

# JEE MICROSERVICE DEVELOPMENT

Custom Training 2023

CGS – IT-Solutions GmbH @ 2020/2023  
Version 1.0.11

Mag. Christian Schaefer  
[cgs@cgs.at](mailto:cgs@cgs.at)

## 1 Inhalt

2 Voraussetzungen und Rahmenbedingungen.....	7
3 Einleitung und Kursüberblick.....	7
3.1 Ziele .....	7
4 Architektur Überblick und Grundlagen .....	8
4.1 Architekturen Überblick - Paketierung .....	8
4.2 (Definition) Micro Service Architecture (MSA).....	8
4.3 Laufzeit versus Entwicklungs-Komplexität.....	9
4.3.1 Verteilte Architektur und Internet-Basis.....	9
4.4 Aufbauende Grundlagen Architektur Standards .....	9
4.5 Standards im Java Umfeld .....	10
4.6 Microservice Profile and Jarkate EE Core Profile.....	11
4.7 Microservice Architektur Muster .....	12
4.7.1 Inversion of Control (IoC).....	12
4.7.2 CDI – Context and Dependency Injection.....	12
4.7.3 Domain Services and DOA/DDD .....	13
4.7.4 CQRS (Command Query Responsibility Segregation) .....	13
4.7.5 Event Sourcing (ES) and Event Driven.....	13
4.8 MS API Architektur Patterns.....	14
4.9 MS Architektur Verteilung .....	15
4.9.1 MS Laufzeit Umgebung.....	15
4.9.2 Micro Service Management (Kubernetes).....	16
4.10 Kapitel Zusammenfassung: Microservice Architektur .....	16
4.10.1 Weiterführende Literatur .....	16
5 Quarkus .....	17
5.1 Quarkus Extension Architektur.....	17
5.1.1 Quarkus Extensions .....	17
5.1.2 Graal VM.....	18
5.2 Quarkus Spring DI Kompatibilität .....	18
5.3 Quarkus Konfiguratton.....	18
5.4 Quarkus DEV UI .....	19
5.5 Kapitel Zusammenfassung: Quarkus Grundlagen.....	19
5.5.1 Weiterführende Literatur .....	19
6 Quarkus – Configuration Management .....	20
6.1 Quarks Configuration Injection .....	20

6.1.1	System Properties.....	21
6.1.2	Quarkus Property Expressions und Umgebungen.....	21
6.2	Quarkus Konfiguration in der Cloud/K8s.....	22
7	Quarkus CDI – Context and Dependency Injection .....	23
7.1	CDI Beans Definition and Injection.....	23
7.2	CDI Bean Dependency Resolution .....	24
7.3	CDI Annotations.....	24
7.4	CDI Beans - Live Cycle Annotations .....	24
7.5	CDI Bean Discovery.....	25
7.6	Injection Grundlagen.....	25
7.6.1	Property Injection.....	25
7.6.2	Setter Injection .....	25
7.6.3	Constructor Injection.....	26
7.6.4	Default Injection .....	26
7.7	Injection mit Interfaces, Default und Alternativen.....	27
7.7.1	Qualifier und Injections .....	27
7.7.2	Qualifier Error.....	27
7.8	Alternative Beans .....	28
7.9	CDI Producer.....	29
7.10	CDI Lifecycle Callbacks.....	29
7.11	CDI Priorities.....	30
7.12	CDI Interceptoren .....	30
7.13	CDI Scopes .....	31
7.14	Application Scoped Bean.....	32
7.15	CDI Events.....	33
7.15.1	CDI Startup Events .....	34
7.16	Quarkus Profiles .....	34
8	Quarkus –Security.....	35
8.1	ORM Panachache Vereinfachung der Persistenz Operationen.....	35
8.2	Quarkus – JaxRS Security Annotaitons .....	35
8.2.1	Quarkus User Roles Allowed.....	35
8.2.2	Quarkus JPA Security Extension und Panache .....	36
8.2.3	Quarkus Beispiel Initialisierung der user .....	37
8.2.4	Quarkus Security Swagger Support .....	38
8.2.5	QuarkusTest Support für Security.....	38

8.3	JaxRS – Rest Services .....	39
8.3.1	Rest Service - Architecture .....	39
8.3.2	Rest Service – HTTP Methods.....	41
8.3.3	Example Restful Application .....	41
8.3.4	JaxRS Restservices .....	42
8.3.5	Java JaxRS – Annotations.....	43
8.3.6	Method and Path Annotations .....	43
8.3.7	@Produces, @Consumes Content Negotiation.....	45
8.3.8	JSON Output/Input Support via Jackson .....	46
8.3.9	JaxRS Exception handling.....	48
9	Open API Spezifikation – und API-Dokumentation.....	48
9.1	OpenAPI maven Dependency.....	49
9.2	OpenAPI Header Section .....	49
9.2.1	Open API Header Beispiel.....	50
9.3	Open API Datatypes.....	52
9.4	OpenAPI in Quarkus .....	52
9.5	OpenAPI Test DTO Pfad und Dokument Beschreibung.....	53
9.5.1	OpenAPI Path Documentation.....	54
9.5.2	OpenAPI DTO Beschreibung .....	54
9.5.3	OpenAPI OperationID .....	55
9.5.4	Open API Query String Parameter.....	57
9.5.5	Open API Query Path Parameter .....	58
10	OpenAPI – Api First (Generate Server Sources).....	59
10.1	OpenAPI – API First – Maven Konfiguration .....	59
10.1.1	OpenAPI – maven target directories konfigurieren.....	60
10.2	Generierte Struktur mit IntelliJ Source Foldern .....	60
10.3	Generierter Java JaxRS Quellcode .....	61
10.4	OpenAPI – API First Limitierungen .....	62
11	Swagger UI.....	63
11.1	Testing APIs.....	63
12	Quarkus Metrics .....	64
12.1	Metrics Quarkus Maven Extensions .....	64
12.2	Quarkus Metrics Example.....	64
13	Quarkus JMS .....	65
13.1	JMS Example.....	65

14	JPA – Java Persistence API – Database Development .....	67
14.1	JPA Introduction.....	67
14.2	JPA Framework Components.....	68
14.3	The JPA API .....	68
14.4	Modules and APIS used with JPA.....	68
14.5	The JPA Entity Manager .....	69
14.5.1	The Entity Manager API.....	69
14.6	JPA – JTA – Transactions.....	69
14.6.1	JTA – Different Transaction Scopes .....	69
14.7	JPA – JTA –Programmatic TX-Management .....	70
14.8	JPA Entities.....	70
14.9	Entity Annotations.....	71
14.10	Entity Column Annotations .....	71
14.11	Entity Mapping Example.....	71
14.12	JPA Entity Relationship Mappings.....	72
14.12.1	Entity Many to Many Example.....	72
14.13	JPA Query Language .....	73
14.14	JPA - JPQL - Query API.....	73
14.15	Quarks JPA Configuration .....	74
14.16	Chapter Summary: JPA .....	74
14.16.1	Additional Material.....	74
15	Quarkus – Testing .....	75
15.1	Quarkus Test Dependencies .....	75
15.2	Quarkus Test Example.....	75
15.2.1	Quarkus Fluent API Rest Assured Test .....	76
15.3	Rest Assured Tests – Basic Introduction .....	76
15.4	Quarkus Testing Concepts with Object DTO Usage .....	77
15.5	Quarkus Testing Security .....	78
16	Quarkus – Bean Validation .....	78
16.1	Bean Validation Annotations .....	78
16.2	Bean Validation Example .....	79
16.3	Manual/Programmatic Validator Usage .....	80
16.4	Cascading Validation.....	81
16.4.1	Fail Fast and further Configuration.....	81
16.5	Chapter Summary: JPA .....	81

16.5.1	Additional Material.....	81
17	Quarkus – Service 2 Service Communication.....	82
17.1	Quarkus .....	82
18	Postman Testing.....	83
18.1	Postman Variablen.....	83
18.2	Postman Response Tests.....	84
18.3	Postman OpenAPI – Editor .....	85
18.4	Postman weiterführende Literatur .....	85
19	IBM Open Liberty.....	86
19.1	Create Initial Project for Open Liberty.....	86
19.2	Open Liberty – Server Startup.....	86
19.3	Open Liberty Swagger Console.....	87
19.4	Open Liberty - Server Configuration .....	88
20	Appendix.....	89
20.1	Abbildungen .....	89
20.2	Source Code Demos .....	90
20.2.1	JPA Implementation for JPQL Query.....	90
20.2.2	JPA Implementation for Criteria Builder:.....	90
20.2.3	Injection Demo Bean .....	91
20.2.4	Appendix: OpenAPI Yaml für Test DTO .....	91
20.3	Exkurs : DevOps .....	94
20.3.1	Dev Ops Pipeline.....	95
20.4	Exkurs : Cloud Computing.....	96
20.5	GIT Branching Beispiele mit klassischem Ansatz.....	97
20.6	Apache Maven – Build und Dependency Management System .....	98
20.6.1	Apache Maven – Dependency Auflösung und Download .....	98
20.6.2	Apache Maven – Directory Struktur und Beispiel Projekt .....	99
20.6.3	Web-Applikationen in Apache Maven .....	101
20.7	.....	101
20.7.1	Apache Maven Settings Konfiguration .....	101
20.8	Das Logging System .....	102
20.8.1	Grundlagen .....	102
20.9	Logging Frameworks und API: .....	102
20.10	SLF4J .....	103
20.11	Log Level & Output Appender .....	103

20.11.1	Log Levels.....	103
20.11.2	Log Appender .....	104
20.12	Logger Hierarchie.....	104
20.13	Begriffe und Grundlagen und Links .....	105
21	Literaturverzeichnis .....	105
21.1	Allgemeine Links.....	105
21.2	Quarkus Dokumentation.....	105
21.3	Rest Services und Open API.....	105
21.4	Java Standards und Spezifikationen .....	106
21.5	Architektur und Konzeption .....	106
21.6	JPA.....	106
21.7	Cloud Computing, AWS .....	107
21.8	Kubernetes, K8s .....	107
21.9	Internet Standards.....	107
21.10	Dev-Ops .....	107
21.10.1	Jenkins .....	108
21.10.2	Apache Maven.....	108
21.10.3	GIT .....	108

## 2 Voraussetzungen und Rahmenbedingungen

Die Voraussetzungen für diesen Kurs für die Teilnehmer sind:

1. Grundlegende bis gute Java Entwicklungs-Kenntnisse
2. Grundlegende Erfahrung mit Apache maven basierter Software-Entwicklung und Paketierung
3. Erfahrung mit der Software-Entwicklung mit IntelliJ IDE
4. Datenbank und SQL Grund-Kenntnisse
5. Grundkenntnisse über Architekturmuster wie HTTP, Rest-Service Spezifikationen, Schicht-Architekturen bzw verteilte Systeme sind von Vorteil.

Da dieser Kurs eine spezifische Zusammenstellung von Themen und Blöcken aus anderen Kursen ist, kann es sein, dass Teile des Inhalts in Deutsch bzw aber auch Englisch formuliert sind.

## 3 Einleitung und Kursüberblick

Dieser Kurs stellt eine Einführung in die Software-Entwicklung für Microservices dar. Die spezifisch für diesen Kurs zusammengestellten Themen beinhalten:

1. Container Management und CDI (Context and Dependency Injection)
2. JAXRS
3. JPA
4. JMS (Überblick)
5. Open Liberty / Quarkus
6. OPENAPI-Spec 3.0
7. Testen der REST APIs mit Postman
8. Bean Validation

### 3.1 Ziele

- Sie haben einen Überblick über den Aufbau von Microservices mittels Quarkus/Open Liberty
- Sie verfügen einen Überblick zur praktischen Anwendung und selbstständiger Vertiefung der vorgestellten Themen.
- Sie können ein einfaches Micro-Service auf Basis von Quarkus selbst entwickeln.

## 4 Architektur Überblick und Grundlagen

### 4.1 Architekturen Überblick - Paketierung

Wie in dieser Architektur Übersicht dargestellt existieren verschiedene Architekturmuster zur Abbildung von Applikationen. Die klassische JEE-Architektur der Java Enterprise Services mit EAR und War-Files implementieren eher Makro-Services bzw werden diese oft auch als Monolith bezeichnet. Dieser Monolith (bzw. auch Unterarten davon wie die Service Domain) werden im Java Bereich oft mittels klassischem Java EE Server wie JBOSS/Wildfly implementiert.

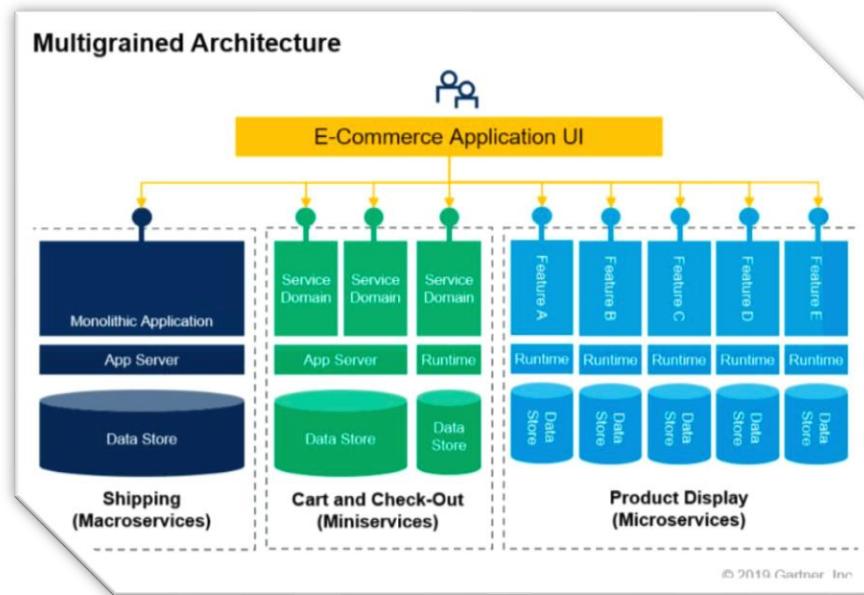


Abbildung 1 Multigrained Architecture (Quelle Gartner)

### 4.2 (Definition) Micro Service Architecture (MSA)

Micro Services (siehe auch [Fowler](#)) stellen 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.

#### 4.3 Laufzeit versus Entwicklungs-Komplexität

Die Architekturen unterscheiden sich vor allem auch in ihrem Vor und Nachteilen.

Während der Monolith in der Infrastruktur tendenziell einfacher zu betreiben ist, stellt er etwas weniger Flexibilität während Entwicklungszeit dar. Insbesondere auch für die agile Entwicklung in kleineren autonomen Teams ist die Macro-Service Paketierung ein Nachteil.

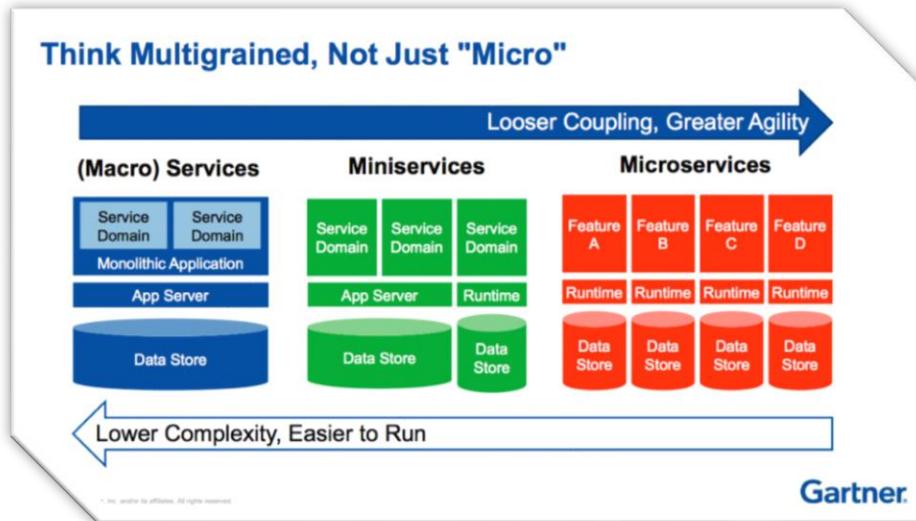


Abbildung 2 Macro vs Micro Services (Quelle Gartner)

##### 4.3.1 Verteilte Architektur und Internet-Basis

MS sind also vor allem auch eine Verteilte Architektur mit allen vor und Nachteilen. Insbesondere stellen verteilte Architekturen jeweils die Performance auf den Prüfstand.

Folgende Features sind daher für eine verteilte Microservice Architektur von höherer Bedeutung:

1. Zentrales Logging
2. Zentrales Funktionalitäts- und Performance- Monitoring
3. Zentrale Konfiguration
4. Softwaredesign für Cloud Umgebungen und skalier fähiger dynamischer Laufzeit Abbildung

#### 4.4 Aufbauende Grundlagen Architektur Standards

Could basierte und MS basierte Architekturen erfordern ein Verständnis der Basis-Architekturen für Internet-Basierte und Verteilte Anwendungen.

Hierzu zählen unter anderem:

1. TCP/IP Internet Standards
2. http/S Standard besonders für die Verarbeitung von Restful Services
3. Weitere Internet RFCs wie MIME Types, DNS-System und Format Grundlagen
4. JSON/XML Schema und Validierung Standards
5. Etc.

#### 4.5 Standards im Java Umfeld

Java als technisches Ökosystem ist weitläufig und nicht immer sofort überblickbar. Oftmals bauen Spezifikationen und deren Lösungen auf allgemeine grundlegende Ideen auf. Wie zum Beispiel Lambda Expression die auch einen Eingang in die JDK fanden. Weiters sind Spezifikationen oftmals auch an allgemeine Internet Standards angelehnt bzw bauen darauf auf, wie zum Beispiel Rest Services.

Im Java und JEE Bereich werden sowohl für die JDK als auch für die JEE (Java Plattform, Enterprise Edition ([Java EE](#)) Spezifikationen über den Java Community Process) ([https://de.wikipedia.org/wiki/Java\\_Community\\_Process](https://de.wikipedia.org/wiki/Java_Community_Process)) durchgeführt.

Wichtig ist zu wissen, dass die Standards wie der CDI-Standard nur eine Spezifikation darstellen die dann durch einen oder mehrere Hersteller zertifiziert implementiert werden/können.

Normalerweise liegt für jede Spezifikation eine Referenzimplementierung vor die die Funktionalität bereits implementiert hat.

CDI 2.0. kompatible Implementierung dieses Standards bieten zum Beispiel

- WildFly
- Glashfish
- Apache TomcatEE
- BM WebSphere Application Server

Neben diesen Enterprise Servern werden auch diese Standards durch die Definition Pakete des MicroProfile Standards implementiert.

## 4.6 Microservice Profile and Jakarta EE Core Profile

Unter dem Begriff "MicroProfile" versteht man im Java Context die Bemühungen und Standards die zur Transformation der JEE-Welt in eine Microservice orientierte Architektur gemacht werden.

Die Informationen zu diesen Standards findet man auf der Homepage <https://microprofile.io>.

Und wird dort folgendermaßen definiert:

*MicroProfile enables Java EE developers to leverage their existing skill set while shifting their focus from traditional 3-tier applications to microservices. MicroProfile APIs establish an optimal foundation for developing microservices-based applications by adopting a subset of the Java EE standards and extending them to address common microservices patterns.*

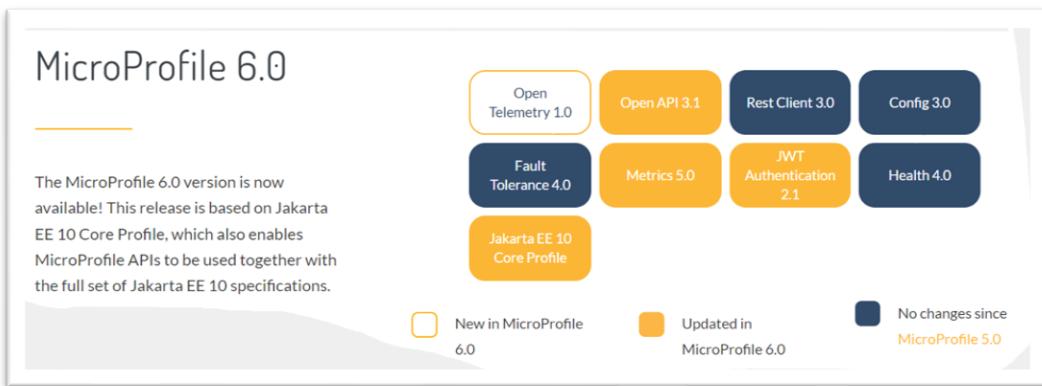


Abbildung 3 Micro Profile Standard Components

Der aktuelle Stand von Micro Profile ist die Version 6.0. mit dem Release Datum

Released Dec 22nd, 2022

<https://github.com/eclipse/microprofile/releases/tag/6.0>

## 4.7 Microservice Architektur Muster

### 4.7.1 Inversion of Control (IoC)

Als wichtiges Architektur-Konzept sowohl in Spring, JEE-Anwendungen als auch Microservices stellt IoC ein Konzept dar, dass die Softwareentwicklung leicht und möglichst nicht invasiv im Kontext der externen Framework Abhängigkeiten gestaltet.

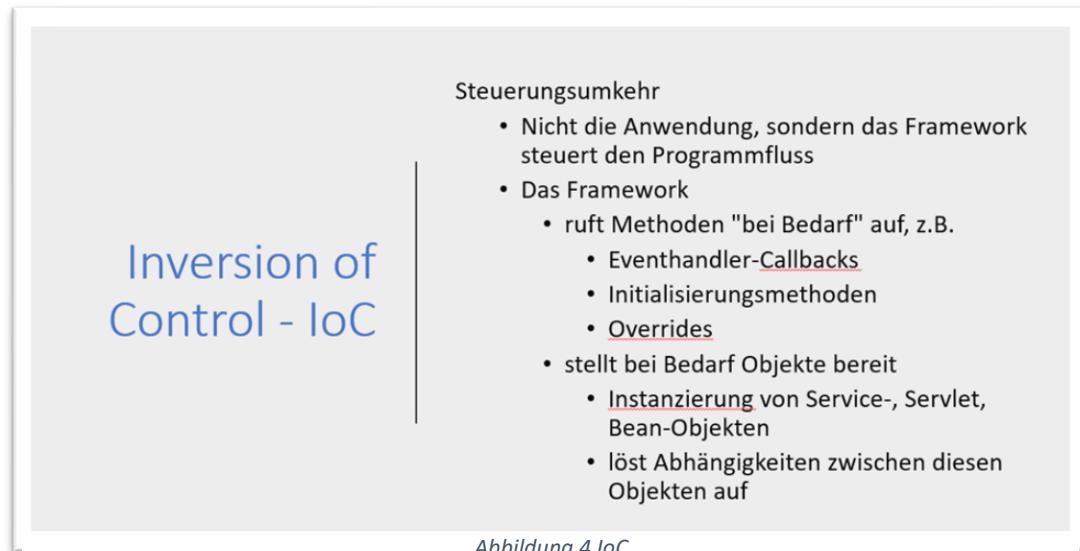


Abbildung 4 IoC

Siehe: Inversion of Control

### 4.7.2 CDI – Context and Dependency Injection

CDI stellt eine IoC Lösung zur Verfügung, die ab Java EE 6 eingeführt wurde. CDI ist neben dem Spring Framework eine Standardlösung.

Folgende Eigenschaften kann man damit in Verbindung bringen:

1. Vermindert Abhängigkeiten zwischen Objekten einer Java EE Anwendung
  - Lose Kopplung
  - Typsicherheit meist über Interfaces
2. Lebenszyklus der Objekte wird vom Container gesteuert
3. Objekte werden vom Container bei Bedarf erzeugt und "injiziert"
4. Verwendbar in allen Komponenten, deren Lebenszyklus vom Container gesteuert wird
  - Servlet, Managed Bean, EJB, Webservice, REST Service, ...

#### 4.7.3 Domain Services and DOA/DDD

DDD (Domain Driven Design, vergleiche Evans 2003).

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

Insofern erlebte DDD mit MS ein wichtiges Revival, um den Gesamtumfang einer Anwendung in definierte Teile zu zerlegen und für eine Microservice Architektur Umsetzung zu strukturieren.

##### 4.7.3.1 *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 bottom-up vor indem zuerst die Domänen Objekte und deren fachlich grundlegenden Beziehungen definiert werden, die danach in BC gruppiert werden.

#### 4.7.4 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 umsetzen.

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>

#### 4.7.5 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.

#### 4.8 MS API Architektur Patterns

MS als verteiltes System benötigt weiterführende zusätzliche Architektur und Implementierungs-Muster. Dazu zählen zum Beispiel:

#### 4.9 MS Architektur Verteilung

Microservices benötigen zur Laufzeit eine entsprechende Umgebung, und dies zieht auch weitere benötigte Infrastruktur Services nach sich.

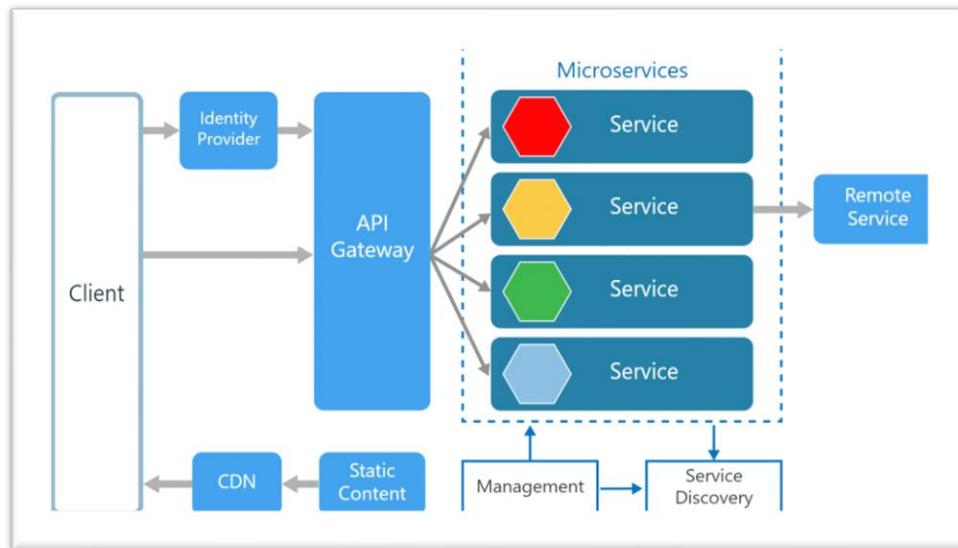


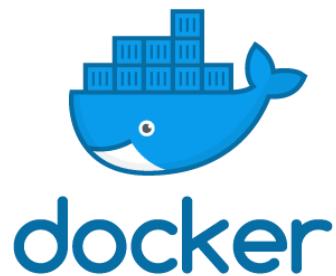
Abbildung 5 Microservices & API Gateway

Elemente einer Verteilten MS Struktur sind unter anderem:

##### 4.9.1 MS Laufzeit Umgebung

MS werden für den operativen Betrieb oftmals mit bzw in Containern betrieben. Der bisher klassische Ansatz dafür sind Docker Container:

1. Ein Docker Image ist **Speicherabbild eines Containers**.
2. Docker Images werden mittels „Layer“ Konzept implementiert.
3. Images werden über eine Repository für die Laufzeit zur Verfügung gestellt
4. als **Container** wird die aktive Instanz eines Images bezeichnet.



Neuere Technologien stellen einfachere und neuere Varianten wie Pod

#### 4.9.2 Micro Service Management (Kubernetes)

Eine Plattform die MS Managen kann ist Kubernetes.

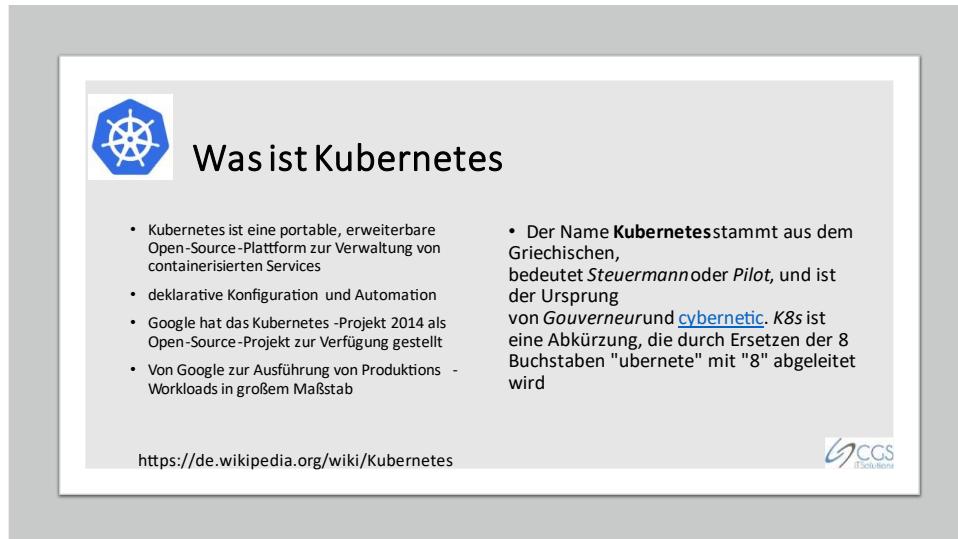


Abbildung 6 Was ist Kubernetes, K8s

Kubernetes bietet folgende Funktionalitäten für Microservices und Container an:

- Containerplattform
- Microservices-Plattform
- Portable Cloud-Plattform
- Es koordiniert die Computer-, Netzwerk- und Speicherinfrastruktur im Namen der Benutzer-Workloads
- Ermöglicht die Portabilität zwischen Infrastrukturanbietern

#### 4.10 Kapitel Zusammenfassung: Microservice Architektur

Dieses Kapitel befasste sich mit den grundlegenden Architektur Konzepten von Cloud und Microservice Applikationen. Zuerst wurde die unterschiedliche Granularität vor allem der Laufzeit und Deployment Abbildung von klassischen als auch Microservice Anwendungen dargestellt.

Danach wurden einige Architekturmuster dargestellt, die im MS Bereich eine Anwendung finden.

##### 4.10.1 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](https://www.goodreads.com/book/show/179133.Domain_Driven_Design)

## 5 Quarkus

Quarkus (<https://quarkus.io/>) ist ein CDI Container basiertes Framework, dass mit starkem Fokus auf Standards eine schnelle und Cloud-Native Software Entwicklung unterstützt. Es setzt dabei auf Eclipse MicroProfile Standards.



### 5.1 Quarkus Extension Architektur

Quarkus stellt dabei als Grundkonzept den Quarkus Core und eine CDI Implementierung (ARC) zur Verfügung. Alle weiteren benötigten Elemente werden als Quarkus Extensions optional zur Verfügung gestellt.

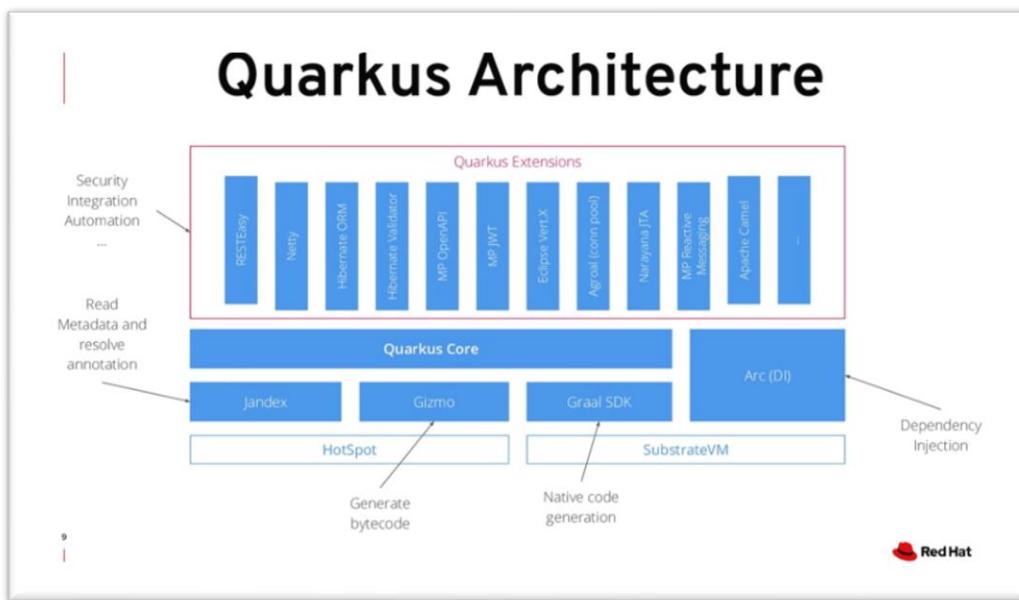


Abbildung 7 Quarkus Architecture

#### 5.1.1 Quarkus Extensions

Quarkus implementiert dabei einen Spring Boot ähnlichen Ansatz, in dem es die Erweiterungen möglichst einfach als Extension mit einer Default Konfiguration anbietet.

## Dealing with extensions

From inside a Quarkus project, you can obtain a list of the available extensions with:

```
./mvnw quarkus:list-extensions
```

You can enable an extension using:

```
./mvnw quarkus:add-extension -Dextensions="hibernate-validator"
```

Extensions are passed using a comma-separated list.

*Abbildung 8 Quarkus Maven Extension*

Dabei ist es sowohl im IntelliJ direkt im Maven pom.xml als auch mittels Maven Quarkus Extension leicht möglich eine Erweiterung wie hier die Hibernate Validator Erweiterung hinzuzufügen.

### 5.1.2 Graal VM

Quarkus ist dabei auch nicht auf die Standard HotSpot Virtuelle Maschine fixiert, sondern stellt mit dem Support für die Graal VM auch native Executables zur Verfügung. Sie erlaubt das Kompilieren des Java-Codes in direkt ausführbaren Maschinencode

#### Performance-Vorteile Quarkus & GraalVM:

- Schnelle Startzeit der Anwendung
- Geringer Speicherverbrauch der laufenden Anwendung
- Beinahe unmittelbare Skalierung von Services
- Geringer Platzbedarf der nativen Images

### 5.2 Quarkus Spring DI Kompatibilität

Quarkus stellt einen Kompatibilitäts-Modus für Spring basierte Annotationen zur Verfügung.

If you already have your Quarkus project configured, you can add the `spring-di` extension to your project by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="spring-di"
```

This will add the following to your `pom.xml`:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-spring-di</artifactId>
</dependency>
```

*Abbildung 9 Spring DI Support*

### 5.3 Quarkus Konfiguration

Eine Übersicht über alle Konfigurationsparameter der meisten Extensions findet sich hier

<https://quarkus.io/guides/all-config>

## 5.4 Quarkus DEV UI

Das Quarkus DEV UI bietet eine komfortable Information für den Entwickler über die Konsole.

<http://localhost:8080/q/dev-v1/>

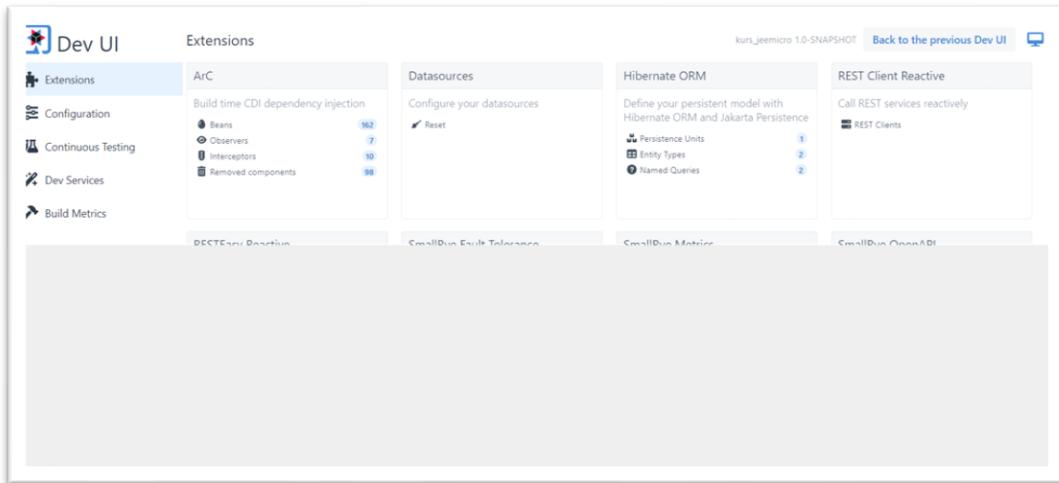


Abbildung 10 Quarkus Dev UI

Beginnend mit Quarkus 3 gibt es eine neue verbesserte Konsole. Über diese Konsole biete jede Quarkus Erweiterung bestimmte Informationen an.

Im IntelliJ kann mittels der Tastenkombinationen „w“ die Quarkus Console direkt geöffnet werden.

## 5.5 Kapitel Zusammenfassung: Quarkus Grundlagen

Das Kapitel stellte den grundlegenden Aufbau der Quarkus Architektur und Modul/Extension Struktur dar.

### 5.5.1 Weiterführende Literatur

- Quarkus Dokumentation  
<https://quarkus.io>  
<https://quarkus.io/quarkus3>
- Microprofile Spezifikation  
<https://microprofile.io>
- Micro Services  
<https://de.wikipedia.org/wiki/Microservices>

## 6 Quarkus – Configuration Management

Quarkus hat eine Konfigurations-Hierarchie die entsprechend von innen nach Außen jeweils mit den Property Werten übersteuert werden kann.

Als Einstiegspunkt für die Entwicklung dient das File „*application.properties*“ im Verzeichnis *src/main/resources*. Diese Konfiguration kann aber entsprechend von außen durch das externe config file, oder System Properties übersteuert werden.

Wobei jeweils der Layer mit der höchsten Priorität als finaler Wert übernommen wird. D.h. ein System Property Wert hat die höchste Priorität, und wird allen anderen Werten vorgezogen.

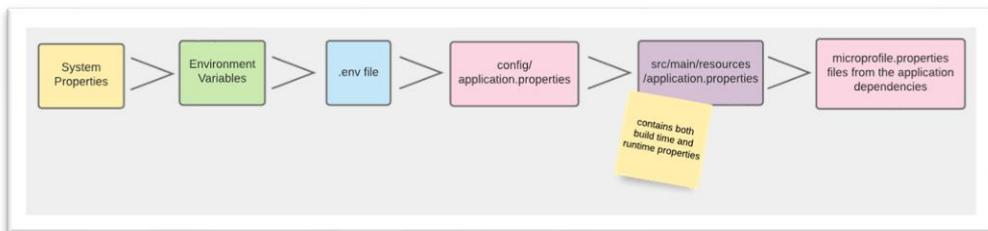


Abbildung 11 Quarkus Configuration Hierarchy (Quelle Quarkus)

By default, Quarkus reads configuration properties from multiple sources:

1. (400) [System properties](#)
2. (300) [Environment variables](#)
3. (295) [.env](#) file in the current working directory
4. (260) [Quarkus Application configuration file](#) in \$PWD/config/application.properties
5. (250) [Quarkus Application configuration file](#) application.properties in classpath
6. (100) [MicroProfile Config configuration file](#) META-INF/micropackage-config.properties in classpath

Konfiguration Properties haben dabei folgendes Format. Sie werden hierarchisch von links nach rechts aufgebaut. Der Einstieg Quarkus definiert Quarkus und Extension spezifische Parameter.

Eigene Parameter können zusätzlich beliebig definiert werden.

```
# Das ist mein erstes Property
greeting.message="greeting configuration new"

# REST, resteasy, jackson
quarkus.jackson.fail-on-unknown-properties=true
```

### 6.1 Quarks Configuration Injection

In der Anwendung können die Konfigurationsparameter einfach mittels Annotation `injected` und verwendet werden.

```
@Path("/helloMicroDemo")
public class DemoMicroResource {
    private static final Logger LOG = Logger.getLogger(DemoMicroResource.class);
```

```

@ConfigProperty(name = "greeting.message", defaultValue = "default-value")
String message;

@GET
@Path("/showMessage")
@Produces(MediaType.TEXT_PLAIN)
public String showMessage() {
    LOG.infov( "INFO :: showMessage {0}", message);

    try {
        return "Hello: " + message;
    } catch (RuntimeException ex) {
        LOG.error("fehler beim message lesen ", ex);
    }
    return "";
}

```

Dabei ermöglicht die Annotation `@ConfigProperty` einen zusätzlichen Default Wert anzugeben.

### 6.1.1 System Properties

System properties can be handed to the application through the -D flag during startup. The following examples assign the value youshallnotpass to the attribute quarkus.datasource.password.

- For Quarkus dev mode:  
`./mvnw quarkus:dev -Dquarkus.datasource.password=youshallnotpass`
- For a runner jar:  
`java -Dquarkus.datasource.password=youshallnotpass -jar target/quarkus-app/quarkus-run.jar`
- For a native executable:  
`./target/myapp-runner -Dquarkus.datasource.password=youshallnotpass`

### 6.1.2 Quarkus Property Expressions und Umgebungen

Quarkus ermöglicht es Property Expressions zu verwenden, die zum Beispiel auf eine andere Variable referenzieren.

```

remote.host=quarkus.io
callable.url=https://${remote.host}/

```

(Quelle Quarkus Dokumentation)

Weiters ist auch ein Feature vorhanden, um Variablen bereits für unterschiedliche Umgebungen zu spezifizieren. Und je nach Umgebung werden diese dann automatisch aktiv:

```

%dev.application.server=localhost
application.server=remotehost

```

Siehe dazu auch:

<https://quarkus.io/guides/config-reference#property-expressions>

## 6.2 Quarkus Konfiguration in der Cloud/K8s

In einer Cloud basierten Umgebung werden Konfigurationen fast zwingend auch via zentrale externe Konfiguration zur Verfügung gestellt.

<https://quarkus.io/guides/kubernetes-config>

Weiters sind auch Extensions verfügbar die einen zentralen Configuration server anbinden können.  
Z.b. Hashicorp Konfiguration Provider.

## 7 Quarkus CDI – Context and Dependency Injection

Quarkus ARC implementiert einen CDI (IoC) Container für die Implementierung von Beans.

### 7.1 CDI Beans Definition and Injection

Ein CDI Bean wird durch die Quarkus und CDI-Spezifikation folgendermaßen definiert:

- A **CDI-bean** is a container-managed object that supports a set of basic services, such as injection of dependencies, lifecycle callbacks and interceptors.
- An application developer can focus on the business logic rather than finding out "where and how" to obtain a fully initialized component with all of its dependencies.

```
import jakarta.inject.Inject;
import jakarta.enterprise.context.ApplicationScoped;
import org.eclipse.microprofile.metrics.annotation.Counted;

@ApplicationScoped ①
public class Translator {

    @Inject
    Dictionary dictionary; ②

    @Counted ③
    String translate(String sentence) {
        // ...
    }
}
```

Abbildung 12 CDI Bean Beispiel

1. This is a scope annotation. It tells the container which context to associate the bean instance with. In this particular case, a single bean instance is created for the application and used by all other beans that inject Translator.
2. This is a field injection point. It tells the container that Translator depends on the Dictionary bean. If there is no matching bean the build fails.
3. This is an interceptor binding annotation. In this case, the annotation comes from the MicroProfile Metrics.

## 7.2 CDI Bean Dependency Resolution

### 7.3 CDI Annotations

Die wichtigsten Annotations sind:

Annotation	Description
@Inject	Identifies injectable constructors, methods, and fields
@Qualifier	Identifies qualifier annotations
@ApplicationScoped, @SessionScoped, @RequestScoped, @Singleton, @Dependent	Set of annotations defining the life cycle of a bean
@Observes	Identifies the event parameter of an observer method

Abbildung 13 CDI Annotations

## 7.4 CDI Beans - Live Cycle Annotations

Live Cycle Annotations bestimmen wie lange und mit wie vielen Objekt Instanzen ein Bean lebt und verwendet wird. Grundsätzlich werden nur CDI Beans gefunden und instanziert die auch mit einer der hier angeführten Annotationen annotiert ist:

Annotation	Description
@javax.enterprise.context.ApplicationScoped	A single bean instance is used for the application and shared among all injection points. The instance is created lazily, i.e. once a method is invoked upon the <a href="#">client proxy</a> .
@javax.inject.Singleton	Just like @ApplicationScoped except that no client proxy is used. The instance is created when an injection point that resolves to a @Singleton bean is being injected.
@javax.enterprise.context.RequestScoped	The bean instance is associated with the current <i>request</i> (usually an HTTP request).
@javax.enterprise.context.Dependent	This is a pseudo-scope. The instances are not shared and every injection point spawns a new instance of the dependent bean. The lifecycle of dependent bean is bound to the bean injecting it - it will be created and destroyed along with the bean injecting it.
@javax.enterprise.context.SessionScoped	This scope is backed by a javax.servlet.http.HttpSession object. It's only available if the quarkus-undertow extension is used.

Quelle:

<https://quarkus.io/guides/cdi#bean-scope-available>

## 7.5 CDI Bean Discovery

Quarks verwendet eine Vereinfachte Methode, um die vorhandenen CDI Beans einer Quarkus Anwendung zu suchen.

Quarkus sucht alle mit Live Cycle Annotierten Klassen.

## 7.6 Injection Grundlagen

Folgende Möglichkeiten sind vorhanden:

1. Das Injected-Objekt kann beim Injection-Point auch ein Interface sein oder ein konkretes Objekt
2. Getter und Setter Methoden sind nicht erforderlich für die Injection
3. Das Inject feld kann auch private sein
4. Ein Konstruktur Inject ist bei Quarkus möglich
5. Ein setter Injekt ist auch erlaubt

Injection Demo Bean:

Siehe Appendix: Injection [Demo Bean](#)

### 7.6.1 Property Injection

Die gängigste Injection Variante ist die Annotation mittels @Inject direkt bei der Deklaration des Properties selbst.

```
@Path("/simplecdi")
public class CDISimpleResource {

    @Inject
    SimpleCDIBean cdiBean;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String requestScope() {
        String new_value = cdiBean.echo("new value");
        return new_value;
    }
}
```

Der CDI Container sucht das entsprechende Bean in der Liste der vorhanden Beans für das Inject. Danach wird das Bean bei Bedarf Lazy erzeugt, und die Variable cdiBean initialisiert. Für die Initialisierung wird die Variable via Java Reflection im verwendenden Bean modifiziert.

### 7.6.2 Setter Injection

Hier wird die @Inject Annotation direkt bei der Setter Methode verwendet

```
@RequestScoped
public class SetterInjection {

    @Inject
    Logger log;

    SimpleCDIBean cdiBean;
```

```

@Inject
public void setCdiBean(SimpleCDIBean cdiBean) {
    this.cdiBean = cdiBean;
}

public String echo(String input) {
    log.info("SetterInjection");
    return cdiBean.echo(input);
}

}

```

### 7.6.3 Constructor Injection

Nur ein Bean Constructor ist bei Quarkus erlaubt mit der Inject Annotation

```

@RequestScoped
public class ConstructorInjection {

    @Inject
    Logger log;

    SimpleCDIBean cdiBean;

    @Inject
    public ConstructorInjection(SimpleCDIBean cdiBean) {
        this.cdiBean = cdiBean;
    }

    public String echo(String input) {
        log.info("SetterInjection");
        return cdiBean.echo(input);
    }
}

```

Mit der Konstruktor oder Setter Injection Variante kann das Schlüsselwort @Inject bei Quarkus auch weggelassen werden.

### 7.6.4 Default Injection

Quarkus bzw CDI bietet das Schlüsselwort

```

@Path("/simplecdi")
public class CDISimpleResource {

    @Inject
    private SimpleCDIBean cdiBean;

    @Inject
    @Default
    private ConstructorInjection constructorInjection;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String requestScope() {

```

## 7.7 Injection mit Interfaces, Default und Alternativen

Mehr Nutzen bringt für die Flexibilität der Anwendung bringt die Benutzung von Interfaces und Implementierungen.

Gibt es nur 1 CDI Bean wird diese einzige Implementierung injected

```
@RequestScoped
public class InterfaceBeanImpl implements InterfaceBean {

    @Override
    public String echo(String input) {
        return "InterfaceBeanImpl: " + input;
    }
}
```

```
@Path("/simplecdi_interface")
public class CDIInterfaceResource {

    @Inject
    InterfaceBean interfaceBean;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String requestScope() {
        String new_value = interfaceBean.echo("interfaceBean");
        return new_value;
    }
}
```

### 7.7.1 Qualifier und Injections

Um bei mehreren vorhandenen Implementierungen eine entsprechende Implementierung auswählen zu können, kann dafür ein eigener CDI Qualifier implementiert werden.

```
import jakarta.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface QualifyA { }
```

### 7.7.2 Qualifier Error

Sind mehrere Implementierungen für einen Injection Point vorhanden, und kein Qualifier angegeben, so wird eine Resolution Exception geworfen.

```
@Path("/simplecdi_qualify")
public class CDIQualifyResource {

    @Inject
    QBean interfaceBean;
```

Fehler:

```
Resulted in: jakarta.enterprise.inject.UnsatisfiedResolutionException: Unsatisfied
dependency for type at.cgsit.jeemicro.cdi.qualify.QBean and qualifiers [@Default]
    - java member: at.cgsit.jeemicro.resource.cdi.CDIQualifyResource#interfaceBean
    - declared on CLASS bean
[types=[at.cgsit.jeemicro.resource.cdi.CDIQualifyResource, java.lang.Object],
qualifiers=[@Default, @Any], target=at.cgsit.jeemicro.resource.cdi.CDIQualifyResource]
    The following beans match by type, but none have matching qualifiers:
        - Bean [class=at.cgsit.jeemicro.cdi.qualify.QBeanImplB, qualifiers=@Any,
@QualifyB]
        - Bean [class=at.cgsit.jeemicro.cdi.qualify.QBeanImplA, qualifiers=@Any,
@QualifyA]
    at io.quarkus.arc.processor.Beans.resolveInjectionPoint(Beans.java:482)
    at io.quarkus.arc.processor.BeanInfo.init(BeanInfo.java:564)
    at io.quarkus.arc.processor.BeanDeployment.init(BeanDeployment.java:293)
```

Erst durch die Qualifizierung bei der Verwendung des Beans wird diese Mehrdeutigkeit aufgelöst

```
@Path("/simplecdi_qualify")
public class CDIQualifyResource {

    @Inject
    @QualifyB
    QBean interfaceBean;
```

## 7.8 Alternative Beans

Die Annotation `@Alternative` kann dazu benutzt werden, um zusätzliche Alternative Beans zur Verfügung zu stellen.

- Diese Beans können mittels Applikation Konfiguration Property aktiviert werden.
- Ersetzten dann die Default Implementierung

```
@Alternative
@ApplicationScoped
public class AlternativeBeanImplDummy implements AlternativeBeanInterface,
Serializable {

    @Inject
    Logger log;

    @Override
    public String echo(String input) {
... }}
```

```
# cdi alternatives
quarkus.arc.selected-alternatives=at.cgsit.jeemicro.cdi.alternatives.mock.*
```

## 7.9 CDI Producer

- `@Produces` erlaubt es die Erzeugung von CDI Beans selbst zu implementieren.
- Dabei kann auch der Scope der Beans definiert werden.
- Hier in Kombination mit einem Konfigurations Property.

```
public class PBProducer {

    private static Logger log = Logger.getLogger(PBProducer.class);

    @ConfigProperty(name = "at.cgs.training.produceBean", defaultValue = "a")
    String beantoProduce;

    @Produces
    @RequestScoped
    PBInterface producePB() {
        log.info("producer called");
        if( "a".equalsIgnoreCase(beantoProduce)){
            return new PBImplA();
        }
        return new PBImplB();
    }
}
```

Konfiguration für den Producer (Beispiel):

```
# wechselt der Implementierung von A auf B via Producer Methode und diesem Konfigurationsparameter
at.cgs.training.produceBean=b
# at.cgs.training.produceBean=a
```

## 7.10 CDI Lifecycle Callbacks

Lifecycle callbacks provides Callbacks for CDI Beans that are called Post Construct and PRE destroy

```
import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;

@ApplicationScoped
public class Translator {

    @PostConstruct ①
    void init() {
        // ...
    }

    @PreDestroy ②
    void destroy() {
        // ...
    }
}
```

1. This callback is invoked before the bean instance is put into service. It is safe to perform some initialization here.
2. This callback is invoked before the bean instance is destroyed. It is safe to perform some cleanup tasks here.

## 7.11 CDI Priorities

`@javax.interceptor.Interceptor.Priority` takes an integer that can be any value. The rule is that values with a lower priority are called first. The `javax.interceptor.Interceptor` annotation defines the following set of constants:

- `PLATFORM_BEFORE = 0`: Start of range for early interceptors defined by the platform,
- `LIBRARY_BEFORE = 1000`: Start of range for early interceptors defined by extension libraries,
- `APPLICATION = 2000`: Start of range for interceptors defined by applications,
- `LIBRARY_AFTER = 3000`: Start of range for late interceptors defined by extension libraries, and
- `PLATFORM_AFTER = 4000`: Start of range for late interceptors defined by the platform.

## 7.12 CDI Interceptoren

Ein Interceptor gibt die Möglichkeit vor und nach dem Aufruf der jeweiligen Ziel Bean Methode eigenen Source Code auszuführen.

Dies kann zum Beispiel für eigenes Logging (wie in diesem Beispiel gezeigt) genutzt werden:

```

@Logged
@Priority(2020)
@Interceptor
public class LoggingInterceptor {

    @Inject
    Logger logger;

    @AroundInvoke
    Object logInvocation(InvocationContext context) throws Exception {
        logger.info("object before");
        Object ret = context.proceed();
        logger.info("object after");
        return ret;
    }
}

```

Dafür muss eine eigene Annotation programmiert werden. Diese Annotation implementiert ein jakarta Interceptor Binding.

Das Binding kann dann auf Klassen, Methoden oder Konstruktor Scope im Ziel verwendet werden.

```
@InterceptorBinding
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.CONSTRUCTOR})
@Inherited
public @interface Logged {
}
```

Die Verwendung sieht für ein klassenweites Logging folgend aus

```
@RequestScoped
@Logged
public class RSBeanInterceptedExample {

    public String echoReverse(String input) {
        StringBuilder inputSB = new StringBuilder();
        StringBuilder reverse = inputSB.append(input).reverse();
        return reverse.toString().toUpperCase(Locale.ROOT);
    }

    public String echoReverse2(String input) {
        String reverse = echoReverse(input);
        return echoReverse(reverse);
    }
}
```

### 7.13 CDI Scopes

Ein RequestScoped Bean wird immer dann neu erzeugt, wenn ein neuer http Request verarbeitet wird.

Wird also dieses Bean in einer JaxRS Service Resource verwendet, wird für jeden Request eine neue eigene Bean Objektinstanz instanziert.

```
@RequestScoped
public class RSBean {

    @Inject
    Logger log;

    private String requestScopedMessage = "default";

    @PostConstruct
    public void postConstruct() {
        log.info("postConstruct called: " + LocalDateTime.now());
    }
}
```

```
public String getRequestScopedMessage() {  
    return requestScopedMessage;  
}  
  
public void setRequestScopedMessage(String requestScopedMessage) {  
    this.requestScopedMessage = requestScopedMessage;  
}
```

## 7.14 Application Scoped Bean

Das Application Scoped Bean wird nur Einmal pro Quarkus Instanz erzeugt.

```
@ApplicationScoped  
public class ApplicationScopeBean {  
  
    public Integer counter = 0;  
  
    public Integer getCounter() {  
        this.counter++;  
        return counter;  
    }  
}
```

## 7.15 CDI Events

Ein CDI Bean kann ein Event erzeugen (fire).

Dieses Event wird mittels Event Injection Annotation deklariert.

Die eigene Event Klasse “SpecialEvent“ kann nach Bedarf implementiert werden

```
@Singleton
public class SpecialService {

    @Inject
    Event<SpecialEvent> event;

    public void doSomething() {
        event.fire(new SpecialEvent("Special Event"));
    }
}
```

- Die eigene Event Klasse “SpecialEvent“ kann nach Bedarf implementiert werden

```
public class SpecialEvent {

    private String message;

    public SpecialEvent(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```

Das Event kann durch die Annotation @Observes <EventType> empfangen werden:

```
@ApplicationScoped
public class SpecialEventListener {

    @Inject
    Logger log;

    void onSpecialEventCompleted(@Observes SpecialEvent event) {
        log.info("SpecialEventListener called: " + event.getMessage());
    }
}
```

TEE: 00:47:05 INFO [at.cg.je.cd.ev.SpecialEventListener] (executor-thread-1) SpecialEventListener  
called: Special Event

### 7.15.1 CDI Startup Events

```
@ApplicationScoped
class CoolService {
    void startup(@Observes StartupEvent event) {
    }
}
```

```
@Startup
@ApplicationScoped
public class EagerAppBean {

    private final String name;

    EagerAppBean(NameGenerator generator) {
        this.name = generator.createName();
    }
}
```

### 7.16 Quarkus Profiles

By default Quarkus comes with three profiles:

1. dev – Activated when in development mode: mvn quarkus:dev
2. test – Activated when running tests: mvn test
3. prod – The default profile when not running in development or test mode

```
@Dependent
public class TracerConfiguration {

    @Produces
    @IfBuildProfile("prod")
    public Tracer realTracer() {
        return new TracerImplTwo();
    }

    @Produces
    @DefaultBean
    public Tracer noopTracer() {
        return new TracerImplTwo();
    }
}
```

## 8 Quarkus –Security

- Das Beispiel basiert auf der Quarkus Basic Security Dokumentation
- Basic HTTP Security wird aktiviert
- JaxRS Security Annotations für Rest Services werden konfiguriert und verwendet

```
<!-- add quarkus security via jpa user entity -->
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-security-jpa</artifactId>
</dependency>
```

### 8.1 ORM Panachache Vereinfachung der Persistenz Operationen

Panachache Vereinfachung der Persistenz Operationen

```
<!-- panachache quarkus jpa extensions -->
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-hibernate-orm-panache</artifactId>
</dependency>
```

### 8.2 Quarkus – JaxRS Security Annotaitons

Eine öffentliche Ressource wird mittels @PermitAll gekennzeichnet. Sie benötigt kein Login

```
@Path("/api/public")
public class PublicResource {

    @GET
    @PermitAll
    @Produces(MediaType.TEXT_PLAIN)
    public String publicResource() {
        return "public";
    }
}
```

#### 8.2.1 Quarkus User Roles Allowed

Die Annotation @RolesAllowed kennzeichnet einen Resource Pfad als geschützt. Der Benutzer benötigt eine gültige User Session.

Der Benutzer muss in der richtigen User-Rolle sein.

```
@Path("/api/users")
public class UserResource {

    @GET
    @RolesAllowed("user")
    @Path("/me")
    public String me(@Context SecurityContext securityContext) {
        return securityContext.getUserPrincipal().getName();
    }
}
```

```
}
```

Der Admin User ist nur eine spezielle Rolle.

Alle Benutzer (User) die diese Rolle zugewiesen haben, dürfen die Methode auch aufrufen

```
@Path("/api/admin")
public class AdminResource {

    @GET
    @RolesAllowed("admin")
    @Produces(MediaType.TEXT_PLAIN)
    public String adminResource() {
        return "admin";
    }
}
```

## 8.2.2 Quarkus JPA Security Extension und Panache

Die Quarkus JPA Security Extension bietet Annotations zur Abbildung der user und Berechtigungen in der Datenbank an.

Zusätzlich konfiguriert diese Extension das Quarkus System und Rest-Easy so, dass diese Benutzer auch an das JaxRS System entsprechend angebunden und richtig konfiguriert werden.

Das Beispiel nutzt zusätzlich Quarkus Panache für ein einfacheres Datenbank Mapping.

```
import io.quarkus.elytron.security.common.BcryptUtil;
import io.quarkus.hibernate.orm.panache.PanacheEntity;
import io.quarkus.security.jpa.Password;
import io.quarkus.security.jpa.Roles;
import io.quarkus.security.jpa.UserDefinition;
import io.quarkus.security.jpa.Username;
import jakarta.persistence.Entity;
import jakarta.persistence.Table;

@Entity
@Table(name = "test_user")
@UserDefinition
public class User extends PanacheEntity {
    @Username
    public String username;
    @Password
    public String password;

    // einfache variante für rollen ohne extra tabelle
    @Roles
    public String role;
```

The **security-jpa** extension initializes only if there is a single entity annotated with **@UserDefinition**.

- The **@UserDefinition** annotation must be present on a single entity and can be either a regular Hibernate ORM entity or a Hibernate ORM with Panache entity.

- @Username: Indicates the field used for the username.
- @Password Indicates the field used for the password. By default, it uses bcrypt-hashed passwords. You can configure it to use plain text or custom passwords.
- @Roles This indicates the comma-separated list of roles added to the target principal representation attributes.

### 8.2.3 Quarkus Beispiel Initialisierung der user Panachache Vereinfachung der Persistenz Operationen

```
@Singleton
public class StartupInitUser {

    @Transactional
    public void loadUsers(@Observes @Priority(1) StartupEvent evt) {
        log.info("StartupInitUser initialization called ");

        // reset and load all test users
        User.deleteAll();
        User.add("admin", "admin", "admin");
        User.add("user", "user", "user");
    }
}
```

#### 8.2.4 Quarkus Security Swagger Support

Swagger stellt für die Basic Security auch eine Login Maske zur Verfügung

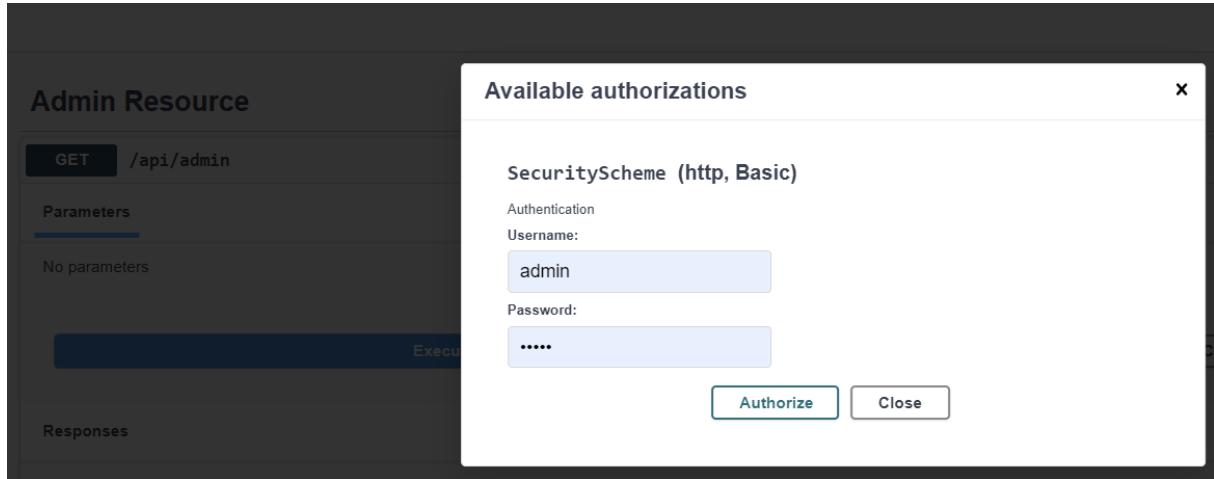


Abbildung 14 Quarkus Swagger Security Login

#### 8.2.5 QuarkusTest Support für Security

Quarkus Test unterstützt das Testen der Security mittels Rest Assured fluent API .basic

```
@Test
void shouldNotAccessUserWhenAdminAuthenticated() {
    given()
        .auth().preemptive()
        .basic("admin", "admin")
        .when()
        .get("/api/users/me")
        .then()
        .statusCode(HttpStatus.SC_FORBIDDEN);
}
```

### 8.3 JaxRS – Rest Services

#### 8.3.1 Rest Service - Architecture

Rest Service as an architectural concept is implementation language independent. It defines a Style and Concept for exchanging Data via HTTP/S and Internet Standards.

The basic idea is to use Http methods (GET, POST, PUT, DELETE) to directly be used for the intended CRUD (create, read, update, delete) business method. And then transport the data like html but without the html formatting overhead and serialize the data in a standard format like JSON or XML.

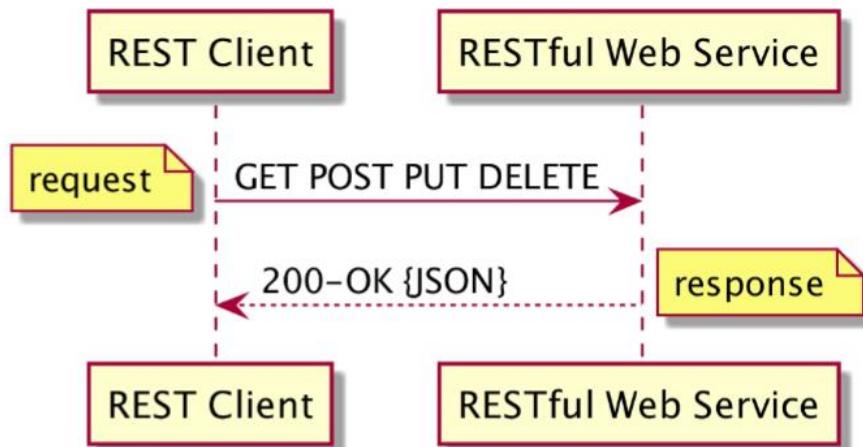


Abbildung 15 Rest Webservice Communication

Also the URI (Uniform Resource Identifiers) are a major design principle here. This means that each Resource is mapped to an URI. For example: <http://host/service/chatRoom/10> would identify the chat Room Number 10 as an URI in the http Rest Context.

This will simplify the overall tech stack compared to other technologies and speed up development.

Restful Services therefore also strongly de-couples Services and makes it easier to develop and maintain distributed Systems. Not only the de-coupling removes specific technologies like RMI, Remote-EJBs it also reduced the complexity on the involved standards since http is a major robust stable standard. In Comparison Soap Services introduce a bit more specification on top of the base and requires more complex standards compared to Restful Services.

### REST : Definition [Wikipedia]

A REST API is an application programming interface (API) that uses a representational state transfer (REST) architectural style. The REST architectural style uses HTTP to request access and use data. This allows for interaction with RESTful web services.

Abbildung 16 REst Services Definition

Characterizations for Rest-Services

## JEE Microservice Development

1. An architectural style, not technology
2. Client/server + Request/response approach.
3. Everything is a RESOURCE.
4. CRUD (Create / Read / Update / Delete)
5. Stateless by nature (excellent for distributed systems)
6. Cacheable (naturally supported)
7. A great way to web-service

See Also:

[https://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)

### 8.3.2 Rest Service – HTTP Methods

To fully understand Rest Services regarding Rest Methods, it would be best to also directly read the https specification also:

<https://www.rfc-editor.org/rfc/rfc9110.html#name-method-definitions>

Rest than maps those methods to the following (simplified) meaning:

Rest Methods	
Method	Description
GET	Retrieve information about the REST API resource
POST	Create a REST API resource
PUT	Update a REST API resource
DELETE	Delete a REST API resource or related component

Abbildung 17 Rest Service – Methods

### 8.3.3 Example Restful Application

The example application descripts a simple Ticket System like this:

Method		
GET	/tickets	Retrieves a list of tickets
GET	/tickets/12	Retrieves a specific ticket
POST	/tickets	Creates a new ticket
PUT	/tickets/12	Updates ticket #12
DELETE	/tickets/12	Deletes ticket #12

Abbildung 18 Exmple Ticket System API

### 8.3.4 JaxRS Restservices

Wikipedia definiert JaxRS Restservices folgend

- Bei den **Jakarta RESTful Web Services** (JAX-RS) handelt es sich um die Spezifikation einer Programmierschnittstelle (API) der Programmiersprache Java,
- die die Verwendung des Software-Architekturstils Representational State Transfer (REST) im Rahmen von Webservices ermöglicht und vereinheitlicht.
- Die in der Spezifikation beschriebenen Funktionalitäten wurden vom Java Community Process erarbeitet
- und im Java Specification Request 311<sup>[1]</sup> verabschiedet.
- Wie auch andere Programmierschnittstellen der Jakarta EE (JEE) benutzt JAX-RS Annotationen, um die Entwicklung und das Deployment von Webservice-Clients und Service-Endpunkten zu vereinfachen.

Further specifications from RedHat defines:

- RESTEasy is a JBoss / Red Hat project that provides various frameworks to help you build RESTful Web Services and RESTful Java applications.
- It is an implementation of the Jakarta RESTful Web Services, an Eclipse Foundation specification that provides a Java API for RESTful Web Services over the HTTP protocol.
- Moreover, RESTEasy also implements the MicroProfile REST Client specification API.
- RESTEasy can run in any Servlet container, WildFly Application Server and Quarkus is also available to make the user experience nicer in those environments.

Siehe auch:

[https://de.wikipedia.org/wiki/Jakarta\\_RESTful\\_Web\\_Services](https://de.wikipedia.org/wiki/Jakarta_RESTful_Web_Services)

Siehe auch

- Cheat Sheet für Resteasy Annotations  
<https://www.mastertheboss.com/jboss-frameworks/resteasy/jax-rs-cheatsheet/>

### 8.3.5 Java JaxRS – Annotations

All available Annotations are defined in the package “jakarta.ws.rs.” since the latest version. Previous versions had the package import “javax.ws.rs.” this was changed by Jakarta since the Jakarta EE 9 release on Dec 8th, 2020.

Developing Rest Services with JEE Jakarta Annotations has the additional advantage of working with an API instead of an implementation. Various compatible implementations are implementing this standard. See

<https://jakarta.ee/blogs/javax-jakartaee-namespace-ecosystem-progress/>

### 8.3.6 Method and Path Annotations

Example:

```

@Path("/library")
public class Library {

    @GET
    @Path("/books")
    public String getBooks() {...}

    @GET
    @Path("/book/{isbn}")
    public String getBook(@PathParam("isbn") String id) {
        // search my database and get a string representation and return it
    }

    @PUT
    @Path("/book/{isbn}")
    public void addBook(@PathParam("isbn") String id, @QueryParam("name") String name) {...}

    @DELETE
    @Path("/book/{id}")
    public void removeBook(@PathParam("id") String id) {...}

}

```

#### 8.3.6.1.1 @Path

Path indicates the URI Path under which the Resource can be found. This is the relative path part for this application. So there might be a base @Path for the Application and within this Application the path /library is then mounted. Like /application/library/10

#### 8.3.6.1.2 @PUT, @GET, @DELETE @POST

Using the Annotations specifies which java Method is provided for the corresponding HTTP Method for this /library API.

Please see that the same Path is provided with the same URI but with different http methods like PUT and POST or DELETE.

#### 8.3.6.1.3 @Path Parameter

Using the @PathParam Annotation enables us to extract information directly from the URI Path into Method Parameters. To use this correctly a placeholder information must be specified within the @PATH Annotation also. Like {inputString} in this example.

```
http://localhost:8080/json/inputParameter/{inputString}
no usages
@GET
@Path("/inputParameter/{inputString}")
@Produces(MediaType.TEXT_PLAIN)
public String inputParameter(@PathParam("inputString") String inputString){
    log.infov( format: "log: {0}", inputString);
    return inputString;
}
```

Abbildung 19 Path Parameter

#### 8.3.6.1.4 @Query Parameter

Using the @QueryParameter enables the mapping of http query parameters into our Java Method Parameters. Like shown in this Example.

Remember that a Query Parameter is not part of the Base URI and instead is located in the Query Section of the URL Part.

```
http://localhost:8080/json/queryParameter
// http://localhost:8080/json/queryParameter?qp=inputText&qp2=text2
no usages
@GET
@Path("/queryParameter")
@Produces(MediaType.TEXT_PLAIN)
public String queryParameter(
    @QueryParam("qp") String qp,
    @QueryParam("qp2") String qP2
){
    log.infov( format: "log QueryParam: {0}", qp);
    return qp + " und " + qP2;
```

Abbildung 20 Query Parameters

### 8.3.6.1.5 @DefaultValue

Its possible to provide Default values for Parameters also directly using the Annotation `@DefaultValue`

```

@GET
@Path("/queryParameter")
@Produces(MediaType.TEXT_PLAIN)
public String queryParameter(
    @QueryParam("qp") String qp,
    @DefaultValue("5") @QueryParam("qp2") Long qP2
){
    log.info("log QueryParam: {0}", qp);
    return "query params ["+ qp + "] und [" + qP2 + "]";
}

```

### 8.3.6.1.6 @Header @Context – Special, additional Information

Using special Annotations like `@HeaderParam` and `@Context` provides additional information for the code:

```

@GET
@Path("header")
public String checkBrowser(@HeaderParam("User-Agent") String whichBrowser) {
    return "Browser is "+whichBrowser;
}

// Reading REST Parameters Programmatically
// z.b. http://localhost:8080/json/context?username=chris
@GET
@Path("context")
public Response login(@Context UriInfo info) {
    String id = info.getQueryParameters().getFirst("username");
    return Response
        .status(200)
        .entity("login called with id: " + id)
        .build();
}

```

### 8.3.7 @Produces, @Consumes Content Negotiation

Http itself already provides a mechanism for content negotiation:

<https://www.rfc-editor.org/rfc/rfc9110.html#name-content-negotiation>

This is also used by Rest-Services and supported by Annotations `@Produces` and `@Consumes`.

As a background Information you should also review the defined Mime-Type Formats for this chapter

[https://en.wikipedia.org/wiki/Media\\_type](https://en.wikipedia.org/wiki/Media_type)

The @Produces is used to map a client request and match it up to the client's Accept header. The Accept-HTTP header is sent by the client and defines the media types the client prefers to receive from the server.

```
@Produces("text/*")
@Path("/library")
public class Library {

    @GET
    @Produces("application/json")
    public String getJSON() {...}

    @GET
    public String get() {...}
```

So, if the client sends:

```
GET /library
Accept: application/json
```

Abbildung 21 @Produces and @Consumes

### 8.3.8 JSON Output/Input Support via Jackson

To enable the serialization and deserialization for java Objects to JSON you have to add an additional dependency to the maven project “Jackson”

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy</artifactId>
</dependency>
<!-- add json output support via jackson library --&gt;
&lt;dependency&gt;
    &lt;groupId&gt;io.quarkus&lt;/groupId&gt;
    &lt;artifactId&gt;quarkus-resteasy-jackson&lt;/artifactId&gt;
&lt;/dependency&gt;</pre>

```

Abbildung 22 Quarkus maven dependency for Jackson

#### 8.3.8.1.1 Deciding about the mapping library JSON-B vs Jackson

There are multiple implementations for mapping java objects from and to json.

- **Jackson** is a established library for working with JSON in Java. It provides a full suite of JSON processing capabilities, including parsing, generating, and manipulating JSON data. It is highly customizable and can be used to handle a wide variety of JSON formats and use cases.
- JSON-B is the reference implementation of JSR-367, which is an official standard for JSON binding in Java. It provides a simple and easy-to-use API for converting between Java objects and JSON, and supports many of the features of JSON-P (Java API for JSON Processing) and JSON-P-1.1. JSON-B is designed to be easy to use and requires minimal configuration and setup.

The examples are using the Jackson implementation.

#### 8.3.8.2 JSON Output Example

Using a json mapping library it is very simple and straightforward to map and produce json input and ouput for a series like this. Even a List result is possible by using the @Produces Annotations with

```

@GET
@Path("/")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public List<TestDTO> listAllObjects(){
    log.info("objectOutput {0}", "");

    TestDTO dto = new TestDTO();
    dto.setName("name");
    dto.setVorname("voranme");

    TestDTO dto2 = new TestDTO();
    dto2.setName("name2");
    dto2.setVorname("voranme2");

    List result = new ArrayList();
    result.add(dto);
    result.add(dto2);

    return result;
}
  
```

Media Type APPLICATION\_JSON.

#### 8.3.8.3 Input and Output Example

Consuming and Producing JSON then uses both Annotations:

```

@PUT
@Path("createTestMessage")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public TestDTO updateTestMessage(TestDTO input) {
    log.info("got object : {0}", input.toString());

    return input;
}
  
```

See Postman or Debugger for the corresponding http header information. Also, the swagger UI provides the Accept Headers in the http header section when json is used correctly.

### 8.3.9 JaxRS Exception handling

Jakarta JaxRS API also specifies Exception Mappers. Using such a mapper it is possible to catch all or some Exceptions from within your Java code or used libraries and define the mapping for the Rest Response as you like.

Example:

```
package at.cgsit.jeemicro.resource.exception;

import com.fasterxml.jackson.databind.exc.InvalidFormatException;
import jakarta.inject.Inject;
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.ext.ExceptionMapper;
import jakarta.ws.rs.ext.Provider;
import org.jboss.logging.Logger;

@Provider
public class JsonExceptionMapper implements ExceptionMapper<InvalidFormatException> {

    @Inject
    Logger LOG;

    @Override
    public Response toResponse(InvalidFormatException ex) {

        LOG.errorv("jackson json format exception {0}", ex.getMessage(), ex);

        return Response.status(422)
            .build();
    }
}
```

Especially for error handling you should review the http specifications for defined Error Codes and their meaning.

See HTTP Status Codes for more details:

<https://www.rfc-editor.org/rfc/rfc9110.html#name-status-codes>

## 9 Open API Spezifikation – und API-Dokumentation

Swagger wurde etwas 2011 als open source framework erstellt, und 2015/2016 in die OpenAPI Initiative und danach als OpenAPI Spezifikation umbenannt.

OpenAPI Spezifikation ist ein API Beschreibungs Format für Rest APIs.

Open API ermöglicht eine maschinen verarbeitbare Beschreibung der Schnittstelle.

Inkludiert

1. Verfügbare Endpoint services /chatmessage etc
2. Input und Output Datentypen bzw Objekte/DTOS
3. Berechtigungs Informationen
4. Lizenz und Benutzungsbedingungen
5. Versionierungs Informationen

## 9.1 OpenAPI maven Dependency

Der benötigte Maven Import ist:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-openapi</artifactId>
</dependency>
```

## 9.2 OpenAPI Header Section

OpenAPI definiert folgende Root Dokument Elemente:

Field Name	Type	Description
openapi	string	REQUIRED. This string MUST be the semantic version number of the OpenAPI
info	<a href="#">Info Object</a>	REQUIRED. Provides metadata about the API. The metadata MAY be used by tooling as required.
Servers	<a href="#">[Server Object]</a>	<b>An array of Server Objects, which provide connectivity information to a target server.</b>
Paths	<a href="#">Paths Object</a>	<b>REQUIRED.</b> The available paths and operations for the API.
Components	<a href="#">Components Object</a>	<b>An element to hold various schemas for the specification.</b>
security	<a href="#">[Security Requirement Object]</a>	A declaration of which security mechanisms can be used across the API.
tags	<a href="#">[Tag Object]</a>	A list of tags used by the specification with additional metadata

externalDocs	<a href="#">External Documentation Object</a>	Additional external Dokumentation.
--------------	---	------------------------------------

Siehe dazu: <https://swagger.io/specification/v3/>

### 9.2.1 Open API Header Beispiel

Eine OpenAPI Spezifikation kann sowohl im YAML als auch JSON Format abgebildet werden.

Jede Spezifikation startet mit dem Tag „openapi“ zur Kennzeichnung der Format und Version der Spezifikation derzeit

**openapi: 3.0.3**

Ein YAML Beispiel:

```

1  openapi: 3.0.3
2  info:
3    title: Swagger Petstore - OpenAPI 3.0
4    description: |
5      This is a sample Pet Store Server based on the OpenAPI 3.0 specification. You can find out more
       about
6      - [The source API definition for the Pet Store](https://github.com/swagger-api/swagger-petstore
         /blob/master/src/main/resources/openapi.yaml)
7    termsOfService: http://swagger.io/terms/
8    contact:
9      email: apiteam@swagger.io
10   license:
11     name: Apache 2.0
12     url: http://www.apache.org/licenses/LICENSE-2.0.html
13     version: 1.0.11
14   externalDocs:
15     description: Find out more about Swagger
16     url: http://swagger.io
17   servers:
18     - url: https://petstore3.swagger.io/api/v3
19   tags:
20     - name: pet
21       description: Everything about your Pets
22       externalDocs:
23         description: Find out more
24         url: http://swagger.io
25     - name: store
26       description: Access to Petstore orders
27       externalDocs:
28         description: Find out more about our store
29         url: http://swagger.io
30     - name: user
31       description: Operations about user
32   paths:

```

OpenAPI stellt alle notwendigen Annotations zur Verfügung um den Java Code mit Annotations soweit zu erweitern um daraus eine API Spezifikation generieren zu können.

Umgekehrt kann auch aus der OpenAPI Spezifikation mittels Generator die Java bzw auch anders Sprachen generiert werden.

Die Annotationen für die Information Header als Beispiel:

```

@OpenAPIDefinition(
  tags = {
    @Tag(name = "widget", description = "Widget operations."),
    @Tag(name = "gasket", description = "Operations related to gaskets")
  },

```

```
info = @Info(  
    title = "Chat Message Example API",  
    version = "1.0.1",  
    contact = @Contact(  
        name = "Chat Message Example API Support",  
        url = "http://exampleurl.com/contact",  
        email = "techsupport@example.com"),  
    license = @License(  
        name = "Apache 2.0",  
        url = "https://www.apache.org/licenses/LICENSE-2.0.html"))  
)  
public class ChatApplication extends Application {  
}
```

### 9.3 Open API Datatypes

- Primitive data types in the OAS are based on the types supported by the JSON Schema Specification Wright Draft 00.
- Primitives have an optional modifier property: **format**. OAS uses several known formats to define in fine detail the data type being used.

The formats defined by the OAS are:

type	format	Comments
integer	int32	signed 32 bits
integer	int64	signed 64 bits (a.k.a long)
number	float	
number	double	
string		
string	byte	base64 encoded characters
string	binary	any sequence of octets
boolean		
string	date	As defined by <a href="#">full-date</a> - <a href="#">RFC3339</a>
string	date-time	As defined by <a href="#">date-time</a> - <a href="#">RFC3339</a>
string	password	A hint to UIs to obscure input.

Abbildung 23 OpenAPI Datentypen

### 9.4 OpenAPI in Quarkus

In Quarkus das automatisch erstelle Quarkus API wird hier zur Verfügung gestellt

```

YAML: curl http://localhost:8080/q/openapi
JSON: curl -H "Accept: application/json" http://localhost:8080/q/openapi

```

## 9.5 OpenAPI Test DTO Pfad und Dokument Beschreibung

Ein Rest Services in Java bildet sich wie folgt in eine Open API spezifikation ab

```
@GET
@Path("/")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public List<TestDTO> listAllObects(){}
```

Für das Test DTO werden die einzelnen implementierten Methoden folgendermaßen beschrieben:

1. /testdto: Definiert den Basis Pfad für diese Beschreibung des TestDTO APIs
2. Get: wird hier ohne weiteren Pfad dokumentiert für die http GET-Aufruf
3. Für den GET-Aufruf ist bisher nur eine Antwort „200“ als OK dokumentiert
4. Es könnten aber auch weitere Fehlerfälle mit anderen http Error Codes beschrieben werden
5. Als Content für den Response werden sowohl eine JSON als auch eine XML-Version des DTOs beschrieben.
6. Die Antwort ist wie im Java Code angegeben eine Liste, die hier als Array abgebildet ist.
7. Die Elemente des Arrays werden hier als Items beschreiben und mittels \$ref in als DTO

```
paths:
/testdto:
  get:
    tags:
      - Test Dto Resource
    responses:
      "200":
        description: OK
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/TestDTO'
          application/xml:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/TestDTO'
  post:
    tags:
      - Test Dto Resource
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/TestDTO'
    responses:
      "200":
        description: OK
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/TestDTO'
```

### 9.5.1 OpenAPI Path Documentation

Describes the operations available on a single path. A Path Item MAY be empty, due to ACL constraints. The path itself is still exposed to the documentation viewer but they will not know which operations and parameters are available.

Field Name	Type	Description
<b>\$ref</b>	<a href="#">string</a>	Allows for an external definition of this path item. The referenced structure MUST be in the format of a <a href="#">Path Item Object</a> . In case a Path Item Object field appears both in the defined object and the referenced object, the behavior is undefined.
<b>summary</b>	<a href="#">string</a>	An optional, string summary, intended to apply to all operations in this path.
<b>description</b>	<a href="#">string</a>	An optional, string description, intended to apply to all operations in this path. <a href="#">CommonMark syntax</a> MAY be used for rich text representation.
<b>get</b>	<a href="#">Operation Object</a>	A definition of a GET operation on this path.
<b>put</b>	<a href="#">Operation Object</a>	A definition of a PUT operation on this path.
<b>post</b>	<a href="#">Operation Object</a>	A definition of a POST operation on this path.
<b>delete</b>	<a href="#">Operation Object</a>	A definition of a DELETE operation on this path.
<b>head</b> <b>patch</b> <b>trace</b>	<a href="#">Operation Object</a>	A definition of a operation on this path.
<b>servers</b>	<a href="#">[Server Object]</a>	An alternative server array to service all operations in this path.
<b>parameters</b>	<a href="#">[Parameter Object   Reference Object]</a>	A list of parameters that are applicable for all the operations described under this path. These parameters can be overridden at the operation level, but cannot be removed there. The list MUST NOT include duplicated parameters.

See Also:

<https://swagger.io/specification/v3/#info-object>

### 9.5.2 OpenAPI DTO Beschreibung

Die OpenAPI Spezifikation inkludiert auch die Abbildung von Java Objekten auf YAML bzw bidirektional eine Möglichkeit DTOS bereits im YAML zu definieren.

Das Java DTO:

```
@XmlRootElement
public class TestDTO {

    private Long id;
    String name;
    String vorname;
```

Mapped auf die OpenAPI Beschreibung in der Components Sektion wie folgt:

```
592   components:
593     schemas:
594       TestDTO:
595         type: object
596         properties:
597           id:
598             format: int64
599             type: integer
600           name:
601             type: string
602           vorname:
603             type: string
604         securitySchemes:
605           SecurityScheme:
606             type: http
607             description: Authentication
608             scheme: basic
609
```

### 9.5.3 OpenAPI OperationID

```
@Operation(summary = "read a Test DTO Object by ID",
            description = "read a Test DTO Object by ID and return it",
            operationId = "readTestDtoById")
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public TestDTO readObjectById(
    @Parameter(name = "input", description = "The TestDTO Input object to store", required = true,
    allowEmptyValue = false)
    @PathParam("id") String id
){
    log.infov("input {} , objectOutput {}", id, "");
```

```
/testdto/{id}:
  get:
    tags:
      - Test Dto Resource
    summary: read a Test DTO Object by ID
    description: read a Test DTO Object by ID and return it
    operationId: readTestDtoById
    parameters:
      - name: id
        in: path
        description: The TestDTO Input object to store
        required: true
        schema:
          type: string
          allowEmptyValue: false
    responses:
      "200":
        description: OK
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/TestDTO'
```

#### 9.5.4 Open API Query String Parameter

Query string parameters must not be included in paths. They should be defined as query parameters instead.

```
/parameter/queryParameter:  
get:  
  tags:  
    - Parameter Resource  
  parameters:  
    - name: qp  
      in: query  
      schema:  
        type: string  
    - name: qp2  
      in: query  
      schema:  
        format: int64  
        default: "1"  
        type: integer  
  responses:
```

```
// http://localhost:8080/parameter/queryParameter?qp=inputText&qp2=text2  
@GET  
@Path("/queryParameter")  
@Produces(MediaType.TEXT_PLAIN)  
public String queryParameter(  
    @QueryParam("qp") String qp,  
    @DefaultValue("1") @QueryParam("qp2") Long qP2  
){  
    log.infov("log QueryParam: {0}", qp);  
    return "query params [" + qp + "] und [" + qP2 + "]";  
}
```

## 9.5.5 Open API Query Path Parameter

Path Parameter werden mit ihrem Platzhalter im Path angegeben und als Parameter dokumentiert:

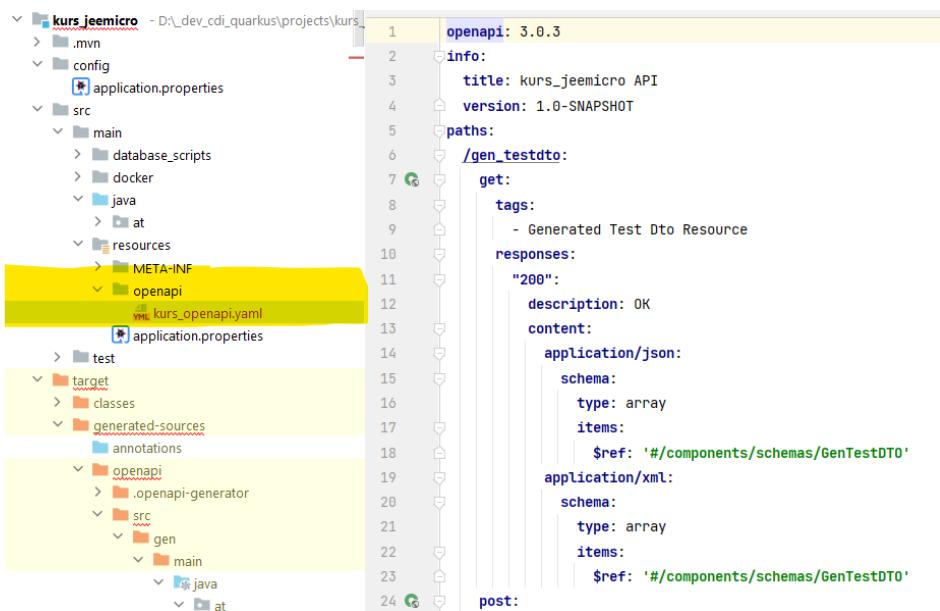
```
/parameter/inputParameter/{inputString}:
get:
tags:
- Parameter Resource
parameters:
- name: inputString
  in: path
  required: true
schema:
  type: string
responses:
"200":
  description: OK
  content:
    text/plain:
      schema:
        type: string
```

```
@GET
@Path("/inputParameter/{inputString}")
@Produces(MediaType.TEXT_PLAIN)
public String inputParameter(
    @PathParam("inputString") String inputString){
    log.infov("log: {0}", inputString);

    StringBuilder sbStr = new StringBuilder();
    sbStr.append(inputString).reverse();
    return sbStr.toString() + "!";
}
```

## 10 OpenAPI – Api First (Generate Server Sources)

Um aus einer OpenAPI Spezifikation den Server Java Code generieren lassen zu können, muss das API YAML File mit eingecheckt werden, um dem Generator zur Verfügung zu stehen.



The screenshot shows a file browser interface with two panes. The left pane displays the project structure:

```

  kurs_jeemicro - D:\_dev_cdi_quarkus\projects\kurs_
  > .mvn
  > config
    application.properties
  > src
    > main
      database_scripts
      docker
      java
        at
      resources
        META-INF
        openapi
          kurs_openapi.yaml
    application.properties
  > test
  > target
  > classes
  > generated-sources
    annotations
    > openapi
      .openapi-generator
      > src
        gen
          main
            java
            at

```

The right pane shows the content of the `kurs_openapi.yaml` file:

```

1  openapi: 3.0.3
2  info:
3    title: kurs_jeemicro API
4    version: 1.0-SNAPSHOT
5    paths:
6      /gen_testdto:
7        get:
8          tags:
9            - Generated Test Dto Resource
10         responses:
11           "200":
12             description: OK
13             content:
14               application/json:
15                 schema:
16                   type: array
17                   items:
18                     $ref: '#/components/schemas/GenTestDTO'
19             application/xml:
20               schema:
21                 type: array
22                 items:
23                   $ref: '#/components/schemas/GenTestDTO'

```

Abbildung 24 OpenAPI Yaml File für Generator

### 10.1 OpenAPI – API First – Maven Konfiguration

Mit dem OpenAPI-generator-Maven-Plugin kann aus dem OpenAPI YAML File der benötigte Source-Code generiert werden, als auch die DTOs.

Zusätzlich wird eine Delegate Klasse generiert, die wir zur Anbindung des vollständig generierten Server Stubs an unsere Server Implementierung verwenden können.

```

<plugins>
  <plugin>
    <groupId>org.openapi.tools</groupId>
    <artifactId>openapi-generator-maven-plugin</artifactId>
    <version>6.4.0</version>
    <executions>
      <execution>
        <goals>
          <goal>generate</goal>
        </goals>
        <configuration>
          <inputSpec>${project.basedir}/src/main/resources/openapi/kurs_openapi.yaml</inputSpec>
          <generatorName>jaxrs-resteasy</generatorName>
          <apiPackage>at.cgsit.jeemicro.openapi.api</apiPackage>
          <modelPackage>at.cgsit.jeemicro.openapi.model</modelPackage>
          <supportingFilesToGenerate>
            ApiUtil.java
          </supportingFilesToGenerate>
          <configOptions>
            <delegatePattern>true</delegatePattern>
            <sourceFolder>src/gen/main/java</sourceFolder>
          </configOptions>
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>

```

```
</execution>
</executions>
</plugin>
```

#### 10.1.1 OpenAPI – maven target directories konfigurieren

Mittels Maven Helper Plugin können die temporären target directories korrekt zum IntelliJ und für Maven in den Classpath hinzugefügt werden.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <version>3.4.0</version>
  <executions>
    <execution>
      <id>add-source</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>add-source</goal>
      </goals>
      <configuration>
        <sources>
          <source>target/generated-sources/annotations</source>
          <source>target/generated-sources/openapi/src/gen/main/java</source>
          <source>target/generated-sources/openapi/src/main/java</source>
        </sources>
      </configuration>
    </execution>
  </executions>
</plugin>
```

#### 10.2 Generierte Struktur mit IntelliJ Source Foldern

Die Generierten OpenAPI Files inkludieren

1. GenTestdtoAPI.java
2. GenTestdtoAPIService.java
3. GenTestDTO.java
4. GenTestdtoApiServiceImpl.java

Die Dateien sind für das IntelliJ Projekt und für Maven als SOURCE Folder sichtbar

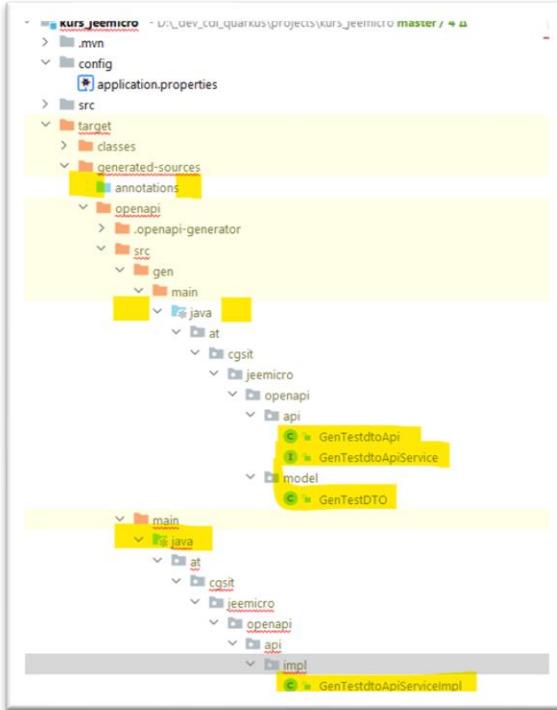


Abbildung 25 Generierte OpenAPI Files

### 10.3 Generierter Java JaxRS Quellcode

Der Quellcode wird für den Server korrekt generiert.

Mittels Generated Service Interface und Implementierung können wir das API-Service Beans implementieren das die Anbindung an unseren manuellen Code erledigt.

```

@Path("/gen_testdto")
@io.swagger.annotations.Api(description = "the gen_testdto API")
@javax.annotation.Generated(value =
"org.openapitools.codegen.languages.JavaResteasyServerCodegen", date = "2023-05-
30T22:30:49.429154200+02:00[Europe/Berlin]")
public class GenTestdtoApi {

    @Inject GenTestdtoApiService service;

    @GET

    @Produces({ "application/json", "application/xml" })
    @io.swagger.annotations.ApiOperation(value = "", notes = "", response = GenTestDTO.class,
responseContainer = "List", tags={ "Generated Test Dto Resource", })
    @io.swagger.annotations.ApiResponses(value =
        @io.swagger.annotations.ApiResponse(code = 200, message = "OK", response = GenTestDTO.class,
responseContainer = "List" ) )
    public Response genTestdtoGet(@Context SecurityContext securityContext)
        throws NotFoundException {
        return service.genTestdtoGet(securityContext);
    }
    @DELETE
    @Path("/{id}")
}

```

#### 10.4 OpenAPI – API First Limitierungen

Mit dem Stand 30.05. kann der openapi generator die import pakete noch nicht für den neuen import pfad generieren

**import** jakarta.ws.rs

Sobald dies behoben ist, kann der Generator entsprechend dem Beispiel genutzt werden.

## 11 Swagger UI

Das Swagger UI ist in der Quarkus Extension via Pfad /q/swagger-ui verfügbar.

Es bietet eine sehr einfache angenehme Möglichkeit für die Entwicklung und den Test zur Visualisierung als auch zum Testen und Aufrufen der Rest APIs einer Deployment Einheit.

<http://localhost:8080/q/swagger-ui/>

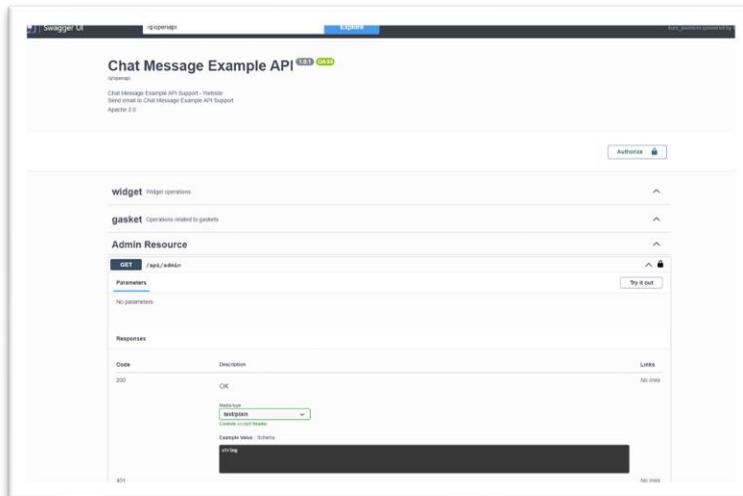


Abbildung 26 Swagger UI

### 11.1 Testing APIs

Das Swagger UI bietet die Möglichkeit direkt die APIs auch aufzurufen, und stellt für die DTOs/Objekte bereits einen Input Vorschlag in JSON zur Verfügung.

Das Beispiel aus dem Quarkus Training:

```
curl -X 'POST' \
'http://localhost:8080/testdto' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
  "id": 0,
  "name": "string",
  "vorname": "string",
  "isOk": true,
  "enumTest": "ENUM1",
  "eventDate": "2022-03-10",
  "ok": true
}'
```

Weiters werden die notwendigen CURL Commands für den Entwickler fertig dargestellt, die für einen Aufruf notwendig sind.

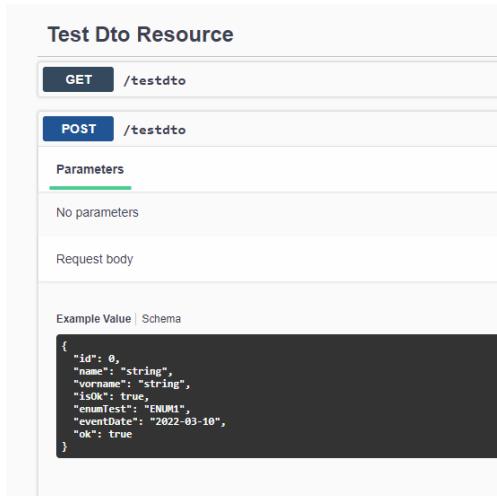


Abbildung 27 Swagger - API Call, Try it Out

## 12 Quarkus Metrics

Quarks includes microprofile.health and metrics as Microprofile Standards. This enables health and metrics checks easily for the Mircro Service. Annotations are provided for easy usage

### 12.1 Metrics Quarkus Maven Extensions

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-metrics</artifactId>
</dependency>
```

### 12.2 Quarkus Metrics Example

The Metrics Example from the JEE Training shows the usage for Counted and Timed Services via Microprofile Metrics extension

```
@Path("/metricsResource")
public class MetricsResorce {

    @Inject
    Logger LOG;

    @GET
    @Path("/chatMessage")
    @Produces(MediaType.APPLICATION_JSON)
    @Counted(name = "chatMessageFindAllCount", description = "How many primality checks have been performed.")
    @Timed(name = "chatMessageFindAllTimer", description = "A measure of how long it takes to perform the primality test.", unit = MetricUnits.MILLISECONDS)
    public List<ChatMessageDTO> chatMessageFindAll() {
        LOG.info("info chatMessage ");
        List<ChatMessageDTO> result = new ArrayList<>();
        return result;
    }
}
```

## 13 Quarkus JMS

```
<dependency>
    <groupId>org.amqphub.quarkus</groupId>
        <artifactId>quarkus-qpid-jms</artifactId>
</dependency>
```

### 13.1 JMS Example

This low level Example shows the basic usage for JMS Communications. Via Connection Factory a JMS Context and Producer or Consumer is created.

This is then used to send or receive data from JMS.

```
@ApplicationScoped
public class PriceProducer implements Runnable {
    @Inject
    ConnectionFactory connectionFactory;

    private final Random random = new Random();
    private final ScheduledExecutorService scheduler =
Executors.newSingleThreadScheduledExecutor();
    void onStart(@Observes StartupEvent ev) {
        scheduler.scheduleWithFixedDelay(this, 0L, 5L, TimeUnit.SECONDS);
    }
    void onStop(@Observes ShutdownEvent ev) {
        scheduler.shutdown();
    }
    @Override
    public void run() {
        try (JMSContext context =
connectionFactory.createContext(JMSContext.AUTO_ACKNOWLEDGE)) {
            context.createProducer().send(context.createQueue("prices"),
Integer.toString(random.nextInt(100)));
        }
    }
}
```

```
@ApplicationScoped
public class PriceConsumer implements Runnable {

    @Inject
    ConnectionFactory connectionFactory;

    private final ExecutorService scheduler =
Executors.newSingleThreadExecutor();

    private volatile String lastPrice;

    public String getLastPrice() {
        return lastPrice;
    }

    void onStart(@Observes StartupEvent ev) {
        scheduler.submit(this);
    }

    void onStop(@Observes ShutdownEvent ev) {
        scheduler.shutdown();
    }

    @Override
    public void run() {
        try (JMSContext context =
connectionFactory.createContext(JMSContext.AUTO_ACKNOWLEDGE)) {
            JMSConsumer consumer =
context.createConsumer(context.createQueue("prices"));
            while (true) {
                Message message = consumer.receive();
                if (message == null) return;
                lastPrice = message.getBody(String.class);
            }
        } catch (JMSEException e) {
            throw new RuntimeException(e);
        }
    }
}
```

## 14 JPA – Java Persistence API – Database Development

JPA is available in Quarkus via JPA/Hibernate and Bean Validation Quarkus Extensions.

To include it simply add the hibernate maven dependency to your project:

```
<!-- Hibernate ORM specific dependencies -->
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-hibernate-orm</artifactId>
</dependency>
```

And decide about the database you want to use:

```
<!-- JDBC driver dependencies -->
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-jdbc-postgresql</artifactId>
</dependency>
```

### 14.1 JPA Introduction

JPA as an Architecture always requires an implementation. Quarkus includes Hibernate as standard ORM Implementation. Finally, a JDBC-Driver is required to connect to a database.

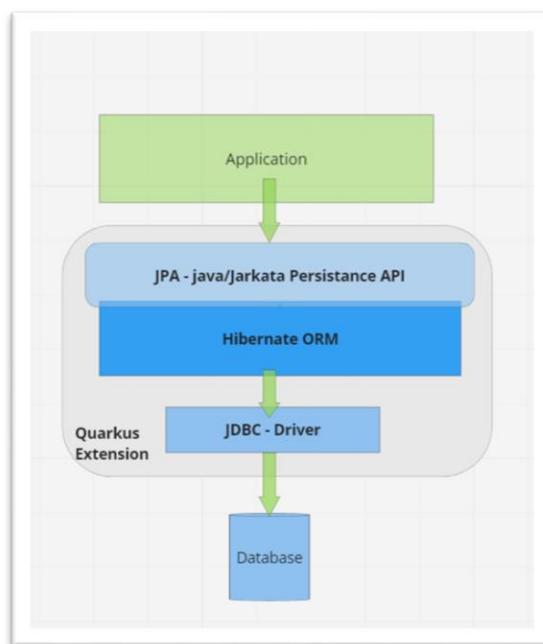


Figure 1 JPA Architecture

- Java Persistence API or now **Jakarta Persistence API**
- JPA provides Java developers with an object/relational mapping facility for managing relational data in Java applications.
- JPA itself is just a specification, not a product, it cannot perform persistence or anything else by itself.
- JPA is a set of interfaces and requires an implementation.

- There are open-source and commercial JPA implementations to choose from and any Java EE 5 application server should provide support for its use.
- JPA also requires a database to persist to

## 14.2 JPA Framework Components

The major elements of JPA are:

- The Java Persistence API
- Object/Relational Mapping Metadata
- The Query Language
- The Java Persistence Criteria API

## 14.3 The JPA API

The JPA API includes:

- JPA specifies interfaces, Annotations and Patterns for OR-Mapping
- JPA enables a more Object-Oriented View to store objects to a relational database.
- Within the standard it also defines Entity Mapping Annotations
- It also defines a Query Language (JPQL) and
- a programmatic Query API.

## 14.4 Modules and APIs used with JPA.

- Java Bean Validation Framework
  - Integrates into JPA for additional bean validations during JPA operations.
- JTA – Java Transaction API
  - Transaction Support via also Annotations for JPA
- Hibernate
  - Is often used in Quarkus as an Implementation for JPA
- JDBC
  - JPA/Hibernate always connects to the database via a JDBC-Driver implementation.

## 14.5 The JPA Entity Manager

The Main-API to JPA is provided via Entity Manager Interface.

The API:

- allows to manage Entities by providing all required.
  - CRURD (create, read, update delete) methods.
  - as well as search and query api utility
- Getting an Instance for this API is simply done via @Inject

### 14.5.1 The Entity Manager API

The Entity Manager API provides some central major API-Methods to handle Entities:

Method	Description
em.persist(Object entity)	Make an instance managed and persistent It will be attached to the EM Session and persisted during commit
Update ?	<ul style="list-style-type: none"> <li>• An Entity is updated whenever you modify an Entity which is attached to a session, so loaded from the entity manager inside an active transaction</li> <li>• An Entity/Table can also be updated by directly using update statement queries</li> </ul>
<T> T em.merge(T entity)	If an Entity was loaded via EM but the Entity Manager and Transaction is not active anymore, the Entity gets the status „DETACHED“. The Return value is the new attached merged Entity for the current EM Session
em.remove(Object entity)	Remove the entity instance from the Session, which leads to a db delete
em.find(Class<T> entityClass, Object primaryKey)	Find by primary key. Search for an entity of the specified class and primary key

Figure 2 JPA Entity Manager API

## 14.6 JPA – JTA – Transactions

To work with a database for update/insert/delete operations a Database Transaction is required.

Within Quarkus and JPA the DB Transaction is used indirectly via JTA Transactions which is a higher level concept used with JPA to implement transactional support

- The Java Transaction API specified by JSR 707
- The API specifies Interfaces and Annotations to demarcate transaction boundaries.
- Also defines an API to directly deal with the Transaction-Manager

### 14.6.1 JTA – Different Transaction Scopes

Annotation	Description
@Transactional	Annoation to define TX Boundaries automatically Default Scope is : REQUIRED
@TransactionScoped	standard CDI scope to define bean instances whose lifecycle is scoped to the currently active JTA transaction
@Transactional( Transactional.TxType.REQUIRED)	Same as default. A Transaction is required. If no
Transactional.TxType. <b>REQUIRES_NEW</b>	Always will create a new transaction for this method, even if another one is already active. If another TX is active the other TX will be suspended
Transactional.TxType. <b>SUPPORTS</b>	Participate in an existing TX or call the method without any additional TX. If you have read only methods todo you can use it
Transactional.TxType. <b>MANDATORY</b>	Mandatory is like Supports but if no TX is active a TransactionalException is thrown
Transactional.TxType. <b>NOT_SUPPORTED</b>	If the caller/client has a TX active, it is suspended before processing continuos
Transactional.TxType. <b>NEVER</b>	If the client is running in a TX context. A TransactionalException is thrown

Figure 3 JTA Transaction Scopes

## 14.7 JPA – JTA –Programmatic TX-Management

The JTA Transaction can also be used programmatically. User Transaction can also be manually injected, and also controlled manually.

```
public void insertChatMessageManualTX(ChatMessageEntity
newObject) throws SystemException, NotSupportedException {
    utx.begin();
    try {
        em.persist(newObject);
        utx.commit();
    } catch (Exception e) {
        utx.rollback();
        throw new RuntimeException("tx error ",e);
    }
}
```



As a rule of thumb use annotations  
if no specific requirement needs to be fulfilled

## 14.8 JPA Entities

JPA models the Entities by Annotating POJOS with JPA Annotations to define the mapping to the Database in a neutral way independently of

- An entity represents a table in a relational database, and each
- entity instance corresponds to a row in that table
- An entity is a lightweight persistence domain object
- The primary programming artifact of an entity is the entity class
- The class must be annoyed with **java.persistence.Entity** annotation .
- The class must not be declared final.
- The class must implement the Serializable interface.
- Entities may extend both entity and non-entity classes, and nonentity classes may extend entity class.

- Persistent instance variable must be declared private, protected, or package private.

#### 14.9 Entity Annotations

The most important Entity Annotations are:

Annotation	Description
@Entity	The POJO annotated with @Entity becomes a persistent object.
@Table	Specifies a primary database table (name) for this entity
@SecondaryTable	Enables the storage into multiple tables for one Entity
@Embeddable	Annotates a pojo used within Entities as an embedded class
@Inheritance @DiscriminatorValue @PrimaryKeyJoinColumn	Enables the Entity to use Java Inheritance and map it correctly to one or multiple database tables.

#### 14.10 Entity Column Annotations

For Field/Properties there are those major Annotations

Annotation	Description
@Column	Specifies the mapped column for a property in the Entity Class. Normally annotated on the property itself
@Id	Markes a property as a primary key column
@GeneratedValue	Used with ID columns to specify how the value is generated. @GeneratedValue(strategy=GenerationType. <i>SEQUENCE</i> )
@Basic	The simplest type of mapping to a database column. Used especially with EAGER or LAZY loading flag
@Enumerated	Annotates the mapping for java enumerations by ordinal or String mapping. <i>ORDINAL</i> is the default currently
@Temporal	Used for date, time and timestamp mappings: @Temporal(DATE) protected java.util.Date endDate;
@Lob	for BLOBs /CLOB mappings
@Transient	A transient field is not persisted at all

Figure 4 Entity Field Annotations

#### 14.11 Entity Mapping Example

Annotations Example for an Entity based on the Training Chat Message Example

- Easy mapping for any Pojo into a Table
- If the @Table annotation is not used it defaults to the entity name
- NotBlank is already added as an example for bean validation framework usage

```

@Entity
@Table( name="chat_message")
public class ChatMessageEntity {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name="user_name", length= 100, nullable = false)
    private String userName;

    @NotBlank(message="ChatRoom may not be blank")
    @Column(name="chat_room", length= 50, nullable = true)
    private String chatRoom;

    ...
}

```

Figure 5 Entity Example

## 14.12 JPA Entity Relationship Mappings

Annotation	Description
@OneToOne	1:1 relation to another Entity
@OneToMany	1:n relation to another entity used as annotation on the first „1“ side of the relation
@ManyToOne	Defines the N-side for a 1:n Relation (is a reference to the primary key of the target)
@ManyToMany	Used for n:m relations and used on both sides
@JoinColumn	Additional information for join columns

Figure 6 Entity Relations Mapping

### 14.12.1 Entity Many to Many Example

- Both sides use `@ManyToMany as Annotation`
- The join table is specified on the owning side (customer)
- See [ManyToMany Java Documentation](#) for more examples

```
// In Customer class:
@ManyToMany
@JoinTable(name="CUST_PHONES")
public Set<PhoneNumber> getPhones() { return phones; }

// In PhoneNumber class:
@ManyToMany(mappedBy="phones")
public Set<Customer> getCustomers() { return customers; }
```

See Javadoc for ManyToMany Relation and Specification for more information and details.

#### 14.13 JPA Query Language

- Provides a SQL Like Query Language but against the mapped entity objects instead of the Database
- It provides select, Join, Where, order, group by and having support.
- See the specification and tutorials for more details

Example:

```
SELECT e from ChatMessageEntity e " +
"WHERE e.chatRoom like
:chatMessageLike " +
"ORDER by e.creationTime DESC
```

The Basic Syntax for the JPQL can be seen here:

```
SELECT <select clause>
FROM <from clause>
[WHERE <where clause>]
[ORDER BY <order by clause>]
[GROUP BY <group by clause>]
[HAVING <having clause>]
```

#### 14.14 JPA - JPQL - Query API

- Enables a programmatic approach to queries by using a Criteria Builder
- and Expressions for the select and where conditions
- Is more save for refactoring

```
public Long countChatMessages() {
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Long> query = cb.createQuery(Long.class);
    Root<ChatMessageEntity> from = query.from(ChatMessageEntity.class);

    Expression<Long> count = cb.count(from);
    query.select(count);

    // ca.where(/*your stuff*/);

    TypedQuery<Long> query1 = em.createQuery(query);

    Long singleResult = query1.getSingleResult();

    return singleResult;
}
```

#### 14.15 Quarks JPA Configuration

- JPA Configuration is simplified via Quarkus to minimal settings
- The db-kind desides on a database type
- to use a database you need to import the driver maven dependency for the jdbc diver
- Datasource url and password

```
# datasource configuration
quarkus.datasource.db-kind = postgresql

quarkus.datasource.username = sc_admin
quarkus.datasource.password = sc_admin

quarkus.datasource.jdbc.url =
    jdbc:postgresql://localhost:5432/simplechat

quarkus.hibernate-orm.database.generation=drop-and-create
# quarkus.hibernate-orm.database.generation=update

quarkus.hibernate-orm.log.sql=true
quarkus.hibernate-orm.log.bind-parameters=true
```



To see the generated SQL use ORM SQL logging parameters, which are also simplified a bit compared to hibernate direct logging configuration



#### 14.16 Chapter Summary: JPA

This Chapter described JPA Concepts and various aspects for using JPA and mapping entities to a database, as well as how to query and process data with JPA.

##### 14.16.1 Additional Material

See: [JPA](#)

## 15 Quarkus – Testing

The Quarkus Test Framework provides a very simple and effective Test Framework for the Development and Test Team.

It Includes:

- Simple Container based Testing Unit + CDI + Quarkus
- Component Tests via @QuarkusTest
- Integration tests via @QuarkusIntegrationTest
- Simple testing for CDI Beans, Rest und DB Objekten
- Support for RestAssured for Rest API Tests

### 15.1 Quarkus Test Dependencies

Include Quarkus Junit5 and Rest Assured to the maven test scope, to be able to perform tests.

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-junit5</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <scope>test</scope>
</dependency>
```

### 15.2 Quarkus Test Example

A simple Quarkus test just Annotates the Class with @QuarkusTest. This enables the Junit-5 Tests with full Quarkus CDI Container support. But the Container initializes only the required minimal dependencies.

The Example shows how to implement a Quarkus test.

Using the Annotation is all you need to do, to be able to @Inject the Logger and the ChatMessage Repository for Database Access.

After this step it is already possible to test the DB Layer without further initialization.

Via application configuration and @Alternatives some Beans can be replaced, but the other beans can be used like in a fully started Quarkus Microservice Container.

```

@QuarkusTest
class ChatMessageRepositoryTest {

    @Inject
    Logger log;

    @Inject
    ChatMessageRepository cmRepository;

    @Test
    void readChatMessage() {
        ChatMessageEntity cmEntity =
cmRepository.readChatMessage(1L);
        assertNotNull(cmEntity);
        log.info("cmresult" + cmEntity.toString());
    }
}
  
```

### 15.2.1 Quarkus Fluent API Rest Assured Test

The same applies to additionally used Rest-Assured framework tests

Using **@TestHTTPEndpoint** Annotation will set all base paths for this test class to the base of the Example Resource

**@Path("/hello")**

Using the Rest Assured framework provides a fluent API to simplify tests in a way like shown here.

The .get() configuration can be left aside because of the TestHttpEndpoint base configuration already.

```

@QuarkusTest
@TestHTTPEndpoint(ExampleResource.class)
public class ExampleResourceTest {

    @Test
    public void testHelloEndpoint() {
        given()
            .when()
            // .get("/hello")
            .get()
            .then()
            .statusCode(200)
            .body(is(notNullValue()));
            // .body(is("Hello RESTEasy"));
    }
}
  
```

### 15.3 Rest Assured Tests – Basic Introduction

Rest Assured Tests have this basic concept and API Idea:

```

Given() .
    param("x", "y") .
    header("z", "w") .
when() .
Method() .
Then() .
    statusCode(XXX) .
    body("x", "y", equalTo("z"));
  
```

Code	Explanation
Given()	'Given' keyword, lets you set a background, here, you pass the request headers, query and path param, body, cookies. This is optional if these items are not needed in the request
When()	'when' keyword marks the premise of your scenario. For example, 'when' you get/post/put something, do something else.
Method()	Substitute this with any of the CRUD operations(get/post/put/delete)
Then()	Your assert and matcher conditions go here

Figure 7 Rest Assured Basic Testing Concepts

For Further Details and Information see also

<https://rest-assured.io/>

#### 15.4 Quarkus Testing Concepts with Object DTO Usage

Quarkus Rest Services using Rest Assured also allows to insert the DTO to the POST Body like shown in this example:

```

@Test
@Disabled
public void createChatMessagePost() {
    ChatMessageDTO chatMessageDto = new ChatMessageDTO();

    RequestSpecification httpRequest = given() // .header("Authorization",
    "Bearer " + token)
        .header("Content-Type", "application/json");

    RequestSpecification rs = httpRequest.body(chatMessageDto);

    Response response = rs.post("/chatMessageResource/chatMessage");

    //Fetching the response code from the request and validating the same
    System.out.println("The response code - " + response.getStatusCode());
    assertEquals(response.getStatusCode(), 200);
}
}
  
```

Extracting a Response into a DTO is also possible like this:

```

@QuarkusTest
class ChatMessageResourceTest {

    @Test
    @Disabled
    public void readChatMessage() {

        ResponseBody body =
            given()
                .when().get("/chatMessageResource/chatMessage/10")
                .then().extract().response().body();

        ChatMessageDTO dto = body.as(ChatMessageDTO.class);
        assertNotNull(dto);
    }
}

```

## 15.5 Quarkus Testing Security

Testing Security with Rest Assured and Quarkus tests is directly possible in Rest Quarkus tests.

The Framework including Jaxrs and Security will be setup correctly.

Testing Security should always be with positive AND negative outcome tests like in this example, to ensure that you get denied, if you have no permissions for this user in this role.

```

@Test
void shouldNotAccessUserWhenAdminAuthenticated() {
    given()
        .auth().preemptive()
        .basic("admin", "admin")
        .when()
        .get("/api/users/me")
        .then()
        .statusCode(HttpStatus.SC_FORBIDDEN);
}

```

## 16 Quarkus – Bean Validation

The Bean Validation Framework us used to validate Data by using Annotations or Validation-Methods or Classes to ensure valid objects.

This is usually done in Rest-APIs or for JPA Entities to be validated when received or stored. Validation can also be used programmatically. Furthermore it can be used to ensure valid Method input or return Values.

The Bean Validation APIs and annotations are defined in the dependency Jakarta.validation. For Quarkus we use the Import

```

<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-hibernate-validator</artifactId>
</dependency>

```

### 16.1 Bean Validation Annotations

All available Annotations can be found in the package “jakarta.validation.constraints”

Annotation	Applies to	Description
@Constraint		Marks an annotation to be a bean Validation Constraint
@Valid		Marks a method parameter, return type or property for validation
@Null - @NotNull	Object	Null or not Null Validation
@NotBlank	CharSequence	The annotated element must not be null and must contain at least one non-whitespace character
@NotEmpty	CharSequence, Collection, map, array	Must not be null or empty
@Min - @Max	BigDecimal, short, int, long And there Object Wrappers	Used for numbers to validate lower or equal or higher or equal to the specified value
@DecimalMin, @DecimalMax , @Digits	BigDecimal, short, int, long And there Object Wrappers	The annotated element must be a number whose value must be lower or equal to the specified maximum or minimum.
@Size	CharSequence, Collection, Map, arrays	The annotated element size must be between the specified boundaries (included).
@Furture - @Past - @FutureOrPresent	Calendar, Date	The annotated element must be an instant, date or time in the future/past
@AssertTrue - @AssertFalse	Boolean, boolean	The annotated element must be true. Supported types are boolean and Boolean.  null elements are considered valid !
@Pattern	CharSequence	must match the specified regular expression. The regular expression follows the Java regular expression conventions see java.util.regex.Pattern
@Email	CharSequence	The String has to be a “well formed” email address



Ein Java-String implementiert auch eine CharSequence

## 16.2 Bean Validation Example

Usage Examples can be found in the Example Repository package “at.cgsit.jeemicro.bean\_validation”

```
package at.cgsit.jeemicro.bean_validation;  
  
import jakarta.validation.constraints.*;
```

```

public class BVTestObject {

    @NotEmpty(message = "name may not be empty")
    @Pattern(regexp = "[a-zA-Z0-9]*", message = "name must be alphanumeric")
    private String name;

    @NotNull
    private String description;

    @Future
    private Calendar futureDate;

    @Email
    private String email;

    @Min(0)
    @Max(100)
    private int percent;

    @PositiveOrZero
    @Digits(integer = 5, fraction = 2) // 5 digits in total, 2 after the decimal point
    private BigDecimal amount;

    @AssertTrue(message = "this chat message is not allowed. because of user name")
    public boolean isChatMessageAllowed() {
        if("chris".equalsIgnoreCase(this.name)) {
            return false;
        }
        return true;
    }
}

```

### 16.3 Manual/Programmatic Validator Usage

Bean Validator can be used programmatically also like this:

```

@QuarkusTest
class BVTestObjectTest {

    @Inject
    Validator validator;

    @Test
    void isChatMessageAllowed() {
        Set<ConstraintViolation<BVTestObject>> validate
            = validator.validate(createTestObject());

        validate.forEach(v -> System.out.println("failed: " + v.getPropertyPath() + " message: " +
v.getMessage()));
        assertEquals(2, validate.size());
    }
}

```

So simply inject an Instance to the Bean Validator Interface. And then call **validator.validate()** on the Object Instance which should be validated.

As a return value a Set of Constraint Violations is returned. It can be empty, but if validation errors where detected, they can be used for processing or Exception Handling.

## 16.4 Cascading Validation

Basically, Bean Validation does not cascade into other POJO Relations. If @Valid Annotation is used on a referenced single or List Object, the validation is cascaded to those referenced objects also

```
@NotNull @Valid  
private BVTestObject2 referenceObject;
```

### 16.4.1 Fail Fast and further Configuration

`"quarkus.hibernate-validator.fail-fast"` When fail fast is enabled the validation will stop on the first constraint violation detected.

## 16.5 Chapter Summary: JPA

This Chapter described howto use java bean validation with simple annotations and custom validation Annotations

### 16.5.1 Additional Material

Java Bean Validation:

<https://javaee.github.io/tutorial/bean-validation001.html>

## 17 Quarkus – Service 2 Service Communication

Quarkus Rest Client enables the simple connection to other Rest Services as shown in the Training Examples:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-rest-client-reactive</artifactId>
</dependency>
```

### 17.1 Quarkus

`@RegisterRestClient` enables a simple connection to other Rest services

MicroProfile Rest Client automatically creates a client for the Rest Services and Provides this Interface to be able to be injected as Bean to call the other Rest-Service correctly.

```
@Path("/createinfo")
@Produces(MediaType.TEXT_PLAIN)
@RegisterRestClient()
public interface CreateInfoProxy {

    @GET
    String createNumber();

    @GET
    @Path("/ext")
    @Produces(MediaType.APPLICATION_JSON)
    @Counted(name = "ciproxy_ext_counter", description = "How many primality checks have been performed.")
    @SimplyTimed(name = "ciproxy_ext_timer", description = "A measure of how long it takes to perform the primality test.", unit = MetricUnits.MILLISECONDS)
    @Retry(delay = 3, delayUnit = ChronoUnit.SECONDS, maxRetries = 3)
    @Fallback(value = SimpleFallback.class)
    @Timeout(value = 200, unit = ChronoUnit.MILLISECONDS)
    SimpleDTO createNumber2();

}
```

To use this client proxy implementation, inject the API and use it

```
@Path("/callproxy")
public class CallProxyResource {

    @Inject
    @RestClient
    CreateInfoProxy cIPProxy;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String callProxy() {

        String number = cIPProxy.createNumber()
```

## 18 Postman Testing

Postman bietet support für einen schnellen Test von Rest APIS. Und ermöglicht es Test Sets zu erstellen die verschiedene Requests auch durchtesten.

Für eine professionelle Verwendung ist ein Postman Account notwendig. Damit können die Test-sets auch entsprechend gespeichert werden.

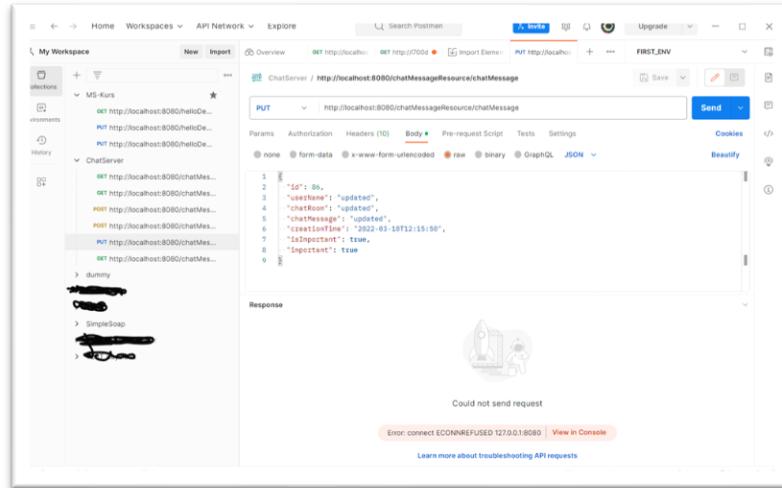


Figure 8 Postmain Hauptmaske

Hier in der Hauptmaske sind links die gespeicherten Requests als auch ein Request sichtbar

- Eingabe von URL, HTTP-Methode
- Parameter & Header
- In den einzelnen TABS unter dem URL möglich
- Hier PUT Request zu
- <http://localhost:8080/chatMessageResource/chatMessage>
- In der Content Sektion das JSON das mitgesendet wird

### 18.1 Postman Variablen

Variablen können im Pfad und auch im JSON Input für den Request platziert werden

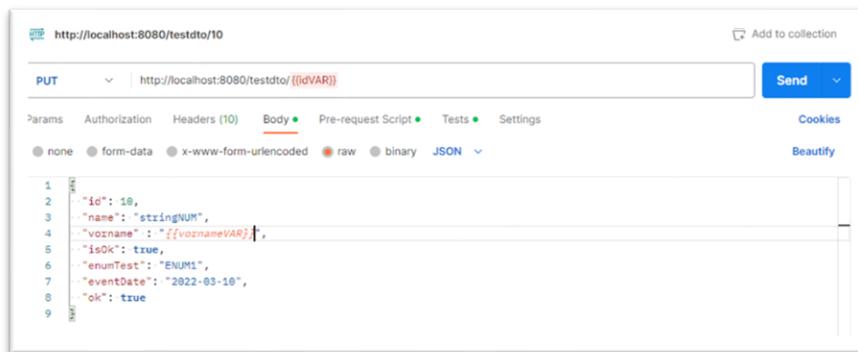


Figure 9 Post man Variablen

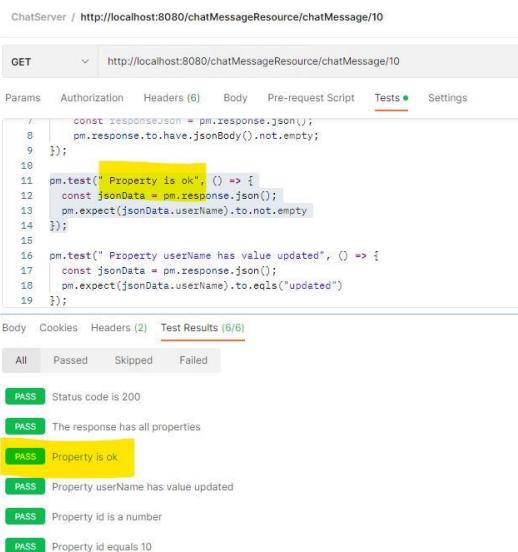
Variablen können dann im TAB PRE-Request Script befüllt werden



Figure 10 Postman Pre Request Variablen

## 18.2 Postman Response Tests

Mittels Tests können die ausgeführten Requests bzw ihre Ergebnisse überprüft werden



```
const responseJson = pm.response.json();
pm.response.to.have.jsonBody().not.empty;
};

pm.test("Property is ok", () => {
  const jsonData = pm.response.json();
  pm.expect(jsonData.userName).to.not.empty
});

pm.test(" Property user Name has value updated", () => {
  const jsonData = pm.response.json();
  pm.expect(jsonData.userName).to.eql("updated")
});
```

Test Result	Status	Message
All	PASS	Status code is 200
	PASS	The response has all properties
	PASS	Property is ok
	PASS	Property user Name has value updated
	PASS	Property id is a number
	PASS	Property id equals 10

## Beispiele für Request Test Scripts

```
pm.test("Status code is 200", () => {
  pm.expect(pm.response.code).to.eql(200);
});

pm.test("The response has all properties", () => {
  //parse the response JSON and test three properties
  const responseJson = pm.response.json();
  pm.response.to.have.jsonBody().not.empty;
});

pm.test(" Property id equals 10 ", () => {
  const jsonData = pm.response.json();
  pm.expect(jsonData.id).not.null;
  pm.expect(jsonData.id).to.eql(10);
});

pm.test(" Property name is ok ", () => {
  const jsonData = pm.response.json();
  pm.expect(jsonData.id).not.null;
  pm.expect(jsonData.name).to.eql("stringNUM");
});

pm.test(" Property name is ok ", () => {
  const jsonData = pm.response.json();
  pm.expect(jsonData.name).to.eql("stringNUM");
});
```

### 18.3 Postman OpenAPI – Editor

Postman mit Account erlaubt es OpenAPI Spezifikationen zu importieren, Exportieren und sie auch zu Editieren. Dabei ist ein Validator hilfreich und erleichtert das Bearbeiten von OpenAPI Files.

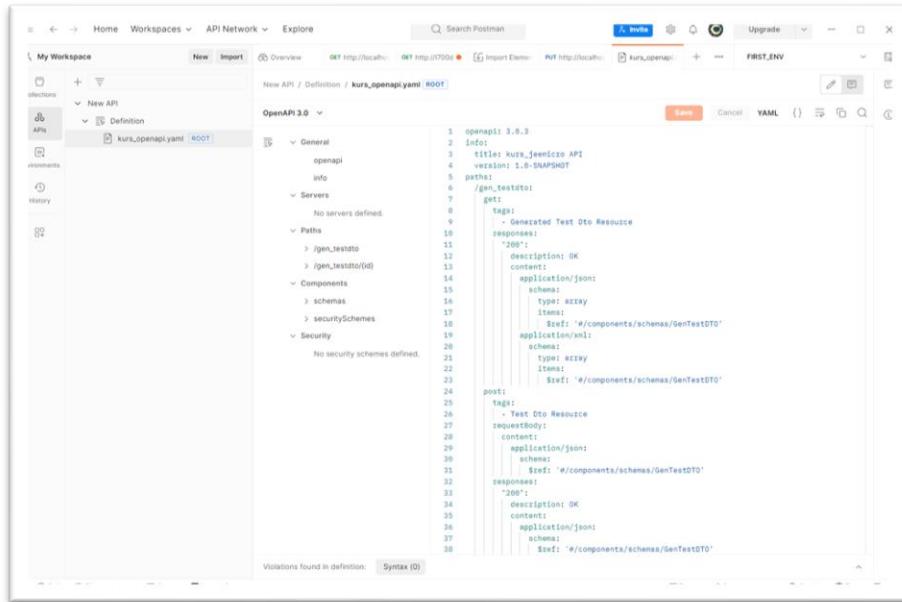


Figure 11 Postman OpenApi Editor

### 18.4 Postman weiterführende Literatur

- Postman  
<https://www.postman.com/>
- Postman Tools  
<https://www.postman.com/product/tools>

## 19 IBM Open Liberty

### 19.1 Create Initial Project for Open Liberty

Open Liberty bietet einen Wizard an um das Startprojekt entsprechend schnell generieren zu können.

<https://openliberty.io/start/>

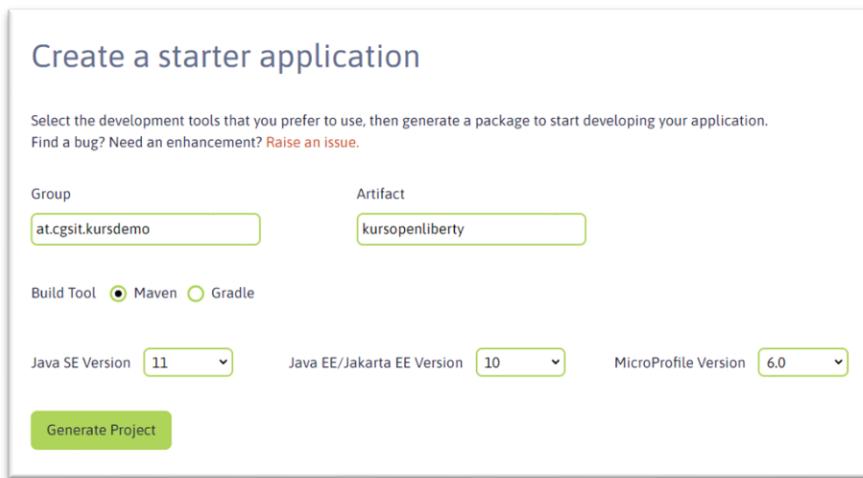


Abbildung 28 Open Liberty Create Project

Siehe auch:

<https://openliberty.io/guides/getting-started.html>

### 19.2 Open Liberty – Server Startup

Wie auch Quarkus bietet OpenLiberty die notwendigen und ähnlichen Server Startup Maven Commands an.

### 19.3 Open Liberty Swagger Console

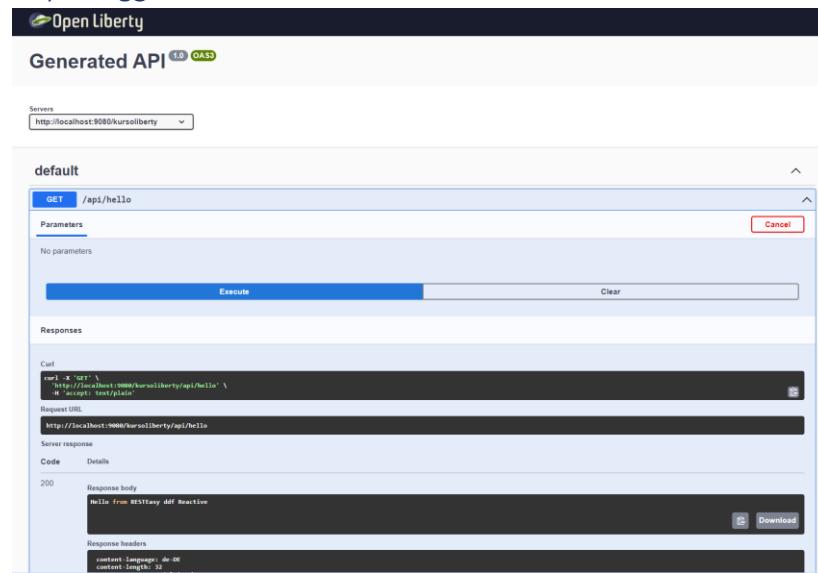


Abbildung 29 Open Liberty Swagger

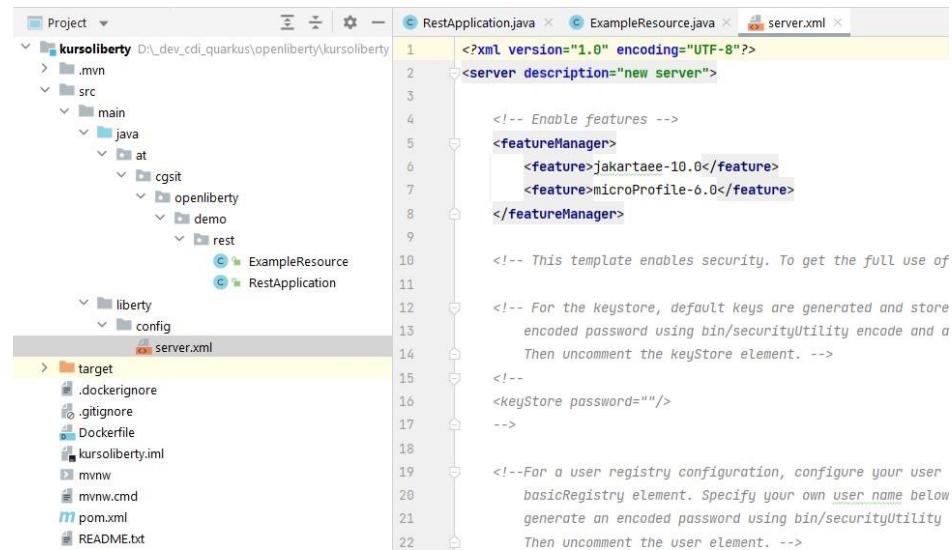
```

    liberty:clean
    liberty:compile-jsp
    liberty:configure-arcillian
    liberty:create
    liberty:debug
    liberty:deploy
liberty:dev
    liberty:devc
    liberty:display-ord
    liberty:dump
    liberty:generate-features
    liberty:help
    liberty:install-feature
    liberty:install-server
    liberty:java-dump
    liberty:package
    liberty:prepare-feature
    liberty:run
    liberty:start
    liberty:status
    liberty:stop
    liberty:test-start
    liberty:test-stop
    liberty:undeploy
liberty:uninstall-feature 31:9080/openapi/
    libt_host: http://localhost:9080/libm/api/
    libt_host: http://localhost:9080/openapi/ui/
    libt_host: http://localhost:9080/health/
    libt_host: http://localhost:9080/metrics/
    libt_host: http://localhost:9080/lat/
    libt_host: http://localhost:9080/kursoliberty/
  
```

t nach 5,680 Sekunden gestartet.

Wie man hier sieht, stellt Open Liberty auch die Einstiegs URLs FÜR Swagger, Application und Metrics zur Verfügung.

## 19.4 Open Liberty - Server Configuration



The screenshot shows a Java project named "kursoliberty" in an IDE. The project structure on the left includes ".mvn", "src" (containing "main" and "java" packages), "target" (containing ".dockerignore", ".gitignore", "Dockerfile", "kursoliberty.iml", "mvnw", "mvnw.cmd", "pom.xml", and "README.txt"), and "config" (containing "server.xml"). The "server.xml" file is open in the editor on the right, showing XML code for server configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<server description="new server">
    <!-- Enable features -->
    <featureManager>
        <feature>jakartaee-10.0</feature>
        <feature>microProfile-6.0</feature>
    </featureManager>
    <!-- This template enables security. To get the full use of
        security, uncomment the KeyStore element. -->
    <!--
        <keyStore password="" />
    -->
    <!--For a user registry configuration, configure your user
        basicRegistry element. Specify your own user name below
        generate an encoded password using bin/securityUtility
        Then uncomment the user element. -->

```

## 20 Appendix

### 20.1 Abbildungen

Abbildung 1 Multigrained Architecture (Quelle Gartner) .....	8
Abbildung 2 Macro vs Micro Services (Quelle Gartner) .....	9
Abbildung 3 Micro Profile Standard Components.....	11
Abbildung 4 IoC .....	12
Abbildung 5 Microservices & API Gateway .....	15
Abbildung 6 Was ist Kubernetes, K8s .....	16
Abbildung 7 Quarkus Architecture .....	17
Abbildung 8 Quarkus Maven Extension .....	18
Abbildung 9 Spring DI Support .....	18
Abbildung 10 CDI Bean Beispiel .....	23
Abbildung 11 CDI Annotations .....	24
Abbildung 12 Quarkus Swagger Security Login .....	38
Abbildung 13 Rest Webservice Communication .....	39
Abbildung 14 REst Services Definition.....	39
Abbildung 15 Rest Service – Methods.....	41
Abbildung 16 Exmple Ticket System API.....	41
Abbildung 17 Path Parameter .....	44
Abbildung 18 Query Parameters .....	44
Abbildung 19 @Produces and @Consumes .....	46
Abbildung 20 Quarkus maven dependency for Jackson.....	46
Abbildung 21 OpenAPI Datentypen .....	52
Abbildung 22 OpenAPI Yaml File für Generator .....	59
Abbildung 23 Generierte OpenAPI Files.....	61
Abbildung 24 Open Liberty Create Project.....	86
Abbildung 25 Open Liberty Swagger .....	87
Abbildung 26 DevOps .....	<b>Fehler! Textmarke nicht definiert.</b>
Abbildung 27 Dev Ops Pipeline .....	95
Abbildung 28 Quelle Wikipedia .....	<b>Fehler! Textmarke nicht definiert.</b>
Abbildung 29 GIT Feature Branches .....	97
Abbildung 30 Apache Maven Ablauf .....	98
Abbildung 31 Apache Maven Modul und Verzeichnisse .....	100
Abbildung 32 Redirecting Java API to SLF4j.....	103
Abbildung 33 Log4J Log levels .....	103
Abbildung 34 Log Appender .....	104

## 20.2 Source Code Demos

### 20.2.1 JPA Implementation for JPQL Query

```
/*
 * suche alle chat messages and order by date descending
 * @param likeStatement
 * @return
 */
@Transactional
public List<ChatMessageEntity> findChatMessagesWithLikeNameAndOrdedByDate(String likeStatement) {
    Query query = em.createQuery(
        "SELECT e from ChatMessageEntity e " +
        "WHERE e.chatRoom like :chatMessageLike " +
        "ORDER by e.creationTime DESC" );
    if( !StringUtil.isNullOrEmpty(likeStatement)) {
        query.setParameter("chatMessageLike", likeStatement);
    } else {
        query.setParameter("chatMessageLike", "%");
    }
    List resultList = query.getResultList();
    return resultList;
}
```

### 20.2.2 JPA Implementation for Criteria Builder:

```
@Transactional
public List<ChatMessageEntity>
findChatMessagesWithLikeNameAndOrdedByDateWithQueryBuilder(String likeStatement ) {
    // "SELECT e from ChatMessageEntity e WHERE e.chatRoom like :chatMessageLike ORDER by
    // e.creationTime DESC"
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<ChatMessageEntity> criteriaQuery = cb.createQuery(ChatMessageEntity.class);
    Root<ChatMessageEntity> from = criteriaQuery.from(ChatMessageEntity.class);

    criteriaQuery.select( from );

    Predicate like = cb.like(from.get("chatRoom"), likeStatement);
    criteriaQuery.where( like );

    Order orderByCT = cb.desc(from.get("creationTime"));
    criteriaQuery.orderBy(orderByCT );

    Query query = em.createQuery(criteriaQuery);
    List<ChatMessageEntity> resultList = query.getResultList();

    return resultList;
}
```

### 20.2.3 Injection Demo Bean

```

@RequestScoped
public class SimpleCDIBean {

    @Inject
    Logger log;

    @PostConstruct
    public void postConstruct() {
        log.info("postConstruct called ");
    }

    public String echo(String input) {
        log.info("SimpleCDIBean");
        return echoReverse(input);
    }

    public String echoReverse(String input) {
        StringBuilder inputSB = new StringBuilder();
        StringBuilder reverse = inputSB.append(input).reverse();
        return reverse.toString();
    }
}
  
```

### 20.2.4 Appendix: OpenAPI Yaml für Test DTO

```

openapi: 3.0.3

info:
  title: kurs_jeemicro API
  version: 1.0-SNAPSHOT

paths:
  /testdto:
    get:
      tags:
        - Test Dto Resource
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/TestDTO'
  
```

```
application/xml:  
  schema:  
    type: array  
    items:  
      $ref: '#/components/schemas/TestDTO'  
  
post:  
  tags:  
    - Test Dto Resource  
  
  requestBody:  
    content:  
      application/json:  
        schema:  
          $ref: '#/components/schemas/TestDTO'  
  
  responses:  
    "200":  
      description: OK  
      content:  
        application/json:  
          schema:  
            $ref: '#/components/schemas/TestDTO'  
  
/testdto/{id}:  
  get:  
    tags:  
      - Test Dto Resource  
  
    summary: read a Test DTO Object by ID  
    description: read a Test DTO Object by ID and return it  
  
    parameters:  
      - name: id  
        in: path  
        description: The TestDTO Input object to store  
        required: true  
        schema:  
          type: string  
          allowEmptyValue: false  
  
    responses:  
      "200":
```

```
description: OK
content:
  application/json:
    schema:
      $ref: '#/components/schemas/TestDTO'

put:
  tags:
    - Test Dto Resource
  parameters:
    - name: id
      in: path
      required: true
  schema:
    type: string
  requestBody:
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/TestDTO'
  responses:
    "200":
      description: OK
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/TestDTO'

delete:
  tags:
    - Test Dto Resource
  parameters:
    - name: id
      in: path
      required: true
  schema:
    type: string
  responses:
```

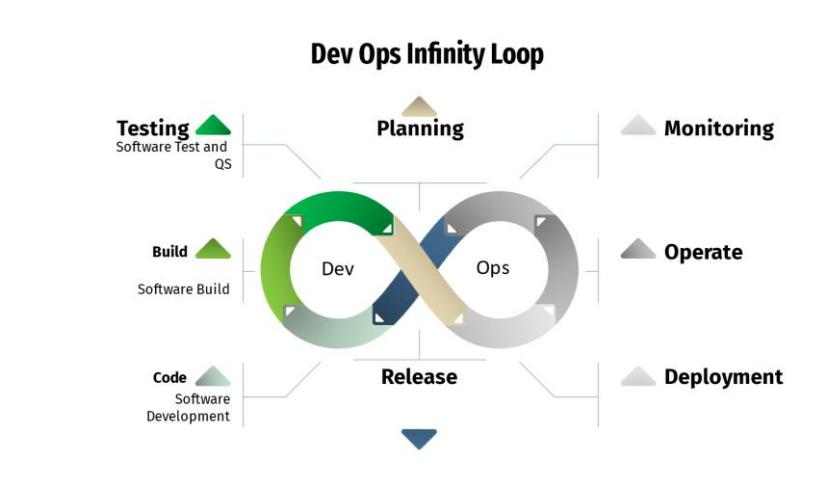
```
"200":  
  description: OK  
  content:  
    text/plain:  
      schema:  
        type: string  
  
components:  
  schemas:  
    TestDTO:  
      type: object  
      properties:  
        id:  
          format: int64  
          type: integer  
        name:  
          type: string  
        vorname:  
          type: string  
  
  securitySchemes:  
    SecurityScheme:  
      type: http  
      description: Authentication  
      scheme: basic
```

### 20.3 Exkurs : DevOps

Microservices werden mit starkem DevOps Fokus entwickelt.

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.



### 20.3.1 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 eines klassischen Java/git Pipeline wird in der Praxis eher in mehrere einzelne Pipelines mit jeweils eigenen Steps unterteilt werden.

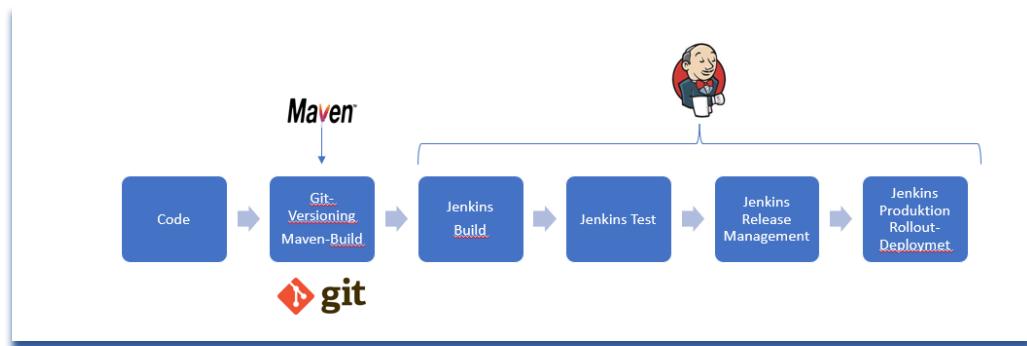


Abbildung 30 Dev Ops Pipeline

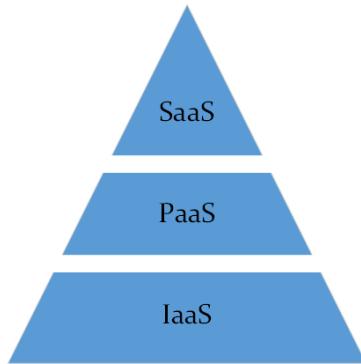
## 20.4 Exkurs : Cloud Computing

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](https://de.wikipedia.org/wiki/Cloud_Computing)

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



## 20.5 GIT Branching Beispiele mit klassischem Ansatz

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

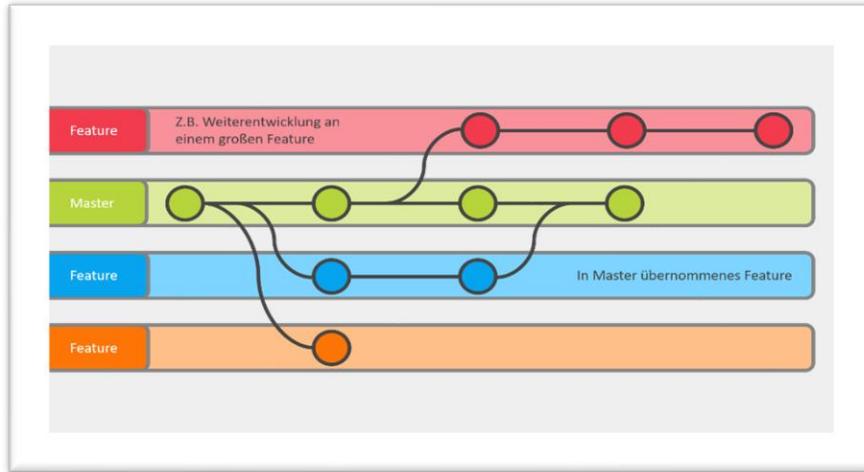


Abbildung 31 GIT Feature Branches

### Beispiel 2:

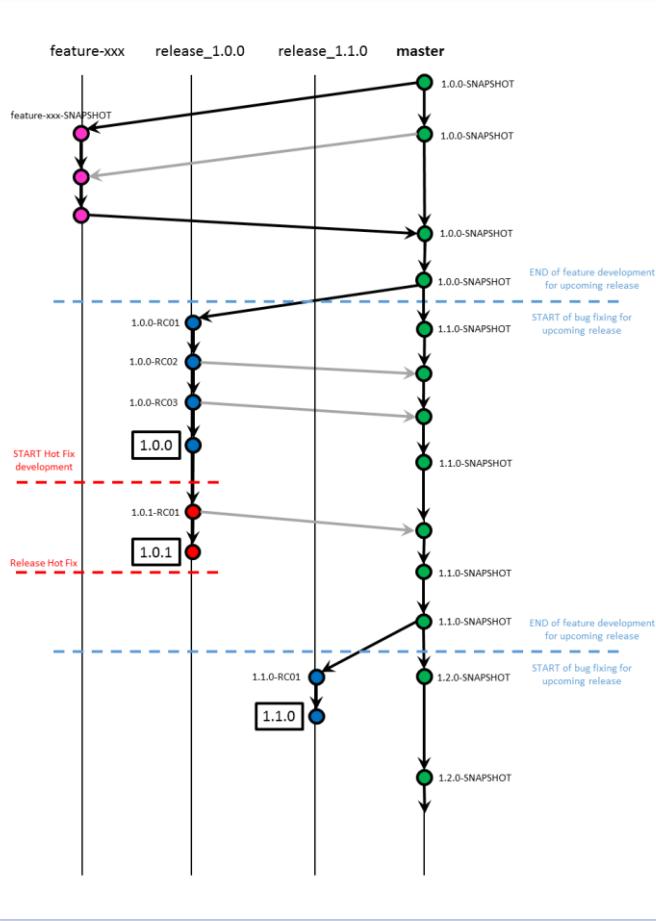


Figure 12 GIT Branching Modell

## 20.6 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:

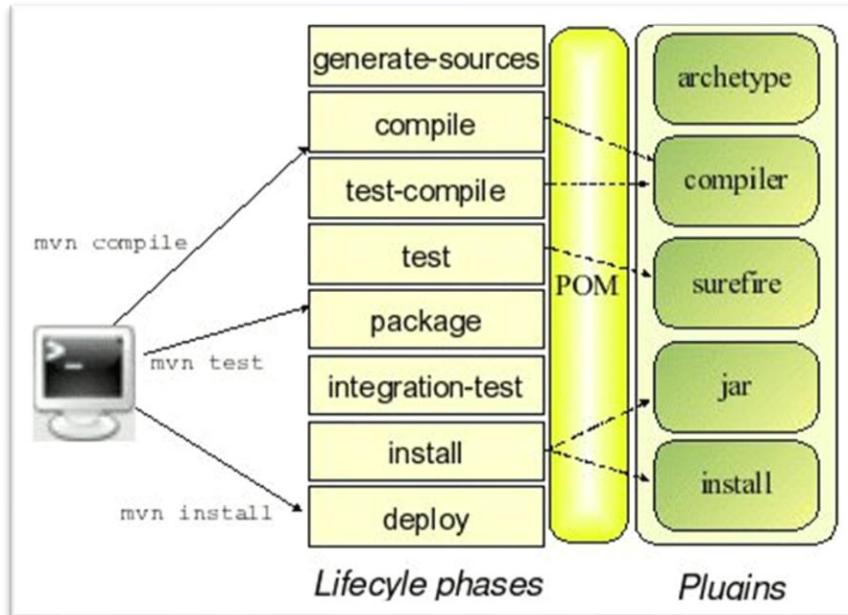
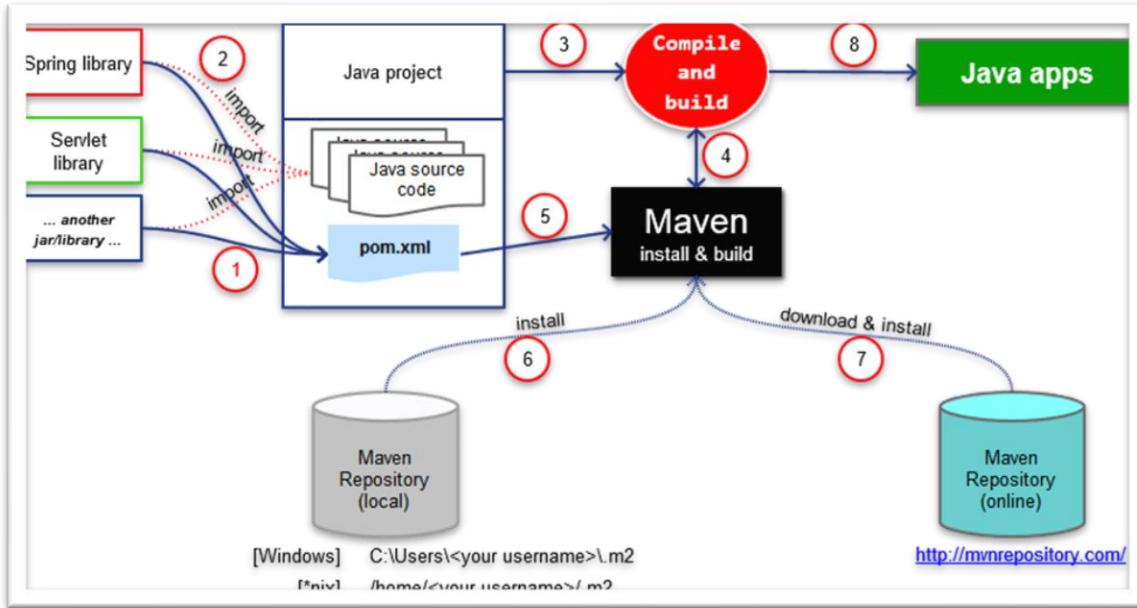


Abbildung 32 Apache Maven Ablauf

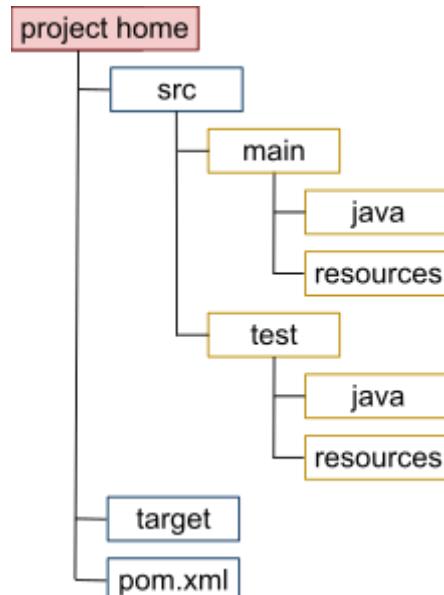
### 20.6.1 Apache Maven – Dependency Auflösung und Download

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



### 20.6.2 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:

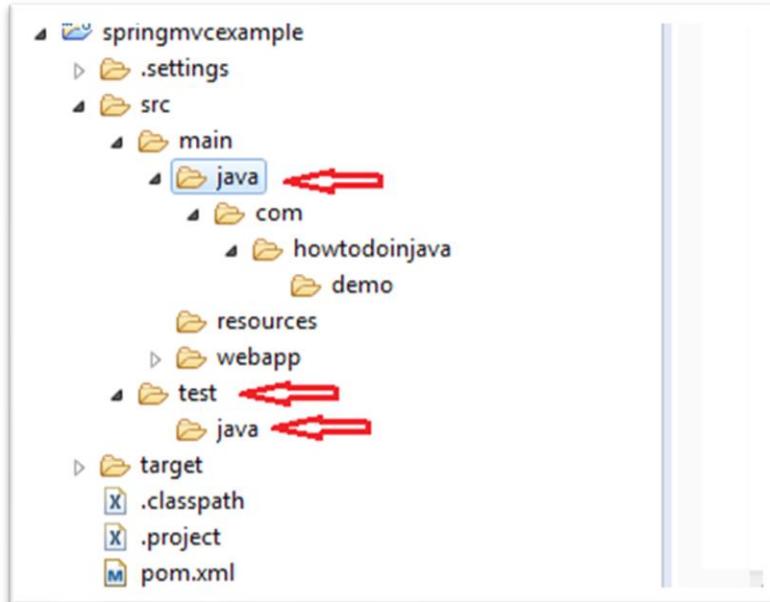
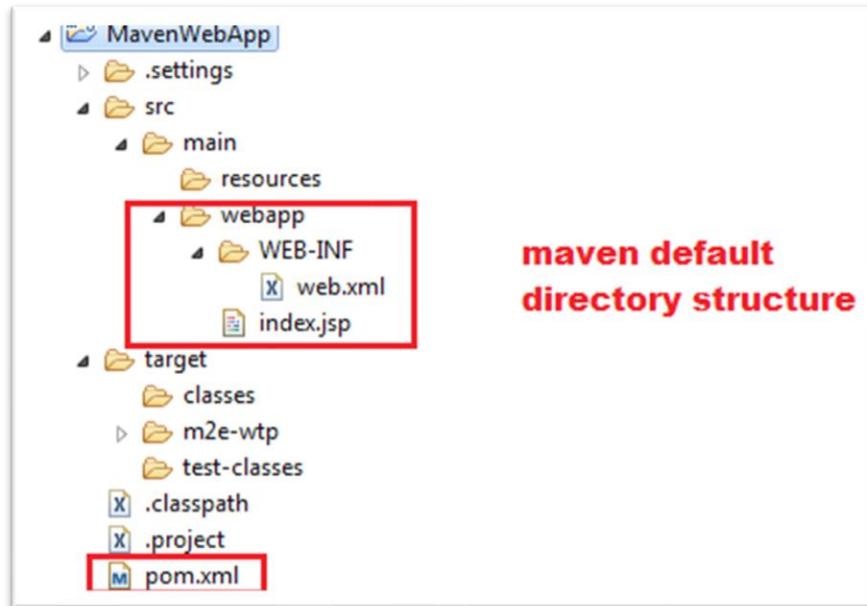


Abbildung 33 Apache Maven Modul und Verzeichnisse

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

### 20.6.3 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.



## 20.7

### 20.7.1 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.

## 20.8 Das Logging System

### 20.8.1 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.

### 20.9 Logging Frameworks 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.

## 20.10 SLF4J

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

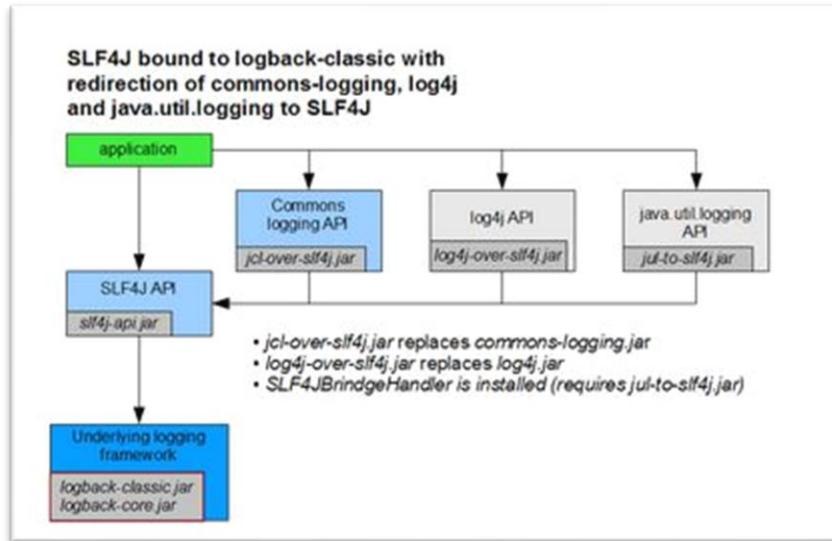


Abbildung 34 Redirecting Java API to SLF4J

## 20.11 Log Level & Output Appender

### 20.11.1 Log Levels

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

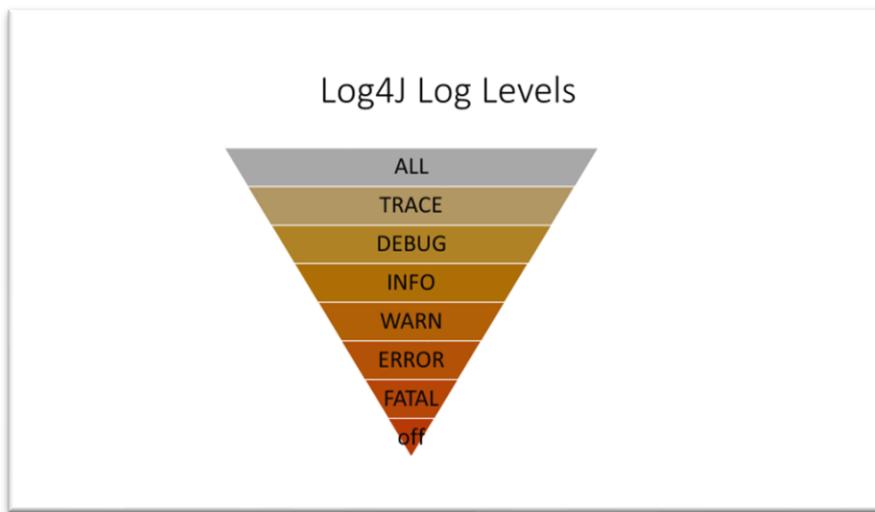


Abbildung 35 Log4J Log levels

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.

## 20.11.2 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 ein Problem da damit Meldungen sehr leicht verloren gehen können.

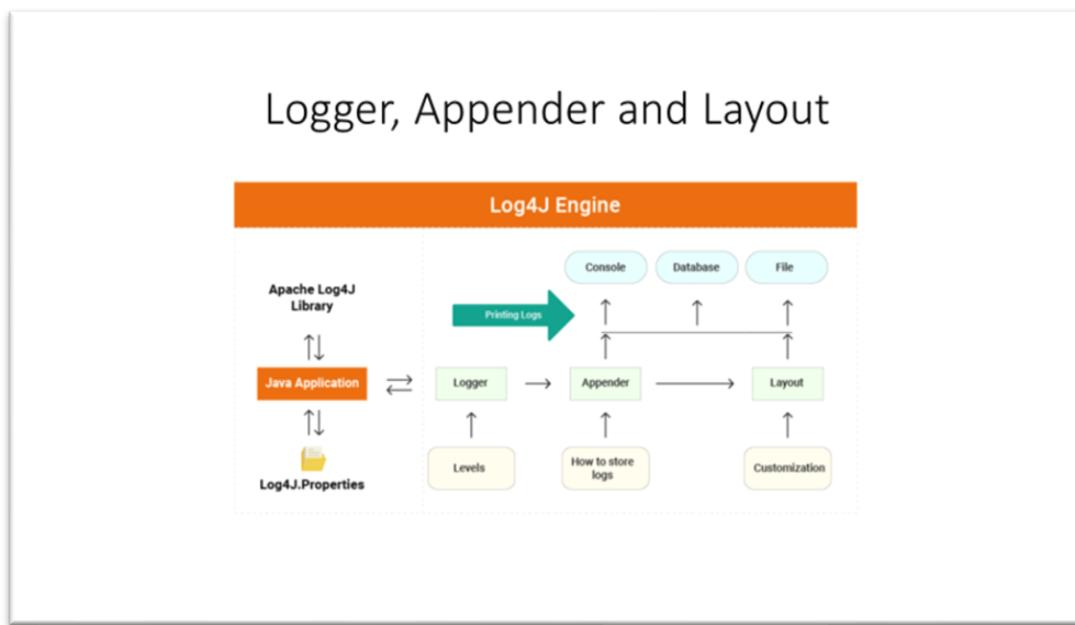


Abbildung 36 Log Appender

## 20.12 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>
```

## 20.13 Begriffe und Grundlagen und Links

## 21 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>>.

### 21.1 Allgemeine Links

- Micro Services  
<https://de.wikipedia.org/wiki/Microservices>
- Domain Driven Design (Eric Evans 2003)  
[https://www.goodreads.com/book/show/179133.Domain\\_Driven\\_Design](https://www.goodreads.com/book/show/179133.Domain_Driven_Design)

### 21.2 Quarkus Dokumentation

- Quarkus Homepage  
<https://quarkus.io>
- Quarkus Configuration Properties  
<https://quarkus.io/guides/config-reference#property-expressions>

### 21.3 Rest Services und Open API

- Open API  
<https://www.openapis.org>
- OpenAPI Specification  
<https://swagger.io/specification/>
- HTTP Specification  
<https://www.rfc-editor.org/rfc/rfc9110.html#name-method-definitions>
- Rest Services:  
[https://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)
- JaxRS Rest Services  
[https://de.wikipedia.org/wiki/Jakarta\\_RESTful\\_Web\\_Services](https://de.wikipedia.org/wiki/Jakarta_RESTful_Web_Services)
- JCP JSR 311: JAX-RS: The Java API for RESTful Web Services  
<https://jcp.org/en/jsr/detail?id=311>
- Rest Easy  
[https://docs.jboss.org/resteasy/docs/6.2.2.Final/userguide/html\\_single/index.html](https://docs.jboss.org/resteasy/docs/6.2.2.Final/userguide/html_single/index.html)
- Cheat Sheet für Rest easy Annotations  
<https://www.mastertheboss.com/jboss-frameworks/resteasy/jax-rs-cheatsheet/>

- MIME Types  
[https://en.wikipedia.org/wiki/Media\\_type](https://en.wikipedia.org/wiki/Media_type)
- 

## 21.4 Java Standards und Spezifikationen

- Context and Dependency Injection 2.0  
<https://www.jcp.org/en/jsr/detail?id=365>
- Jakarta CDI  
<https://jakarta.ee/specifications/cdi/2.0/cdi-spec-2.0.html>

## 21.5 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>
- Inversion of Control (IoC)  
[https://de.wikipedia.org/wiki/Inversion\\_of\\_Control](https://de.wikipedia.org/wiki/Inversion_of_Control)

## 21.6 JPA

- Oracle: Java Persistence API:  
<http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html>
- The Java Persistence API - A Simpler Programming Model for Entity Persistence:-  
<http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html>
- JPA Annotation Reference:-  
<http://www.oracle.com/technetwork/middleware/ias/toplink-jpaannotations-096251.html>
- Hibernate Reference Documentation:-  
<http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/>
- Java Bean Validation:  
<https://javaee.github.io/tutorial/bean-validation001.html>
-

## 21.7 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/B09C2VLVFF>

## 21.8 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>

## 21.9 Internet Standards

- http Protokoll  
Siehe Wikipedia und IETF ORG Standard Pages
  - [https://de.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](https://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol)
  - <https://datatracker.ietf.org/doc/html/rfc9113>
- Mime Types  
[https://en.wikipedia.org/wiki/Media\\_type](https://en.wikipedia.org/wiki/Media_type)
- RFC3339 - Date and Time on the Internet: Timestamps  
<https://datatracker.ietf.org/doc/html/rfc3339>

## 21.10 Dev-Ops

- [https://en.wikipedia.org/wiki/DevOps\\_toolchain](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>

#### 21.10.1 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/>

#### 21.10.2 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/>

#### 21.10.3 GIT

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

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