TECHNISCHE
HOCHSCHULE
LÜBECK

Bachelor-Thesis

Automatisierung von Datenbanken in DevOps-Szenarien

Vorgelegt von: Arne Bruhns

Fachbereich: Elektrotechnik und Informatik

Studiengang: Informatik / Softwaretechnik

Erstprüfer/in: Professor Dr. Kratzke

Ausgabedatum: 20. Mai 2020
Abgabedatum: 20. August 2020

A Hanemann

(Professor Dr. Andreas Hanemann)
Vorsitzender des Prüfungsausschusses

Aufgabenstellung:

Innerhalb von Continuous Integration und Delivery (CI/CD) Prozessen gibt es Integrationstests, die Datenbanken (DB) benötigen. Damit solche Tests automatisiert durchgeführt werden können, sind Schnittstellen zu DB-Management-Systemen (DBMS) erforderlich, um DBen

- dynamisch erzeugen,
- mit Testdaten befüllen
- und löschen zu können.

Damit diese Schnittstelle universell eingesetzt werden kann, ist zu untersuchen, ob diese mittels einem DBMS Controller (Server mit REST-basierter API) und Build-System-spezifischen Plugins (Client) konzipiert werden kann. Bei der Konzeption der REST-basierten API des DBMS Controllers und erforderlicher Plugins sind folgende Aspekte zu berücksichtigen:

Unterschiedliche Datenbank-Systeme sind geeignet zu berücksichtigen (der Nachweis kann in der Arbeit anhand von zwei geeigneten Typvertretern erfolgen).
Unterschiedliche Build-Systeme sind geeignet zu berücksichtigen (der Nachweis kann in der Arbeit anhand von zwei geeigneten Typvertretern erfolgen).

- DBen müssen angelegt und gelöscht werden können.
- DB-Zustände (Schema + Daten) müssen geeignet verglichen werden können.
- DB-Zustände (Schema + Daten) müssen verwaltet werden können (erzeugen, klonen, zurücksetzen, auflisten, löschen, etc.).

Das Problem soll anhand des Fallbeispiels von Jenkins-getriggerten Integrationstests in Continuous Delivery-Builds des Praxispartners AEB untersucht, analysiert und (prototypisch) gelöst werden.

Prof. Dr. Kratzke

TECHNISCHE HOCHSCHULE LÜBECK

Fachbereich
Elektrotechnik und Informatik
Dekanat - Prüfungsausschuss

Erklärung zur Abschlussarbeit

Ich versichere, dass ich die Arbeit selbständig, ohne fremde Hilfe verfasst habe.

Bei der Abfassung der Arbeit sind nur die angegebenen Quellen benutzt worden. Wörtlich oder dem Sinne nach entnommene Stellen sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, insbesondere dass die Arbeit Dritten zur Einsichtnahme vorgelegt oder Kopien der Arbeit zur Weitergabe an Dritte angefertigt werden.

10.08.2020 — (Datum)

Bruhns — Unterschrift

Danksagung

Ich möchte mich bei Herrn Prof. Dr. rer. nat. Dipl.-Inform. Kratzke für die Betreuung meiner Bachelorarbeit sowie für die sehr schnelle Beantwortung aller meiner Fragen bedanken. Die Anmerkungen und Hinweise zu den Zwischenständen waren immer hilfreich und präzise gestellt.

Ebenso gilt mein Dank den Mitarbeitern des Unternehmens AEB SE, die ein angenehmes Arbeitsumfeld vorgewiesen haben. Insbesondere möchte hier mein Team Software Infrastructure am Standort Lübeck, bestehend aus Herrn Schubert, Herrn Seidel und Herrn Harn, hervorheben, die mir ein Thema für meine Bachelorarbeit angeboten und mich in der Zeit als Werkstudent in die Grundlagen eingearbeitet haben. Während der Bearbeitung der Bachelorarbeit habe ich jederzeit Tipps und konstruktives Feedback zu meinen Fragen, Problemen und Zwischenständen erhalten.

Weiterhin möchte ich mich bei Herrn Breuker für die Bereitschaft, die Position des Zweitprüfers zu übernehmen, bedanken.

Ich möchte mich bei meiner Familie bedanken, die mich während meines gesamten Studiums unterstützt und an mich geglaubt hat.

Zum Schluss möchte ich mich bei allen Personen bedanken, die die schriftliche Ausarbeitung meiner Bachelorarbeit Korrektur gelesen und Anmerkungen dazu gemacht haben.

Zusammenfassung der Arbeit / Abstract of Thesis

Fachbereich: **Department:**	Elektrotechnik und Informatik
Studiengang: **University course:**	Informatik / Softwaretechnik
Thema: **Subject:**	Automatisierung von Datenbanken in DevOps-Szenarien
Zusammenfassung:	In der vorliegenden Arbeit wird eine Software entwickelt, die es ermöglicht, automatisiert und mit wenig Konfigurationsaufwand Datenbanken, die für die Integrationstest innerhalb von Continuous Integration und Continuous Delivery Prozessen benötigt werden, anzulegen und zu verwalten. Es wird untersucht, inwiefern die Software universell eingesetzt werden kann. Universell heißt in diesem Kontext, dass die Software mit möglichst unterschiedlichen Datenbank-Systemen und Build-System-spezifischen Plugins interagieren kann.
Abstract:	In this thesis a software is developed, which allows to create and manage databases, which are needed for integration testing within Continuous Integration and Continuous Delivery processes, automatically and with little configuration effort. It is investigated to what extent the software can be used universally. Universal in this context means that the software can interact with as many different database systems and build system-specific plug-ins as possible.
Verfasser: **Author:**	Arne Bruhns
Betreuender ProfessorIn: **Attending professor:**	Prof. Dr. rer. nat. Dipl.-Inform. Nane Kratzke
WS /SS	SS 2020

Inhaltsverzeichnis

1 Einleitung

1.1 Hintergrund

Mit der Digitalisierung hat der Bereich IT und Softwareentwicklung in den letzten Jahren einen starken Zuwachs erhalten. Stetig wird neue Software entwickelt und zum Einsatz gebracht. Dabei steigt die Anforderung, dass sich die Qualität der neuen Software verbessert und die Entwicklungszeit verkürzt. Um diesen hohen Anforderungen gerecht zu werden, etablieren sich neue Vorgehensweisen in der Softwareentwicklung.

Eine neue Methodik ist der DevOps-Ansatz[1]. Mit diesem Ansatz soll die Zusammenarbeit von bereichsübergreifenden Teams[2] in einem Unternehmen, die alle an einem Produkt bzw. an einer Software beteiligt sind, optimiert werden. Das bedeutet, dass die Kooperation zwischen den einzelnen Beteiligten an einem Softwareprodukt stark angeregt wird. Der Vorteil bezüglich der engen Zusammenarbeit ist, dass die Geschwindigkeit der Entwicklung einer Software steigt. Zudem unterstützt der DevOps-Ansatz den Ansatz der agilen Softwareentwicklung[3], wodurch die Flexibilität in der Entwicklung erhöht wird und auf Veränderungen an den Anforderungen schneller reagiert werden kann.

Damit die Softwareprodukte dem Anspruch der stetig steigenden Qualität gerecht werden können, ist es essenziell wichtig, dass diese ausreichend getestet werden. Damit aber auch der Anspruch an eine kurze Entwicklungszeit und eine schnelle Auslieferung erfüllt werden kann, müssen wiederkehrende Aktionen im Testzyklus automatisiert werden.

[1] DevOps beschreibt die Verschmelzung zwischen Entwicklung (eng.: *Development*) und IT-Betrieb (eng.: IT *Operations*) [36].
[2] Damit sind alle Teams der Bereiche Entwicklung und (IT-)Betrieb gemeint.
[3] Die agile Softwareentwicklung bezeichnet den Ansatz, die Flexibilität und Transparenz im Entwicklungsprozess zu erhöhen, wodurch ein schnellerer Einsatz der entwickelten Software ermöglicht wird [38].

1.2 Ziele der Arbeit

In dieser Arbeit soll untersucht werden, ob und auf welche Art und Weise es möglich ist, Integrationstests von (neuen) Softwaremodulen[4], die Datenbanken benötigen, zu automatisieren, indem zur Testlaufzeit automatisiert Datenbanken angelegt und manipuliert werden. Dazu ist eine Software zu entwickeln, die eine Schnittstelle bereitstellt, mit der Build-System-spezifische-Plugins (Clients) mit einem Server kommunizieren können, der anhand der ihm übermittelten Befehle, Datenbanken anlegt und ggf. mit Testdaten befüllt. Des Weiteren soll der Server Funktionalitäten aufweisen, mit der die Zustände der jeweiligen Datenbank zurückgesetzt, aufgelistet, geklont und gelöscht werden können. Zusätzlich soll der Server ermöglichen, dass Datenbanken unterschiedlicher Datenbankmodelle angelegt und verwaltet werden können.

Der Nachweis, dass die Software universell - d.h. mit unterschiedlichen Clients und mit unterschiedlichen Datenbank-Systemen - agieren kann, soll wie folgt stattfinden: Es werden je zwei Typvertreter von Build-System-spezifischen Plugins und von Datenbank-Systemen gewählt und analysiert. Auf Grundlage der Ergebnisse aus der Analyse soll eine entsprechende Architektur entwickelt und umgesetzt werden. Ziel ist es, die Architektur so zu konzipieren, dass diese leicht erweiterbar ist, um weitere Typvertreter anbinden zu können.

Es ist kein grundlegendes Ziel, dass die Software am Ende im Unternehmen eingesetzt werden kann, jedoch hat die Systemintegration seitens der AEB SE einen Einfluss auf jegliche Entwurfsentscheidungen.

[4] Ein Softwaremodul ist ein Baustein einer Software. Der Baustein stellt dabei eine in sich funktional geschlossene Einheit dar und einen bestimmten Dienst oder Teildienst zur Verfügung.

1.3 Ist-Zustand der AEB SE

Dieses Kapitel befasst sich damit, aus welchem Grund es für die AEB SE interessant und wichtig ist, sich mit der Problematik dieser Thesis zu beschäftigen. Es werden die zwei Hauptgründe näher betrachtet, aus der die Motivation seitens der AEB SE entstand.

Entlastung des DevOps-Teams:

Grundsätzlich werden neue Features und Erweiterungen eines Softwareproduktes auf einem Featurebranch[5] entwickelt. Bevor dieser Branch mit den neuen Funktionalitäten in den Masterbranch[6] integriert wird, werden Unit- und Integrationstest durchgeführt. Spätestens bei den Integrationstests wird eine Anbindung zu einer Datenbank benötigt, sofern das Softwareprodukt mit einer Datenbank interagiert. Bisher war es so, dass die Entwicklerteams manuell eine Datenbank bereitgestellt und verwaltet haben. Mit der hier zu entwickelnden Software soll dieser Schritt nun entfallen. Die Datenbanken sollen mit geringem Konfigurationsaufwand automatisiert angelegt, gelöscht und verwaltet werden können.

Automatisierte Bereitstellung von Datenbanken für kundenspezifische Zwecke:

Während der Entwicklung eines Softwareproduktes ist es wichtig, Kunden Zwischenstände zu präsentieren, um mögliche Missverständnisse u.a. durch ein unterschiedliches Verständnis der Aufgabenstellung frühzeitig zu beheben. Damit sich aber der Kunde nicht nur in einer passiven Rolle befindet, ist es auch wichtig, dass er den Zwischenstand seiner in Auftrag gegebenen Software selbst testet. Hier ist es ebenfalls bisher so, dass der Entwickler eine temporäre Datenbank für den Kunden manuell bereitstellen muss. Auch in diesem Fall soll die hier zu entwickelnde Software diesen manuellen Schritt übernehmen.

[5] Das Konzept des Featurebranch wird eingesetzt, damit neue Features einer Software auf einem zum Masterbranch dedizierten Branch entwickelt werden. Treten Fehler bei der Entwicklung neuer Features auf, so haben diese keinen Einfluss auf den Zustand des Masterbranch [37].
[6] Der Masterbranch ist der Standardbranch, der sich jederzeit in einem produktionsbereiten Zustand befindet [37].

1.4 Gliederung des weiteren Dokuments

Es folgt ein kurzer Überblick über den weiteren Aufbau der Arbeit:

Kapitel 2 (Grundlagen)

In diesem Kapitel werden die Grundlagen, die für das Verständnis der Arbeit erforderlich sind, umfassend erläutert. Dabei werden nur Themen vermittelt, die direkten Einfluss in die Erstellung der Arbeit und in der Entwicklung hatten.

Kapitel 3 (Anforderungsanalyse)

Alle Anforderungen an die Software werden vorab ermittelt und spezifiziert. In diesem Kapitel werden diese Anforderungen dargestellt, erklärt und miteinander in Zusammenhang gebracht.

Kapitel 4 (Systemarchitektur)

Dieses Kapitel befasst sich mit der Systemarchitektur der Software. Es wird ein Einblick in die grobe Aufteilung der Software und die Integration in die bestehenden Systeme gegeben.

Kapitel 5 (Softwarearchitektur und Entwurfsentscheidungen)

Dieses Kapitel befasst sich mit der Softwarearchitektur der Software und baut auf den Beschreibungen aus dem vorigen Kapitel auf. Es werden alle technischen Aspekte, die für die Implementierung relevant sind, näher beschrieben. Zusätzlich wird auf die getroffen Entwurfsentscheidungen näher eingegangen.

Kapitel 6 (Implementierung)

In Kapitel 6 wird die Implementierung beschreiben. Der Fokus liegt dabei auf der Umsetzung der wichtigen Entwurfsentscheidungen.

Kapitel 7 (Nachweisführung)

Dieses Kapitel beschäftigt sich mit der Nachweisführung anhand einer zur Anfangszeit gewählten Nachweisstrategie.

Kapitel 8 (Evaluation)

In Kapitel 8 wird die implementierte Software hinsichtlich der Anforderungen evaluiert. Dies wird exemplarisch an den Tests und an einem Beispielprojekt gezeigt.

Kapitel 9 (Fazit)

Die Ergebnisse werden in einem Fazit reflektiert. Es folgt in diesem Kapitel ebenfalls ein Ausblick.

2 Grundlagen

In diesem Kapitel werden die aus der Aufgabenstellung hervorgehenden und für die Nachvollziehbarkeit dieser Arbeit wichtigen Grundlagen beschrieben. Es werden die jeweiligen Grundkonzepte vorgestellt und in Zusammenhang gebracht, sowie Kandidaten für die Build- und die Datenbank-Systeme vorgestellt und bezüglich der Ziele dieser Arbeit analysiert. Die einzelnen Grundlagen sollen in einer Reihenfolge beschrieben werden, die den Verlauf der allgemeinen Einarbeitung dieser Arbeit widerspiegelt.

2.1 DevOps

Die Methodik DevOps beschreibt eine Organisationsform in Unternehmen, bei der die Bereiche Entwicklung (eng.: Development) und Betrieb (eng.: Operation) zu einem gemeinsamen Team – dem DevOps-Team – zusammenschmelzen und im Gegensatz zur klassischen Organisationsform, in der die Bereiche mit ihren Aufgaben und Zielen überwiegend getrennt voneinander arbeiten, gemeinsam an einem Ziel mit gemeinsamen Interessen und bereichsübergreifendem Wissen arbeiten. Der Vorteil dabei ist, dass neben der hohen Produktivität auch hohe Stabilität, Sicherheit, Verfügbarkeit und Zuverlässigkeit gewährleistet wird [1].

In Unternehmen, in denen die Bereiche Entwicklung und Betrieb nach der klassischen Organisationsform getrennt voneinander an ihren Aufgaben eines gemeinsamen Projektes arbeiten und erst in der finalen Phase ihre Ergebnisse zusammenführen, treten häufig Probleme auf, die zur Folge haben, dass Nacharbeiten von den Teams geleistet werden müssen. Durch die Nacharbeiten können Zeitpläne meistens nicht eingehalten werden, die Qualität der Arbeit nimmt ab und das Potenzial des Teams wird nicht vollkommen ausgeschöpft. Die Konsequenz ist, dass die Ziele nicht erreicht werden und die Motivation sowie das Engagement der Teammitglieder nachlässt [1].

Der Grund, weshalb die genannten Bereiche in der klassischen Organisationsform getrennt arbeiten, liegt an den vollkommen unterschiedlichen Zielen. Während die Entwicklung versucht, auf Änderungen im Markt schnell zu reagieren und Features in die Produktivumgebung zu deployen[7], hat der Betrieb das Ziel, den IT-Service für den Kunden stabil, sicher und zuverlässig zu halten. Durch das Verfolgen der unterschiedlichen Ziele entsteht zwischen den beiden Bereichen ein Konflikt, der so mächtig ist, dass das Erreichen

[7] Deployment bezeichnet einen (automatisierten) Prozess zur Installation und Konfiguration einer Software.

der gewünschten Geschäftsergebnisse verhindert wird [1].

Durch das Prinzip der Organisationsform DevOps wird verhindert, dass ein solcher Konflikt in einem Unternehmen entsteht, wobei gleichzeitig die Arbeitsbedingungen verbessert werden. Das Ideal von DevOps sieht vor, dass innerhalb des DevOps-Teams kleinere Teams gebildet werden, die unabhängig voneinander an ihren Features arbeiten, diese auf Korrektheit in einer produktivähnlichen Umgebung überprüfen und letztendlich schnell und zuverlässig in die echte Produktivumgebung deployen. Somit werden Code-Deployments zur Routine und die Teams können jederzeit vorhersagen, wie sich die Software mit allen Features verhält [1].

Ein sehr beliebtes Konzept aus der Softwareentwicklung, welches sich sehr gut in DevOps integrieren lässt, ist Continuous Delivery, da für dieses Konzept das Wissen aus beiden Bereichen – Entwicklung und Betrieb – benötigt wird. Durch eine engere Zusammenarbeit der Bereiche wird eine bessere und schnellere Kommunikation aller Beteiligten ermöglicht und Entscheidungen deutlich schneller getroffen. Dies verringert den Koordinierungsaufwand deutlich, was dem Konzept Continuous Delivery sehr entgegenkommt [2].

2.2 Continuous Delivery

Continuous Delivery (kurz: CD) ist in der Softwareentwicklung ein Konzept, bei der die Software so erstellt wird, dass diese für die Produktion zu jederzeit freigegeben werden kann. Dies wird erreicht, indem sichergestellt wird, dass sich der Quellcode der Software immer in einem einsatzfähigen Zustand befindet, auch wenn eine beliebig große Anzahl von Entwicklern täglich Änderungen an der Software vornehmen [3].

Das Prinzip von Continuous Delivery setzt darauf, die Schritte, die für die ununterbrochene Einsatzfähigkeit einer Software notwendig sind, möglichst zu automatisieren, damit diese in kurzen Zeitabständen automatisiert und nicht mehr manuell vom Entwickler ausgeführt werden müssen. Dadurch soll eine Regelmäßigkeit entstehen, durch die die Qualität und die Zuverlässigkeit der Software erhöht wird. Durch eine Versionsverwaltung[8] soll zudem gewährleistet werden, dass jegliche Änderungen an einer Software nachvollziehbar sind und jeder Stand der Software rekonstruierbar ist. Dies trägt zu Verbesserung der Fehleranalyse bei [2].

Damit die Schritte automatisiert ausgeführt werden können, wird eine Deployment-Pipeline eingesetzt, die den Software-Build in Phasen unterteilt und automatisiert ausführt, sobald eine

[8] Eine Versionsverwaltung ist ein System, bei dem alle Änderungen an Dateien als Versionen mit einem Zeitstempel und Benutzererkennung gesichert und verwaltet werden. Es ist möglich Versionen wiederherzustellen [37].

Änderung in die Versionsverwaltung der Software überführt wurde [4].

Entwickler, die nach dem Konzept von Continuous Delivery arbeiten, setzen ihren Fokus auf die Einsatzfähigkeit der Software anstatt auf die Entwicklung neuer Funktionen, um damit sicherzustellen, dass die Software während ihres gesamten Lebenszyklus einsatzfähig ist. Voraussetzung für die Einsatzfähigkeit ist dabei allerdings, dass die Entwickler in kurzen Zyklen Änderungen an der Software integrieren und automatisierte Tests laufen lassen, um schnell Feedback zu erhalten, ob sich die Software mit den Änderungen korrekt verhält. Das Integrieren sowie das automatisierte Testen sind Teile des Konzeptes Continuous Integration, welches im nachfolgenden Kapitel näher betrachtet wird [5].

Ein weiteres Konzept, welches auf Continuous Delivery aufbaut und häufig mit diesem verwechselt wird, ist Continuous Deployment. Dieses Konzept erweitert Continuous Delivery um den Schritt, dass die Software nach jedem Build oder nach Anforderung eines Entwicklers in die Produktivumgebung bereitgestellt (eng.: deploy) wird. Continuous Deployment ist nicht Thema dieser Arbeit, jedoch für eine klare Abgrenzung der beiden Begriffe notwendig zu erwähnen [1].

2.3 Continuous Integration

Continuous Integration (kurz: CI) ist in der Softwareentwicklung ein Konzept, bei dem Entwickler ihren neu entwickelten Code in regelmäßigen kurzen Abständen in die Code-Basis der Software, die meistens in einem Versionsverwaltungssystem verwaltet wird, integrieren. Das Konzept umfasst die Schritte Build und Integration, welche automatisiert ausgeführt werden [2]. Der Software-Build wird generell von einem Build-System übernommen. Build-Systeme sind Thema des nächsten Kapitels.

Die Aufgabe, Änderungen einer Software in die Code-Basis zu integrieren, wird von einem Build-Server übernommen, der die aktuelle Version der Software auscheckt und mithilfe eines Build-Systems den Software-Build anstößt. Durch die kontinuierliche Integration können Fehler frühzeitig erkannt und behoben werden, da mit jedem Build die Tests ausgeführt werden, wodurch sich die Arbeitseffizienz des Teams und die Software-Qualität verbessern [2].

Da die Software-Builds in dieser Arbeit mittels Jenkins-getriggerten Integrationstests untersucht werden sollen, wird im Folgenden nur der Dienst Jenkins kurz vorgestellt. Alternativen zu Jenkins sind beispielsweise TravisCI oder GoCD von ThoughtWorks. Die Voraussetzung für die Nutzung solcher Dienste ist es, diese mit einem Versionsverwaltungssystem – wie Bitbucket oder GitHub - zu kombinieren, damit der Dienst

Änderungen am Code automatisch erkennt und den Prozess der kontinuierlichen Integration und kontinuierlichen Bereitstellung starten kann [2].

Jenkins ist eine Anwendung, die es ermöglicht, den Quellcode einer Software automatisiert zu bauen und die daraus entstandenen Artefakte auszuliefern. Dieser Prozess bildet die Schritte der Konzepte von Continuous Integration bzw. Continuous Delivery ab [6]. Ein Software-Build in Jenkins wird dabei auf einen Job abgebildet, der in einem eigenen Workspace läuft und durch den Entwickler beliebig konfiguriert werden kann [2].

Wieso Jenkins und bspw. nicht Gitlab CI (das hätten Sie ja als THL managed Service einfach in GitLab nutzen können?)

2.4 Build-Automatisierung und Build-Systeme

In diesem Kapitel wird der Begriff Build-Automatisierung in Zusammenhang mit Continuous Integration definiert und geklärt, welche Aufgaben Build-Systeme in der Softwareentwicklung übernehmen. Anschließend werden zwei Build-Systeme für diese Arbeit gewählt und vorgestellt.

In der Softwareentwicklung werden Build-Systeme eingesetzt, um den Quellcode einer Software in ein ausführbares Softwareprodukt zu übersetzen. Dieser Prozess, das Bauen (engl.: build) der Software, besteht aus einer Vielzahl von Schritten, die in einer bestimmten Reihenfolge ausgeführt werden. Nicht nur das Kompilieren des Quellcodes, sondern auch das Paketieren des kompilierten Codes in verteilbare Artefakte wie JAR- oder WAR-Dateien oder die automatische Ausführung von Tests, die durch den Entwickler geschrieben wurden, sind Teile dieses Prozesses [7].

Viele Schritte dieses Prozesses sind Routinearbeiten, wie auch das zuvor erwähnte Ausführen der Tests. Mit jedem erneuten Bauen der Software werden die Routinearbeiten automatisiert ausgeführt, wodurch die Produktivität der Entwickler gesteigert wird, da diese die wiederkehrenden Tätigkeiten nicht eigenständig ausführen müssen [7]. Die Build-Automatisierung stellt dabei eine wichtige Grundlage für die Konzepte Continuous Integration und Continuous Delivery dar [2].

Eine weitere wichtige Aufgabe, die Build-Systeme erfüllen, ist das Abhängigkeitsmanagement. Entwickler haben mit Build-Systemen die Möglichkeit, Abhängigkeiten (engl.: Dependencies) eines Projektes zu verwalten. Mit Abhängigkeiten sind in diesem Fall Fremdbibliotheken, die von Drittanbietern, wie beispielsweise JUnit[9], zur Verfügung gestellt werden, gemeint. Die Einbindung von Fremdbibliotheken ist essenziell wichtig für die Entwicklung einer Software. Die Build-Systeme stellen zur Einbindung von Fremdbibliotheken eine Datei zur Verfügung, in der die Abhängigkeiten eindeutig über ein Tupel aus Gruppen-ID, Artefakt-ID und

[9] JUnit ist ein Framework zum Testen von Java-Anwendungen.

Versionsnummer deklariert werden können. Die Fremdbibliotheken werden automatisch vom Build-System in einen Cache heruntergeladen [8].

Die beliebtesten drei Tools für die Build-Automatisierung in der Java-Welt sind Apache Ant, Apache Maven und Gradle, welche sich deutlich in ihren Ansätzen unterscheiden. Apache Ant, das älteste Build-System der drei Kandidaten, verfolgt einen imperativen Ansatz, der von den Entwicklern einen großen Konfigurationsaufwand verlangt. Die Entwickler müssen bei Ant den Software-Build vollständig Schritt für Schritt implementieren. Beispielsweise ist es notwendig, dass die Verzeichnisse für den kompilierten Quellcode manuell angelegt oder Abhängigkeiten innerhalb des Builds explizit angegeben werden müssen. Apache Maven dagegen verfolgt einen deklarativen Ansatz, bei dem das Tool einen großen Anteil der Konfiguration mittels Konventionen, die jedoch vom Entwickler überschrieben werden können, übernimmt. So ist beispielsweise der Speicherort des Quellcodes und der Tests sowie die Abfolge der Build-Phasen fest definiert. Gradle ist das jüngste Build-System der drei Kandidaten und baut auf den Konzepten von Ant und Maven auf, um die Vorteile beider Tools in einem zu vereinen. Dabei übernimmt Gradle die Flexibilität von Ant und die Prinzipien zur Projektdefinition und Projektverzeichnisstruktur von Maven [2].

Aufgrund der Tatsache, dass bei Apache Ant viel Konfiguration notwendig ist und ein Software-Build weitestgehend manuell implementiert werden muss, was nicht den Prinzipien von Continuous Delivery und Continuous Integration sowie dem eigentlichen Ziel dieser Arbeit, prototypisch eine Software zu entwickeln, die möglichst viel manuelle Arbeit abnimmt, indem Automatisierung bevorzugt wird, entspricht, wird Apache Ant nicht weiter berücksichtigt.

2.4.1 Apache Maven

Apache Maven ist ein Build-Tool für Softwareprojekte, die mit der Programmiersprache Java entwickelt wurden. 2002 wurde es als Teilprojekt des Apache Turbine Servlet Framework[10] erstellt und zwei Jahre später in ein eigenständiges Apache Projekt umgewandelt [9].

Das Hauptziel von Maven ist es, dass die Entwickler die Möglichkeit haben, den kompletten Stand eines Softwareproduktes, welches mit Maven aufgesetzt wurde, in kürzester Zeit erfassen zu können [10]. Auf dieses Ziel bezieht sich auch die Philosophie von Maven: Mit Maven sollen Softwareprodukte die Eigenschaften Sichtbarkeit, Wiederverwendbarkeit, Wartbarkeit und Verständlichkeit aufweisen können, um die Produktivität eines Teams zu erhöhen [11].

[10] Mit dem Apache Turbine Servlet Framework können Java-Entwickler Webanwendungen erstellen.

Mit dem Projekt-Objektmodell (kurz: POM), eine XML-Datei, stellt Maven die Möglichkeit, alle Konfigurationen in einer Datei zu speichern. In dieser Datei können beispielsweise Fremdbibliotheken oder externe Plugins deklariert werden, damit diese in das jeweilige Projekt eingebunden werden können [10].

Mit der einfachen Projekt-Einrichtung definiert Maven eine Richtlinie, wie Projekte aufgebaut sein müssen. Eine festdefinierte Verzeichnisstruktur, in der der Quellcode für die Tests in einem separaten Verzeichnis abgelegt wird, sollen „Best-Practies-Prinzipien“ stärken. Auch andere Dateien wie Textdateien werden in einem separaten Verzeichnis abgelegt. Maven zielt hierbei auf eine klare Trennung verschiedener Dateitypen ab [10].

Um ein Projekt zu bauen, folgt Maven einem bestimmten Konzept, einem Build-Lebenszyklus, der ausgeführt wird, wenn ein Projekt gebaut werden soll. Der Lebenszyklus besteht aus Phasen[11], die in einer fest definierten Sequenz ausgeführt werden [12].

Entwickler haben bei Maven die Möglichkeit, eigene Plugins zu schreiben. In diesen Plugins können Goals, die eine spezifische Funktionalität widerspiegeln, definiert werden, welche wiederum bei der Ausführung des Build-Lebenszyklus an eine oder mehrere Phasen gekoppelt werden können. Ein Plugin kann aus mehreren Goals bestehen, um Funktionalitäten zu bündeln [13].

2.4.2 Gradle

Gradle ist 2007 auf dem Markt erschienen [14]. Es baut auf den Konzepten von Maven und Ant auf, um die Vorteile beider in einem Tool zu vereinen. Dabei übernimmt Gradle die Flexibilität von Ant und die Prinzipien zur Projektdefinition und Projektverzeichnisstruktur von Maven [2]. Gradle zeichnet sich durch seine hohe Leistung, Flexibilität und Erweiterbarkeit aus [15].

Das Hauptziel von Gradle ist es, Unternehmen dabei zu unterstützen, ihre Software schneller ausliefern zu können. Um dieses Ziel zu erreichen, hat Gradle den Fokus auf schnelle Software-Builds gerichtet [16].

Zur Beschreibung der zu bauenden Projekte nutzt Gradle eine auf Groovy basierende domainspezifische Sprache (kurz: DSL), eine Skriptdatei, die direkt ausführbar ist. Diese Datei besteht u.a. aus einer Vielzahl von Aufgaben[12]. Die Aufgaben bilden das Kernmodell von Gradle und ähneln im weitesten Sinne den Phasen von Maven. Eine Aufgabe besteht optional aus einer Aktion sowie aus einer Eingabe und einer Ausgabe. Zum Ausführen von Projekt-

[11] Eine Phase entspricht einem Befehl (vgl. compile oder test).
[12] In der Literatur wird oft der englische Begriff „Task(s)“ verwendet. In dieser Thesis wird die deutsche Übersetzung „Aufgabe(n)“ verwendet.

Builds modelliert Gradle über die Reihenfolge der benötigten Aufgaben einen gerichteten azyklischen Graphen[13]. Dadurch können Aufgaben parallel ausgeführt werden, wodurch Gradle Projekte potenziell schneller bauen kann als Maven [15].

Gradle kann auf unterschiedliche Weise erweitert werden. Es besteht die Möglichkeit, Code in die Skriptdatei einzubetten, die Aufgaben zu erweitern und eigene Aufgaben zu schreiben, wenn keine der von Gradle vordefinierten Aufgaben die gewünschte Aktion ausführen kann. Zudem können Entwickler eigene Plugins schreiben. Dazu ist es notwendig, neben dem eigentlichen Plugin auch einen benutzerdefinierten Aufgabentyp zu implementieren, um die gewünschte Funktionalität ausführen zu können. Ein Plugin kann entweder in Java, Kotlin oder direkt in der Skriptdatei implementiert werden [15].

In Hinblick auf die Automatisierung von Software-Builds hat Gradle einige Features, die sehr hilfreich sind. Mit diesen Features hat Gradle einen Vorteil gegenüber Apache Maven hinsichtlich der Automatisierung von Builds. Zu diesen Features zählen der Gradle Wrapper, der Gradle Daemon und die inkrementelle Kompilierung [2].

Mithilfe des Gradle Wrappers ist es möglich, die verwendete Gradle-Version in der Build-Datei zu konfigurieren und somit Gradle selbst zum Teil des Projektes zu machen. Der Vorteil dabei ist, dass beim Auschecken des Projektes auf einem Build-Server oder einem anderen Rechner alles Nötige vorhanden ist, um das Projekt zu bauen. Es ist nicht notwendig, dass auf dem Build-Server oder auf dem anderen Rechner eine Installation von Gradle gepflegt wird [2].

Der Gradle Daemon ist ein langlebiger Prozess, der Informationen über die Builds eines Projektes im Cache abspeichert. Wird ein Projekt erneut gebaut, so ruft der Daemon die Informationen der vorherigen Builds des Projektes ab und nutzt diese wieder. Dadurch werden die Projekte bis zu 75% schneller gebaut [15].

Bei der inkrementellen Kompilierung prüft Gradle, ob sich die Eingabe, Ausgabe oder Implementierung einer Aufgabe zum vorherigen Projekt-Build geändert hat. Wurden keine Änderungen durchgeführt, so wird diese Aufgabe auch nicht ausgeführt [2].

2.5 Datenbank-Systeme

Können Sie daa dennoch einmal erläutern?
Was versteht man unter DB-Modell? Welche gibt es?
Wieso ist SQL so wichtig in diesem Zusammenhang?
Gibt es andere Sprachen?

Für dieses Kapitel wird der Begriff *Datenbankmodell* sowie dessen Bedeutung und ein Grundverständnis der Datenbanksprache *SQL* (Structured Query Language) vorausgesetzt.

Im Folgenden wird das grundlegende Konzept von Datenbank-Systemen (kurz: Datenbank) vorgestellt und beschrieben, welche Aufgaben sie erfüllen. Im Anschluss werden zwei

[13] Ein gerichteter azyklischer Graph ist ein Graph, der keinen Zyklus und keine gerichteten Kanten enthält.

spezifische Datenbank-Systeme mit unterschiedlichen Datenbankmodellen – relational und dokumentorientiert – vorgestellt und miteinander verglichen. Für die Bearbeitung der Thesis wurde eine relationale Datenbank gewählt, da dessen Datenbankmodell ein über viele Jahre etablierter Standard für Datenbanken ist. Des Weiteren wurde sich für eine dokumentorientierte Datenbank entschieden, da dessen Typvertreter MongoDB das populärste nicht relationale Datenbank-System ist [17].

Datenbank-Systeme werden zur strukturierten und dauerhaften Speicherung von Datenmengen sowie deren Bereitstellung für Anwendungen und Benutzer eingesetzt. Ein Datenbank-System besteht aus zwei Komponenten, einem Datenbankmanagementsystem (kurz: DBMS) und einer Datenbasis, die Menge der gespeicherten Daten. Das DBMS hat die Aufgabe, die gespeicherten Daten intern zu organisieren sowie alle lesenden und schreibenden Zugriffe auf die Datenbank zu kontrollieren. Damit Daten gespeichert, organisiert und abgefragt werden können, bieten Datenbankmanagementsysteme eine formale Sprache, eine Datenbanksprache, zur Beschreibung und Abfrage der Datenbank an [18].

Datenbank-Systeme müssen einige Eigenschaften aufweisen. So ist es notwendig, dass die Daten auf der Datenbank vollständig sind und die Regeln des Datenbankmodells einhalten. Diese Eigenschaft wird Datenintegrität genannt. Zudem muss sichergestellt werden, dass die Veränderung der Daten vollständig erfolgt, sodass sich die Datenbank nach einer Transaktion in einem konsistenten Zustand befindet. In einigen Fällen sollte eine redundante Speicherung von identischen Daten für eine Performance-Optimierung möglich sein, auch wenn damit die Regel der Datenintegrität verletzt wird und es nicht immer realisierbar ist, dass die Datenbank sich direkt nach Ende einer Transaktion in einem konsistenten Zustand befindet [18].

Der Zugriff auf die Daten einer Datenbank darf nur durch autorisierte Benutzer erfolgen, um Datensicherheit zu gewährleisten. Aus diesem Grund ist es notwendig, dass die Zugriffe auf die Daten mittels Berechtigungen erfolgen. Damit kann auch ein gleichzeitiger Zugriff auf die identischen Daten mehrerer Anwender realisiert werden. Beispielsweise könnte ein Anwender schreibend und die anderen Anwender lesend auf die Daten zugreifen [18].

2.5.1 Relationale Datenbanken

In Datenbank-Systemen mit einem relationalen Datenbankmodell werden die Daten in verschiedenen Tabellen organisiert und sind über Beziehungen (Relationen) miteinander verknüpft. Der Vorteil des relationalen Datenbankmodells liegt darin, dass relationale Datenbanken sehr einfach erstellt werden können und flexibel einsetzbar sind. Allerdings zeigen sie Schwächen in der Skalierung und in der Verarbeitung großer Datenmengen auf.

Mithilfe der Datenbanksprache SQL können Datenbanken manipuliert und Abfragen durchgeführt werden [18].

Im Gegensatz zu anderen Datenbankmodellen ist es bei dem relationalen Modell notwendig, die Daten auf das von der Datenbank vordefinierte Schema anzupassen, bevor diese u.a. in die Datenbank gespeichert werden können. Diese Vorverarbeitung der Daten ist mit einem großen Mehraufwand verbunden, wodurch dieses Modell an Effizienz verliert [19].

Als Typvertreter dieses Datenbank-Systems wurde Microsoft SQL Server (kurz: MSSQL) gewählt, welches in Datawarehouse- und Business-Intelligence-Anwendungen eingesetzt werden kann. Microsoft verwendet für sein DBMS die SQL-Variante Transact-SQL (kurz: T-SQL), die den SQL-Standard um einige Features wie die Transaktionssteuerung und die Fehlerbehandlung erweitert [20].

In Erweiterung zu SQL, welches eine datenorientierte Sprache ist und generell zur Verarbeitung der Daten einer Datenbank mittels den Grundbefehlen *INSERT*, *SELECT*, *DELETE* und *UPDATE* dient, ist T-SQL eine transaktionsorientierte Sprache, mit der durch die Verwendung von mehreren Programmiertechniken komplexere Abfragen erstellt werden können. Mit T-SQL ist es möglich einer Datenbank über Abfragen Anweisungen zu geben, auf welche Art die Datenbank die Abfrage bearbeiten soll. Des Weiteren können T-SQL-Abfragen aus mehreren einzelnen SQL-Abfragen bestehen, sodass mehrere Befehle in einer Abfrage ausgeführt werden können [21].

Kann Ihr Tool nur Relational + Dokument-basierte DBen?
Wieso greifen Sie ausgerechnet diese beiden heraus?
Kennen Sie noch weitere Datenbankmodelle?
Funktioniert Ihre Lösung mit diesen genauso?

2.5.2 Dokumentbasierte Datenbanken

Bei dem dokumentorientierten Datenbankmodell werden die Daten in keiner festen Struktur in der Datenbank gespeichert und verwaltet, sondern die Datenbasis selbst definiert das Schema, wie die Daten organisiert werden. Die dokumentorientierten Datenbanken zählen zu den NoSQL[14]-Datenbanken, da sie nicht dem relationalen Ansatz folgen und auf die Datenbanksprache SQL verzichten [18]. NoSQL-Datenbanken sind primär für Anwendungen entwickelt worden, bei denen Datenbanken, die den relationalen Ansatz verfolgen, an ihre Grenzen stoßen, wodurch sie eine gute Alternative zu den relationalen Datenbank-Systemen bilden [2]. Als Typvertreter dieses Datenbank-Systems wurde MongoDB gewählt.

MongoDB ist ein dokumentbasiertes Datenbank-System, welches durch hohe Skalierbarkeit und hohe Flexibilität charakterisiert wird. Es ist möglich Daten in flexiblen, JSON-ähnlichen Dokumenten zu speichern, wobei das Schema der Dokumente einer Datenbank variieren

[14] **NoSQL** steht für **N**ot **o**nly **SQL**.

kann. Zudem ist es möglich die Datenstruktur eines Dokuments mit der Zeit zu ändern [22].

Ende der 2000er Jahren entstanden die ersten NoSQL-Datenbanken, als die Kosten für die Organisation von großen Datenmengen in Datenbanken drastisch sanken. Das Verfolgen des relationalen Ansatzes, bei dem ein komplexes Datenmodell verwaltet werden musste, um Datenduplikate möglichst gering zu halten, war nicht mehr notwendig. Mit den NoSQL-Datenbanken wurde es den Entwicklern ermöglicht, große Datenmengen schemalos in Datenbanken zu speichern. Es war nicht mehr notwendig, die Daten auf das von der Datenbank vordefinierte Schema anzupassen, wodurch die Flexibilität anstieg. Durch die Flexibilität war es ebenfalls möglich, schneller auf die wechselnden Anforderungen an eine Software zu reagieren. Mit dem Aufkommen des Cloud Computings stieg gleichzeitig die Anforderung, Daten auf mehrere Server in unterschiedlichen Regionen zu verteilen, damit Anwendungen robuster und ausfallsicherer werden. Das Konzept der NoSQL-Datenbanken insbesondere auch das Konzept von MongoDB ermöglichen es, Daten in verteilten Datenbanken zu organisieren [23].

Gibt es denn noch andere Ansätze bei DBen?
Was ist bspw. mit Distributed Key-Value-Stores? (ETCD)
Was ist mit horizontal skalierenden DBen? Ohne (oder mit wechselndem Leader)

2.6 Client-Server-Modell

Das Client-Server-Modell ist eines der häufigsten verwendeten Modelle von verteilten Systemen. Ein solches System besteht aus einem Server und mindestens einem Client. Die Aufgaben zwischen den einzelnen Komponenten des Systems sind klar verteilt. Der Server stellt einen Dienst zur Verfügung und ist der Hauptknotenpunkt des Systems. Er bearbeitet Anfragen (engl. requests) der Clients, die den Dienst für ihre Funktionalität nutzen, und schickt, nachdem die Anfragen vollständig bearbeitet wurden, eine Antwort (engl.: response) an den jeweiligen Client zurück. Die Dienste, die ein Server zur Verfügung stellt, können sehr unterschiedlich voneinander sein. Beispielsweise könnte ein Server mit einer Datenbank interagieren. Die Anfragen der Clients würden in diesem Fall u.a. zur Datenabfrage oder Datenmanipulation dienen. Die Interaktion zwischen Client und Server erfolgt generell nach dem zuvor beschriebenen Schema [24].

Ein wichtiges Merkmal des Client-Server-Modells ist, dass die Clients zueinander in keinem Bezug stehen und aus diesem Grund auch voneinander keine Kenntnis besitzen [24]. Zudem zeichnet sich dieses Modell – wie alle anderen Modelle der verteilten Systeme - u.a. durch Ausfallsicherheit, Robustheit und Skalierbarkeit aus [25].

3 Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an die Software ermittelt und erläutert. Im ersten Schritt werden die funktionalen Anforderungen sowie die Produktfunktionen der Software betrachtet und in Zusammenhang gebracht. Im zweiten Schritt werden die nichtfunktionalen Anforderungen an die Software analysiert und aufgelistet.

3.1 Funktionale Anforderungen und Produktfunktionen

In Kapitel 1.3 wurde bereits der Ist-Zustand der AEB SE vorgestellt und die Motivation aufgezeigt, weshalb es interessant und wichtig ist, sich mit der Problematik dieser Thesis zu beschäftigen. Aus dieser Beschreibung sowie auch aus den Zielen dieser Arbeit lassen sich die grundlegenden Anforderungen an die Software ermitteln. Die folgende Tabelle zeigt die dazugehörigen User Stories, um einen allgemeinen Überblick zu geben und kurz zu definieren, was in dem jeweiligen Anwendungsfall das gewünschte Ziel ist.

Titel	Beschreibung
Anlegen einer Datenbank	Als Benutzer dieser Software möchte ich über ein Build-System-spezifisches Plugin eine Datenbank, die keine Daten beinhaltet, anlegen können, um diese anschließend nutzen zu können.
Anlegen einer Datenbank mit initialen Daten	Als Benutzer dieser Software möchte ich über ein Build-System-spezifisches Plugin eine Datenbank anlegen können, um diese anschließend nutzen zu können. Dabei möchte ich dem Server Befehle übermitteln können, damit die Datenbank bei ihrer Erstellung initial befüllt wird.
Anlegen einer Datenbank mittels eines Backups	Als Benutzer dieser Software möchte ich über ein Build-System-spezifisches Plugin eine Datenbank mittels eines Backups anlegen können, um diese anschließend nutzen zu können. Der Zustand der Datenbank soll anschließend dem Zustand des Backups gleich sein.
Anlegen einer Datenbank mittels Duplizierens einer Datenbank	Als Benutzer dieser Software möchte ich über ein Build-System-spezifisches Plugin eine Datenbank duplizieren können, um eine identische Version der aktuellen Datenbank zu haben, damit der Zustand gesichert bleibt.
Auflisten aller Datenbanken	Als Benutzer dieser Software möchte ich über ein Build-System-spezifisches Plugin alle Datenbanken auflisten können, um mir einen Überblick über die Datenbanken verschaffen zu können.

Löschen einer Datenbank	Als Benutzer dieser Software möchte ich über ein Build-System-spezifisches Plugin eine Datenbank löschen können, um Speicherplatz freizugeben. Dabei möchte ich angeben können, ob alle zur Datenbank zugehörigen Backups mit gelöscht werden sollen.
Anlegen eines Backups einer Datenbank	Als Benutzer dieser Software möchte ich über ein Build-System-spezifisches Plugin ein Backup einer Datenbank anlegen können, damit der aktuelle Zustand einer Datenbank für zukünftige Zwecke gespeichert wird.
Datenbank auf den Stand eines Backups zurücksetzen	Als Benutzer dieser Software, möchte ich über ein Build-System-spezifisches Plugin eine Datenbank auf den Stand eines Backups zurücksetzen können, um mit dem Zustand des Backups weiterarbeiten zu können.
Vergleichen zweier Backups	Als Benutzer dieser Software möchte ich über ein Build-System-spezifisches Plugin zwei Backups von Datenbanken miteinander vergleichen können, um Unterschiede auflisten zu können.
Löschen eines Backups	Als Benutzer dieser Software möchte ich über ein Build-System-spezifisches Plugin ein Backup einer Datenbank löschen können, damit nur benötigte Backups verwaltet werden müssen.
Auflisten aller Backups einer Datenbank	Als Benutzer dieser Software möchte ich über ein Build-System-spezifisches Plugin alle Backups einer Datenbank auflisten können, um mir einen Überblick über die Backups verschaffen zu können.
Auflisten aller Backups	Als Benutzer dieser Software, möchte ich über ein Build-System-spezifisches Plugin alle Backups (Datenbank übergreifend) auflisten können, um mir einen Überblick über die Backups verschaffen zu können.

Tabelle 1 User Stories

Wie sind sie zu diesen Anforderungen gekommen? Was haben Sie wie analysiert? Können Sie bitte einmal den Prozess der Datenbanknutzung in einem typischen CI/CD-Prozess aufzeichnen?

Aus den User Stories ergeben sich die funktionalen Anforderungen an die Software. Es folgt eine Liste mit den funktionalen Anforderungen:

- **[SF 1]** Eine Datenbank muss angelegt werden können.
- **[SF 2]** Eine Datenbank muss angelegt und mit initialen Daten befüllt werden können.
- **[SF 3]** Eine Datenbank muss angelegt werden können, die sich nach ihrer Erzeugung auf dem Stand eines gewünschten und existierenden Backups befindet.
- **[SF 4]** Eine Datenbank muss dupliziert werden können.
- **[SF 5]** Datenbanken müssen gelistet werden können.
- **[SF 6]** Eine Datenbank muss gelöscht werden können.
- **[SF 7]** Von einer Datenbank muss ein Backup (Schema + Daten) erzeugt werden können.
- **[SF 8]** Eine Datenbank muss auf den Stand eines Backups zurückgesetzt werden können.
- **[SF 9]** Zwei Backups (Schema + Daten) müssen verglichen werden können.
- **[SF 10]** Ein Backup (Schema + Daten) muss gelöscht werden können.
- **[SF 11]** Backups (Schema + Daten) müssen gelistet werden können.
- **[SF 12]** Backups (Schema + Daten) einer Datenbank müssen gelistet werden können.

Die Produktfunktionen, welche die zu entwickelnde Software erfüllen muss, lassen sich aus den Zielen ableiten. Es folgt eine Liste der Produktfunktionen:

- **[PF 1]** Die Schnittstelle zu Datenbankmanagementsystemen, die von der Software bereitgestellt wird, muss es ermöglichen, dass Datenbanken erzeugt werden können.
- **[PF 2]** Die Schnittstelle zu Datenbankmanagementsystemen, die von der Software bereitgestellt wird, muss es ermöglichen, dass die Datenbanken verwaltet werden können.
- **[PF 3]** Die Schnittstelle zu Datenbankmanagementsystemen, die von der Software bereitgestellt wird, muss es ermöglichen, dass Backups von Datenbanken erzeugt werden können.
- **[PF 4]** Die Schnittstelle zu Datenbankmanagementsystemen, die von der Software bereitgestellt wird, muss es ermöglichen, dass die Backups der Datenbanken verwaltet werden können.
- **[PF 5]** Die Schnittstelle zu Datenbankmanagementsystemen, die von der Software bereitgestellt wird, muss es ermöglichen, dass die Backups der Datenbanken verglichen werden können.

Die folgende Tabelle zeigt die Zuordnung der Produktfunktionen zu den funktionalen Anforderungen, um zu zeigen, dass jede funktionale Anforderung durch eine Produktfunktion abgedeckt ist:

Anforderung x Produktfunktion	PF 1	PF 2	PF 3	PF 4	PF 5
SF 1 (Anlegen einer DB)	X				
SF 2 (Anlegen einer DB mit initialen Daten)	X				
SF 3 (Anlegen einer DB durch ein Backup)	X				
SF 4 (Duplizieren einer DB)	X				
SF 5 (Auflisten aller DBs)		X			
SF 6 (Löschen einer DB)		X			
SF 7 (Anlegen eines Backups einer DB)			X		
SF 8 (DB auf Stand eins Backups setzen)		X			
SF 9 (Vergleichen zwei Backups)					X
SF 10 (Löschen eines Backups)				X	
SF 11 (Auflisten aller Backups einer DB)				X	
SF 12 (Auflisten aller Backups)				X	

Tabelle 2 Zuordnung der Produktfunktionen zu den funktionalen Anforderungen

3.2 Nichtfunktionale Anforderungen

Die folgenden nichtfunktionalen Anforderungen wurden in Zusammenarbeit mit dem Team Software Infrastructure am Standort Lübeck ermittelt.

Die Software soll vollständig in Java implementiert werden. Bei der Auswahl der Java-Version ist es erforderlich, dass diese Version langfristig (Sicherheits-)Updates erhält, sodass die Java-Version der Software nicht halbjährlich aktualisiert werden muss. Eine solche Version wird von dem Hersteller Oracle als eine Long-Term-Support Version (kurz: LTS Version) bezeichnet und erhält mehrere Jahre lang Support und Updates. Java 11 ist aktuell die letzte LTS-Version und wird noch bis September 2023 bzw. bis September 2026 Support erhalten. Aus diesem Grund soll die Software mit der Java-Version 11 entwickelt werden [26].

Bei der Entwicklung der Software ist darauf zu achten, dass der Datenaustausch über die Schnittstelle zustandslos erfolgt. Alle Anfragen eines Clients müssen daher alle notwendigen Informationen enthalten, da der Server keine Informationen speichern soll. Dadurch soll die Zuverlässigkeit erhöht werden. Des Weiteren ist es notwendig, dass die Logik zwischen Client und Server klar getrennt wird und der Client keine weiteren Informationen über den Aufbau und die Komplexität des Servers besitzt. Aus diesem Grund soll der Server eine auf der REST-API basierende Schnittstelle implementieren, die eine einheitliche Programmierschnittstelle

zum Austausch von Daten mittels HTTP-Methoden auf verteilten Systemen bereitstellt [27].

Der Body der HTTP-Anfrage soll in Form von JSON sein. Der Grund liegt darin, dass die menschliche Lesbarkeit von JSON im Gegensatz zu XML beispielsweise deutlich besser ist und JSON einfach zu verarbeiten ist.

Des Weiteren soll bei der Konzeption und Entwicklung des Servers darauf geachtet werden, die einzelnen Komponenten voneinander zu entkoppeln, um die Komplexität der Software zu reduzieren, die Wiederverwendbarkeit von Objekten zu steigern sowie die Wartbarkeit und Testbarkeit der Software zu erhöhen. Durch die Entkopplung ist es nicht mehr notwendig, dass die Komponenten Objekte anderer Komponenten erzeugen, sofern zwischen ihnen eine Abhängigkeit besteht, sondern die Komponenten werden beispielsweise bei einem Framework registriert, wo sie verwaltet und an den entsprechenden Stellen durch das Framework injiziert werden. Ein geeignetes Framework, welches diese Aufgabe erfüllt, ist das Spring Framework (kurz: Spring), welches von Rod Johnsen entwickelt und im Juni 2003 unter der Apache 2.0-Lizens veröffentlicht wurde [28]. Das Hauptziel von Spring ist es, die Komponenten einer Software mittels Inversion of Control und Dependency Injection voneinander zu entkoppeln [29]. Aus diesem Grund eignet es sich gut, die Server-Komponente mithilfe des Spring Frameworks zu entwickeln. Ein weiterer Vorteil von Spring ist, dass mit wenig Aufwand und einem hohen Anteil automatischer Konfiguration, Anwendungen schnell entwickelt werden können [30].

Es folgt eine Liste der nicht-funktionalen Anforderungen:

- **[NFR 1]** Die Entwicklung der Software erfolgt in der Programmiersprache Java in der Version 11.
- **[NFR 2]** Die Serverkomponente der Software soll eine auf der REST-API basierende Schnittstelle bereitstellen.
- **[NFR 3]** Der Inhalt der zu übermittelende Anfrage muss in „JSON-Form“ sein.
- **[NFR 4]** Die Server-Komponente der Software ist mit dem Framework Spring umzusetzen.

4 Systemarchitektur

Dieses Kapitel befasst sich mit der Systemarchitektur der Software. Aus der Problemstellung der Aufgabe sowie aus den ermittelten Anforderungen lässt sich diese ableiten. Bereits in diesem frühen Stadium ist es äußerst wichtig, bei der Konzeption der Systemarchitektur darauf zu achten, dass sich aus dieser eine Softwarearchitektur ableiten lässt, die leicht erweiterbar und änderbar ist, sodass die Schnittstelle der Software universell eingesetzt werden kann und somit das Ziel der Arbeit erreicht ist. Aus diesem Grund ist es äußerst sinnvoll, die Problemstellung der Aufgabe sowie die Anforderungen zu analysieren und Schritt für Schritt eine Architektur daraus abzuleiten.

Der erste und naheliegendste Punkt, der sich aus den Anforderungen ableiten lässt, ist, dass die Architektur dieser Software auf dem Client-Server-Modell basiert, sodass zwischen den Build-System-spezifischen Plugins – Clients – und der Schnittstelle zu den Datenbank-Systemen – Server – eine klare Trennung der Aufgabenbereiche hervorgeht. Aus diesem Grund ist es möglich, die Architektur des Clients und des Servers getrennt voneinander zu betrachten. Es ist nur notwendig, darauf zu achten, dass die Schnittstelle zwischen den beiden Komponenten gleich ist.

Wieso ist das naheliegend? Was für andere Alternativen haben Sie denn berück-sichtigt?

Aus der Anforderungsanalyse ist hervorgegangen, dass die Schnittstelle des Servers dieser Software auf der REST-API basieren soll. Eine Anforderung dieses Konzeptes ist, dass die Logik des Servers in mehrere Schichten unterteilt werden soll, damit der Client keine Informationen über den Aufbau und die Komplexität der Architektur des Servers besitzt. Der Vorteil dabei ist, dass der Client nur die erste Schicht des Servers – die restbasierte Schnittstelle – kennt. Vor diesem Hintergrund lässt sich der Server in zwei Schichten unterteilen. Die erste Schicht beinhaltet die restbasierte Schnittstelle des Servers und die zweite Schicht die Komponenten, die direkt mit einem Datenbank-System interagieren.

Die bisher konzipierte Systemarchitektur kommt der Zielarchitektur schon sehr nahe. Der einzige Punkt, der noch fehlt, ist, wie die beiden Schichten des Servers miteinander interagieren und kommunizieren. Mit einer weiteren Schicht, die zwischen der restbasierten Schnittstelle und den Datenbankkomponenten einzuordnen ist, ist eine Funktionalität beim Server gegeben, die die Anfragen, die bei der Schnittstelle eingehen, den entsprechenden Datenbankkomponenten zuordnet und weiterleitet.

Es wurde nun eine Systemarchitektur konzipiert, bei der die Aufgabenbereiche zwischen den Build-System-spezifischen Plugins und der Schnittstelle zu den Datenbank-Systemen klar voneinander getrennt sind. Des Weiteren wurde beim Server erreicht, dass dieser in Schichten unterteilt ist, sodass die hier erwähnte Anforderung der REST-API erfüllt worden ist. Die

Aufgaben der Schichten des Servers sind somit nahezu atomar voneinander getrennt und bauen aufeinander auf. Die drei Schichten des Servers kann man zudem in zwei Bereiche zusammenfassen. Die ersten beiden Schichten des Servers fungieren als eine Art Service, der Anfragen von Clients entgegennimmt und an die gewünschte Komponente, die direkt mit einem Datenbank-System interagiert, weiterleitet. Die dritte Schicht des Servers arbeitet dagegen direkt mit den Datenbank-Systemen zusammen.

Die Abbildung 1 gibt einen Überblick über die konzipierte Systemarchitektur. Wie aus dieser zu entnehmen, ist dort zusätzlich repräsentiert, wie sich die Software in der AEB SE integriert. Die Software kann entweder aus anderen Softwareprojekten der AEB SE oder aktiv von einem Entwickler aufgerufen werden. Bei den Softwareprojekten wäre es notwendig, dass sie das entsprechende Plugin in ihrer Build-Datei einbinden und das gewünschte Goal bei Maven bzw. die gewünschte Aufgabe bei Gradle aufrufen. Die Entwickler hingegen müssten das Goal bzw. die Aufgabe manuell anstoßen.

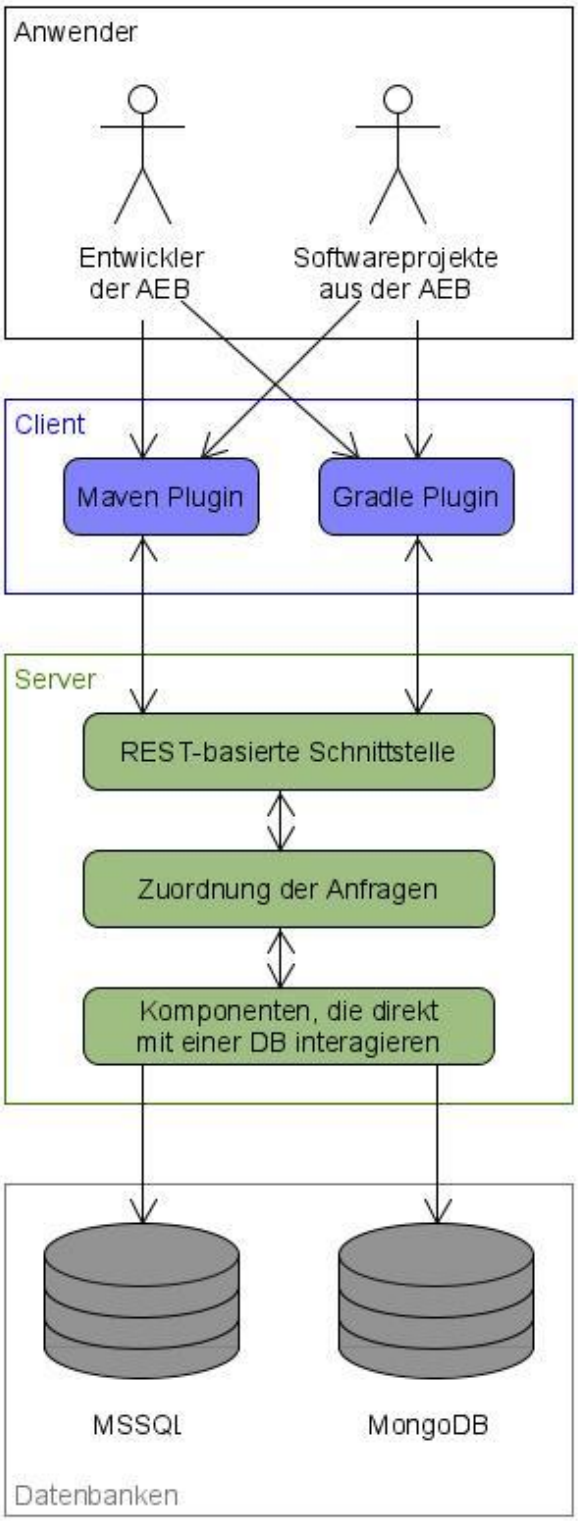


Abbildung 1 Systemarchitektur

5 Softwarearchitektur und Entwurfsentscheidungen

In diesem Kapitel wird die Softwarearchitektur, die sich aus der Systemarchitektur ableiten lässt, beschrieben. Dabei wird auf Entwurfsentscheidungen eingegangen. Aufgrund der Tatsache, dass die Architektur der Software auf dem Client-Server-Modell basiert, werden die Architekturen der beiden Komponenten getrennt voneinander betrachtet. In diesem Kapitel werden die entsprechenden UML-Diagramme in vereinfachter Form gezeigt. Im Anhang sind die vollständigen UML-Diagramme zu finden, um aus diesen weiteren Informationen zu entnehmen.

Damit beide Komponenten – Server und Client – sinnvoll miteinander kommunizieren können und sichergestellt wird, dass beide Komponenten die gleichen Informationen für die gewünschte Anfrage besitzen, ist der Einsatz von Data Transfer Objects (kurz: DTOs) sinnvoll. Aufgrund der Tatsache, dass beide Komponenten mit Objekten der DTOs interagieren, sind diese wegen der besseren Wartbarkeit und Erweiterbarkeit in einem eigenen Unterprojekt ausgelagert. Sowohl der Server als auch der Client können dieses Unterprojekt über eine eindeutige Abhängigkeit einbinden und Objekte der DTOs erzeugen und verwenden. Die Abbildung 2 zeigt den Aufbau der Architektur in vereinfachter Form inklusive der Einteilung der einzelnen Klassen in Pakete.
Wie aus der Abbildung zu entnehmen, gibt es zu jeder funktionalen Anforderung ein DTO für die Anfrage und für die Antwort. Die DTOs beinhalten dabei jegliche Informationen, die für die jeweilige Anfrage bzw. für die jeweilige Antwort benötigt wird. Sowohl für die Anfragen als auch die Antworten gibt es ein abstraktes DTO, welches die Informationen enthält, die bei jeder Anfrage bzw. jeder Antwort benötigt werden[15].

Zusätzlich zu den DTOs müssen der Server und der Client dieselben Informationen darüber besitzen, über welchen URL-Pfad die jeweilige Methode zu erreichen ist. Aus diesem Grund ist es sinnvoll diese Pfade ebenfalls in dem zuvor beschriebenen Unterprojekt auszulagern, um sicherzustellen, dass die URL-Pfade immer einheitlich sind.

[15] Bei jeder Anfrage muss angegeben sein, mit welchem Datenbank-System interagiert werden soll. Bei jeder Antwort ist eine Fehlermeldung und eine dazugehörige Exception angegeben, sofern ein Fehlerfall aufgetreten ist.

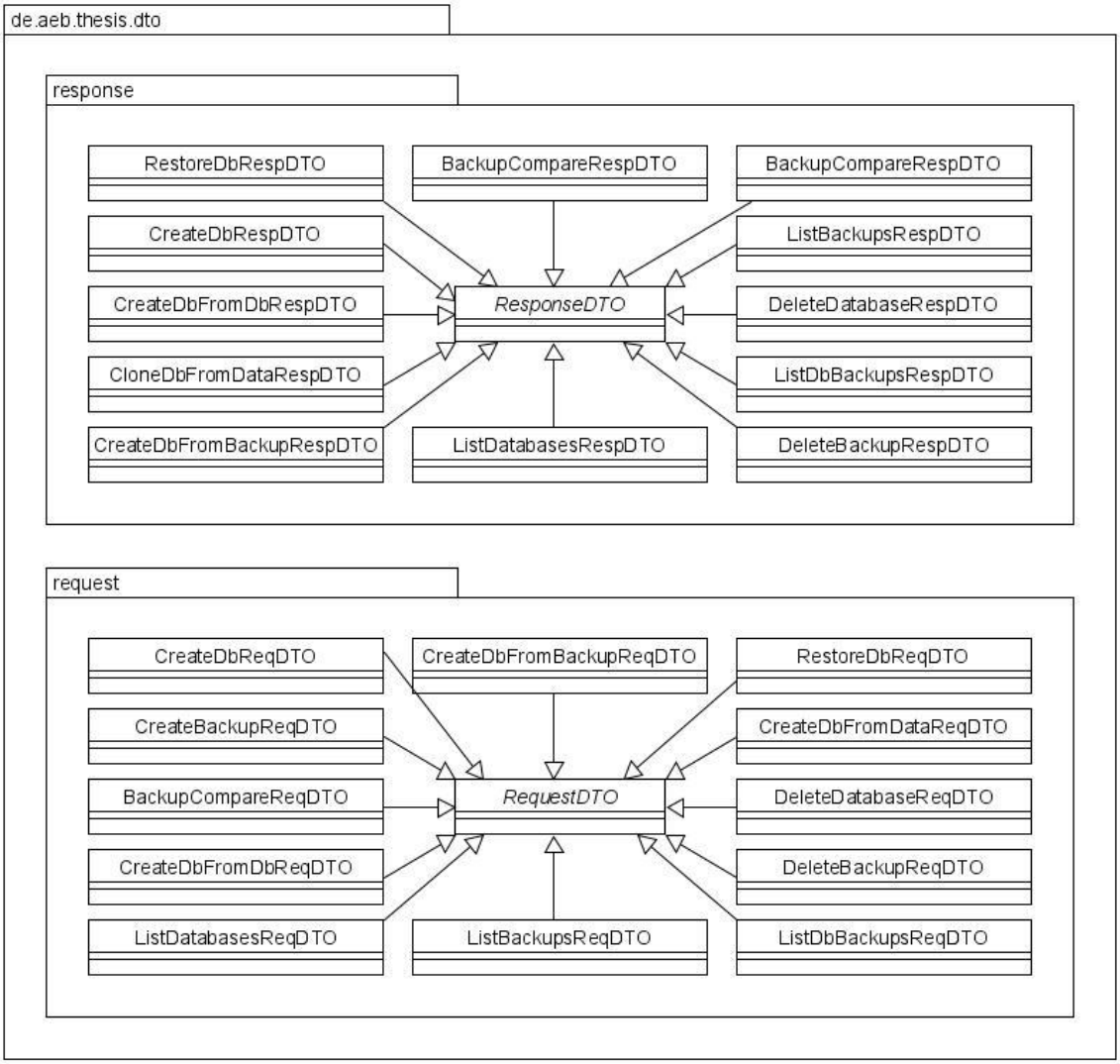


Abbildung 2 Vereinfachtes UML-Diagramm Data Transfer Objects

5.1 Server

Bereits bei der Konzeption der Systemarchitektur wurde darauf geachtet, dass diese hinsichtlich der Anforderungen und der Ziele leicht erweiterbar ist. Der Server wurde in zwei Bereiche eingeteilt, wodurch sich zwei voneinander klar getrennte Aufgabenbereiche des Servers ergeben. Aus diesem Grund wird die Architektur der beiden Bereiche getrennt voneinander vorgestellt.

Eine Anforderung an den Server, die sich während der Analyse ergab und vorgestellt wurde, war, dass die Komponenten des Servers voneinander entkoppelt werden müssen. Durch diese Entkopplung ist es erst möglich, die beiden Bereiche des Servers getrennt voneinander zu betrachten. In der Vorstellung der Architektur der beiden Bereiche wird genauer auf die Art und Weise eingegangen, wie die Entkopplung der beiden Bereiche erreicht wurde und welche Entscheidungen dabei getroffen worden sind.

5.1.1 Der Bereich Service

Zuerst wird die Architektur des Bereiches des Service betrachtet. Die folgende Abbildung zeigt den Aufbau der Architektur in vereinfachter Form inklusive der Einteilung der einzelnen Klassen in Pakete.

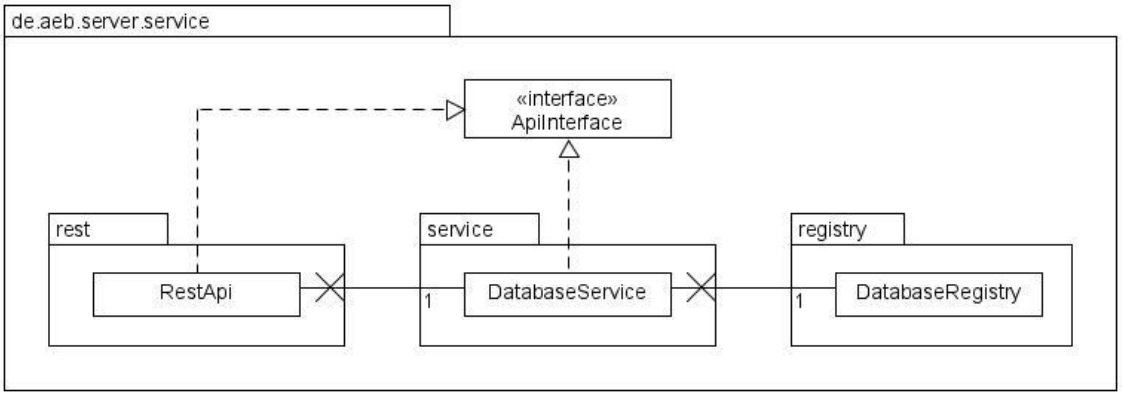


Abbildung 3 Vereinfachtes UML-Diagramm des Bereichs Service

Auch bei der Konzeption der Softwarearchitektur dieses Bereiches wurde sehr darauf geachtet, dass die Architektur leicht änder- und erweiterbar ist. Aus diesem Grund wurde ein Interface – *ApiInterface* – entwickelt, welches alle benötigten Schnittstellen, die aus den funktionalen Anforderungen hervorgegangen sind, als Methoden bereitstellt. Damit können andere Klassen dieses Interface implementieren und die Methoden um die Funktionalitäten für ihre spezifische Aufgabe erweitern. Durch den Einsatz eines Interfaces werden die Methoden für die verschiedenen Anwendungsfälle einheitlich und an zentraler Stelle definiert, sodass im Allgemeinen die Architektur übersichtlicher und leichter erweiterbar ist. Die folgende Abbildung zeigt das Klassendiagramm des Interfaces.

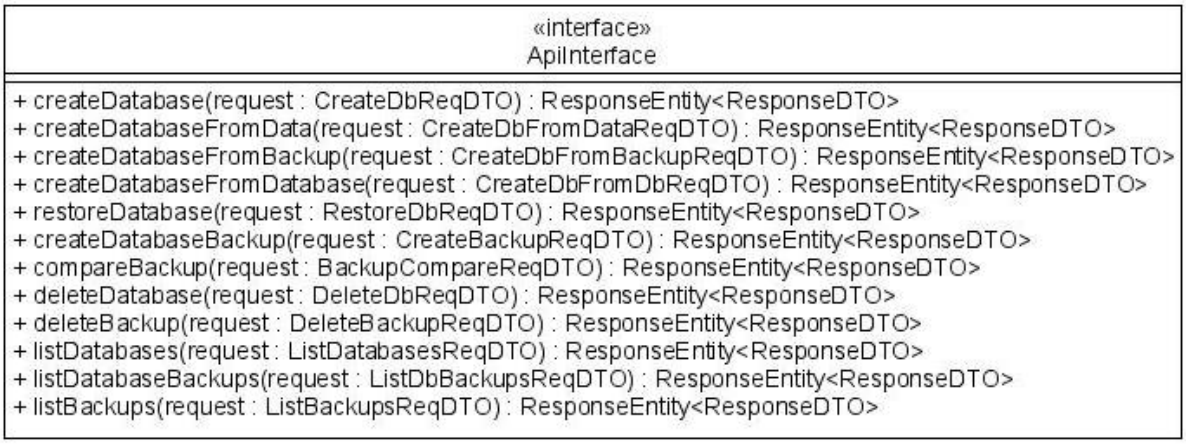


Abbildung 4 UML-Diagramm ApiInterface

Wie in der Abbildung zu erkennen, werden den Methoden die passenden Request-DTOs übergeben, damit direkt alle benötigten Informationen über ein entsprechendes Objekt zur Verfügung stehen und abgerufen werden können.

Es gibt zwei Klassen, die das zuvor vorgestellte Interface implementieren, die Klasse *RestApi* und die Klasse *DatabaseService*. Die Klasse *RestApi* implementiert die Methoden des Interface als eine auf der REST-API basierende Schnittstelle, über die Clients Anfragen via HTTP-Befehlen stellen können, und ist daher zur ersten Schicht des Servers zuzuordnen. Die Klasse *DatabaseService* implementiert die Methoden des Interfaces für die Funktionalität, dass die Anfragen an die gewünschte Komponente, die direkt mit einer Datenbank interagieren, weitergereicht werden, sofern dies möglich ist. Somit stellt diese Klasse eine Kommunikation zwischen der restbasierten Schnittstelle und der gewünschten Datenbankklassen her und sorgt dafür, dass die Anfragen immer korrekt zugeordnet werden. Aus diesem Grund ist diese Klasse der zweiten Schicht des Servers zuzuordnen.

Es ist zu beachten, dass die in dieser Arbeit entwickelte Schnittstelle zwischen dem Server und den Clients etwas von der klassischen Variante einer auf der REST-API basierenden Schnittstelle abweicht. Das bedeutet, dass die Schnittstelle zwischen dem Server und den Clients in Anlehnung zu einer klassischen restbasierten Schnittstelle entwickelt wurde. Der Grund für die Abweichung liegt in dem Mehraufwand, der die Realisierung einer klassischen restbasierten Schnittstelle mit sich bringt. Bei einer solchen Schnittstelle wäre es notwendig, ein Objektmodell zu entwickeln und zu pflegen, in dem alle Datenbanken unabhängig des Datenbank-Systems eine eindeutige ID zur Identifizierung erhalten. Da die Zuordnung zwischen Datenbanken und IDs über einen Neustart des Servers hinaus bestehen bleiben muss, wäre es notwendig, diese Informationen persistent zu speichern. Die Identifizierung der Datenbanken über eindeutige IDs ist nicht notwendig, da bei der Konzeption der Software die Entscheidung getroffen wurde, dass Datenbanken bezüglich eines Datenbank-Systems eindeutige Namen haben müssen. Das bedeutet, dass das Tupel bestehend aus dem Namen des Datenbank-Systems und Name der Datenbank für die eindeutige Identifizierung einer Datenbank vollkommen ausreicht. Die Art und Weise, wie diese Schnittstelle von den Clients aufgerufen werden kann, bleibt zur klassischen Variante identisch, bis auf die Ausnahme, dass in den einzelnen Pfaden keine IDs zu finden sind.

Über die Schnittstelle des Servers können die HTTP-Befehle *DELETE*, *GET* und *POST* aufgerufen werden, um u.a. Datenbanken oder Backups erstellen, löschen oder listen zu können. Alle zu den HTTP-Befehlen benötigten Parameter werden entweder im Body der Anfrage (*POST*) oder als Parameter in der URL (*DELETE* und *GET*) gesetzt. Abbildung 5 gibt einen Überblick über alle HTTP-Befehle, die existieren, und zeigt deren genaueren Aufbau.

Der Aufbau der Befehle folgt dabei einem bestimmten Schema:

1. Zuerst wird die gewünschte Ressource (Datenbank oder Backup einer Datenbank) definiert.
2. Als Zweites wird optional die gewünschte Aktion (vgl. Erstellen einer Datenbank mit initialen Daten oder Vergleichen zweier Backups) definiert. Die gewünschte Aktion referenziert dabei auf eine bestimmte Methode, die die gewünschte Aktion ausführt.

Abbildung 5 HTTP-Methoden der Schnittstellen

Das wichtigste Entwurfsmuster, welches in diesem Bereich des Servers eingesetzt wird, ist das Registry-Pattern [31], welches durch die Klasse *DatabaseRegistry* realisiert wird (siehe Abbildung 6). In dieser Klasse werden Objekte aller Klassen, die direkt mit einer Datenbank

interagieren, registriert. Begünstigt wird dieses Entwurfsmuster durch das Spring Framework, welches alle vom Entwickler gekennzeichneten Klassen intern verwaltet und beim Start des Servers automatisch die gewünschten Klassen in der Klasse *DatabaseRegistry* registriert. Der Grund für den Einsatz dieses Entwurfsmusters ist, dass ohne dieses die Klasse *DatabaseService* zu jeder Klasse, die direkt mit einem Datenbank-System interagiert, eine Assoziation hätte und somit eine starke Kopplung zwischen den beiden Bereichen des Servers existieren würde. Durch die Entkopplung der beiden Bereiche wird es ermöglicht, dass sie fast unabhängig voneinander erweiterbar und änderbar sind. Es ist beispielsweise möglich, weitere Komponenten dem Bereich Datenbankkomponenten hinzuzufügen, ohne etwas an der Komponente *DatabaseService* oder *DatabaseRegistry* zu ändern.

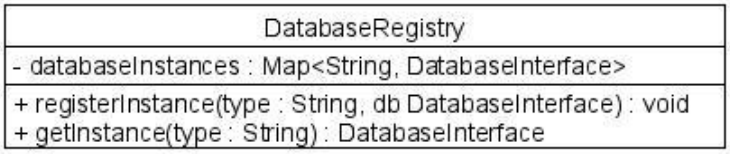


Abbildung 6 UML-Diagramm DatabaseRegistry

5.1.2 Der Bereich Datenbankkomponenten

Es wird die Architektur der Datenbankkomponenten betrachtet. Die folgende Abbildung zeigt den Aufbau der Architektur in vereinfachter Form inklusive der Einteilung der einzelnen Komponenten in Pakete.

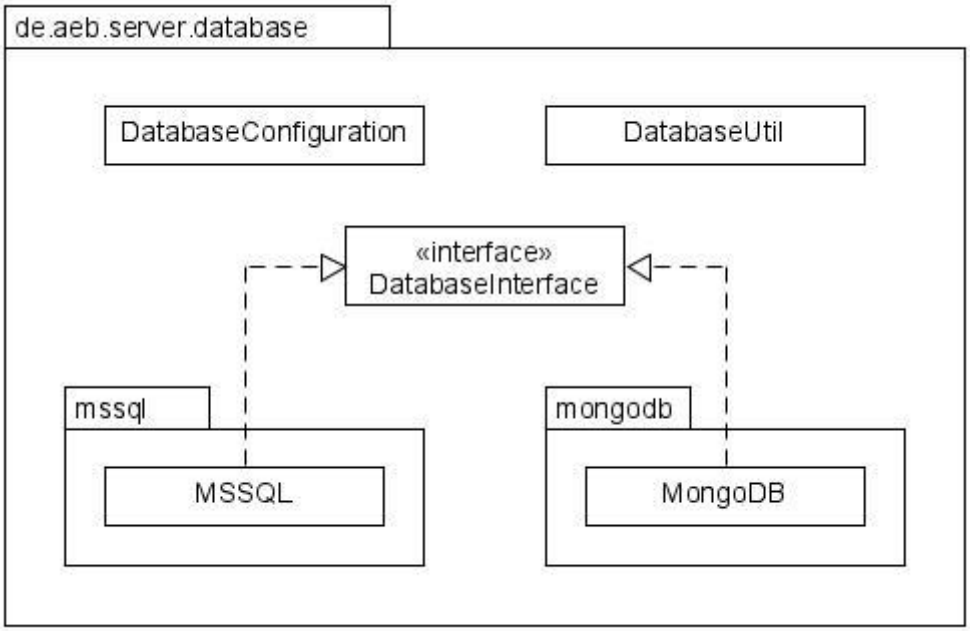


Abbildung 7 Vereinfachtes UML-Diagramm des Bereichs Datenbankkomponenten

Auch in diesem Bereich ist ein Interface – *DatabaseInterface* – definiert, welches alle benötigten Methoden, die aus den funktionalen Anforderungen hervorgegangen sind, zur Verfügung stellt. Diese Methoden können die Klassen, die direkt mit einem Datenbank-System interagieren, implementieren und um die entsprechenden Funktionalitäten erweitern. Auch hier ist der Vorteil des Einsetzens eines Interface, dass die Methoden an einer zentralen Stelle einheitlich definiert werden und somit die Schnittstellen der Klassen, die dieses Interface implementieren, einheitlich sind. Die folgende Abbildung zeigt das Klassendiagramm des Interfaces.

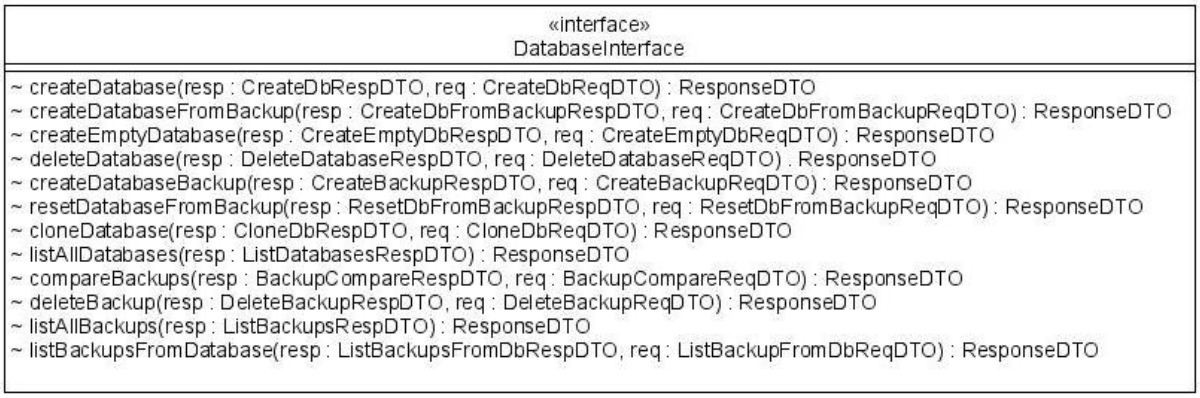


Abbildung 8 UML-Diagramm DatabaseInterface

Wie in der Abbildung zu erkennen, werden auch hier den Methoden die passenden Request-DTOs übergeben, damit direkt alle benötigten Informationen über ein entsprechendes Objekt zur Verfügung stehen und abgerufen werden können. Zusätzlich zu den Request-DTOs werden die entsprechenden Response-DTOs übergeben, damit die jeweiligen Parameter der DTOs für die Generierung der Antwort gesetzt werden können.

Eine wichtige Entwurfsentscheidung bezüglich des Interface ist, dass jegliche Hilfsmethoden, die von mehr als einer Klasse, die direkt mit einem Datenbank-System interagieren, verwendet werden kann, in eine Hilfsklasse – *DatabseUtil* – ausgelagert werden. Dadurch wird erreicht, dass das Interface nur die Methoden bereitstellt, die aus den Anwendungsfällen hervorgehen, und die Hilfsmethoden an zentraler Stelle einheitlich implementiert werden.

Die Klasse *DatabaseConfiguration* dient zur Konfiguration der einzelnen Klassen dieses Bereiches durch das Framework Spring. Beispielsweise werden die Befehle für die Datenbankaktion für die Klassen *MSSQL* und *MongoDB* beim Start des Servers aus einer externen Datei gelesen und so aufbereitet, dass die Komponenten auf eine einfache Weise die Befehle abfragen können, sobald sie eine Anfrage eines Clients bearbeiten.

5.2 Client

Aus der Beschreibung der Systemarchitektur ging hervor, dass der Client im Wesentlichen aus zwei Plugins – Maven und Gradle– besteht. Da beide Plugins mit dem Server interagieren und dessen Antworten verarbeiten, besitzen sie größtenteils die gleichen Funktionalitäten[16]. Aus diesem Grund ist es sinnvoll, die Realisierung dieser Funktionalitäten in ein Unterprojekt – *Plugin Helper* – auszulagern, damit an der Architektur des Clients Änderungen und Erweiterungen, wie das Hinzufügen weiterer Build-System-spezifischen Plugins, leichter vorgenommen werden können.

5.2.1 Plugin Helper

Es wird die Architektur des Plugin Helpers betrachtet. Der Plugin Helper stellt Funktionalitäten zur Verfügung, über die die Build-System-spezifischen Plugins Befehle an den Server stellen können. Dabei verarbeitet er die Antwort des Servers und übergibt das Ergebnis, welches entweder die gewünschte Antwort des Servers oder eine spezifische Fehlermeldung enthält, an das Build-System-spezifische Plugin zurück. Die folgende Abbildung zeigt den Aufbau der Architektur der Klasse, die die ganzen Funktionalitäten dieses Unterprojektes beinhaltet. Es ist zu beachten, dass diese Klasse in vereinfachter Form dargestellt ist und die Klassen *ServerResponseDTO*[17] und *ServerException*[18] fehlen.

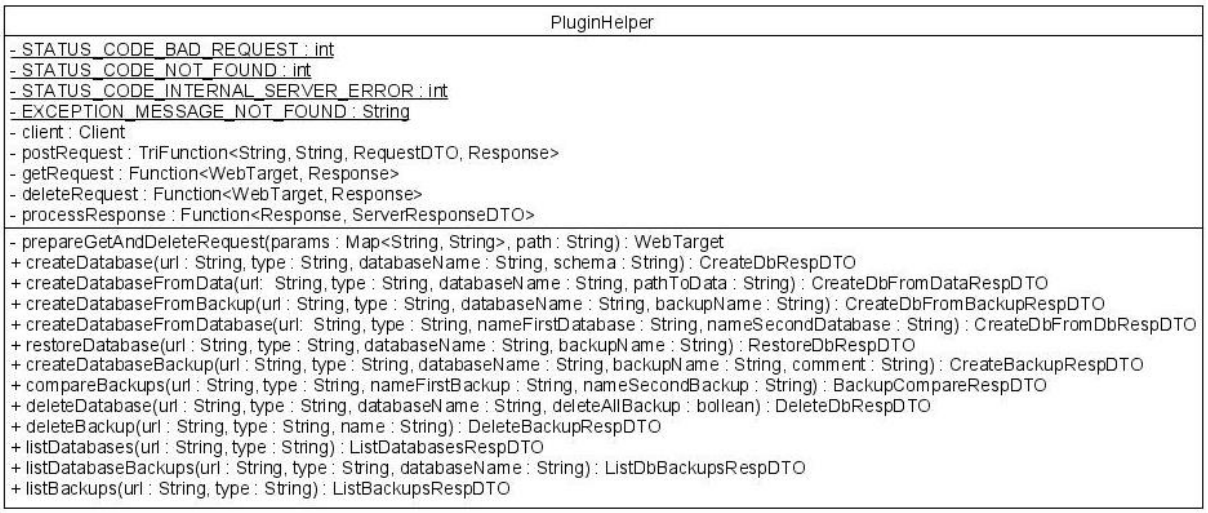


Abbildung 9 Vereinfachtes UML-Diagramm Plugin Helper

[16] Bereitstellung der Schnittstelle zum Server und Verarbeitung der Antwort des Servers.
[17] Die Klasse *ServerResponseDTO* ist ein vergleichbares Data Transfer Object zu den bereits vorgestellten Response-DTOs. Dieses DTO beinhaltet die Vereinigungsmenge der Parameter aller Response-DTOs (siehe Abbildung 2).
[18] Ein Objekt dieser Klasse wird die Informationen – Exception des Servers und Fehlermeldung - der spezifischen Fehlermeldung enthalten

Wie aus der Abbildung zu entnehmen, existiert zu jedem Anwendungsfall aus der Anforderungsanalyse eine Methode, die die Informationen des Build-System-spezifischen Clients an die entsprechende Schnittstelle des Servers übermittelt, damit dieser den Befehl ausführen und eine Antwort zurücksenden kann.

Bei der Konzeption dieser Klasse wurde die Entwurfsentscheidung getroffen, sowohl die Ausführung der spezifischen HTTP-Befehle (GET, POST und DELETE) als auch die Antwortverarbeitung über Funktionen zu realisieren. Dadurch wird gewährleistet, dass die wiederkehrenden Funktionalitäten, die für jeden Anwendungsfall identisch sind, zentral und einmalig implementiert sind, sodass Änderungen und Erweiterungen leichter vorgenommen werden können. Über die Funktionen *getRequest*, *postRequest* und *deleteRequest* können die *GET*-, *POST*- und *DELETE*-Befehle an den Server gesendet werden. Bei den *GET*- und *DELETE*-Befehlen ist zu beachten, dass über die Methode *prepareGetAndDeleteRequest* ein Vorverarbeitungsschritt notwendig ist, bei dem an die URL die benötigten Parameter drangesetzt werden.

5.2.2 Maven Plugin

Es wird die Architektur des Maven Plugins betrachtet. Aufgrund der Tatsache, dass ein Großteil der Funktionalität des Clients in den Plugin Helper ausgelagert ist, beinhaltet das Maven Plugin nur Funktionalitäten zur Informationsausgabe. Die folgende Abbildung zeigt den Aufbau der Architektur in vereinfachter Form inklusive der Einteilung der einzelnen Komponenten in Pakete.

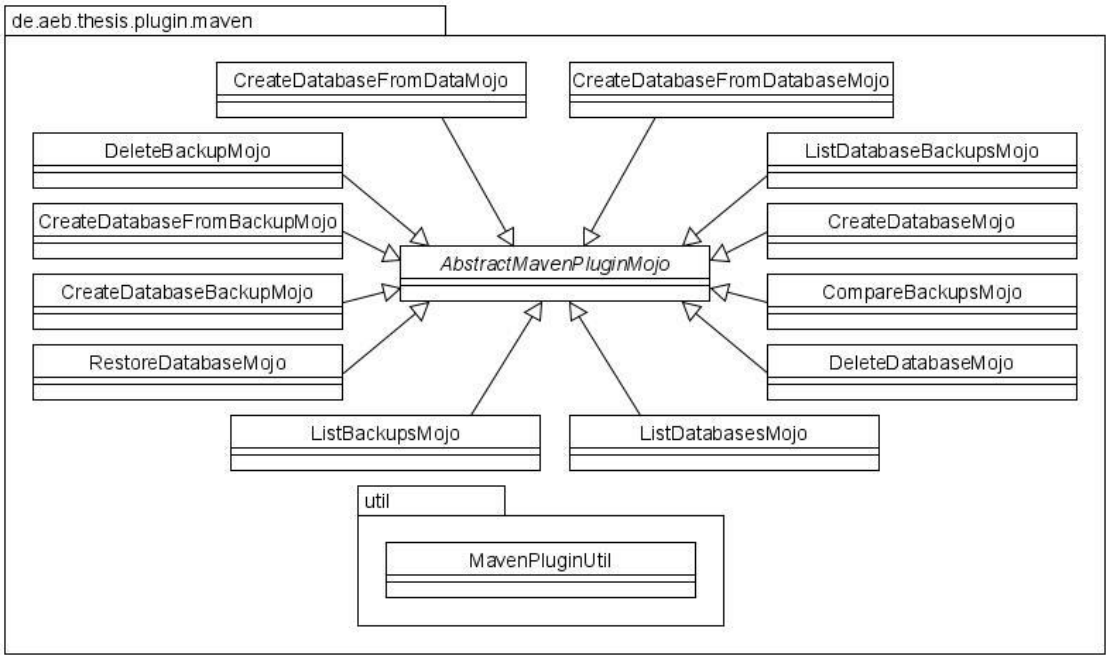


Abbildung 10 Vereinfachtes UML-Diagramm Maven Plugin

Wie in der Abbildung zu erkennen, beinhaltet das Maven Plugin für jeden Anwendungsfall aus der Anforderungsanalyse ein Mojo. Ein Mojo ist gleichzusetzen mit einem Goal, welches in einem Software-Build eines Projektes eingebunden werden kann. Durch das Einbinden eines Goals können dessen Funktionalitäten in einer bestimmten Phase des Build-Lebenszyklus aufgerufen und gleichzeitig die gewünschten Parameter gesetzt werden. Damit die Funktionalität eines Goals ausgeführt werden kann, ist es notwendig, dass diese die *execute* Methode der abstrakten Klasse *AbstractMojo* überschreibt [32].

Die in diesem Plugin entwickelten Mojos beinhalten lediglich die Parameter, die benötigt werden, um für die spezifische Anfrage an den Server die benötigten Informationen zu setzen. Die folgende Abbildung zeigt das Klassendiagramm des *DeleteDatabaseMojo*. Dort ist zu erkennen, dass zur korrekten Ausführung dieses Mojos die Parameter *databaseName*, *deleteAllBackups* und *accessDataToDatabaseDir* gesetzt werden müssen.

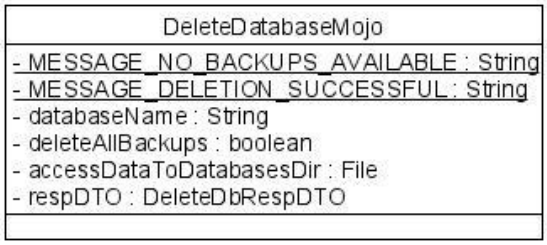


Abbildung 11 UML-Diagramm DeleteDatabaseMojo

5.2.3 Gradle Plugin

Es wird die Architektur des Gradle Plugins betrachtet. Aufgrund der Tatsache, dass die meiste Funktionalität des Clients in dem Plugin Helper ausgelagert ist, beinhaltet das Gradle Plugin nur Funktionalitäten zur Informationsausgabe. Die Abbildung 12 zeigt den Aufbau der Architektur in vereinfachter Form inklusive der Einteilung der einzelnen Komponenten in Pakete.

Das Gradle Plugin beinhaltet für jeden Anwendungsfall aus der Anforderungsanalyse einen Task und eine dazugehörige Extension. Die Klassen in dem Package *task* spiegeln eigenständige Aufgaben wider, die während des Build-Lebenszyklus aufgerufen werden können, damit die gewünschte Aktion – Erstellen einer leeren Datenbank, Löschen einer Datenbank etc. – ausgeführt wird. Dazu ist es notwendig, dass diese Klassen die von der Gradle Api bereitgestellte Klasse *DefaultTask* erweitern, um die Aktion der gewünschten Aufgabe über eine Methode zu definieren.

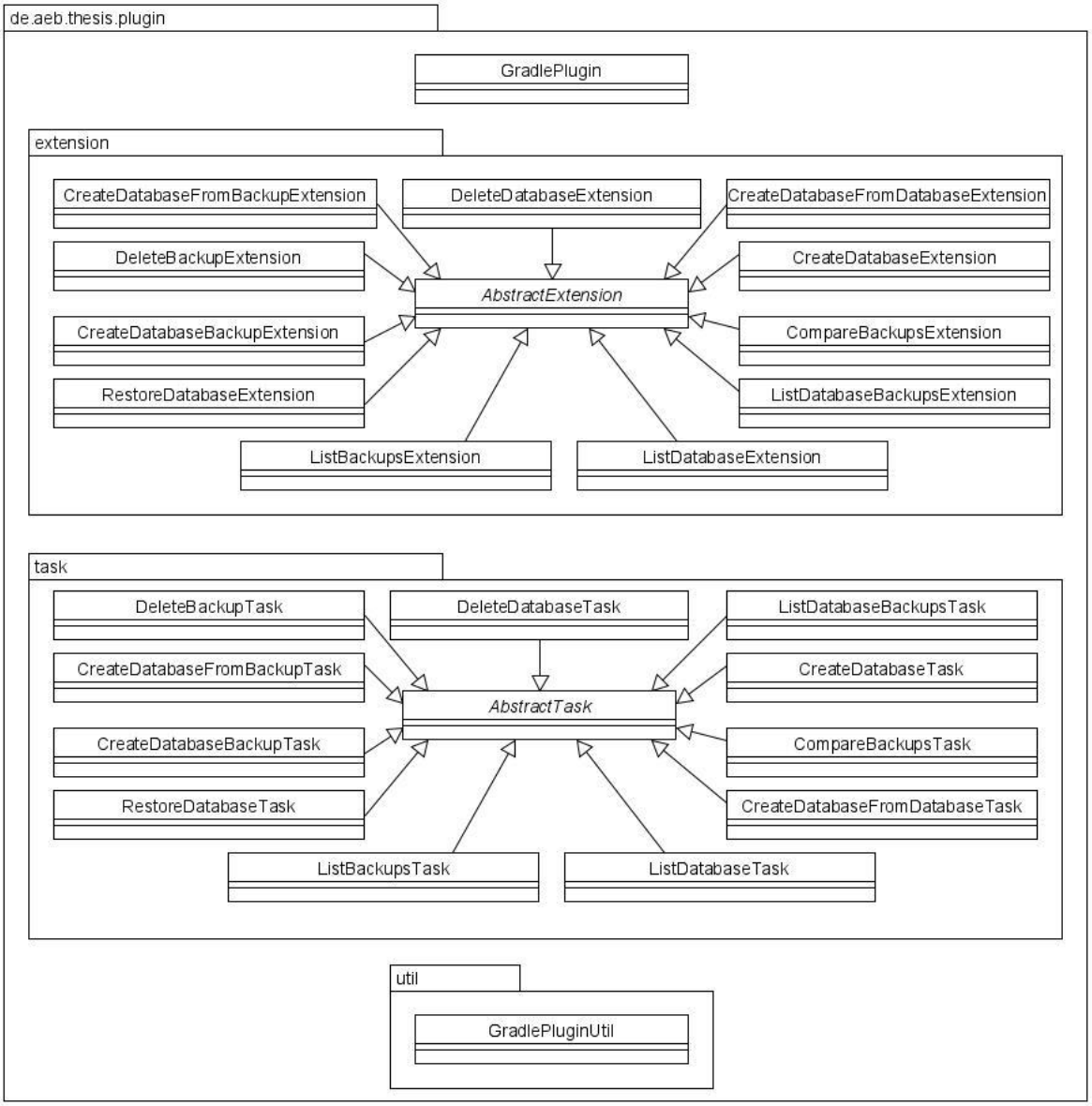


Abbildung 12 Vereinfachtes UML-Diagramm Gradle Plugin

Die folgende Abbildung zeigt das Klassendiagramm der Aufgabe *CreateDatabaseTask* zur Veranschaulichung. Dort ist zu erkennen, dass es eine Methode *createDatabase()* gibt, die die Aktion der spezifischen Aufgabe implementiert.

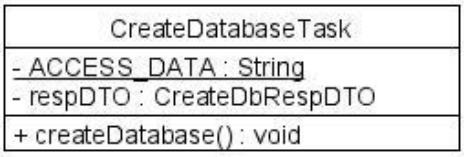


Abbildung 13 UML-Diagramm CreateDatabaseTask

Zu jeder Klasse aus dem Package *task* gehört eine Klasse aus dem Package *extension*, um die Aufgabe konfigurierbar zu machen. Diese Klassen dienen der Datenhaltung und besitzen somit nur Parameter und dazugehörige Getter- und Setter-Methoden. Die folgende Abbildung zeigt die Extension *CreateDatabaseExtension*, die zu der Klasse *CreateDatabaseTask* gehört. Dort ist zu erkennen, dass zur korrekten Ausführung dieser Aufgabe die Parameter *databaseName* und *schema* gesetzt werden müssen. Die Parameter, die Informationen über die URL des Servers und über die Auswahl des Datenbank-Systems beinhalten, sind in der abstrakten Klasse *AbstractTask* definiert.

Abbildung 14 UML-Diagramm CreateDatabaseExtension

6 Implementierung

Dieses Kapitel befasst sich mit der Implementierung der Software. Es wird darauf eingegangen, wie die Entwurfsentscheidungen realisiert wurden. Dazu werden die entsprechenden Ausschnitte des Quellcodes gezeigt. Auf die Implementierungsdetails des Unterprojektes, welches u.a. die Data Transfer Objects beinhaltet, wird nicht näher eingegangen, da diese Klassen der Datenerhaltung dienen und dementsprechend nur Konstruktoren, Getter- und Setter-Methoden sowie Variablen beinhalten.

Im Folgenden werden auch hier die Komponenten Server und Client getrennt voneinander betrachtet.

6.1 Server

Wie im Kapitel Softwarearchitektur werden auch hier die beiden Bereiche des Servers getrennt voneinander betrachtet. Die Prinzipien der Implementierung werden anhand der Anforderung „Anlegen einer DB“ [SF1] beschrieben, sodass nicht jede Methode, die aus den Anwendungsfällen aus der Anforderungsanalyse hervorgegangen ist, beschrieben werden muss, denn diese Methoden sind alle nach dem gleichen Schema entwickelt worden.

6.1.1 Der Bereich Service

Auch wenn die Klasse *RestApi* die erste Schicht des Servers präsentiert und Anfragen entgegennimmt, wird zuerst die Implementierung der Klasse *DatabaseRegistry* vorgestellt, da die Registrierung der Klassen, die direkt mit einem Datenbank-System interagieren, für die korrekte Funktionsweise des Servers sehr wichtig ist. Die folgende Abbildung zeigt die Implementierung der Klasse *DatabaseRegistry*.

```
1  @Component("registry")
2  @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
3  public class DatabaseRegistry {
4
5      @Autowired
6      private Map<String, DatabaseInterface> databaseInstances;
7
8      public void registerInstance(String type, DatabaseInterface db) {
9          databaseInstances.put(type, db);
10     }
11
12     public DatabaseInterface getInstance(String type) {
13         return databaseInstances.get(type);
14     }
15
16 }
```

Abbildung 15 Implementierung DatabaseRegistry

Wie in der Abbildung zu erkennen, besteht die Klasse aus zwei Methoden und einer Variablen, durch die das Entwurfsmuster *Registry* [31] realisiert wird. Die Klassen, die direkt mit einem Datenbank-System interagieren, werden in einer *Map* mittels eines eindeutigen Schlüssels gespeichert. Über die Methode *registerInstance* können Objekte dieser Klassen registriert und über die Methode *getInstance* können diese Objekte wieder abgefragt werden, sofern diese registriert sind.

Die Klasse *DatabaseRegistry* ist mit zwei Annotationen versehen, die aus dem Spring Framework hervorgehen. Die Annotation *@Component("registry")* signalisiert dem Framework, dass diese Klasse mittels des eindeutigen Namens *registry* verwaltet werden soll, sodass u.a. Objekte dieser Klasse durch Spring in andere Klassen injiziert werden können, sofern diese dort benötigt werden. Durch die Annotation *@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)* wird angegeben, dass Spring nur ein einziges Objekt dieser Klasse erzeugen darf.

Die Variable *databaseInstance* ist ebenfalls mit einer Annotation, die aus dem Spring Framework hervorgeht, versehen. Diese wird allerdings erst bei der Implementierungsbeschreibung der Klasse *DatabaseService* näher erklärt.

Die folgende Abbildung zeigt einen Ausschnitt der Implementierung der Klasse *RestApi*.

```
1  @RestController
2  public class RestApi implements ApiInterface {
3
4      private DatabaseService service;
5
6      @Override
7      @PostMapping(value = RestUtil.REST_PATH_CREATE_EMPTY_DATABASE)
8      public ResponseEntity<ResponseDTO> createDatabase(@RequestBody CreateDbReqDTO json) {
9          return json.isComplete() ? generateErrorMessage() : service.createDatabase(json);
10     }
11
12     private ResponseEntity<ResponseDTO> generateErrorMessage() {
13         return ResponseEntity.status(HttpStatus.BAD_REQUEST)
14                 .body(new ResponseDTO(MESSAGE_MISSING_PARAMETERS));
15     }
16
17 }
```

Abbildung 16 Implementierung der Klasse RestApi – Erstellen einer leeren DB

Auch diese Klasse ist mit einer Annotation, die aus dem Spring Framework hervorgeht, versehen. Mithilfe der Annotation *@RestController* kann in Spring ein restbasierter Webservice entwickelt werden, der es u.a. ermöglicht, den Inhalt der Anfragen in ein Data Transfer Object und die dazugehörigen Antworten in JSON oder XML zu konvertieren. In der Methode *createDatabase* wird der Body der HTTP-Anfrage in das Data Transfer Object *CreateDbReqDTO* konvertiert, damit die Anfrage bearbeitet werden kann. Gekennzeichnet

wird dies durch die Annotation *@RequestBody.* Durch den Rückgabewert *ResponseEntity<ResponseDTO>* dieser Methode wird gekennzeichnet, dass der Inhalt des Objektes der Klasse *ResponseDTO* zu JSON konvertiert werden soll. Die Antwort beinhaltet somit den Inhalt des Objektes in JSON.

Damit die Methode *createDatabase* von Spring als eine Schnittstelle erkannt wird und Clients Anfragen an diese stellen können, muss die Methode mit der Annotation *@PostMapping(value = RestUtil.REST_PATH_CREATE_EMPTY_DATABASE)* versehen werden. *PostMapping* definiert dabei, dass nur POST-Befehle an diese Methode über den Pfad, der in den Klammern angegeben ist, gestellt werden können.

Die Methode *createDatabase* überprüft mit *json.isComplete()*, ob alle Parameter des Objektes *CreateDbReqDTO* gesetzt sind und somit die Anfrage vollständig ist. Ggf. wird durch die Methode *generateErrorMessage* eine Fehlermeldung mit dem HTTP-Statuscode *BAD REQUEST* erzeugt. Sofern die Anfrage vollständig ist, wird das Objekt *json* an die Methode *createDatabase* der Klasse *DatabaseService* weitergeleitet.

Zu der Weiterleitung passend wird nun die Implementierung der Klasse *DatabaseService* betrachtet. Der folgende Ausschnitt zeigt die Implementierung dieser Klasse.

```
1  @Component("service")
2  @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
3  public class DatabaseService implements ApiInterface {
4
5      private DatabaseRegistry databaseRegistry;
6
7      @Autowired
8      public DatabaseService(DatabaseRegistry databaseRegistry) {
9          this.databaseRegistry = databaseRegistry;
10     }
11
12     @Override
13     public ResponseEntity<ResponseDTO> createDatabase(CreateDbReqDTO req) {
14         DatabaseInterface db = databaseRegistry.getInstance(req.getType());
15         Supplier<ResponseDTO> supplier = () -> db.createDatabase(new CreateDbRespDTO(), req);
16         return function.apply(HttpStatus.CREATED, req, supplier);
17     }
18
19     private ResponseEntity<ResponseDTO> generateResponse(ResponseDTO resp, HttpStatus status) {
20         return resp.getErrorMessage() != null ?
21             ResponseEntity.status(HttpStatus.BAD_REQUEST).body(resp) :
22             ResponseEntity.status(status).body(resp);
23     }
24
25 }
```

Abbildung 17 Implementierung der Klasse DatabaseService – Erstellen einer leeren DB

Die Klasse *DatabaseService* hat mit der Klasse *DatabaseRegistry* eine Assoziation und erhält über den Konstruktor ein Objekt dieser Klasse. Der Konstruktor ist mit der Annotation *@Autowired* versehen, die dem Spring Framework kennzeichnet, dass ein Objekt der Klasse

DatabaseRegistry via Konstruktor-Injection injiziert werden soll. Dazu ist es notwendig, dass Spring diese Klasse verwaltet, was bereits am Anfang dieses Kapitels gezeigt wurde.

Die Methode *createDatabase* erwartet ein Objekt der Klasse *CreateDbReqDTO*, welches den Inhalt der Anfrage repräsentiert. In der ersten Anweisung dieser Methode wird mit *databaseRegistry.getInstance(req.getType())* der Name des gewünschten Datenbank-Systems aus der Anfrage und mittels dieses Namens das entsprechende Objekte in der Registrierung - repräsentiert durch die Klasse *DatabaseRegistry* - abgefragt. Anschließend wird ein *Supplier* definiert, der ein *ResponseDTO* für die Antwort als Rückgabewert hat. Der *Supplier* beinhaltet die Anweisung, die Anfrage des Clients an die entsprechende Komponente, die direkt mit einem Datenbank-System interagiert, weiterzuleiten. In der dritten Anweisung dieser Methode wird eine Funktion, die drei Eingaben und eine Ausgabe besitzt, ausgeführt. Die folgende Abbildung zeigt die Definition und Implementierung dieser Funktion.

```
1  @Component("service")
2  @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
3  public class DatabaseService implements ApiInterface {
4
5      @FunctionalInterface
6      public interface TriFunction<F, S, T, R> {
7          public R apply(F first, S second, T third);
8      }
9
10     private TriFunction<HttpStatus, RequestDTO, Supplier<ResponseDTO>, ResponseEntity<ResponseDTO>>
11         function = (status, req, sup) -> {
12             DatabaseInterface db = databaseRegistry.getInstance(req.getType());
13             if (db == null) {
14                 ResponseDTO resp = new ResponseDTO();
15                 resp.setErrorMessage(MESSAGE_INCORRECT_DATABASE_TYPE);
16                 return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(resp);
17             } else {
18                 return generateResponse(sup.get(), status);
19             }
20     };
21
22 }
```

Abbildung 18 Implementierung der TriFunction der Klasse DatabaseService

Die definierte Funktion ist auf Grund der drei Eingaben eine *TriFunction* und erwartet in der Implementierung einen *HttpStatus*, ein Objekt der Klasse *RequestDTO* und den *Supplier*, der bereits in der vorherigen Methode definiert wurde, als Eingaben. Als Rückgabewert gibt diese Funktion ein Objekt der Klasse *ResponseEntity<ResponseDTO>* zur Antwortgenerierung wieder.

In der ersten Anweisung dieser Funktion wird erneut das gewünschte Objekt der Klasse, die direkt mit einem Datenbank-System interagiert, abgefragt. Im negativen Fall – es existiert kein Objekt der gewünschten Klasse - wird eine Fehlermeldung mit dem HTTP-Statuscode *BAD REQUEST* erzeugt. Im positiven Fall wird die Anweisung, die in dem Supplier definiert wurde,

ausgeführt. Anschließend wird mittels der Methode *generateResponse*, der Antwort, die aus der Ausführung des *Suppliers* hervorgeht, und dem gewünschten HTTP-Statuscode eine Antwort für den Client generiert.

In der Methode *generateResponse()* wird geprüft, ob bei der Ausführung der Anfrage auf Seiten des gewünschten Objektes der Klasse, die direkt mit einem Datenbank-System interagiert, ein Fehler aufgetreten ist. Ist dies der Fall, so wird in der Antwort für den Client statt dem gewünschten HTTP-Statuscode – in diesem Fall *CREATED* – der Statuscode *BAD REQUEST* festgelegt.

6.1.2 Der Bereich Datenbankkomponenten

Bevor direkt auf die Implementierung der Klassen *MSSQL* und *MongoDB* bezüglich des festgelegten Anwendungsfalles näher eingegangen wird, wird die Implementierung der Klassen *DatabaseConfiguration* und *DatbaseUtils* betrachtet. Der folgende Ausschnitt zeigt die Implementierung der Klasse *DatabaseConfiguration*.

```
1  @Configuration
2  public class DatabaseConfiguration {
3
4      @Bean(name = "mssqlCommands")
5      @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
6      public Map<String, String> getMSSQLCommands() throws IOException{
7          File commands = Paths.get("src")
8              .resolve("main").resolve("resources").resolve("mssql_commands.json")
9              .toFile();
10         return getCommands(commands);
11     }
12
13     @Bean(name = "mongodbCommands")
14     @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
15     public Map<String, String> getMongoDBCommands() throws IOException{
16         File commands = Paths.get("src")
17             .resolve("main").resolve("resources").resolve("mongodb_commands.json")
18             .toFile();
19         return getCommands(commands);
20     }
21
22 }
```

Abbildung 19 Implementierung DatabaseConfiguration

Wie bereits im Kapitel Softwarearchitektur beschrieben, dient diese Klasse dazu, die Klassen *MSSQL* und *MongoDB* zu konfigurieren. Dazu werden die entsprechenden JSON-Dateien, die alle benötigten Befehle beinhalten, eingelesen und so aufbereitet, dass auf diese Befehle schnell zugegriffen werden kann. Der Prozess des Einlesens und der Aufbereitung wird wieder größtenteils von dem Spring Framework übernommen. Dazu sind in dieser Abbildung zwei neue Annotation - *@Configuration* und *@Bean* - zu entnehmen.

Mittels der Annotation *@Bean* wird Spring darauf hingewiesen, dass die annotierte Methode

ein Objekt zurückgibt, welches in dem Anwendungskontext von Spring verwaltet werden soll. In diesem Fall sind es die Methoden *getMSSQLCommands()* und *getMongoDBCommands()*, die die jeweiligen Befehle des Datenbank-Systems in einer *Map* zurückgeben. Es ist zu beachten, dass in den Klammern der Annotation *@Bean* der Name des zu verwaltenden Objektes angegeben ist, um dieses im weiteren Anwendungsverlauf eindeutig zu identifizieren.

Sowohl die Methode *getMSSQLCommands()* als auch *getMongoDBCommands()* rufen die Methode *getCommands(commands)* auf. In dieser Methode wird mithilfe eines Objektes der Klasse *ObjectMapper* der Inhalt der JSON-Datei in eine *Map* übertragen.

Die zweite neue Annotation, die in dieser Klasse definiert ist, ist die *@Configuration*-Annotation. Diese Annotation zeigt Spring an, dass diese Klasse Methoden beinhaltet, die mit der Annotation *@Bean* versehen sind.

Die Klasse *DatabaseUtils* ist eine Hilfsklasse und beinhaltet u.a. vordefinierte Nachrichten für die spezifischen Fehlermeldungen, die verwendet werden, falls es bei der Verarbeitung der Anfrage des Clients aufseiten der Datenbank zu einem Fehler gekommen ist, und Methoden, die von beiden Klassen - *MSSQL* und *MongoDB* – genutzt werden können. Beispielsweise gibt es die Methode *String getFileChecksum(File file)* zur Ermittlung des Hash-Wertes einer Datei oder die Methode *List<String> sortListOfFiles(Path path)*, um die Dateien eines gegebenen Pfades in einer Liste alphabetisch zu sortieren.

Im Folgenden wird die Klasse *MSSQL* vorgestellt. Dazu wird erst ein Ausschnitt der Konfiguration der Klasse gezeigt und anschließend die Implementierung des festgelegten Anwendungsfalles. Die folgende Abbildung zeigt die Implementierung der Konfiguration dieser Klasse.

```
1  @Component("MSSQL")
2  @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
3  @PropertySource("classpath:application.properties")
4  public class MSSQL implements DatabaseInterface {
5
6      @Autowired
7      @Value("${mssql.db.url}")
8      private String url;
9      @Autowired
10     @Value("${mssql.db.name}")
11     private String servername;
12     @Autowired
13     @Value("${mssql.db.user}")
14     private String user;
15     @Autowired
16     @Value("${mssql.db.password}")
17     private String password;
18     @Autowired
19     @Value("${mssql.db.path.databases}")
20     private String pathToDatabases;
21     @Autowired
22     @Value("${mssql.db.path.backups}")
23     private String pathToBackups;
24     @Autowired
25     @Value("${mssql.db.path.export.files}")
26     private String pathToExportFiles;
27
28     @Autowired
29     Map<String,String> mssqlCommands;
30
31 }
```

Abbildung 20 Implementierung der Konfiguration der Klasse MSSQL

Die Klasse *MSSQL* wird durch zwei Quellen konfiguriert. Zum einen wird sie durch die Klasse *DatabaseConfiguration* und zum anderen durch eine Property-Datei mit dem Namen *application.properties* konfiguriert.

Zur Konfiguration wird die Property-Datei über die Annotation *@PropertySource* – eine Annotation von Spring - eingebunden. Auf diese Weise kann Spring auf die Key-Value-Paare, die in dieser Datei definiert sind, zugreifen und die Parameter *url*, *servername*, *user*, *password* etc. setzen. Damit diese Parameter durch Spring gesetzt werden können, müssen diese neben der Annotation *@Autowired* zusätzlich mit der Annotation *@Value* versehen sein. Mittels *${mssql.db.name}* kann beispielsweise der Parameter *servername* gesetzt werden. Dazu holt Spring über die Annotation *@Value* den Value-Wert des angegebenen Paares, der in der Property-Datei definiert ist.

Die folgende Abbildung zeigt das Key-Value-Paar für den Parameter *servername* aus der Property-Datei zur Veranschaulichung.

```
1  mssql.db.name=PC-XDDEV0222\\XE4AEBDEV
```

Abbildung 21 Property-Datei Key-Value-Paar

Der Parameter *mssqlCommands* der Klasse *MSSQL* wird nicht über die Property-Datei konfiguriert, sondern durch die Klasse *DatabaseConfiguration*. Zuvor wurde bereits diese Klasse vorgestellt und gezeigt, dass es eine Methode gibt, die ein Objekt zurückgibt, welches von Spring verwaltet wird und die benötigten SQL-Befehle beinhaltet. Der Name des zu verwalteten Objektes wurde auf den identischen Namen des Parameters *mssqlCommands* gesetzt, damit Spring dieses Objekt eindeutig zuordnen und den Parameter setzen kann.

Die folgende Abbildung zeigt die Implementierung der Klasse *MSSQL* für den festgelegten Anwendungsfall.

```
1  @Component("MSSQL")
2  @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
3  @PropertySource("classpath:application.properties")
4  public class MSSQL implements DatabaseInterface {
5
6      @Override
7      public ResponseDTO createDatabase(CreateDbRespDTO resp, CreateDbReqDTO req) {
8          try(Connection connection = createConnection()) {
9              if(checkIfDatabaseAlreadyExist(req.getDatabaseName())) {
10                 resp.setErrorMessage(DatabaseUtils.ERROR_DATABASE_ALREADY_EXISTS);
11             } else {
12                 connection.setAutoCommit(false);
13                 try(Statement stmt = connection.createStatement()) {
14                     stmt.execute(
15                         String.format(mssqlCommands.get("CREATE_DATABASE"), req.getDatabaseName())
16                     );
17                     stmt.execute(
18                         String.format(mssqlCommands.get("USE_DATABASE"), req.getDatabaseName())
19                     );
20                     stmt.execute(
21                         String.format(mssqlCommands.get("CREATE_SCHEMA"), req.getSchema())
22                     );
23                 }
24
25                 connection.commit();
26                 resp.setAccessData(setAccessDataToResponse());
27             }
28         } catch (SQLException e) {
29             resp.setErrorMessage(DatabaseUtils.ERROR_CREATION_DATABASE_FAILED);
30             resp.setException(e.getMessage());
31         }
32
33         return resp;
34     }
35
36 }
```

Abbildung 22 Implementierung der Klasse MSSQL – Erstellen einer leeren DB

Dort wird zuerst über die Methode *createConnection()* eine Verbindung zu dem Datenbank-System mittels den Parametern *url*, *user* und *password* hergestellt. Anschließend wird mit der Methode *checkIfDatabaseAlreadyExist(req.getDatabaseName())* überprüft, ob es bereits eine Datenbank mit dem gewünschten Namen, der in der Anfrage enthalten ist, gibt. Existiert bereits eine Datenbank mit dem gewünschten Namen, so wird eine entsprechende Fehlermeldung in dem Objekt der Klasse *CreateDbRespDTO* – spezifisches Data Transfer

Object für diesen Anwendungsfall – gesetzt. Bei dem anderen Entscheidungsfall wird ein Objekt der Klasse Statement erzeugt, um die SQL-Befehle zum Anlegen einer leeren Datenbank mit dem gewünschten Namen und Schema ausführen zu können. Anschließend werden die Zugangsdaten zur erstellten Datenbank – *url*, *user* und *password* - in dem DTO für die Antwort gesetzt.

Tritt bei der Bearbeitung der SQL-Befehle aufseiten des Datenbank-Systems ein Fehler auf, so wird dieser über eine SQLException abgefangen und in dem Objekt der Klasse *CreateDbRespDTO* eine entsprechende Fehlermeldung mit dem Inhalt der SQLException gesetzt.

Die Abbildung 23 zeigt die Implementierung der Methode *boolean checkIfBackupAlreadyExist(String name) throws SQLException* zur Veranschaulichung. Diese Methode wurde ausgelagert, da diese bei manchen Methoden der anderen Anwendungsfälle ebenfalls benötigt wird. In dieser Methode wird erneut eine Verbindung zu dem Datenbank-System aufgebaut, da nicht jede Methode, die sie aufruft, eine bereits bestehende Verbindung übergeben kann. Die Methode *boolean checkIfBackupAlreadyExist(String name) throws SQLException* ist prinzipiell genauso aufgebaut wie die Methode *ResponseDTO createDatabase(CreateDbRespDTO resp, CreateDbReqDTO req).* Nachdem eine Verbindung zu dem Datenbank-System aufgebaut wurde, wird ein Objekt der Klasse *Statement* erzeugt, um den SQL-Befehl auszuführen. Da dieser SQL-Befehl ein Ergebnis zurückgibt, wird ein Objekt der Klasse *ResultSet* benötigt, welches das Ergebnis beinhaltet und verarbeiten kann. In diesem Fall muss nur überprüft werden, ob das Ergebnis leer – es existiert keine Datenbank mit dem gewünschten Namen - oder nicht leer - es existiert eine Datenbank mit dem gewünschten Namen – ist.

```
1  private boolean checkIfDatabaseAlreadyExist(String name) throws SQLException {
2      try(Connection connection = createConnection();
3              Statement stmt = connection.createStatement();
4              ResultSet res = stmt.executeQuery(
5                  String.format(mssqlCommands.get("SELECT_DATABASE_BY_NAME"), name)
6              )
7      ) {
8          return res.next();
9      }
10 }
```

Abbildung 23 Implementierung der Methode checkIfDatabaseAlreadyExist der Klasse MSSQL

Im Folgenden wird die Klasse *MongoDB* vorgestellt. Auch hier wird erst ein Ausschnitt der Konfiguration der Klasse und anschließend die Implementierung des festgelegten Anwendungsfalles gezeigt. Die folgende Abbildung zeigt die Implementierung der Konfiguration dieser Klasse.

```
1  @Component("MONGODB")
2  @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
3  @PropertySource("classpath:application.properties")
4  public class MongoDB implements DatabaseInterface {
5
6      @Autowired
7      private MongoClient client;
8
9      @Autowired
10     @Value("${spring.data.mongodb.host}")
11     private String host;
12     @Autowired
13     @Value("${spring.data.mongodb.port}")
14     private String port;
15     @Autowired
16     @Value("${spring.data.mongodb.backup.path}")
17     String pathToBackups;
18
19     @Autowired
20     Map<String,String> mongodbCommands;
21
22 }
```

Abbildung 24 Implementierung der Konfiguration der Klasse MongoDB

Die Klasse *MongoDB* wird analog zu der Klasse *MSSQL* konfiguriert. Die Parameter *host*, *port* und *pathToBackups* bzw. *mongodbCommands* werden ebenfalls durch Spring mittels der Property-Datei bzw. der zugehörigen Methode aus der Klasse *DatabaseConfiguration* gesetzt. Zusätzlich zu diesen Parametern wird ein Objekt der Klasse *MongoClient* gesetzt. Das Spring Framework bringt bereits einen Client mit, mit dem es möglich ist, Befehle auszuführen.

Die Abbildung 25 zeigt die Implementierung der Klasse *MongoDB* für den festgelegten Anwendungsfall. Der Ablauf dieser Methode ist analog zu dem Ablauf der gleichen Methode aus der Klasse *MSSQL*. Zuerst wird überprüft, ob die Datenbank mit dem gewünschten Namen bereits existiert, und ggf. die Datenbank über den *MongoClient* mit der Methode *getDatabase(req.getDatabaseName())* erstellt. Bei der Erstellung der Datenbank ist es notwendig, dass eine (leere) Collection unter Angabe eines Namens erstellt wird. Anschließend können wie bei MSSQL die Zugangsdaten in dem zugehörigen Data Transfer Object für die Antwort gesetzt werden.

In der Methode *boolean checkIfDatabaseAlreadyExist(String name)* wird über den *MongoClient* eine Liste aller Namen existierender Datenbanken abgefragt und mittels Iteration überprüft, ob der gewünschte Name in der Liste enthalten ist.

```java
@Component("MONGODB")
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
@PropertySource("classpath:application.properties")
public class MongoDB implements DatabaseInterface {

    @Override
    public ResponseDTO createDatabase(CreateDbRespDTO resp, CreateDbReqDTO req) {
        if(checkIfDatabaseAlreadyExist(req.getDatabaseName())) {
            resp.setErrorMessage(DatabaseUtils.ERROR_DATABASE_ALREADY_EXISTS);
        } else {
            MongoDatabase db = client.getDatabase(req.getDatabaseName());
            db.createCollection(COLLECTION_DEFAULT);
            resp.setAccessData(setAccessDataToResponse());
        }
        return resp;
    }

    private boolean checkIfDatabaseAlreadyExist(String name) {
        try(MongoCursor<String> dbs = client.listDatabaseNames().iterator()) {
            while(dbs.hasNext()) {
                if(name.equals(dbs.next())) {
                    return true;
                }
            }
            return false;
        }
    }

}
```

Abbildung 25 Implementierung MongoDB – Erstellen einer leeren DB

Aufgrund der Tatsache, dass nicht alle Anwendungsfälle mithilfe des *MongoClients* realisiert werden können, sondern einige durch die Befehle *mongudump* – Erstellen eines Backups – und *mongorestore* – Wiederherstellen einer Datenbank mittels eines Backups – ausgeführt werden müssen, wird im Folgenden der Anwendungsfall „Anlegen eines Backups einer DB“ [SF7] betrachtet.

Die Abbildung 26 zeigt die Implementierung dieses Anwendungsfalles der Klasse *MongoDB*. Der Einstiegspunkt dieses Anwendungsfalles in dieser Klasse bildet die Methode *ResponseDTO createDatabaseBackup(CreateBackupRespDTO resp, CreateBackupReqDTO req)*, die in der Abbildung nicht enthalten ist. In dieser Methode wird im ersten Schritt überprüft, ob die angegebene Datenbank existiert, und im zweiten Schritt, ob es bereits ein Backup mit dem gewünschten Namen gibt. Existieren die Datenbank und das Backup nicht, wird die Methode *void createDatabaseBackup(String databaseName, String backupName, String comment) throws IOException, InterruptedException* aufgerufen (siehe Abbildung 26). Dort wird zuerst der Pfad mit dem gewünschten Namen des Backups, in dem die Dateien gespeichert werden sollen, bestimmt und anschließend der Befehl *mongodump* ausgeführt. Dieser Befehl erwartet neben den Zugangsdaten den Namen der Datenbank und den zuvor bestimmten Pfad. Zum Schluss wird am gleichen Speicherort eine Informationsdatei mit dem Kommentar zur Erstellung des Backups des Clients und eine Datei, die die aktuelle

Zeit beinhaltet, gespeichert.

```
@Component("MONGODB")
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
@PropertySource("classpath:application.properties")
public class MongoDB implements DatabaseInterface {

    private void createDatabaseBackup(String databaseName, String backupName, String comment)
    throws IOException, InterruptedException {
        Path backupDestinationPath = Path.of(pathToBackups).resolve(databaseName).resolve(backupName);

        Runtime.getRuntime()
            .exec(String.format(
                mongodbCommands.get("MONGO_DUMP"),
                host,
                port,
                databaseName,
                backupDestinationPath.toString()
            ))
            .waitFor();

        Files.writeString(
            backupDestinationPath.resolve(INFO_FILE_NAME),
            comment
        );

        Files.writeString(
            backupDestinationPath.resolve(DATE_FILE_NAME),
            new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date())
        );
    }

}
```

Abbildung 26 Implementierung MongoDB – Erstellen eines Backups

6.2 Client

Es wird die Implementierung der Komponente Client beschrieben, dabei wird zuerst der Plugin Helper und anschließend die beiden Plugins – Maven und Gradle - betrachtet. Die Prinzipien der Implementierung werden anhand der Anforderung „Anlegen einer DB“ [SF1] beschrieben, sodass nicht jede Methode, die aus den Anwendungsfällen aus der Anforderungsanalyse hervorgegangen ist, beschrieben werden muss, denn diese Methoden sind alle nach dem gleichen Schema entwickelt worden.

6.2.1 Plugin Helper

Die Abbildung 27 zeigt einen Ausschnitt der Implementierung des Plugin Helpers. Über die Methode *CreateDbRespDTO createDatabase(String url, String type, String databaseName, String schema)* wird eine Anfrage an den Server zum Erstellen einer leeren Datenbank gesendet. Diese Methode erwartet die dazu notwendigen Informationen wie URL des Servers, Name des Datenbank-Systems und Name der Datenbank. In der ersten Anweisung dieser

Methode werden die übergebenen Informationen in ein Data Transfer Object übergeben. Anschließend wird mittels der Funktion *postRequest*, die drei Eingaben und eine Ausgabe besitzt, die Anfrage an den Server gesendet und die Antwort gespeichert. Die Antwort wird mittels der Funktion *processResponse* verarbeitet. Nach der Verarbeitung wird das entsprechende Data Transfer Object für diesen Anwendungsfall erzeugt und zurückgegeben.

```
1  public class PluginHelper {
2
3      public static CreateDbRespDTO createDatabase(String url,String type,String databaseName,String schema) {
4          CreateDbReqDTO dto = new CreateDbReqDTO(type, databaseName, schema);
5          Response resp = postRequest.apply(url, RestUtil.REST_PATH_CREATE_EMPTY_DATABASE, dto);
6          ServerResponseDTO respDTO = processResponse.apply(resp);
7          return new CreateDbRespDTO(
8              respDTO.getErrorMessage(),
9              respDTO.getException(),
10             respDTO.getDatabaseUrl(),
11             respDTO.getUser(),
12             respDTO.getPassword()
13         );
14     }
15
16 }
```

Abbildung 27 Implementierung PluginHelper - Erstellen einer leeren DB

Die folgende Abbildung zeigt die Implementierung der Funktion *postRequest*.

```
1  private static TriFunction<String, String, RequestDTO, Response> postRequest = (url, path, dto) -> client
2          .target(url)
3          .path(path)
4          .request(MediaType.APPLICATION_JSON)
5          .post(Entity.entity(dto, MediaType.APPLICATION_JSON));
```

Abbildung 28 Implementierung der Funktion postRequest der Klasse PluginHelper

Diese Funktion erwartet die URL des Servers, den Pfad zu der Schnittstelle zum Server und ein Data Transfer Object – in diesem Fall ein Objekt der Klasse *CreateDbReqDTO*. Aus diesen Informationen wird der POST-Befehl generiert und an den Server geschickt. Die Antwort des Servers gibt diese Funktion direkt wieder zurück. Die GET- und DELETE-Befehle für die anderen Anwendungsfälle sind analog aufgebaut.

Die Abbildung 29 zeigt die Implementierung der Funktion *processResult*. Diese Funktion verarbeitet die Antwort, die die Funktion *postRequest* zurückgegeben hat. Im ersten Schritt wird überprüft, ob der Server erreicht worden ist, und im zweiten Schritt, ob ein Fehler mit dem Statuscode BAD_REQUEST oder INTERNAL_SERVER_ERROR aufgetreten ist. In beiden Fällen wird ein Objekt der Klasse *ServerException* mit allen Informationen zu diesen Fehlern erzeugt, falls das erwartete positive Ergebnis bezüglich des Anwendungsfalles ausbleibt. Ist kein Fehler aufgetreten, so wird der Inhalt der Antwort des Servers auf das Objekt der Klasse *ServerResponseDTO* übertragen.

```
1  private static Function<Response, ServerResponseDTO> processResponse = resp -> {
2      if(resp.getStatus() == STATUS_CODE_NOT_FOUND) {
3          throw new ServerException(STATUS_CODE_NOT_FOUND, EXCEPTION_MESSAGE_NOT_FOUND, null);
4      } else if (resp.getStatus() == STATUS_CODE_BAD_REQUEST
5              || resp.getStatus() == STATUS_CODE_INTERNAL_SERVER_ERROR) {
6          String result = resp.readEntity(String.class);
7          ResponseDTO respDTO = new Gson().fromJson(result, ResponseDTO.class);
8          throw new ServerException(resp.getStatus(), respDTO.getErrorMessage(), respDTO.getException());
9      } else {
10         return resp.readEntity(ServerResponseDTO.class);
11     }
12 };
```

Abbildung 29 Implementierung der Funktion processResult der Klasse PluginHelper

6.2.2 Maven Plugin

```
1  @Mojo(name = "createDatabase")
2  public class CreateDatabaseMojo extends AbstractMavenPluginMojo {
3
4      @Parameter(required = true)
5      String databaseName;
6
7      @Parameter(required = true)
8      String schema;
9
10     @Parameter(required = true)
11     File accessDataToDatabasesDir;
12
13     private CreateDbRespDTO respDTO;
14
15     @Override
16     public void execute() throws MojoExecutionException, MojoFailureException {
17         try {
18             respDTO = PluginHelper.createDatabase(url, type, databaseName, schema);
19             getLog().info(String.format(
20                 LOG_ACCESS_DATA, respDTO.getDatabaseUrl(), respDTO.getUser(), respDTO.getPassword()
21             ));
22             MavenPluginUtil.saveAccessData(
23                 accessDataToDatabasesDir.toPath(),
24                 databaseName,
25                 respDTO.getDatabaseUrl(),
26                 respDTO.getUser(),
27                 respDTO.getPassword()
28             );
29         } catch (ServerException e) {
30             getLog().error(String.format(
31                 SERVER_EXCEPTION_MESSAGE_STATUSCODE, e.getStatuscode(), e.getMessage()
32             ));
33             getLog().error(String.format(
34                 SERVER_EXCEPTION_MESSAGE_FROM_SERVER, e.getExceptionMessageFromServer()
35             ));
36             throw new MojoFailureException(e.getMessage(), e);
37         } catch (ProcessingException e) {
38             getLog().error(PROCESSING_EXCEPTION_MESSAGE);
39             throw new MojoFailureException(e.getMessage(), e);
40         } catch (Exception e) {
41             getLog().error(EXCEPTION_MESSAGE, e);
42             throw new MojoExecutionException(e.getMessage(), e);
43         }
44     }
45
46 }
```

Abbildung 30 Implementierung des Maven Plugins - Erstellen einer leeren DB

In Abbildung 30 ist die Implementierung des Goals zum Erstellen einer leeren Datenbank abgebildet. Die Klasse, die dieses Goal repräsentiert, ist mit der Annotation *@Mojo* versehen. Der Namenstag identifiziert dieses Mojo eindeutig. Es ist in der Abbildung zu erkennen, dass dieses Goal zwei Parameter besitzt, die mit der Annotation *@Parameter* versehen sind. Diese Annotation kennzeichnet die Parameter, über die ein Goal in einem Software-Build konfiguriert werden kann. In diesem Fall bedeutet es, dass u.a. der Name der Datenbank und der Name des Schemas in einem Software-Build gesetzt werden können. Mit dem Tag *required* = *true* wird angegeben, dass diese Parameter immer gesetzt werden müssen.

In der Methode *execute()*, die bei der Ausführung des Goals aufgerufen wird, wird über den Plugin Helper die entsprechende Methode zum Erstellen einer leeren Datenbank mit den benötigten Informationen aufgerufen. Es ist zu beachten, dass die übergebenen Parameter *url* und *type* in der abstrakten Klasse *AbstractMavenPluginMojo* definiert und konfigurierbar sind. Das Ergebnis, welches der Plugin Helper liefert, wird auf der Konsole ausgegeben. Tritt ein Fehler in der Ausführung der Anfrage auf, wird die entsprechende Fehlermeldung ausgegeben.

6.2.3 Gradle Plugin

Die folgende Abbildung zeigt die Implementierung des Gradle Plugins. Es ist zu beachten, dass die abgebildete Implementierung nur den Anwendungsfall zum Erstellen einer leeren Datenbank beinhaltet.

```
1  public class GradlePlugin implements Plugin<Project> {
2
3      @Override
4      public void apply(Project project) {
5          project.getExtensions()
6              .create("createDatabase", CreateDatabaseExtension.class);
7
8          project.getTasks()
9              .create("createDatabase", CreateDatabaseTask.class);
10     }
11
12 }
```

Abbildung 31 Implementierung des Gradle Plugins - Erzeugung der Extension und des Tasks

Die Klasse *GradlePlugin* ist der Einstiegspunkt zur Ausführung des Plugins und muss die Klasse *Plugin<T>* erweitern. Über die Methode *void apply(Project project)* wird das Plugin aufgerufen und ausgeführt. In der ersten Anweisung dieser Methode wird ein Objekt der Klasse *CreateDatabaseExtension* mit der gewünschten Konfiguration, die aus dem Software-Build hervorgeht, erzeugt. Anschließend wird mit der zweiten Anweisung die gewünschte Aufgabe, die durch die Klasse *CreateDatabaseTask* repräsentiert wird, ausgeführt.

Die folgende Abbildung zeigt die Implementierung der Klasse *CreateDatabaseTask*.

```
1  public class CreateDatabaseTask extends AbstractTask {
2
3      private CreateDbRespDTO respDTO;
4
5      @TaskAction
6      public void createDatabase() {
7          CreateDatabaseExtension extension = getProject()
8                  .getExtensions()
9                  .findByType(CreateDatabaseExtension.class);
10
11         if(extension.isNotComplete()) {
12             throw new GradleException(MESSAGE_MISSING_PARAMETER);
13         } else {
14             try {
15                 respDTO = PluginHelper.createDatabase(
16                     extension.getUrl(),
17                     extension.getType(),
18                     extension.getDatabaseName(),
19                     extension.getSchema()
20                 );
21                 getProject().getLogger().lifecycle(String.format(
22                     ACCESS_DATA,
23                     respDTO.getDatabaseUrl(),
24                     respDTO.getUser(),
25                     respDTO.getPassword()
26                 ));
27                 GradlePluginUtil.saveAccessData(
28                     Path.of(extension.getAccessDataToDatabasesDir()),
29                     extension.getDatabaseName(),
30                     respDTO.getDatabaseUrl(),
31                     respDTO.getUser(),
32                     respDTO.getPassword()
33                 );
34             } catch (ServerException e) {
35                 getProject().getLogger().lifecycle(String.format(
36                     SERVER_EXCEPTION_MESSAGE_STATUSCODE,
37                     e.getStatuscode(),
38                     e.getMessage(
39                 )));
40                 getProject().getLogger().lifecycle(String.format(
41                     SERVER_EXCEPTION_MESSAGE_FROM_SERVER,
42                     e.getExceptionMessageFromServer()
43                 ));
44                 throw e;
45             } catch (ProcessingException e) {
46                 getProject().getLogger().lifecycle(PROCESSING_EXCEPTION_MESSAGE);
47                 throw new GradleException(e.getMessage(), e);
48             } catch (Exception e) {
49                 getProject().getLogger().lifecycle(EXCEPTION_MESSAGE, e);
50                 throw new GradleException(e.getMessage(), e);
51             }
52         }
53     }
54
55 }
```

Abbildung 32 Implementierung der Gradle Aufgabe zum Erstellen einer leeren DB

Es ist aus dieser zu entnehmen, dass die Methode *createDatabase()* die Funktionalität dieser Aufgabe implementiert. Dazu ist diese mit der Annotation *@TaskAction* versehen, die Gradle kennzeichnet, dass eine Methode die Aktion einer spezifischen Aufgabe implementiert. Die Methode *createDatabase()* ist weitestgehend analog zu der gleichen Methode des Maven

Plugins definiert. In der ersten Anweisung dieser Methode wird die zu dieser Klasse bzw. Aufgabe gehörende Extension abgerufen, um auf die Konfiguration zugreifen zu können. Anschließend wird überprüft, ob die Konfiguration vollständig ist und ggf. eine Fehlermeldung ausgegeben. Ist die Konfiguration vollständig, so wird über den Plugin Helper die entsprechende Methode zum Erstellen einer leeren Datenbank mit den benötigten Informationen aufgerufen. Das Ergebnis, welches der Plugin Helper liefert, wird auf der Konsole ausgegeben. Tritt ein Fehler in der Ausführung der Anfrage auf, wird die entsprechende Fehlermeldung ausgegeben.

7 Nachweisführung

In der Einleitung dieser Arbeit wurde eine Strategie definiert, auf welche Art und Weise nachgewiesen werden soll, dass die Software universell einsetzbar ist. Ziel war es, die Architektur der Software so zu konzipieren, dass der Server mit unterschiedlichen Datenbank-Systemen und unterschiedlichen Clients – Build-System-spezifische Plugins – interagieren kann, um zu zeigen, dass die Software um weitere Datenbank-Systeme und Clients erweitert werden kann. Im Folgenden wird die Nachweisstrategie und dessen Umsetzung reflektiert.

Aufgrund der Tatsache, dass die Serverkomponente der Software durch die Verwendung des Spring Frameworks und des Entwurfsmusters Registry-Pattern in zwei Bereiche einteilbar ist, die voneinander entkoppelt sind, ist es möglich, die Software um weitere Klassen, die direkt mit einem Datenbank-System interagieren, zu erweitern, ohne umfassende Änderungen an anderen Klassen vorzunehmen. Dabei war die Entscheidung, ein Framework zu nutzen, welches die gewünschten Klassen intern verwaltet und Objekte der Klassen in anderen Klassen injiziert, sofern es vom Entwickler gewünscht ist, essenziell wichtig für die Umsetzung der Nachweisstrategie bezüglich der Datenbank-Systeme. So wurde eine enge Kopplung zwischen einzelnen Klassen verhindert und Assoziationen zwischen einzelnen Klassen aufgebrochen. Speziell bei der Entwicklung neuer Klassen muss dem Framework nur über eine entsprechende Annotation angezeigt werden, dass diese Klasse ebenfalls verwaltet werden soll.

Bei der Entwicklung weiterer Klassen, die direkt mit einem Datenbank-System interagieren, ist zu beachten, dass die Architektur insofern erweiterbar ist, dass zwischen dem Interface und den Klassen, die sich auf Datenbank-Systeme mit einem gleichen Datenbankmodell beziehen, eine abstrakte Klasse implementiert werden kann, um Funktionalitäten zu abstrahieren, die spezifisch für das Datenbankmodell sind, aber nicht spezifisch für die Datenbank-Systeme. Da in dieser Arbeit zwei Datenbank-Systeme betrachtet worden sind, deren Datenbankmodelle unterschiedlich sind, wurde auf abstrakte Klassen verzichtet.

Es wurde bei der Beschreibung der Softwarearchitektur des Clients gezeigt, dass die beiden Clients – Maven Plugin und Gradle Plugin – größtenteils die gleichen Funktionalitäten besitzen müssen, damit beide das gleiche Verhalten aufweisen. Aus diesem Grund wurde sich dazu entschieden, diese Funktionalitäten in eine andere unabhängige Komponente – *Plugin Helper* – auszulagern, damit jeder Client die Funktionalitäten dieser Komponente nutzen kann. Durch die Realisierung dieser Entwurfsentscheidung wurde erreicht, dass das Verhalten des Maven Plugins und Gradle Plugins nahezu identisch ist. Zudem können weitere Plugins auf eine einfache und schnelle Art hinzugefügt werden, da der Großteil der Funktionalitäten, die für die

korrekte Arbeitsweise eines neuen Plugins notwendig ist, bereits im *Plugin Helper* implementiert ist. Der Plugin Helper unterstützt somit die Vereinheitlichung der unterschiedlichen Clients, die sicherstellt, dass die unterschiedlichen Plugins im Wesentlichen alle identisch funktionieren, sofern sie die Funktionalitäten des Plugin Helpers korrekt aufrufen.

Rückblickend wurde die Nachweisstrategie sinnvoll gewählt und umgesetzt. Die Architektur der Software weist eine hohe Abstraktion auf und besitzt einige Komponenten, die Funktionalitäten bündeln oder Assoziationen aufbrechen. Dadurch ist es auf eine einfache Art und Weise möglich, weitere Plugins und weitere Komponenten, die direkt mit einem Datenbank-System interagieren, hinzuzufügen. Zudem ist es beispielsweise möglich, alle Plugins identisch und einmalig zu ändern, indem Änderungen am Plugin Helper vorgenommen werden. Die Architektur der gesamten Software ist somit leicht änderbar, beliebig erweiterbar und wartbar.

8 Evaluation

In diesem Kapitel wird die Software auf die korrekte Funktionsweise und Gebrauchstauglichkeit in Bezug auf die ermittelten Anforderungen evaluiert. Um zu überprüfen, ob die definierten Anforderungen erfüllt sind, wird getestet, ob die technische Eignung gegeben ist. Zusätzlich wird gezeigt, inwiefern sich die Software in einen bestehenden Prozess integrieren lässt.

8.1 Tests

In der Softwareentwicklung gibt es unterschiedliche Arten von Tests, die für unterschiedliche Zwecke eingesetzt werden können. In dieser Arbeit wurden bei der Server-Komponente Unit-Tests und Integrationstests und bei der Client-Komponente Akzeptanztests angewendet.

Unit-Tests werden in der Softwareentwicklung eingesetzt, um einzelne Komponenten einer Software bzw. einzelne Methoden zu testen. Es soll mit diesen Tests sichergestellt werden, dass eine einzelne Einheit unabhängig von anderen korrekt funktioniert. Bei den Integrationstests wird dagegen das Zusammenwirken mehrerer aufeinander abgestimmter Komponenten einer Software getestet, damit sichergestellt wird, dass die Schnittstellen zwischen den einzelnen Komponenten korrekt funktionieren und das Zusammenwirken keine unerwünschten Seiteneffekte erzeugt. Die Akzeptanztests testen die gesamte Software aus Sicht des Benutzers. Wichtige Aspekte bei dieser Testart sind, dass die Software korrekt funktioniert und vom Benutzer akzeptiert wird [33].

Im Folgenden wird die Strategie gezeigt, auf welche Art und Weise getestet wurde. Anschließend wird sich näher mit den Testarten in Zusammenhang mit der entwickelten Software befasst und die Testabdeckung betrachtet.

Strategie

Vor der Entwicklung der einzelnen Tests wurden Testszenarien[19] definiert, um sicherzustellen, dass alle ermittelten Anforderungen geprüft werden - auch in ihrem Zusammenspiel. Diese Szenarien wurden, sofern es sinnvoll war, bei allen drei Testarten analog umgesetzt, um ein einheitliches Testverfahren zu gewährleisten. Nach jedem durchgeführten Test wird die jeweilige Test-Datenbank auf den Stand vor Ausführung des Tests zurückgesetzt.

Es ist zu beachten, dass in dieser Arbeit die korrekte Funktionsweise des Maven Plugins nur

[19] Die Testszenarien sind im Anhang zu finden.

durch das Beispielprojekt geprüft wird. Der Plugin Helper wird nicht einzeln, sondern im Zusammenspiel mit den beiden Plugins getestet. Die Klassen, die die Data Transfer Objects repräsentieren, und ähnliche Klassen werden nicht getestet, da sie ausschließlich zur Datenhaltung dienen. Das Verhalten der Software beim Auftreten von unerwarteten Exceptions wie die SQLException oder die IOException werden nicht in die Tests miteinbezogen, da diese in den Anwendungsfällen relativ schwer zu simulieren sind.

Unit-Tests

Die Unit-Tests beziehen sich auf die Klassen, die direkt mit einem Datenbank-System interagieren. Um die korrekte Funktionsweise der Klassen *MSSQL* und *MongoDB* zu überprüfen, ist es nur notwendig, eine Anbindung zu dem jeweiligen Datenbank-System zu haben. Die Schnittstelle zu den anderen Klassen des Servers – *DatabaseService* oder *RestApi* – ist in diesem Fall uninteressant, da die Interaktion über die Schnittstelle ausschließlich von der Klasse *DatabaseService* gestartet und geführt wird.

Integrationstests:

Die Integrationstests beziehen sich auf die Klasse *DatabaseService* bzw. auf die Klasse *RestApi*, um das Zusammenwirken der Komponenten *DatabaseService* und *MSSQL*/*MongoDB* bzw. *RestApi*, *DatabaseService* und *MSSQL*/*MongoDB* bezüglich der definierten Testszenarien zu testen. Auf diese Art und Weise wird schrittweise die korrekte Funktionsweise des Servers und das korrekte Zusammenwirken der einzelnen Schichten getestet.

Akzeptanztests

Die Akzeptanztests testen das Zusammenwirken des Servers und des Gradle Plugins. Dazu ist es notwendig, dass der Server zur Testlaufzeit gestartet ist und ununterbrochen läuft. Bei diesen Tests wird gleichzeitig das korrekte Verhalten des Plugin Helpers mitgetestet, sodass die korrekte Funktionsweise der Software anhand eines Plugins überprüft wird.

Testabdeckung

Die Abbildung 33 zeigt die Testabdeckung der Server-Komponente der Software. Es ist aus dieser zu entnehmen, dass dort mit 90% eine hohe Testabdeckung gewährleistet ist. Die restlichen 10% spiegeln die unerwarteten Exceptions und die Klasse, die den Server startet, wider. Des Weiteren wurde bei einigen if-else-Anweisungen nur die erste Bedingung überprüft oder nur ein Zweig ausgeführt. Beispielseise wurde bei der Anweisung *if(firstBackupPath == null || secondBackupPath == null)* der Test so geschrieben, dass nur der erste Teil der

Bedingung *true* ergibt. Würde man bei solchen Anweisungen die komplette Bedingung überprüfen und beide Zweige ausführen, so würde die Prozentzahl der Testabdeckung steigen und die Anzahl der fehlenden Zweige verringern.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
de.aeb.thesis.server.database.mssql		91%		87%
de.aeb.thesis.server.database.mongodb		88%		91%
de.aeb.thesis.server		5%		n/a
de.aeb.thesis.server.database		94%		87%
de.aeb.thesis.server.service.service		100%		100%
de.aeb.thesis.server.service.rest		100%		100%
de.aeb.thesis.server.service.registry		100%		n/a
Total	264 of 2,858	90%	15 of 170	91%

Abbildung 33 Testabdeckung des Servers

Die folgende Abbildung zeigt die Testabdeckung des Gradle Plugins.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
de.aeb.thesis.plugin.task		76%		90%
de.aeb.thesis.plugin.util		90%		100%
de.aeb.thesis.plugin.extension		98%		68%
de.aeb.thesis.plugin		100%		n/a
Total	359 of 2,108	82%	22 of 98	77%

Abbildung 34 Testabdeckung des Gradle Plugins

Die Testabdeckung des Gradle Plugins ist mit 82% im Gegensatz zur Testabdeckung des Servers schlechter. Dies liegt daran, dass auch hier die unerwarteten Exceptions nicht berücksichtigt werden und bei einigen if-else-Anweisungen nur eine Bedingung überprüft und nur ein Zweig ausgeführt wird.

In Anbetracht der ermittelten Anforderungen und deren Zusammenhänge, sofern eine Sequenz von Anforderungen ausgeführt wird, wurden jegliche mögliche Szenarien in Testfälle umgesetzt, sodass immer ihr Idealfall ausgeführt, aber auch Fehlerfälle[20] simuliert wurden. Die möglichen Fehlerfälle, dass der Server bei den Akzeptanztests nicht erreichbar ist oder das Datenbank-System aufgrund eines internen Fehlers bei der Ausführung einer Anfrage abstürzt und somit eine *SQLException* oder eine *IOException* liefert, sind sehr schwer zu simulieren. Aus diesem Grund kann eine noch höhere Testabdeckung nicht erreicht werden. Die Überprüfung des Verhaltens der Software, sofern es nicht zu einem unerwarteten Fehler führt, ist vollständig abgedeckt und gewährleistet die korrekte Funktionsweise.

[20] Die Datenbank mit dem gewünschten Namen existiert bereits oder das angegebene Backup existiert nicht.

8.2 Beispielprojekt

In diesem Kapitel wird gezeigt, wie sich die entwickelte Software in ein bestehendes Maven-Projekt integrieren lässt. Dies wird anhand des Maven Plugins gezeigt. Bereits im Kapitel der Grundlagen wurde vorgestellt, wie ein externes Plugin in ein Projekt eingebunden werden kann. Über das eindeutige Tupel aus Gruppen-ID, Artefakt-ID und Versionsnummer wird das Plugin in der pom.xml eingebunden und die gewünschten Goals ausgeführt und beliebig konfiguriert.

Die Abbildung 35 zeigt den Ausschnitt aus der pom.xml eines Maven-Projektes, wie sich das Plugin einbinden lässt und welche weiteren Plugins notwendig sind, damit die Integrationstest korrekt ausgeführt werden können. Neben dem Maven Plugin wird das Failsafe Plugin benötigt. Die beiden Plugins werden innerhalb des Tags *build* eingebunden. In diesem Bereich werden alle Plugins definiert, die während der Erstellung des Projektes ausgeführt werden [34].

Das Failsafe Plugin (Zeile 3 bis 22) wird im Build-Lebenszyklus während den Integrationstests verwendet. Dieser Teil des Build-Lebenszyklus besteht aus vier Phasen. Die erste Phase ist die Pre-Integration-Test-Phase, die für die Einrichtung der Integrationstestumgebung dient. Die darauffolgende Phase – Integration-Test-Phase – dient für die Durchführung der Integrationstests. In der dritten Phase – Post-Integration-Test-Phase – wird die Integrationstestumgebung wieder gelöscht. Die letzte Phase – Verify-Phase – dient zur Überprüfung der Ergebnisse der Integrationstests. Es ist zu beachten, dass beim Auftreten eines Fehlers in der Durchführung der Tests die entsprechende Phase nicht abgebrochen wird und die nachfolgenden Phasen ausgeführt werden. Das bedeutet, dass der Build nicht fehlschlagen und die Ausführung der Post-Integrationstest-Phase ermöglicht wird, damit die Integrationstestumgebung nicht ungewollt bestehen bleibt. Beim Failsafe-Plugin werden die Goals *integration* und *verify* ausgeführt [35].

In Zeile 23 bis 59 ist die Einbindung des Maven Plugins dargestellt. Es ist als Anwendungsbeispiel[21] gewählt worden, dass eine leere Datenbank in der Pre-Integration-Phase erstellt und diese anschließend wieder in der Post-Integration-Phase gelöscht wird. Die Erstellung bzw. das Löschen der Datenbank wird durch das Ausführen der entsprechenden Goals des Maven Plugins – *createDatabase* bzw. *deleteDatabase* – realisiert. In den Zeilen 28 bis 42 bzw. 43 bis 57 ist dargestellt, wie die Goals konfiguriert sind. Die Goals haben eine eindeutige ID und sind an die entsprechende Phase gebunden, um die

[21] Weitere Anwendungsbeispiele sind im Anhang zu finden.

Integrationstestumgebung zu erstellen bzw. zu löschen. Zusätzlich sind die benötigten Parameter gesetzt, damit das jeweilige Goal korrekt ausgeführt werden kann.

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>3.0.0-M5</version>
            <dependencies>
                <dependency>
                    <groupId>org.junit.jupiter</groupId>
                    <artifactId>junit-jupiter-engine</artifactId>
                    <version>5.7.0-M1</version>
                </dependency>
            </dependencies>
            <!-- Configuration missing. Definition of the included tests -->
            <executions>
                <execution>
                    <goals>
                        <goal>integration-test</goal>
                        <goal>verify</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>de.aeb.thesis</groupId>
            <artifactId>MavenPlugin</artifactId>
            <version>0.0.1-SNAPSHOT</version>
            <executions>
                <execution>
                    <id>create-databases</id>
                    <phase>pre-integration-test</phase>
                    <configuration>
                        <url>http://localhost:8080</url>
                        <type>MSSQL</type>
                        <databaseName>DemoDb</databaseName>
                        <schema>demo</schema>
                        <accessDataToDatabasesDir>${project.basedir}/src/test/resources/access-data-databases
                        </accessDataToDatabasesDir>
                    </configuration>
                    <goals>
                        <goal>createDatabase</goal>
                    </goals>
                </execution>
                <execution>
                    <id>delete-databases</id>
                    <phase>post-integration-test</phase>
                    <configuration>
                        <url>http://localhost:8080</url>
                        <type>MSSQL</type>
                        <databaseName>DemoDb</databaseName>
                        <deleteAllBackups>true</deleteAllBackups>
                        <accessDataToDatabasesDir>${project.basedir}/src/test/resources/access-data-databases
                        </accessDataToDatabasesDir>
                    </configuration>
                    <goals>
                        <goal>deleteDatabase</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Abbildung 35 Integration des Maven Plugins in ein bestehendes Maven-Projekt

9 Fazit

Zum Abschluss werden die Arbeit und die Ergebnisse aus dieser zusammengefasst sowie ein Ausblick über die weitere Verwendung der entwickelten Software gegeben.

9.1 Zusammenfassung und Ergebnisse

Die Kernaufgabe dieser Thesis besteht darin, den bestehenden Produkten der AEB SE eine Schnittstelle über eine verteilte Software bereitzustellen, über die während der Integrationstests mit wenig Konfigurationsaufwand automatisiert Datenbanken angelegt und verwaltet werden können. Darüber hinaus wurde untersucht, inwiefern die Schnittstelle universell eingesetzt werden kann. Universell heißt in diesem Kontext, dass die Software mit möglichst unterschiedlichen Datenbank-Systemen und Build-System-spezifischen Plugins interagieren soll.

Zur Einführung in das Thema dieser Arbeit wurde ein Überblick in die grundlegenden und wichtigen Begriffe, die aus der Kernaufgabe und den Zielen der Arbeit hervorgegangen sind, behandelt und, sofern es möglich war, in Zusammenhang gebracht. Es wurden die Anforderungen an die Software – auch im Zusammenhang mit dem Team Software Infrastructure am Standort Lübeck – ermittelt und festgehalten, um anschließend die Architektur der Software zu entwickeln. Da die Software u.a. aus einem Maven Plugin, einem Gradle Plugin, einem Plugin für MSSQL und einem Plugin für MongoDB besteht und die Architektur im Allgemeinen einen hohen Grad von Änderbarkeit, Erweiterbarkeit und Wartbarkeit aufweisen muss, war im Vorfeld die praktische Einarbeitung in die Ökosysteme von Maven, Gradle, MSSQL und MongoDB sowie die Recherche nach hilfreichen Architekturprinzipien und Frameworks notwendig. Diese Einarbeitung ist zum Großteil in einer dreimonatigen Tätigkeit als Werkstudent bei der AEB SE geschehen.

Aufbauend auf den ermittelten Anforderungen wurde die Systemarchitektur und anschließend die Softwarearchitektur der zu entwickelnden Software konzipiert. Dabei wurden in Hinblick auf die recherchierten Architekturprinzipien und Frameworks grundlegende Entwurfsentscheidungen getroffen, um die Architektur der Software so zu gestalten, dass sie universell einsetzbar ist. Es folgte die Implementierung, sodass ein funktionsfähiger Prototyp entstand.

Die Software ist nun in der Lage alle Szenarien aus den User Stories, die in der Anforderungsanalyse entwickelt wurden, auszuführen. Es ist möglich eine Sequenz von Szenarien unter der Bedingung, dass diese Sequenz sinnvoll gestellt und die Szenarien aufeinander aufbauend sind, auszuführen. Beispielsweise sollte eine Datenbank zuerst erstellt

werden, bevor diese wieder gelöscht werden kann. Die Sequenz, bestehend aus zwei Szenarien, würde ansonsten als nicht sinnvoll angesehen werden.

Bei der Entwicklung der Software lag der Fokus auf dessen einfacher Änderbarkeit, Erweiterbarkeit und Wartbarkeit. Die dazu getroffenen Entwurfsentscheidungen wie der Einsatz des Spring Frameworks [30] mit der verbundenen Dependency Injection, die Auswahl des Registry-Patterns [31] und die Entwicklung eines Plugin Helpers waren sehr sinnvoll und haben zum Erreichen des Zieles dieser Arbeit nennenswert beigetragen. Mit diesen Entwurfsentscheidungen wurde eine Architektur für die Software entwickelt, bei der die einzelnen Komponenten bzw. Klassen so atomar und unabhängig voneinander wie möglich sind, indem Abhängigkeiten aufgebrochen und ähnliche oder gleiche Funktionalitäten an zentraler Stelle implementiert wurden. Mit dem Einsatz des Registry-Patterns wurde erreicht, dass der Server die Plugins für die Datenbank-Systeme verwaltet, ohne dass andere Komponenten des Servers die konkreten Plugins kennen müssen.

Die an die Software gestellten Anforderungen konnten komplett umgesetzt werden. Die Funktionsfähigkeit wurde mit einer Vielzahl von Tests aus unterschiedlichen Testkategorien und anhand eines kleines Beispielsprojektes überprüft. Die universelle Einsetzbarkeit der Software wurde durch eine Nachweisführung mit einer entsprechenden Strategie belegt.

9.2 Ausblick

Die entwickelte Software kann als ein Prototyp angesehen werden, die, sofern die Software bereitgestellt wird und Server für die Datenbank-Systeme aufgesetzt werden, in die bestehenden Prozesse der AEB SE integriert werden kann. Am Anfang der schriftlichen Ausarbeitung wurde die Motivation und das Interesse seitens der AEB SE zu diesem Thema und dessen Umsetzung aufgezeigt. Nun soll die Integration erfolgen und die Software um die Anbindung weiterer Datenbank-Systeme erweitert werden, damit die Software in vielen Abteilungen des Unternehmens eingesetzt werden kann.

Vor der Integration der Software in die bestehenden Systeme kann eine Anpassung bei der Verarbeitung der Antworten des Servers aufseiten der Plugins oder bei der Realisierung des Anwendungsfalles zum Vergleichen zweier Backups sinnvoll sein. Die Verarbeitung der Antworten wurden prototypisch so umgesetzt, dass sie bei den spezifischen Plugins nur geloggt werden. Der Vergleich der Backups kann (beliebig) genauer umgesetzt werden. Beispielsweise könnte bei dem Datenbank-System MSSQL die Backups bezüglich weiterer Aspekte wie Stored Procedures etc. intensiver verglichen werden.

Literaturverzeichnis

[1] G. Kim, J. Humble, P. Debois und J. Willis, Das DevOps Handbuch, O'Reilly, 2017.

[2] E. Wolff, Continuous Delivery, dpunkt.verlag, 2016.

[3] J. Humble, „Continuous Delivery," [Online]. Available: https://continuousdelivery.com/. [Zugriff am 17 Mai 2020].

[4] M. Fowler, „DeploymentPipeline," Martin Fowler, 30 Mai 2013. [Online]. Available: https://martinfowler.com/bliki/DeploymentPipeline.html. [Zugriff am 18 Mai 2020].

[5] M. Fowler, „Continuous Delivery," Martin Fowler, 30 Mai 2013. [Online]. Available: https://martinfowler.com/bliki/ContinuousDelivery.html. [Zugriff am 17 Mai 2020].

[6] S. Beyer, „Einführung in Jenkins," Softwareforen Leipzig, 31 Oktober 2018. [Online]. Available: https://blog.softwareforen.de/2018/10/einfuehrung-in-jenkins/. [Zugriff am 01 April 2020].

[7] S. McIntosh, M. Nagappan, B. Adams, A. Mockus und A. E. Hassan, „A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance," 2014.

[8] M. Rasmussen, „Java Build Tools for Dependency Management," JRebel, 21 November 2013. [Online]. Available: https://www.jrebel.com/blog/java-build-tools-for-dependency-management. [Zugriff am 23 März 2020].

[9] Wikipedia, „Apache Maven," Wikipedia, 20 April 2020. [Online]. Available: https://en.wikipedia.org/wiki/Apache_Maven. [Zugriff am 19 Mai 2020].

[10] Apache Maven Project, „What is Maven?," 23 März 2020. [Online]. Available: https://maven.apache.org/what-is-maven.html. [Zugriff am 23 März 2020].

[11] Apache Maven Project, „Project Philosophy of Maven," 23 März 2020. [Online]. Available: https://maven.apache.org/background/philosophy-of-maven.html. [Zugriff am 23 März 2020].

[12] Apache Maven Project, „Introduction to the Build Lifecycle," 26 März 2020. [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html. [Zugriff am 26 März 2020].

[13] Apache Maven Project, „Introduction to Maven Plugin Development," 26 März 2020. [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-plugins.html. [Zugriff am 26 März 2020].

[14] Wikipedia, „Gradle," 10 März 2020. [Online]. Available: https://de.wikipedia.org/wiki/Gradle. [Zugriff am 23 März 2020].

[15] Gradle, „Gradle User Manual Version 6.2.2,“ [Online]. Available: https://docs.gradle.org/6.2.2/userguide/userguide.pdf. [Zugriff am 03 April 2020].

[16] Gradle, „Gradle vs Maven: Performance Comparison,“ [Online]. Available: https://gradle.org/gradle-vs-maven-performance/. [Zugriff am 26 März 2020].

[17] „DB-Engines Ranking,“ solid IT gmbh, Mai 2020. [Online]. Available: https://db-engines.com/de/ranking. [Zugriff am 27 Mai 2020].

[18] E. Schicker, Datenbanken und SQL, Springer Vieweg.

[19] A. Sharma, „Difference between SQL and NoSQL,“ GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/difference-between-sql-and-nosql/. [Zugriff am 18 April 2020].

[20] S. Luber und N. Litzel, „Was ist Microsoft SQL Server?,“ BigData-Insider, 24 Oktober 2017. [Online]. Available: https://www.bigdata-insider.de/was-ist-microsoft-sql-server-a-655905/. [Zugriff am 17 April 2020].

[21] SQL World, „Difference Between SQL and TSQL,“ [Online]. Available: https://www.complexsql.com/difference-between-sql-and-tsql-sql-vs-tsql/. [Zugriff am 17 April 2020].

[22] MongoDB, „What Is MongoDB?,“ [Online]. Available: https://www.mongodb.com/what-is-mongodb. [Zugriff am 17 April 2020].

[23] MongoDB, „NoSQL Databases Explained,“ [Online]. Available: https://www.mongodb.com/nosql-explained. [Zugriff am 17 April 2020].

[24] G. Bengel, Grundkurs Verteilte Systeme, Springer Vieweg, 2014.

[25] A. Schill und T. Springer, Verteilte Systeme, Springer Vieweg, 2012.

[26] Oracle Java SE, „Java SE Support Roadmap,“ Oracle Java SE, 13 Mai 2020. [Online]. Available: https://www.oracle.com/java/technologies/java-se-support-roadmap.html. [Zugriff am 19 Juli 2020].

[27] R. T. Fielding, „Architectural Styles and the Design of Network-based Software Architectures,“ 2000.

[28] Tutorialspoint, „Spring Framwork - Overview,“ [Online]. Available: https://www.tutorialspoint.com/spring/spring_overview.htm. [Zugriff am 29 März 2020].

[29] Wikipedia, „Spring (Framework),“ 03 November 2019. [Online]. Available: https://de.wikipedia.org/wiki/Spring_(Framework). [Zugriff am 29 März 2020].

[30] Spring, „Why Spring?,“ [Online]. Available: https://spring.io/why-spring. [Zugriff am 29 März 2020].

[31] M. Fowler, „Registry,“ Martin Fowler, [Online]. Available:

https://martinfowler.com/eaaCatalog/registry.html. [Zugriff am 08 Juni 2020].

[32] Apache Maven Project, „Guide to Developing Java Plugins," Apache Maven Project, 02 Juli 2020. [Online]. Available: https://maven.apache.org/guides/plugin/guide-java-plugin-development.html. [Zugriff am 06 Juli 2020].

[33] H. Vocke, „The Practical Test Pyramid," Martin Fowler, 26 Februar 2018. [Online]. Available: https://martinfowler.com/articles/practical-test-pyramid.html. [Zugriff am 27 Juli 2020].

[34] Apache Maven Project, „POM Reference," Apache Maven Project, 26 Juli 2020. [Online]. Available: https://maven.apache.org/pom.html. [Zugriff am 27 Juli 2020].

[35] Apache Maven Project, „Maven Failsafe Plugin," Apache Maven Project, 13 Juni 2020. [Online]. Available: https://maven.apache.org/surefire/maven-failsafe-plugin/. [Zugriff am 27 Juli 2020].

[36] R. Wilsenach, „DevOpsCulture," Martin Fowler, 09 Juli 2015. [Online]. Available: https://martinfowler.com/bliki/DevOpsCulture.html. [Zugriff am 17 Mai 2020].

[37] M. Fowler, „Patterns for Managing Source Code Branches," Martin Fowler, 28 Mai 2020. [Online]. Available: https://martinfowler.com/articles/branching-patterns.html. [Zugriff am 08 Juni 2020].

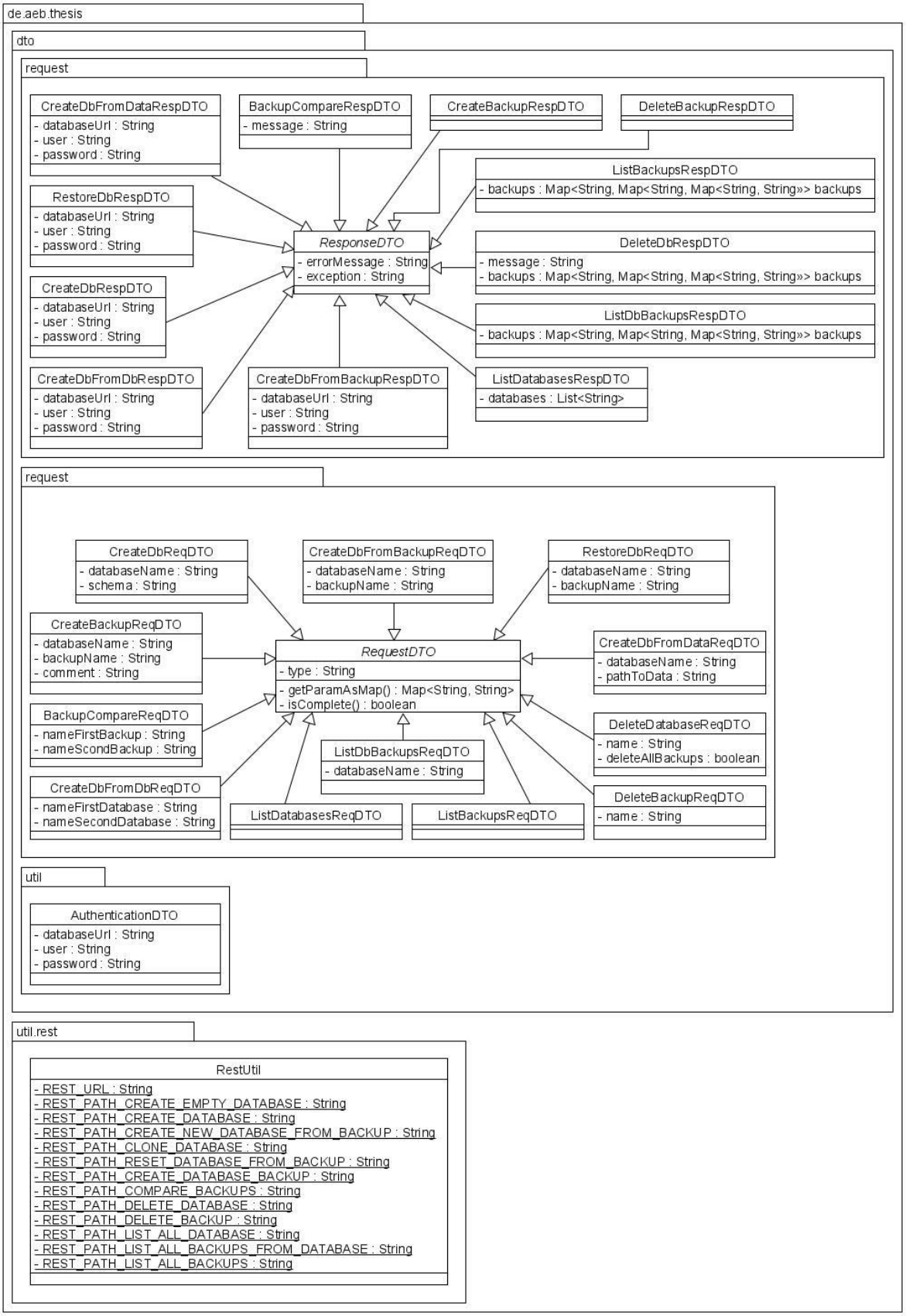
[38] M. Siepermann, „Agile Softwareentwicklung," Gabler Wirtschaftslexikon, 19 Februar 2018. [Online]. Available: https://wirtschaftslexikon.gabler.de/definition/agile-softwareentwicklung-53460/version-276549. [Zugriff am 22 Juli 2020].

Abbildungsverzeichnis

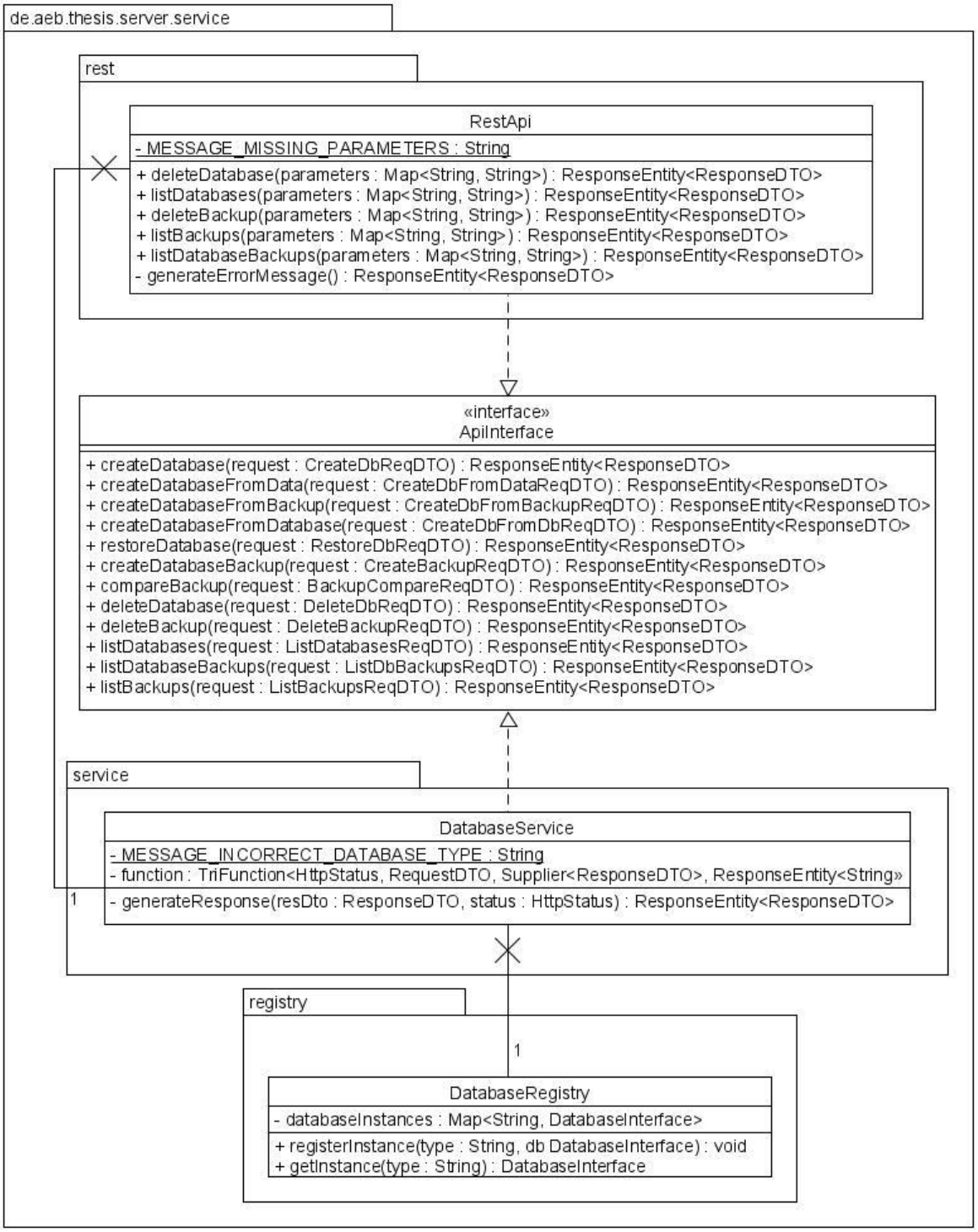
Tabellenverzeichnis

Anhang: Vollständige UML-Klassendiagramme

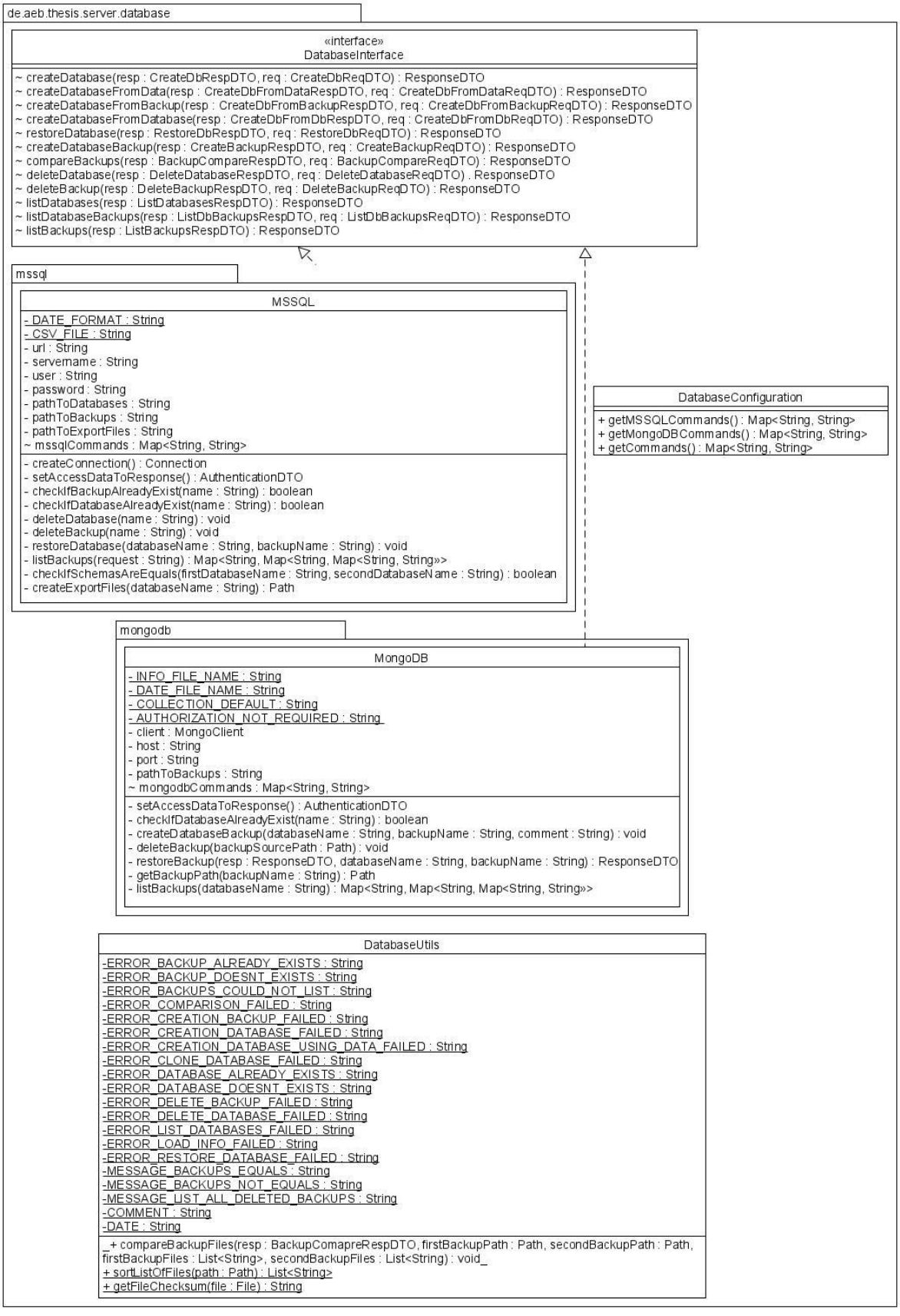
UML-Klassendiagramm zu den Data Transfer Objects:



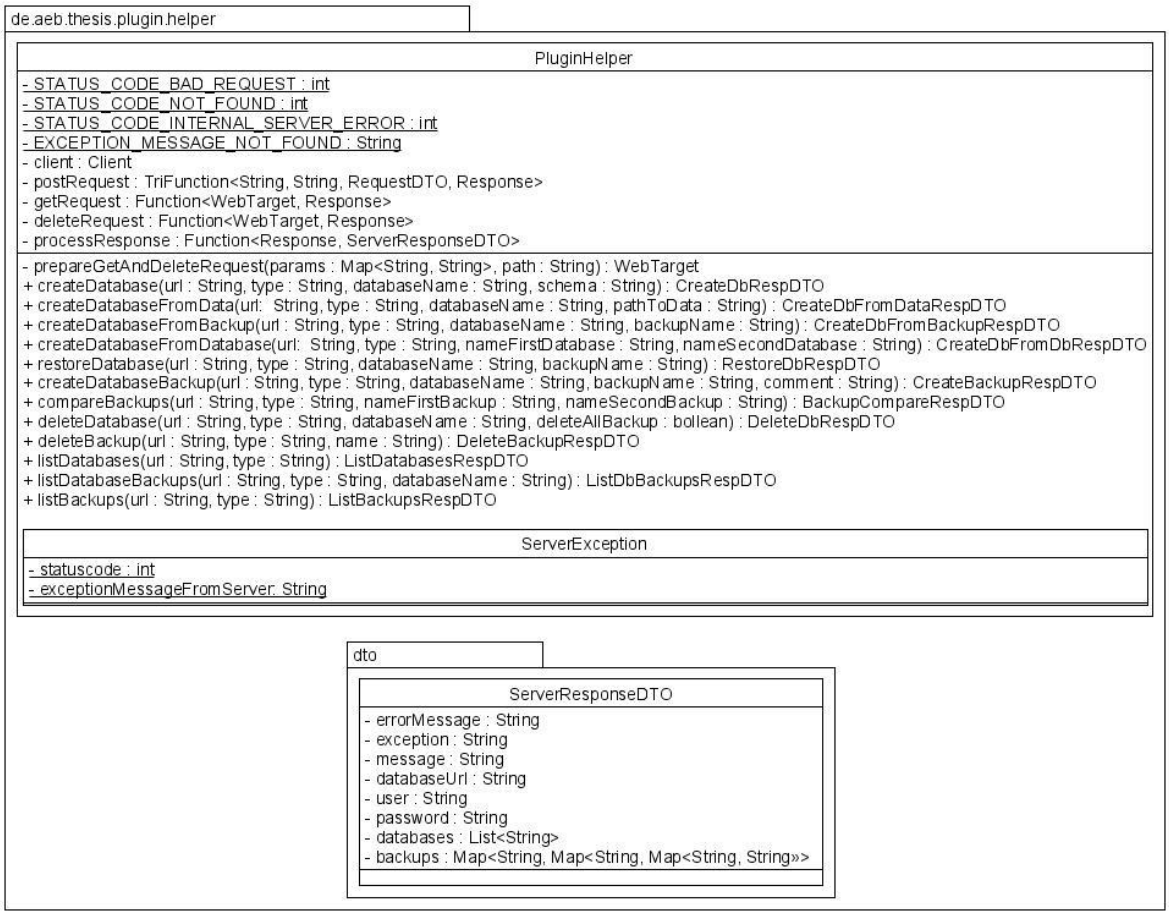
UML-Klassendiagramm zum Bereich Service des Servers:



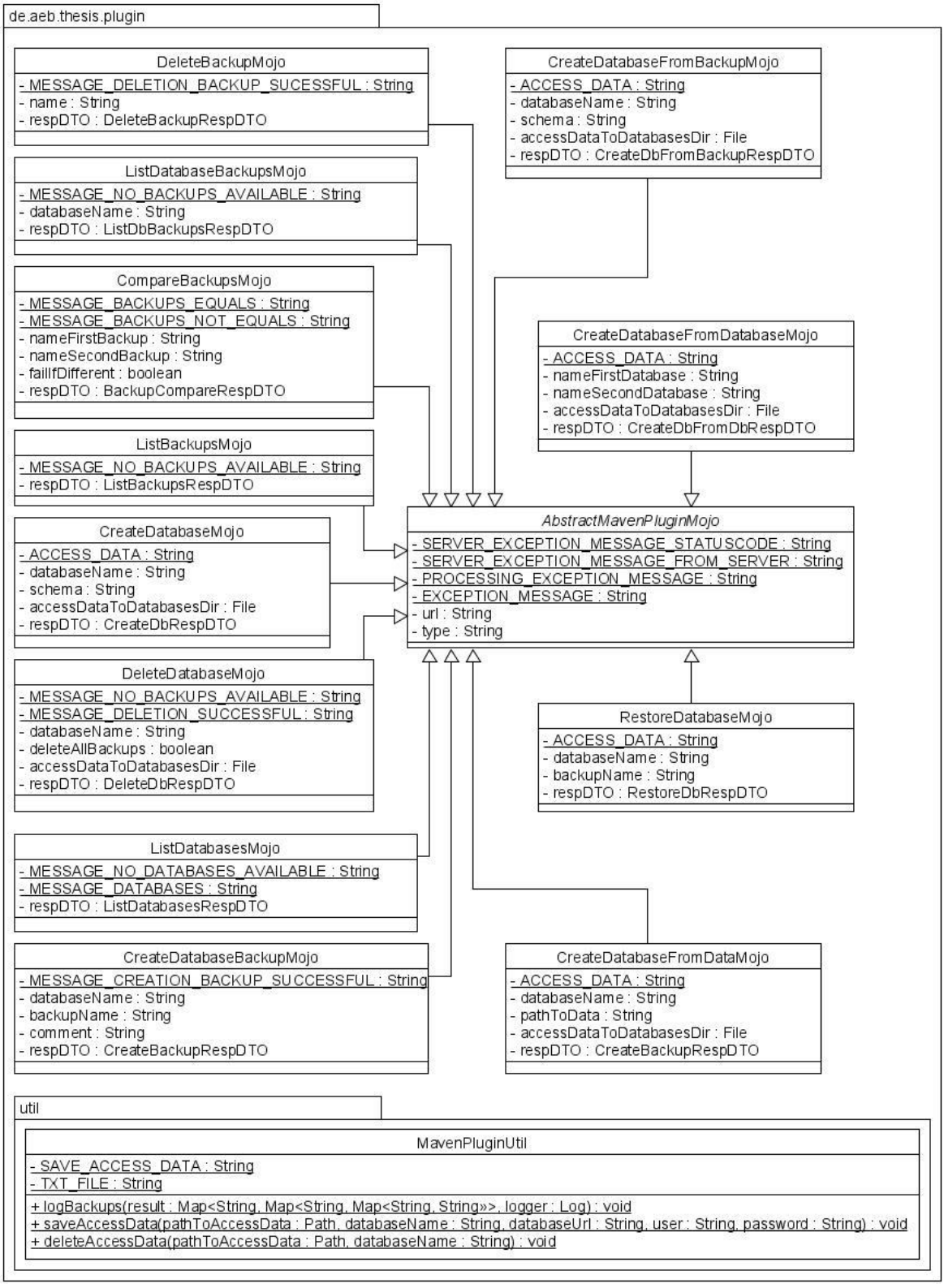
UML-Klassendiagramm zum Bereich Datenbankkomponenten des Servers:



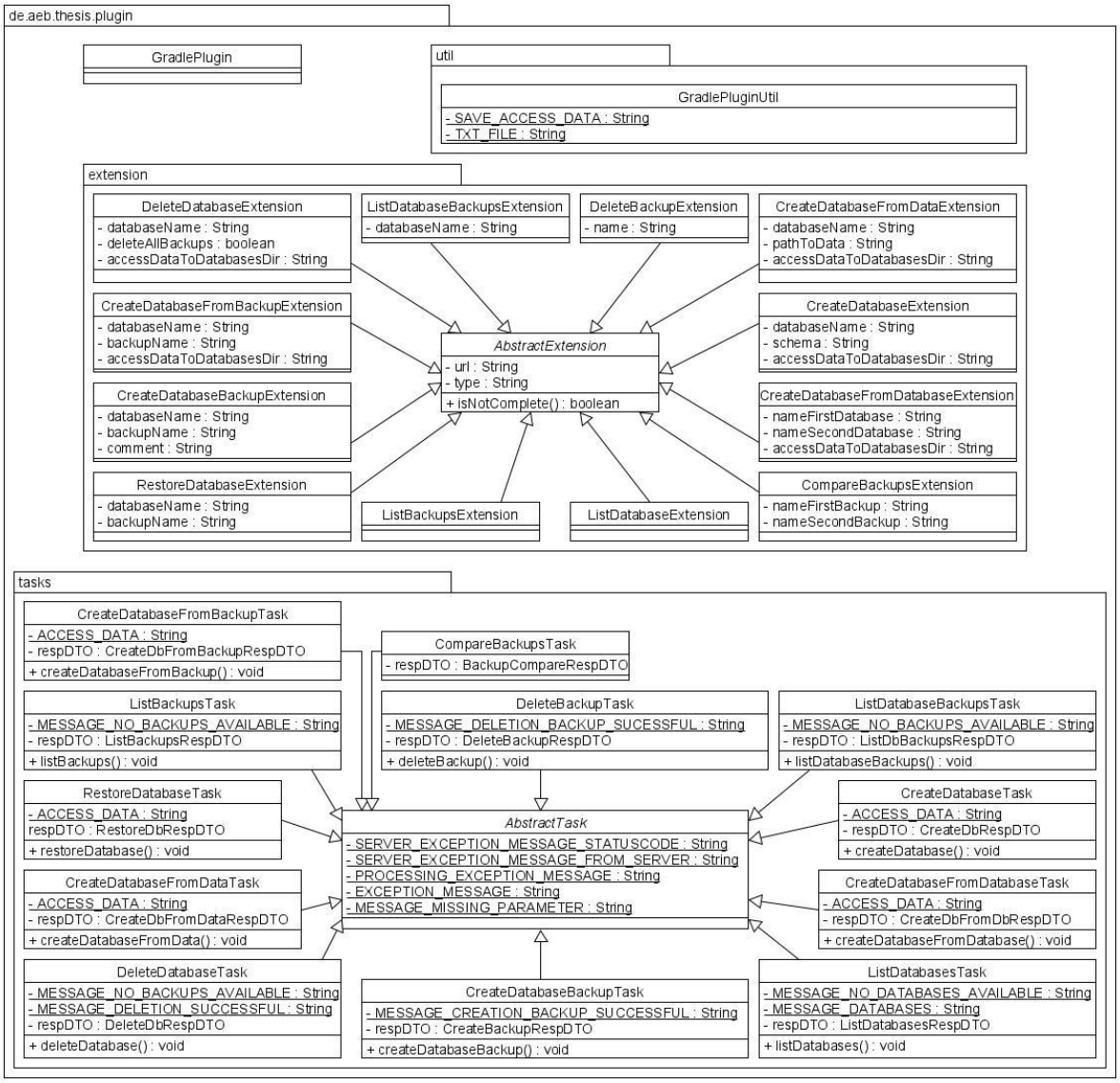
UML-Klassendiagramm zum Plugin Helper:



UML-Klassendiagramm zum Maven Plugin:



UML-Klassendiagramm zum Gradle Plugin:



Anhang: Testszenarien

Im Folgenden werden die Testszenarien aufgelistet, mit denen die Software, wie im Kapitel 8.1 vorgestellt, getestet wurde. Es ist zu beachten, dass nach jedem Testszenario die Testumgebung zurückgesetzt wird. Die Liste ist wie folgt zu lesen: $Aktion_1$ → … → $Aktion_n$ → Ergebnis (Fehlermeldung → Nachricht).

1. Erstellen einer Datenbank → Erfolgreich
2. Erstellen einer Datenbank (doppelt) → Nicht erfolgreich (Fehlermeldung → Datenbank bereits vorhanden)
3. Löschen einer nicht vorhandenen Datenbank → Nicht erfolgreich (Fehlermeldung → Datenbank nicht vorhanden)
4. Erstellen einer Datenbank → Löschen der Datenbank → Erfolgreich
5. Erstellen einer Datenbank → Löschen der Datenbank (doppelt) → Nicht Erfolgreich (Fehlermeldung → Datenbank nicht vorhanden)
6. Erstellen einer Datenbank → Listen aller Datenbanken → Erfolgreich
7. Erstellen eines Backups einer nicht vorhandenen Datenbank → Nicht erfolgreich (Fehlermeldung → Datenbank nicht vorhanden)
8. Erstellen einer Datenbank → Erstellen eines Backups → Erfolgreich
9. Erstellen einer Datenbank → Erstellen einer Datenbank → Erstellen eines Backups → Nicht erfolgreich (Fehlermeldung → Backup bereits vorhanden)
10. Löschen eines nicht vorhandenen Backups → Nicht erfolgreich (Fehlermeldung → Backup nicht vorhanden)
11. Erstellen einer Datenbank → Erstellen eines Backups → Löschen des Backups → Erfolgreich
12. Erstellen einer Datenbank → Erstellen eines Backups → Löschen des Backups (doppelt) → Nicht erfolgreich (Fehlermeldung → Backup nicht vorhanden)
13. Erstellen einer Datenbank → Erstellen eines Backups → Listen aller Backups → Erfolgreich
14. Erstellen einer Datenbank → Erstellen eines Backups → Listen aller Backups der Datenbank → Erfolgreich
15. Erstellen einer Datenbank → Erstellen eines Backups → Erstellen einer Datenbank auf Grundlage des Backups → Erfolgreich
16. Erstellen einer Datenbank auf Grundlage eines nicht vorhandenen Backups → Nicht erfolgreich (Fehlermeldung → Backup nicht vorhanden)
17. Erstellen einer Datenbank → Erstellen eines Backups → Erstellen einer Datenbank mit dem Namen der ersten Datenbank auf Grundlage des Backups → Nicht erfolgreich (Fehlermeldung → Datenbank bereits vorhanden)

18. Erstellen einer Datenbank → Erstellen eines Backups → Wiederherstellen der Datenbank auf Grundlage des Backups → Erfolgreich
19. Erstellen einer Datenbank → Wiederherstellen der Datenbank durch ein nicht vorhandenes Backup → Nicht erfolgreich (Fehlermeldung → Backup nicht vorhanden)
20. Erstellen einer Datenbank → Duplizieren der Datenbank → Erfolgreich
21. Duplizieren einer nicht vorhanden Datenbank → Nicht vorhanden (Fehlermeldung → Datenbank nicht vorhanden)
22. Erstellen einer Datenbank → Duplizieren der Datenbank (doppelt) → Nicht erfolgreich (Fehlermeldung → Datenbank bereits vorhanden)
23. Erstellen einer Datenbank mit initialen Daten → Erfolgreich
24. Erstellen einer Datenbank mit initialen Daten (doppelt) → Nicht erfolgreich (Fehlermeldung → Datenbank bereits vorhanden)
25. Vergleichen zweier nicht vorhandenen Backups → Nicht erfolgreich (Fehlermeldung → Backup nicht vorhanden)
26. Erstellen einer Datenbank mit Daten → Erstellen eines Backups → Simulation eines lesenden Zugriffs auf die Datenbank → Erstellen eines weiteren Backups → Vergleichen beider Backups → Backups identisch
27. Erstellen einer Datenbank mit Daten → Erstellen eines Backups → Simulation eines schreibenden Zugriffs auf die Datenbank → Erstellen eines weiteren Backups → Vergleichen beider Backups → Backups nicht identisch
28. Erstellen einer Datenbank → Erstellen zweier namentlich unterschiedlichen Backups → Vergleichen beider Backups → Backups identisch
29. Erstellen zweier unterschiedlichen Datenbanken → Erstellen zweier Backups von den Datenbanken → Vergleichen beider Backups → Backups nicht identisch
30. Stellen von unvollständigen Anfragen → Nicht erfolgreich (Fehlermeldung → Anfrage unvollständig)

Anhang: Beschreibung des Anwendungsszenarien des Beispielprojektes

Anwendungsszenario I

1. Pre-Integration-Test-Phase
    - Erstellen einer Datenbank mit initialen Daten
    - Erstellen einer Datenbank
2. Integration-Test-Phase
    - Manipulieren der zweiten Datenbank durch die Klasse *ApplicationTestUseCaseOne* (Datenbank wird mit den gleichen Daten befüllt wie die erste Datenbank)
    - Erstellen eines Backups der ersten Datenbank
    - Erstellen eines Backups der zweiten Datenbank
    - Vergleichen beider Backups
3. Post-Integration-Test-Phase
    - Löschen der ersten Datenbank (inklusive zugehörigen Backups)
    - Löschen der zweiten Datenbank (inklusive zugehörigen Backups)

Anwendungsszenario II

1. Pre-Integration-Test-Phase
    - Erstellen einer Datenbank mit initialen Daten
    - Duplizieren der Datenbank
2. Integration-Test-Phase
    - Erstellen eines Backups der ersten Datenbank
    - Erstellen eines Backups der zweiten Datenbank
    - Vergleichen beider Backups
3. Post-Integration-Test-Phase
    - Löschen der ersten Datenbank (inklusive zugehörigen Backups)
    - Löschen der zweiten Datenbank (inklusive zugehörigen Backups)

Anwendungsszenario III

1. Pre-Integration-Test-Phase
    - Erstellen einer Datenbank mit initialen Daten
    - Erstellen eines Backups (1) der Datenbank
2. Integration-Test-Phase
    - Manipulieren der Datenbank durch die Klasse *ApplicationTestUseCaseThree*
    - Erstellen eines Backups (2) der Datenbank
    - Vergleichen beider Backups
    - Widerherstellen der Datenbank auf Grundlage des ersten Backups
    - Erstellen eines Backups (3) der Datenbank
    - Vergleichen des ersten und dritten Backups
3. Post-Integration-Test-Phase
    - Löschen der Datenbank (inklusive zugehörigen Backups)

Anwendungsszenario IV

1. Pre-Integration-Test-Phase
    - Erstellen einer Datenbank mit initialen Daten
    - Erstellen eines Backups
2. Integration-Test-Phase
    - Erstellen einer Datenbank auf Grundlage der Backup
    - Erstellen eines Backups der zweiten Datenbank
    - Vergleichen beider Backups
3. Post-Integration-Test-Phase
    - Löschen der ersten Datenbank (inklusive zugehörigen Backups)
    - Löschen der zweiten Datenbank (inklusive zugehörigen Backups)