https://de.wikipedia.org/wiki/Google_App_Engine) oder Amazon Beanstalk. Beanstalk stellt einen vorgefertigen Tomcat Container mit Load Balancing und Auto Skalierung zur verfügung.

3.2.4 SaaS - Definiton

Software as a Service stellt schliesslich fertige Software bereit die in der Cloud verwendet werden können.

Auch für die Software Entwicklung selbst gibt es SaaS Angebote wie Atlassian Jira/Confluence oder auch als Beispiel DataDog SaaS Monitoring (https://www.datadoghq.com/saas-monitoring/)

3.2.5 Serverless Architecture

„Serverless" als Begriff bezieht sich auf ein Cloud-natives Applikations und Entwicklungs-konzept, bei dem Entwickler Anwendungen erstellen und ausführen können, ohne Server verwalten zu müssen.

Serverless-Apps werden in Containern bereitgestellt, die bei Bedarf automatisch aktiviert werden.

Serverless Computing unterteilt sich bei Cloud Anbietern zwei Gruppen:

1. BaaS (Backend-as-a-Service) und
2. FaaS (Function-as-a-Service).

Serverless Lösungen eignet sich am besten für asynchrone und zustandslose Anwendungen, die unmittelbar gestartet und verwendet werden können, vor allem mit sehr heterogener Lastverteilung.

Weiterführende Literatur:

- https://martinfowler.com/articles/serverless.html
- https://aws.amazon.com/de/serverless/

3.2.6 Serverless Kubernetes

Auch für Anwendungen die vor allem Auf Docker/Kubernetes aufsetzten gibt es Implementierungen zur Umsetzung.

Weiterführende Literatur:

- https://cloud.google.com/knative/

4 Jenkins Build und Deployment Automatisierung

4.1 Continuous Integration and Delivery

- Jenkins CI (Continuous Integration)
- Harness CD Überblick
- Ansible Grundlagen

5 Ziele der Automatisierung mit Jenkins

Die Automatisierung mit Jenkins soll die folgenden Aufgabenbereiche abdecken:

1. **Automatisierter Build und Test:**
    - Bei jedem Code-Push oder Commit in das GIT-Repository wird automatisch ein Build erstellt und Tests durchgeführt.
2. **Automatisierte Builds und Tests für Feature-Branches:**
    - Mit Hilfe der Multibranch-Pipeline (siehe Anhang: Multibranch-Pipeline) können auch Feature-Branches integriert werden.
3. **Erstellung von Software-Releases:**
    - Mit Maven-Release-Befehlen wird ein Release erstellt und die generierten Binärdateien werden in das Nexus-Release-Repository hochgeladen.
4. **Softwareauslieferung und Verteilung:**
    - Nutzung von Jenkins-Pipelines zur Verwaltung der Auslieferung und Verteilung der Software.

5.1 Jenkins Pipeline Example

Abbildung 4 Jenkins Build Pipeline

Siehe Appendix für ein Jenkins Beispiel:

Jenkinsfile Beispiel

6 Abbildung 5: Jenkins Build-Pipeline

Die Konsolenausgabe der Build-Pipelines dient als Dokumentation aller durchgeführten Schritte und kann insbesondere bei der Softwareauslieferung als Auslieferungs-Logdatei verwendet werden.

Ein weiterer Ausbau der Pipelines ist flexibel möglich. Beispielsweise können folgende Funktionen integriert werden:

- **Parametersteuerung:**
  - Verwendung von Parametern, um die Ausführung der Pipeline dynamisch zu beeinflussen.
- **Pipeline-Verkettung:**
  - Nach Abschluss eines Jobs können weitere Pipelines aufgerufen werden.
- **SonarQube-Integration:**
  - Einbindung von SonarQube zur Analyse des Quellcodes und zur Qualitätssicherung.
- **Manuelle Haltepunkte:**
  - Integration von manuellen Haltepunkten, beispielsweise für manuelle Überprüfungen oder Genehmigungsschritte während der Auslieferung.
- **Zeitgesteuerte Ausführung:**
  - Konfiguration einer Cron-basierten Zeitsteuerung für die automatische Ausführung von Jobs.

7 Jenkins Zusammenfassung

Jenkins ist ein leistungsstarkes Werkzeug zur Automatisierung von Build-, Test- und Deployment-Prozessen. Durch die Nutzung von Pipelines können komplexe Workflows abgebildet werden, die flexibel erweiterbar sind.

Die Integration von Tools wie Maven, SonarQube und Nexus unterstützt eine nahtlose Softwareentwicklung und Bereitstellung.

8 Ansible Automation Framework

8.1 What is Ansible?

Ansible is an open-source IT automation framwork developed by Red Hat. It simplifies the management, provisioning, and configuration of IT infrastructure by providing a simple, agentless mechanism to automate tasks across multiple systems.

Ansible is widely appreciated for its simplicity, scalability, and ease of use, making it a popular choice among DevOps professionals and IT administrators.

8.2 Core Features of Ansible

1. **Agentless Architecture**: Unlike other automation tools, Ansible does not require agents to be installed on the target systems. It uses SSH for communication, which simplifies the setup and reduces overhead.
2. **Declarative Configuration**: Ansible uses a declarative language to define the desired state of the system, making configurations easy to read and maintain.
3. **Idempotency**: Ansible ensures that tasks produce the same result, no matter how many times they are executed.
4. **Extensibility**: With a wide range of modules, plugins, and integrations, Ansible can handle various automation tasks, from provisioning to application deployment.
5. **YAML-Based Playbooks**: Configurations and tasks are written in YAML, a human-readable language that makes playbooks easy to understand.
6. **Community Support**: Ansible Galaxy and other community-driven platforms offer reusable roles and modules.

8.3 Key Concepts in Ansible

1. **Playbook**: A YAML file that contains a series of tasks to be executed on the target systems.
2. **Inventory**: A list of managed hosts or nodes, organized in groups.
3. **Module**: A reusable script that performs a specific task (e.g., file management, package installation).
4. **Role**: A set of reusable configurations and tasks, logically grouped for modularity.
5. **Facts**: System variables collected by Ansible about managed hosts.
6. **Task**: A single operation to be executed (e.g., starting a service, copying a file).
7. **Handlers**: Tasks triggered only when notified by another task.
8. **Variable**: A placeholder for data that can be reused across playbooks.

8.4 Installing and Setting Up Ansible

1. **Installation**:
   - **Linux**: Use package managers such as yum, apt, or dnf.
   - **MacOS**: Use Homebrew with the command brew install ansible.
   - **Windows**: Use Windows Subsystem for Linux (WSL) or a virtual environment.
2. **Setting Up Inventory**:
   - Create an inventory file, typically named inventory.ini.
   - Define target systems with their IP addresses or hostnames.

Example:

```
[web_servers]
web1.example.com
web2.example.com

[db_servers]
db1.example.com
```

3. **Configuring Ansible**:
   - Modify **ansible.cfg** to specify defaults such as inventory location and SSH settings.

8.5 Writing a Simple Playbook

Below is an example of a simple playbook to install Apache on web servers:

```
---
- name: Install Apache Web Server
  hosts: web_servers
  become: yes

  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present

    - name: Start Apache Service
      service:
        name: apache2
        state: started
        enabled: true
```

Steps:

1. Save this playbook as install_**apache.yml**.
2. Run the playbook with the command:

```
ansible-playbook -i inventory.ini install_apache.yml
```

8.6 Ansible Advanced

8.7 Advanced Topics

1. **Dynamic Inventory**: Use scripts or plugins to dynamically generate inventory data.
2. **Custom Modules**: Develop custom modules to handle specific use cases.
3. **Vault**: Secure sensitive data using Ansible Vault.
4. **Callbacks and Plugins**: Extend Ansible functionality with custom callbacks and plugins.
5.

8.8 Real-World Use Cases

1. **Infrastructure Provisioning**: Automate cloud resource provisioning using Ansible modules for AWS, Azure, or GCP.
2. **Configuration Management**: Maintain consistent configurations across servers.
3. **Application Deployment**: Deploy and update applications with zero downtime.
4. **Security Compliance**: Enforce security policies and perform patch management.

8.9 Ansible – Conclusion – Zusammenfassung

Ansible is a powerful and simple tool that can significantly improve IT automation workflows.

With its agentless design, human-readable syntax, and robust community support, it is a valuable asset for organizations striving to enhance efficiency and reliability in their infrastructure management processes

See Ansible Examples in the GIT Repository for practical examples:

```
2024_12_clouddev_wifi_code\200_scripts\005_ansible
```

9 Terraform

9.1 What is Terraform?

Terraform is an open-source Infrastructure as Code (IaC) tool developed by HashiCorp.

It enables users to define, provision, and manage infrastructure resources across multiple cloud providers and services using a declarative configuration language.

Terraform is widely adopted for its ability to manage complex environments and provide a consistent workflow for infrastructure automation.

9.2 Terraform Use Cases

1. **Infrastructure Provisioning**: Automate the provisioning of servers, storage, and networks in the cloud.
2. **Multi-Cloud Deployments**: Manage resources across multiple cloud providers from a single configuration.
3. **Compliance Enforcement**: Ensure infrastructure adheres to organizational policies using pre-defined configurations.
4. **Disaster Recovery**: Create consistent and repeatable recovery environments using Terraform.
5. **CI/CD Pipelines**: Integrate Terraform into CI/CD workflows to provision infrastructure during deployment.

9.3 Core Features of Terraform

1. **Multi-Cloud Support**: Terraform integrates with major cloud providers like AWS, Azure, GCP, and on-premises solutions.
2. **Declarative Configuration**: Users define the desired state of infrastructure in a human-readable format using HashiCorp Configuration Language (HCL).
3. **Execution Plans**: Terraform generates a detailed execution plan before applying changes, showing what will be created, modified, or destroyed.
4. **State Management**: Terraform tracks resources using a state file, ensuring that infrastructure matches the declared configurations.
5. **Modules**: Terraform supports reusable modules to simplify and standardize infrastructure management.
6. **Provider Ecosystem**: Terraform uses providers to interact with APIs of cloud platforms and other services.

9.4 Key Concepts in Terraform

1. **Providers**: Plugins that enable Terraform to interact with cloud services or APIs (e.g., AWS, Azure).
2. **Resources**: The components of your infrastructure, such as virtual machines, databases, and networks.
3. **Variables**: Input parameters to customize configurations.
4. **Outputs**: Values that Terraform generates and provides after applying configurations.
5. **Modules**: Containers for multiple resources grouped together for reusability.
6. **State**: A JSON file that Terraform uses to map resources to real-world objects.
7. **Plan**: A command that previews the changes Terraform will make to infrastructure.

9.5 Installing and Setting Up Terraform

1. **Installation**:
    - Download Terraform from HashiCorp's official website.
    - Extract the binary and add it to your system's PATH.

Example for Linux/MacOS:

```
curl -O
https://releases.hashicorp.com/terraform/<VERSION>/terraform_<VER
SION>_linux_amd64.zip

unzip terraform_<VERSION>_linux_amd64.zip

sudo mv terraform /usr/local/bin/
```

9.5.1 Verify Installation:

```
terraform --version
```

Initialize a Working Directory:

1. Create a directory for your Terraform configuration files.
2. Run terraform init to download required providers and set up the working environment.

In the Training Session we used

```
terraform_work
```

9.5.2 Terraform Configuration

Below is an example of a simple configuration to provision an EC2 instance on AWS:

```
provider "aws" {
  profile = "default"
  region  = "eu-central-1"
}

resource "aws_instance" "terraform_test_server" {
  ami           = "ami-0d527b8c289b4af7f"
  instance_type = "t2.micro"

  tags = {
    Name = var.instance_name # TerraformTestInstance
  }
}
```

9.5.3 Steps:

1. Save the configuration file as main.tf.
2. Run terraform init to initialize the directory.
3. Preview changes with:

```
terraform plan
```

4. Apply the configuration:

```
terraform apply
```

5. Destroy the resources when no longer needed:

```
terraform destroy
```

10 Terraform - Best Practices

1. **Organize Configurations**: Group resources logically using modules.
2. **Use Remote State**: Store the state file in a remote backend (e.g., S3) for collaboration.
3. **Implement Version Control**: Use Git to track changes in your configurations.
4. **Lock Provider Versions**: Specify provider versions to avoid unexpected changes.
5. **Test Configurations**: Use tools like Terratest to validate configurations before deployment.
6. **Environment Separation**: Maintain separate configurations for development, staging, and production environments.

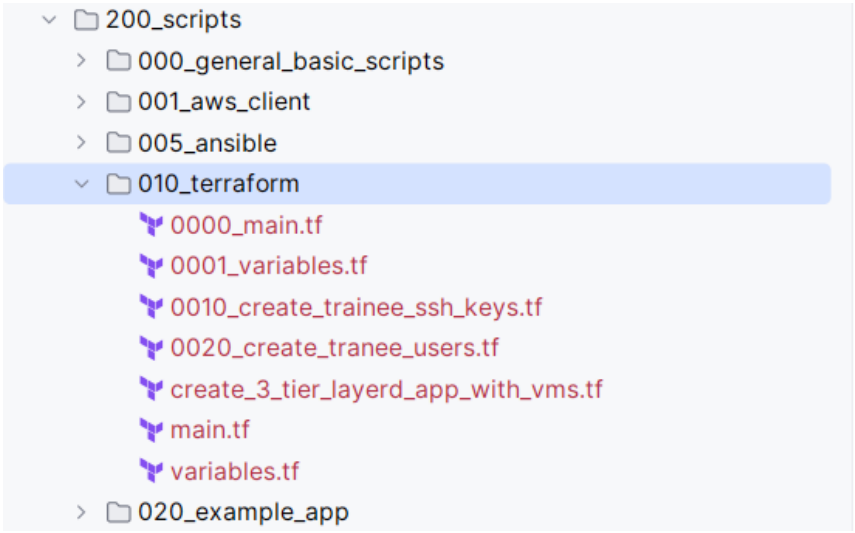
10.1.1 Advanced Topics

1. **Modules**: Create reusable modules for standardizing infrastructure components.
2. **Workspaces**: Manage multiple environments (e.g., dev, prod) within a single configuration.
3. **Terraform Cloud**: Use HashiCorp's SaaS solution for collaboration and remote state management.
4. **Custom Providers**: Develop custom providers to extend Terraform's functionality.
5. **Data Sources**: Use data sources to fetch information about existing resources.

11 Terraform – Conclusion - Zusammenfassung

Terraform provides a powerful and flexible framework for managing infrastructure as code. By enabling automation, consistency, and scalability, it has become an essential tool for modern DevOps practices. Mastering Terraform involves hands-on practice, exploring advanced features, and leveraging its robust ecosystem of providers and modules.

See Terraform Examples from Training Session and the 3 Tier demonstration application:



12 Architektur Grundedanken und weitere Konzepte

12.1 Micro Service Architecture (MSA)

Micro Services (siehe auch Fowler) stellen gerade für Java und Java EE Anwender eine Antwort zur Verfügung um monolithische EAR/WAR Anwendungen in mehrere bzw. viele kleine Deployment Einheiten zu zerteilen und damit besonders für agile Teams die Entwicklung, den Test und das Deployment tendenziell einfacher machen.

Neben den vielen Vorteilen der einfachen kleingranularen Entwicklung wird für eine MS Architektur auch ein entsprechend professioneller Container, Netzwerk, Monitoring und Logging Konzept und Umgebung benötigt.

Weiterführende Literatur:

- Micro Services
  https://de.wikipedia.org/wiki/Microservices
- Domain Driven Design (Eric Evans 2003)
  https://www.goodreads.com/book/show/179133.Domain_Driven_Design

12.1.1 Domain Services and DOA/DDD

Die Aufteilung der benötigten Micro Services ist ein zentrales Erfolgs-Element für MS-Architekturen. Besonders DDD Design Ansätze verbessern die Strukturierung und sinnvolle Aufteilung in MS.

Insofern erlebte DDD (Domain Driven Design, vergleiche Evens 2003) mit MS ein wichtiges Revival, um ein Gesamtumfang einer Anwendung in definierte Teile zu zerlegen und für eine Microservice Architektur Umsetzung zu definieren.

Insbesondere die am Anfang der Micro Service Entwicklung vorhandene zu starke Mini Granularisierung kann und wird durch DDD in auch fachlich sinnvolle Elemente strukturiert, die oftmals wieder etwas umfangreicher sind als ohne diese fachliche Basis.

12.1.1.1 Bounded Context & Ubiquitous Language

DDD versucht dabei die fachliche Domäne in abgegrenzte Teilgebiete BC „Bounded Context“ zu zerlegen. Innerhalb dieser BC werden Geschäftsobjekte und Prozesse mit definierter Namensgebung versehen und so eine sogenannte „Ubiquitous Language“ der Anwendung erzeugt.

Die Vorgangsweise geht aber eher buttom-up vor indem zuerst die Domänen Objekte und deren fachlich grundlegenden Beziehungen definiert werden, die danach in BC gruppiert werden.

12.1.2 CQRS (Command Query Responsibility Segregation)

Cloud und Microservice Architekturen korrelieren oft nicht direkt mit den klassischen eher horizontalen Schichten Architekturen (siehe (Wikipedia) sondern können allein schon aufgrund ihrer Anforderungen andere Lösungskonzepte umsetzten.

Ein sehr wichtiges Konzept dabei ist die Trennung von Command/Befehls- und der Query/Abfrageseite. Dadurch können flexiblere vor allem auch im Massen-Daten Lesebereich notwendige Konzepte umgesetzt werden.

Weiterführende Literatur

1. https://martinfowler.com/bliki/CQRS.html

12.1.3 Event Sourcing (ES) and Event Driven

Ein weiteres wichtiges Umsetzungs- und Architektur Konzept für verteilte Cloud Anwendungen sind Event orientierte Lösungsansätze.

Während Event Driven Ansätze Nachrichten/Events meist als asynchrone Nachrichten zur grundlegenden Entkopplung von (Micro)-Services benutzen, steht im Event Sourcing Konzept eine viel umfangreichere Konzeption dahinter.

Event Sourcing (Siehe auch https://martinfowler.com/eaaDev/EventSourcing.html) definiert hingegen ein Lösungskonzept, bei dem alle Events die zu einer bestimmten Aufgabe einlangen in einem Event-Store (Datenbank) zeitlich geordnet gespeichert werden, die bei Abarbeitung aller Events schlussendlich einen konsistentes Ergebnis liefern. Dabei aber auch direkt ein Audit Log bzw eine zeitliche Historie bereitstellen wie es genau zu diesem Ergebnis gekommen ist .

ES speichert also nicht nur den zuletzt aktuellen finalen Zustand, sondern speichert die Einzelschritte ab, die über den Zeitablauf zum eigentlich aktuellen Datenzustand geführt haben.

Ein Beispiel dafür wäre (ein Klassiker) eines Kontos dessen Event Reihenfolge so aussehen könnte:

2. Tag 1 12:00:00 eröffnet mit Euro 0
3. Tag 2 12:00:00 Einlage 10 Euro
4. Tag 2 12:10:00 Einlage 100 Euro
5. Tag 3 12:10:00 Abhebung 60 Euro

Der aktuelle Kontostand wird durch einen Scan (Replay) der Events ermittelt und beträgt am Tag 3 damit 50 Euro. Das Zustandekommen der Summe ist aber jederzeit leicht zu jedem Zeitpunkt zu ermitteln.

Natürlich ist der Replay aufwändig und wird mit Methoden wie Snapshots teilweise behoben. Snapshots speichern dabei Zwischenstände ab, die damit ein zu langes Replay vermeiden und die Performance entsprechend weiterhin garantieren.

Eine weitere Herausforderung dabei sind auch zeitliche Änderungen der Event Strukturen also damit auch eine Versionierung.

Ein wichtiges Konzept dabei ist auch der Begriff **Aggregate**. Das Aggregate umfasst alle notwendigen Informationen, die für dieses Event notwendig sind.

Wird z.b. via CQRS ein Read Model für Event-Sourcing Lösungen bereitgestellt, so werden diese Read Modelle als **Projections** bezeichnet.

12.1.4 Design und Architektur Grundüberlegungen

Cloud Computing eröffnet unendlich viele neue Möglichkeiten, und daher ist eine entsprechende Konzeption umso wichtiger.

12.2 Grundüberlegungen

Wichtige Grundüberlegungen dazu sind:

1. Das Umfeld:
   OnPremise Inhouse Cloud, Public Cloud oder ein hybrides Setup
2. Die Anwendung:
   Wie sieht meine Anwendung aus, bzw wird sie aussehen
3. Anwendungs-Landschaft:
   a. Wie viele Anwendungen entwickle/betreibe ich
   b. Multi-Mandanten Fähigkeit
4. Das Team und das Know-How

12.2.1 Das Umfeld

Vor allem die technischen Möglichkeiten hinsichtlich Netzwerks sind meist in In-House Cloud Lösungen nicht so ausgeprägt wie in den gängigen Public Cloud Lösungen.

Insofern tendieren solche Lösungen eher dazu mehr oder möglichst alle Bestandteile einer Anwendung direkt im K8s-Cluster zu betreiben, wogegen Public Cloud Lösungen oftmals leichter einige Elemente nicht direkt im z.B. K8s-Cluster betreiben, wie zum Beispiel die Backend Datenbank/en, Content- und Storage (Accelerator) Lösungen.

12.2.2 Die Anwendung

Nicht jede Anwendung ist als maximal verteilte Micro-Service Anwendung die beste Wahl. Kleinere Anwendungen können durchaus mit Apache, Java, NodeJS, Skript und einer lokalen oder zentralen DB einen einfach zu verwaltenden Stack ergeben, der mit Docker-Compose auf einer AWS Ansible/Terraform manged Platform aufgehoben ist.

12.2.3 Die Anwendungs-Landschaft

Werden viele Anwendungen benötigt, sollte über die einzelne Anwendung hinaus ein Plattform Konzept erstellt werden, dass die Anforderungen aller erstellten Applikationen des Unternehmens abbilden

12.2.4 Das Team und das Know-How

Einer der wichtigsten Faktoren ist das Team und die Akzeptanz der jeweiligen Lösung. Viele Wege führen zum Ziel und die gemeinsame Vision auch die technische Vision ist oft wichtiger als die konkret gewählte Technologie.

12.3 Server Placement Varianten

Klassische Varianten der Applikationsverteilung anhand der Server Verteilung.

- Variante 1: One Big-Server:
  Ein großer Server mit vielen Anwendungen die sich eine Betriebssystem Instanz teilen
- Variante 2: Virtuelle-Server:
  Mehrere (kleinere) virtuelle Server die jeweils pro Server nur eine Anwendung bereitstellen
- Variante 3: Ein großer Server der mehrere (Docker) Container betreibt
- Variante 4: Mehrere virtuelle Server mit mehreren Docker Containern
- Variante 5: Anwendung in Kubernetes Cluster (k8s)
- Variante 6: Serverless Container Architecture

Die Variante 1 wird so kaum mehr verwendet, da die Applikationstrennung doch eher problematisch ist, und aus dieser Variante sich die Lösungen mit mehreren Virtuellen-Servern bzw auch die Container zur besseren Kapselung entwickelt haben.

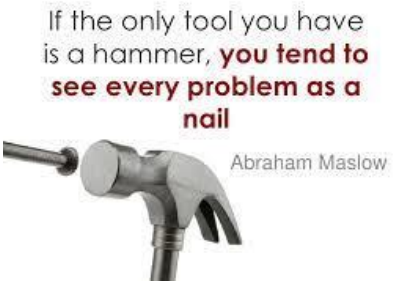
Gerade aber auch Variante 2 ist eine einfache Möglichkeit eine skalierbare vollständig automatisierte Anwendung zu entwerfen.

Z.b. kann ein AWS ELB eine Auto-Scaling Gruppe mit jeweils einem kleinen t2.micro Instanz auf sehr leicht und ziemlich beliebig auch Anwendungslast skalieren und vor allem wieder de-provisionieren, so dass ohne Last kaum Basiskosten entstehen, und nicht mal ein K8s Cluster laufen muss.
Dabei könnte die Anwendung als Spring Boot JAR File via Ansible auf ein Standard VM-Image deployed werden und dieses Image wird mittels ELB beliebig skaliert.

Setzt aber das Team bzw das Unternehmen stark auf K8S bzw auf eine entsprechende K8s Plattform wie OpenShift dann sollte man auch einfachere Use-Cases in K8s abbilden, um die Anzahl der Lösungskonzepte klein zu halten.

12.4 Allgemeine Architektur Richtlinien

12.5 Not only a Hammer

Abraham Maslow Dieser Satz von Maslow wurde in vielen Abwandlungen auch für Software und Architektur verwendet. Es ist wichtig seinen Baukasten entsprechend zu Pflegen und nicht nur auf das gerade hippe neue Tool zu schauen. Verschiedene Probleme verlangen verschiedene Lösungen. Umgekehrt sollte aber die Toolbox nicht zu umfangreich sein, sondern im Kontext der Probleme und der Teamgröße gut verwaltbar.	 *Abbildung 5 Abraham Maslow*

12.6 The First Law of Distributed Objects

Martin Fowlers (https://martinfowler.com/bliki/FirstLaw.html) Regel sollte in jedem Fall immer Berücksichtigung finden.

Verteilte Systeme wie auch Micro-Services bieten einen hohen Grad an Flexibilität während der Entwicklung und des Einzeltests der Services. Jedoch muss die Gesamtanwendung auch einfach, wartbar und überschaubar bleiben.

In diesem Zusammenhang sein nochmals auf das Kapitel DDD hingewiesen, dass hier für die Komponentisierung und damit den Erfolg der Anwendung sehr wichtig ist. Siehe Domain Services and DOA/DDD

12.7 KISS (keep it simple and stupid)

Auch ein Klassiker, der insbesondere auch in (neuen) Cloud Umgebungen Berücksichtigung finden sollte. Es ist sehr verlockend immer neue technsiche Services und Features zu integrieren, aber man sollte hier wieder auf jene Elemente zurückgreifen die den unter Punkt „Not only a Hammer“ erwähnten sinnvollen Baukasten ergeben.

13 Appendix

13.1 Abbildungen

13.2 Jenkinsfile Beispiel

```
pipeline {
    agent any

    tools {
        maven 'maven_3.6.3'
        jdk 'jdk_1.8.0'
    }

    stages {
        stage('Introduction and Pre Checks') {
            steps {
                echo 'This is a minimal pipeline.'
                script {
                    sh 'df -h'
                    sh 'java -version'
                }
            }
        }
        stage('build') {
            steps {
                echo 'This is the build step'
                sh 'mvn -B -DskipTests clean package'
            }
        }
        stage('Build and Unit Test') {
            steps {
                echo "Build and Unit Test"
                sh "mvn -B -nsu clean install"
            }
            post {
                always {
                    junit "**/surefire-reports/*.xml"
                }
            }
        }
    }
    post {
        failure {
            echo "Build POST  FAILURE action "
        }
        always {
            echo "Build POST action"
        }
    }
}
```

13.3 Dev Ops Tool-Chain

Eine beispielhafte Darstellung einer Tool-Chain für eine klassische Java Anwendung kann so aussehen:

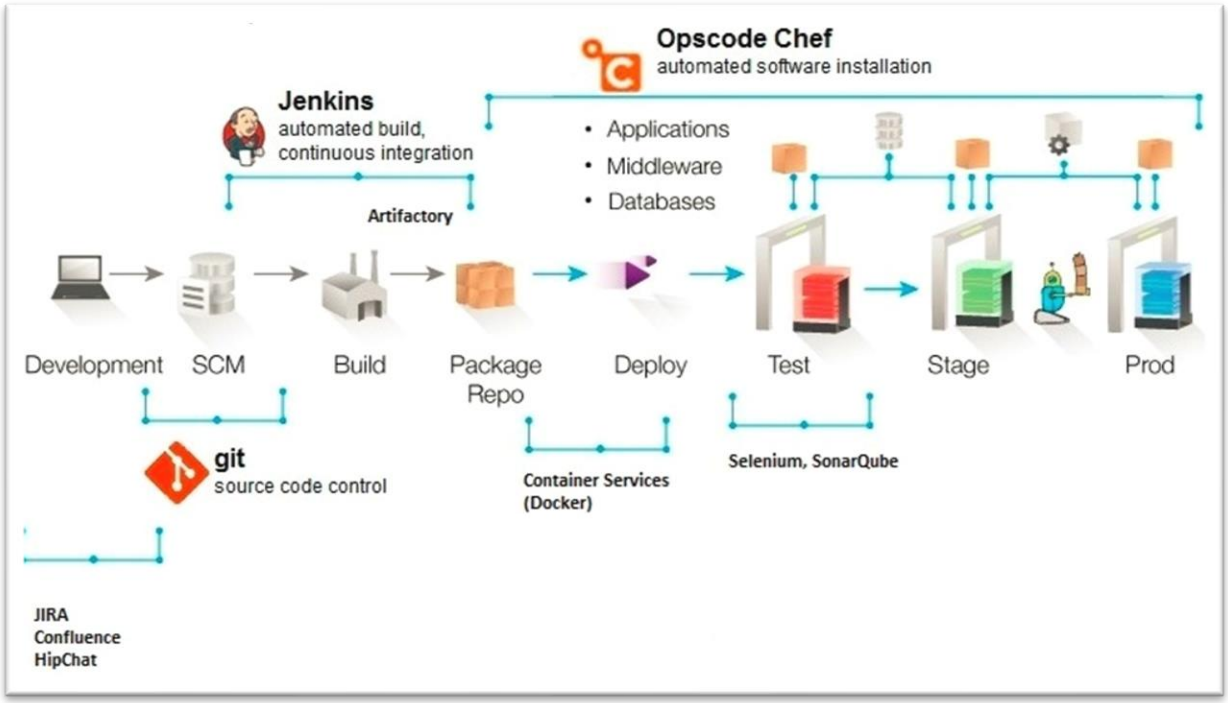


Abbildung 6 Dev Ops – Toolchain

13.4 GIT Branching Beispiele mit klassischem Ansatz

Beispiel für eine klassische Verwendung von GIT Branches für Feature und Release Entwicklung mit einem Default Master Branch.

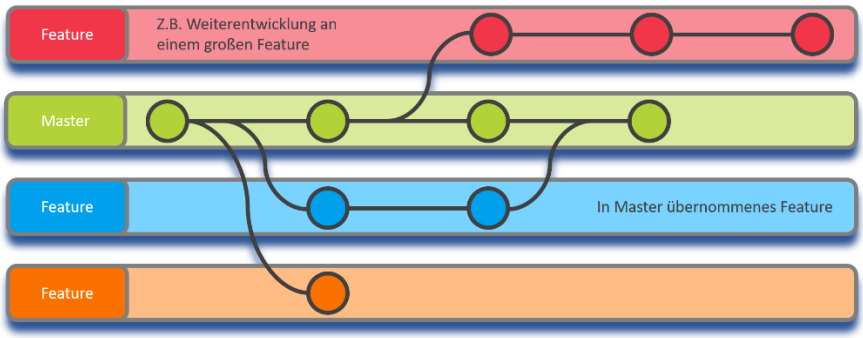


Abbildung 7 GIT Feature Branches

Beispiel 2 :

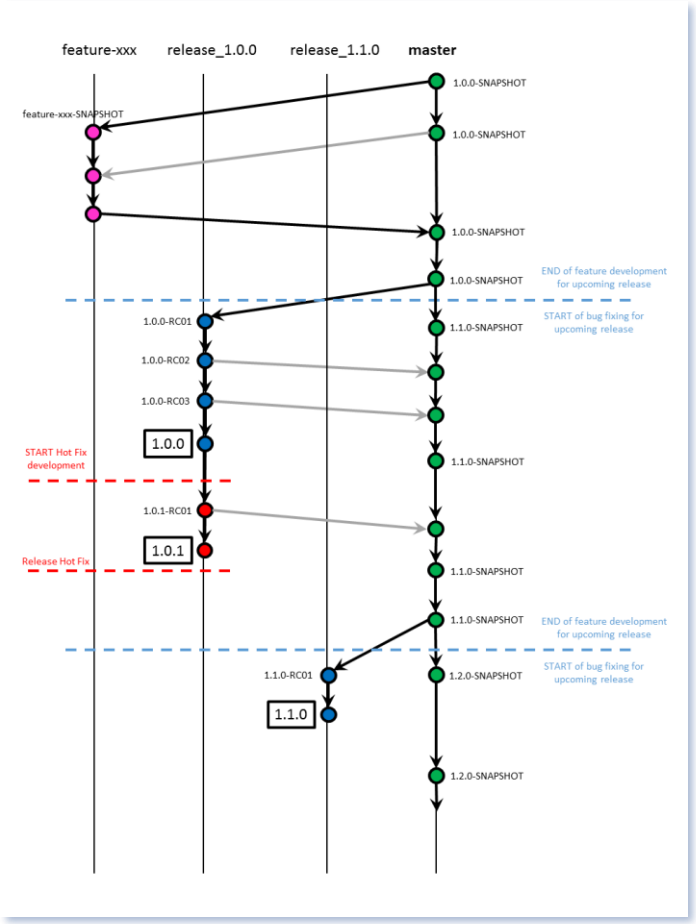


Figure 1 GIT Branching Modell

13.5 Apache Maven – Build und Dependency Management System

Apache Maven ist ein Tool/Framework, dass die Abhängigkeiten für Java Anwendungen verwaltet und auch den Build, Test und Paketierungsprozess unterstützt bzw definiert:

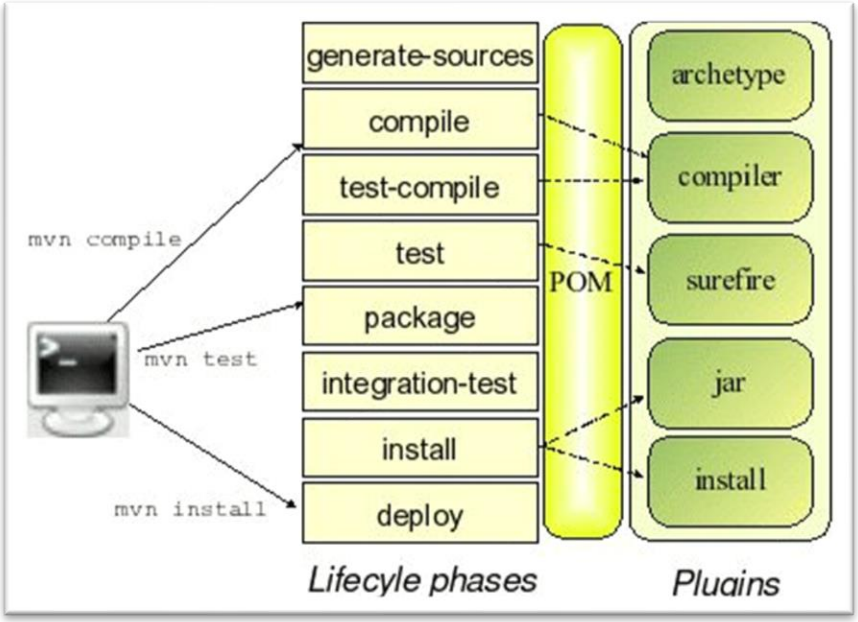
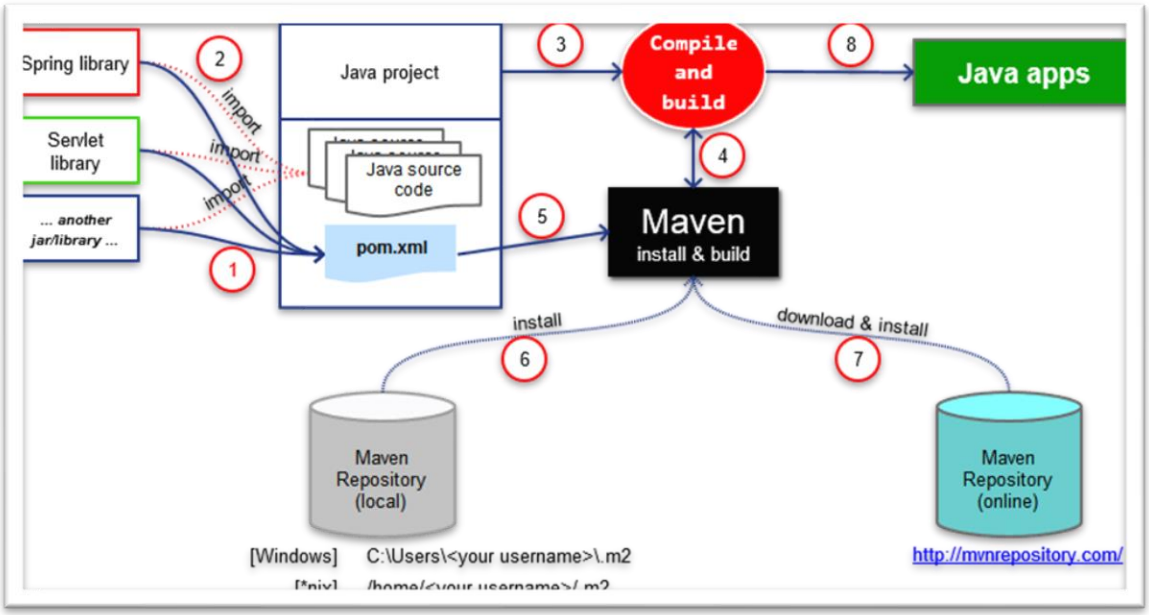


Abbildung 8 Apache Maven Ablauf

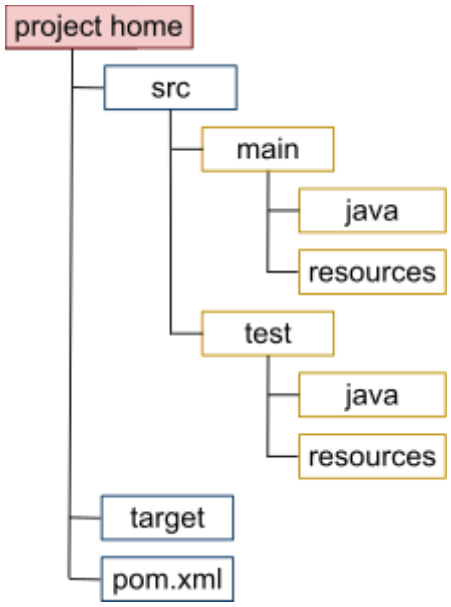
13.5.1 Apache Maven – Dependency Auflösung und Download

Die im pom.xml angegebenen Abhängikeiten werden beim Compile und Test jeweils von sogenannten Maven Repositories geladen (Download via http aus zentralen Repositories).



13.5.2 Apache Maven – Directory Struktur und Beispiel Projekt

Ein Maven Projekt hat für die Grundstruktur der Module eine Vorgabe an die man sich im allgemeinen hält:



Im Beispiel einer Spring Demo Anwendung:

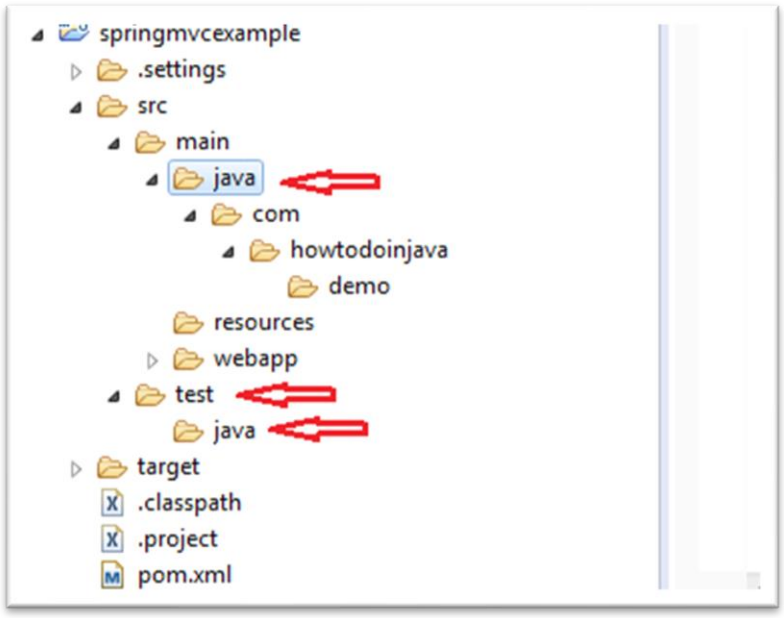
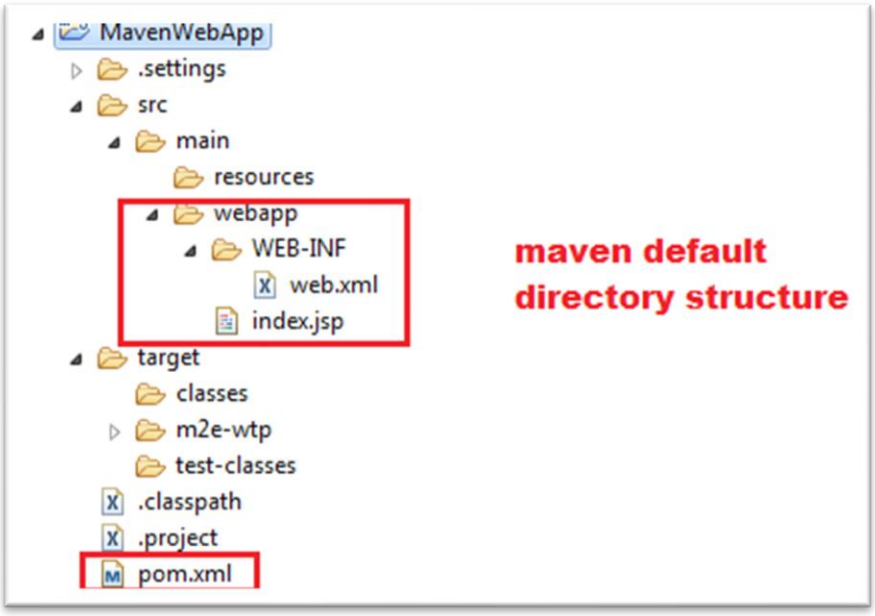


Abbildung 9 Apache Maven Modul und Verzeichnisse

Dadurch wird eine Art Industriestandard erzeugt und eine klare Trennung der Java Klassen, von benötigten Ressourcen und Testklassen erreicht.

13.5.3 Web Applikationen in Apache Maven

Der Unterschied zu normalen Maven Modulen und Maven Web-Applikationen ist, dass das Packaging im pom.xml auf WAR gestellt ist, und die Verzeichnisse für die Webapp auch definiert sind.



13.6

13.6.1 Apache Maven Settings Konfiguration

Wichtig für maven ist die Konfiguration der lokalen Settings.xml im maven/config directory.

Bitte nicht vergessen in der Entwicklungsumgebung die entsprechende Settings Datei auch zu konfigurieren.

Siehe dazu settings.xml aus dem Kurs Beispiel im ZIP File

Weiters auch die SCM und Maven Settings im parent pom.xml des Beispiels, insbesondere für die Maven Release werden sowohl im GIT (für den Release TAG) als auch im Nexus zum Upload der Software die Zugänge benötigt.

13.7 Das Logging System

13.7.1 Grundlagen

Logging stellt eine Kommunikation zwischen der Entwicklung und dem Betrieb zur Verfügung indem Log-Nachrichten in ein oder mehrere dafür bestimmte Files oder Systeme geschrieben werden, um Informationen zur Anwendung zur Verfügung zu stellen.

Diese Informationen haben unterschiedliche Aufgaben und Empfänger.

Die grundlegende Schnittstelle zwischen dem Logging System und dem Source-code bildet ein Logging API. Mit diesem API kann man Log Meldungen/Messages/Nachrichten schreiben:

```
logger.info("id erhalten: {}", id );
```

Diese Info kann sich direkt im Quellcode befinden, wodurch sehr individuelle Nachrichten ge-logged werden können, oder aber auch in nicht direkt sichtbaren CDI-Interceptoren oder Spring.

13.8 Logging Framworks und API:

Es haben sich im Laufe der Zeit einige Frameworks entwickelt die als Logging Frameworks verwendet werden:

1. Java JDK standard Logging API
2. Log4j und Log4j2
3. Logback (http://logback.qos.ch/)
   (Logback is intended as a successor to log4j project)
4. SLF4J –Logging Facade for Java

SLF4J hat sich als allgemeines API für Logging stark durchgesetzt. Logback als Nachfolger von Log4J.

13.9 SLF4J

Auch die direkte Benutzung des Java Logging Apis kann via jul-to-slf4j entsprechend einfach in das slf4j System umgeleitet werden:

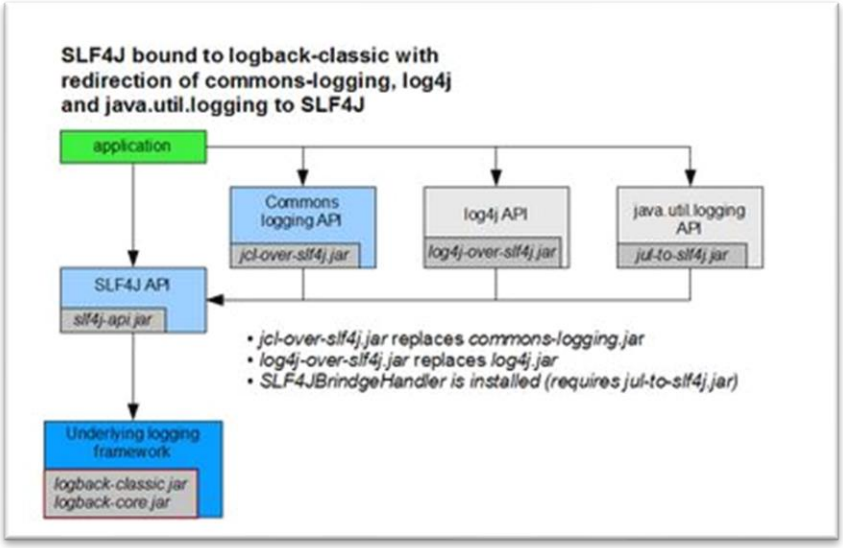


Abbildung 10 Redirecting Java API to SLF4j

13.10 Log Level & Output Appender

13.10.1 Log Levels

Stellen den wichtigsten Grundmechanismus zur Verfügung, um die Wichtigkeiten der Log Nachrichten direkt beim Aufruf des Apis darzustellen:

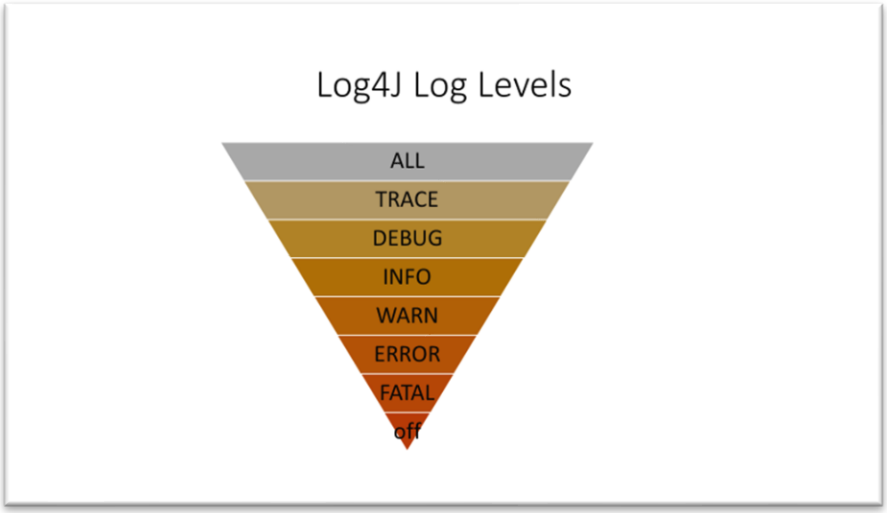


Abbildung 11 Log4J Log levels

Wichtig dabei ist es ein klares Gesamtkonzept zu haben. Das bedeutet es muss klar definiert werden welche Information auf welchem Level gelogged werden soll und welche nicht.

Zuviel Informationen auf INFO Level führen dazu, dass die Anwendung tendenziell Performance Probleme bekommen kann, aber vor allem auch dazu, dass zu viel Information auch sehr verwirrend sein kann.

Informationen für detaillierte Informationen, die nach dem Auftreten eines Fehlers zur Fehlersuche benötigt werden üblicherweise als DEBUG geloggt.

13.10.2 Log Appender

Appender stellen die Schnittstelle zum Output System bzw Format dar. Appender können darüber hinaus auch entscheiden welche Log Level sie noch „durchlassen“ bzw. ausfiltern.

Das ist immer wieder in Problem da damit Meldungen sehr leicht verloren gehen können.

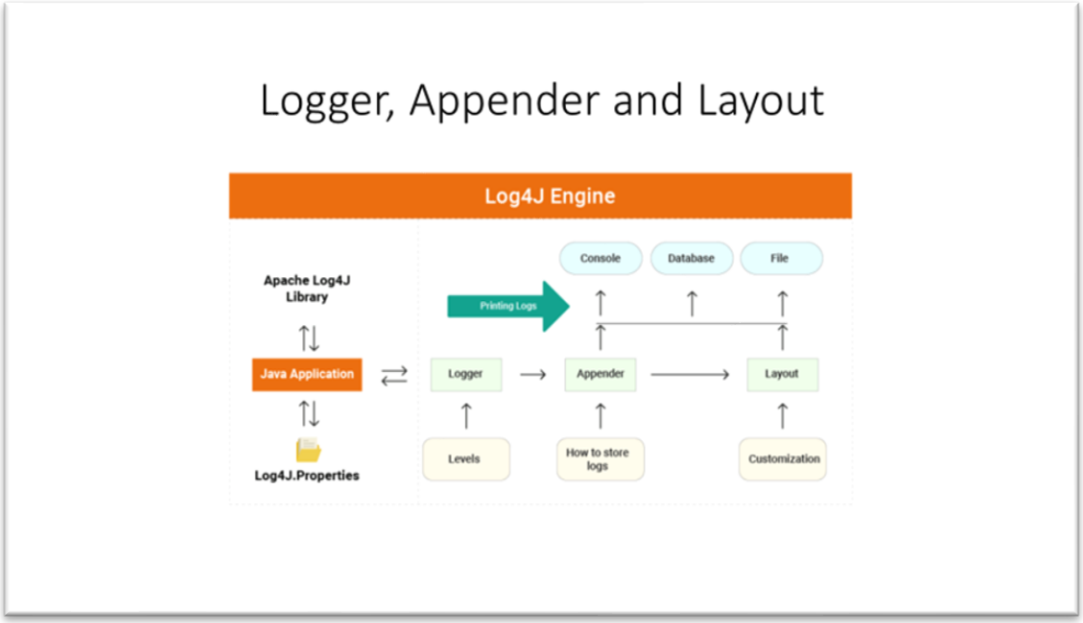


Abbildung 12 Log Appender

13.11 Logger Hierarchie

Über Logger können in der jeweiligen Konfiguration auch sehr einfach für bestimmte Java Pakete und die darunter liegenden Klassen konfiguriert werden, und somit bestimmte Output Filterungen vorgenommen werden:

Hier das Beispiel aus der Simple-Chat Anwendung:

```
<root level="debug">
    <appender-ref ref="RollingFile" />
    <appender-ref ref="Console" />
</root>

<Logger name="org.hibernate" level="info" >
</Logger>

<Logger name="at.cgsit.training" level="trace">
</Logger>
```

13.13 Begriffe und Grundlagen und Links

14 Literaturverzeichnis

t2informatik. *t2informatik*. Conways Law. 11 2021. <https://t2informatik.de/wissen-kompakt/gesetz-von-conway/>.

„Wikipedia.“ kein Datum. <https://de.wikipedia.org/wiki/DevOps>.

Wikipedia. kein Datum. <https://de.wikipedia.org/wiki/Schichtenarchitektur>.

14.1 Architektur und Konzeption

- Cloud Strategy: A Decision-based Approach to Successful Cloud Migration
  Gregor Hohpe
  https://www.goodreads.com/book/show/58037772-cloud-strategy

- Solutions Architect's Handbook: Kick-start your solutions architect career by learning architecture design principles and strategies
  https://www.goodreads.com/book/show/53600892-solutions-architect-s-handbook

- Get Your Hands Dirty on Clean Architecture:by Tom Hombergs, published by Packt Publishing Ltd. (September 2019)
  https://www.goodreads.com/book/show/52774354-get-your-hands-dirty-on-clean-architecture

14.2 Cloud Computing, AWS

- Günstiger Überblick:
  AWS: AMAZON WEB SERVICES: The Complete Guide From Beginners For Amazon Web Services https://www.goodreads.com/book/show/49382225-aws

- Microservices on AWS (AWS Whitepaper), AWS Whitepapers, September 2017
  https://www.goodreads.com/book/show/36266267-microservices-on-aws

- Overview of Amazon Web Services, AWS Whitepapers , Kindle
  https://www.amazon.com/-/de/dp/B09C2VLVPF

14.3 Kubernetes, K8s

- Kubernetes: Eine kompakte Einführung
  https://www.goodreads.com/book/show/38335102-kubernetes

- The Kubernetes Book: Version 2.2 - January 2018
  https://www.goodreads.com/book/show/35494978-the-kubernetes-book

- Kubernetes in Action – 2018
  https://www.goodreads.com/book/show/34013922-kubernetes-in-action

- https://www.goodreads.com/book/show/51931292-aws-security-cookbook

14.4 Dev-Ops

- https://en.wikipedia.org/wiki/DevOps_toolchain

- The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations
  https://www.goodreads.com/book/show/26083308-the-devops-handbook

14.4.1 Jenkins

The leading open-source automation server, Jenkins provides hundreds of plugins to support building, deploying and automating any project.
Siehe https://www.jenkins.io/

14.4.2 Apache Maven

Maven ist ein auf Java basierendes Build-Management-Tool der Apache Software Foundation, mit dem insbesondere die Erstellung von Java-Programmen standardisiert verwaltet und durchgeführt werden kann.

Siehe: https://maven.apache.org/

14.4.3 GIT

Git ist eine freie Software zur verteilten Versionsverwaltung von Dateien, die durch Linus Torvalds initiiert wurde.

Siehe: https://de.wikipedia.org/wiki/Git