



**WHZ Westsächsische
Hochschule Zwickau**
Hochschule für Mobilität

flatEX=DEGIRO

Bachelorarbeit

**Analyse, Konzeption & Neu-Implementierung eines
RESTful Microservice zur Integration von Kunden-Logins
über OAuth2 mittels Keycloak**

an der Fakultät Physikalische Technik/Informatik

Betreuerin an der Hochschule:	Prof. Dr. Tina Geweniger Westsächsische Hochschule Zwickau
Betreuer an der Hochschule:	Prof. Dr. Wolfgang Golubski Westsächsische Hochschule Zwickau
Betreuer beim Unternehmen:	Thomas Kuchs flatexDEGIRO SE
vorgelegt von:	Erzhan Beishenbiev Fachgruppe Informatik MatrikelNr. 47270
Abgabedatum:	3. April 2026

Inhaltsverzeichnis

Abkürzungsverzeichnis	iii
Abbildungsverzeichnis	iv
Auflistungsverzeichnis	v
Tabellenverzeichnis	vi
1 Einleitung	1
1.1 Unternehmensprofil der flatexDEGIRO SE	1
1.2 Motivation	1
1.3 Problemstellung	2
1.4 Zielsetzung	2
1.5 Forschungsfragen	3
1.6 Aufbau der Arbeit	3
2 Literaturrecherche	5
2.1 Microservice-Architekturen im Vergleich zu monolithischen Systemen	5
2.2 Moderne Java-Frameworks: Quarkus vs. Spring Boot	6
2.3 Protokolle für die Identitätsverwaltung: OAuth 2.0 und OpenID Connect (OIDC)	6
2.3.1 Der Standard OAuth 2.0	7
2.3.2 Token-basierte Authentifizierung vs. klassische Sessions	7
2.4 Identitäts- und Zugriffsmanagement (IAM) mit Keycloak	8
2.4.1 Keycloak im Vergleich zu Marktalternativen	8
2.4.2 Erweiterbarkeit durch Custom Provider (SPI)	8
3 Technische Grundlagen	10
3.1 Java 17	10
3.2 Quarkus-Framework	10
3.2.1 Technische Architektur von Quarkus	10
3.2.2 Wirtschaftlichkeit und Ressourceneffizienz	11
3.3 REST-Architekturstil und der API-First-Ansatz (OpenAPI)	11
3.3.1 Grundlagen des REST-Architekturstils	11
3.3.2 Kernprinzipien von RESTful Services	12
3.3.3 Ressourcen	13
3.3.4 Repräsentationen	13
3.3.5 Hypermedia (HATEOAS)	13
3.3.6 Einheitliche Schnittstelle	14
3.3.7 API-First-Ansatz und OpenAPI-Spezifikation	15
3.4 Identitätsmanagement und Autorisierung	15
3.4.1 Das OAuth 2.0 Framework	16
3.4.2 OpenID Connect (OIDC)	17
3.4.3 Keycloak als Identity and Access Management System	17

3.4.4	JSON Web Token (JWT)	19
4	Anforderungsanalyse	20
4.1	Methodik der Anforderungserhebung	20
4.2	Ist-Zustand	20
4.3	Soll-Zustand	22
4.4	Funktionale Anforderungen	23
4.5	Nicht-funktionale Anforderungen	24
4.6	Priorisierung der Anforderungen	24
5	Entwurf und Konzeption	26
5.1	Architekturüberblick	26
5.1.1	Infrastruktur und Netzwerksegmentierung	26
5.1.2	Komponenten und Datenfluss	26
5.2	Technologiewahl	27
5.2.1	Java 17	27
5.2.2	Anwendungsframework Quarkus	28
5.2.3	Auswahl des IAM-Systems: Keycloak	29
5.3	Build-Management und Deployment-Infrastruktur	29
5.4	API-Design	31
5.4.1	Strukturierung der Endpunkte	31
5.4.2	Ressourcen-Orientierung und Pfad-Struktur	31
5.4.3	Beispielhafte Endpunkt-Definition	31
5.5	Ablauf der Kommunikation	32
5.5.1	Sequenzdiagramm: Authentifizierungsprozess	33
5.5.2	Trennung der Verantwortlichkeiten	33
5.6	Teststrategie: Behavior-Driven Development (BDD)	33
6	Implementierung des pin-tan-service	34
6.1	API-Design und automatisierte Artefakt-Generierung	34
6.2	Implementierung der Service-Ressourcen	36
6.3	Funktionale Implementierung der Ressourcen	36
6.4	Qualitätssicherung	38
6.4.1	Unit-Tests mit JUnit	39
6.4.2	Integrationstests mit Cucumber	40
6.5	Bewertung der Zielerreichung	41
7	Fazit und Ausblick	43
7.1	Beantwortung der Forschungsfragen	43
7.2	Ausblick und zukünftige Entwicklung	44
	Quellenverzeichnis	I

Abkürzungsverzeichnis

API Application Programming Interface

BDD Behavior-Driven Development

DTO Data Transfer Object

HTML HyperText Markup Language

IAM Identity and Access Management

JSON JavaScript Object Notation

JWKS JSON Web Key Set

JWT JSON Web Token

PIN Personal Identification Number

REST Representational State Transfer

SOAP Simple Object Access Protocol

TAN Transaktionsnummer

URI Uniform Resource Identifier

XML eXtensible Markup Language

Abbildungsverzeichnis

3.1	Hypermedia-Struktur	14
4.1	Ist-Zustand	21
4.2	Soll-Zustand	22
5.1	Detaillierter Architekturüberblick	27
5.2	Vollständiger CI/CD-Arbeitsablauf und Orchestrierung mit Kubernetes . .	30
5.3	UML-Sequenzdiagramm des Authentifizierungsablaufs	32

Auflistungsverzeichnis

6.1	Auszug aus der OpenAPI-Spezifikation (YAML)	35
6.2	Signatur der generierten Interface-Methode im API-Artefakt	36
6.3	Implementierung der PIN-Verifizierungslogik in der Ressourcen-Klasse . . .	37
6.4	JUnit 5 Testbeispiele für die PIN-Verifizierung	39
6.5	Cucumber-Szenarien für die PIN-Prüfung	40

Tabellenverzeichnis

3.1	Übersicht der HTTP-Methoden und deren Eigenschaften	14
4.1	Priorisierungsmatrix der Anforderungen	25
5.1	Auszug aus der API-Endpunkt-Spezifikation	32
6.1	Detaillierter Soll-Ist-Vergleich der Anforderungen	42

1 Einleitung

1.1 Unternehmensprofil der flatexDEGIRO SE

Die im vorliegenden Abschnitt enthaltenen Informationen basieren auf Angaben der offiziellen Website von flatexDEGIRO SE¹.

Die flatexDEGIRO SE ist einer der führenden und am schnellsten wachsenden Online-Broker in Europa. Das Unternehmen mit Hauptsitz in Frankfurt am Main wurde vor allem durch die Übernahme des niederländischen Brokers DEGIRO im Jahr 2020 gegründet. Das Unternehmen bedient Millionen von Kunden in Europa und bearbeitet jährlich Hunderte Millionen von Börsentransaktionen.

Eines der Hauptmerkmale von flatexDEGIRO SE ist seine Positionierung als Fintech-Unternehmen mit einer eigenen Vollbanklizenz. Im Gegensatz zu vielen Mitbewerbern basiert sein Geschäftsmodell auf einer eigenen, hochautomatisierten IT-Infrastruktur. Die selbst entwickelte Basis-Bankplattform hilft dem Unternehmen, Transaktionen effizient, sicher und mit einem hohen Maß an Skalierbarkeit abzuwickeln.

Neben dem Hauptsitz in Frankfurt verfügt das Unternehmen über weitere wichtige Technologiezentren, von denen sich eines in Zwickau befindet. An diesem Standort befindet sich ein Großteil der IT-Entwicklungs- und Systemadministrationsabteilung, die für die Wartung und regelmäßige Modernisierung der Plattform verantwortlich ist.

1.2 Motivation

Heutzutage müssen Programme schnell laufen und dürfen nicht ausfallen, wenn etwas daran geändert wird. In der Vergangenheit wurden wichtige Funktionen wie Authentifizierung oder Passwortüberprüfung in zentralen Bibliotheken zusammengefasst und anschließend in die entsprechenden Anwendungen integriert. Aufgrund der ständigen technologischen Weiterentwicklung werden heute moderne Standards wie OAuth2 sowie Tools wie Keycloak verwendet. Um eine hohe technologische Entwicklung und die Anwendung moderner Standards zu gewährleisten, muss auf die Verwendung veralteter Bibliotheken in neuen Programmen verzichtet werden. Die Motivation für diese Arbeit besteht darin, die geschäftskritische Logik (PIN/TAN-Verwaltung) in eine isolierte, moderne Umgebung zu übertragen, die es dem Unternehmen ermöglicht, operative Risiken schneller zu reduzieren.

¹ <https://flatexdegiro.com>

1.3 Problemstellung

Das derzeitige System weist erhebliche Mängel auf, die die Effizienz der Entwicklung einschränken. Die wichtigsten Nachteile lassen sich wie folgt zusammenfassen:

- **Monolithische Abhängigkeiten und enge Kopplung:** Die bestehende Legacy-Bibliothek ist in mehrere Anwendungen integriert. Jede Modifikation der Geschäftslogik erfordert von den Entwicklern eine vollständige Neukompilierung und eine aufwändige Bereitstellung aller abhängigen Systeme. Dies erhöht die Wartungskosten und verringert die Flexibilität bei Systemänderungen.
- **Wissensverlust und mangelnde Dokumentation:** Fehlende detaillierte Dokumentation sowie Weggang von Mitarbeitern, die den ursprünglichen Code geschrieben haben (vor 10–15 Jahren) und über die verfügbaren Informationen zur internen Funktionalität Kenntnis hatten. Unzureichende Dokumentation erschwert die Einarbeitung neuer Mitarbeiter und erhöht das Risiko von Fehlern bei der Modernisierung des Systems.
- **Veraltete Geschäftsprozesse:** Der Code enthält immer noch viele ungenutzte Funktionen, die das System komplexer machen und sein Verständnis erschweren.
- **Mangelnde Skalierbarkeit:** Derzeit kann die Legacy-Bibliothek nicht unabhängig von den Anwendungen, die sie verwenden, skaliert werden. Dies verhindert eine effiziente Nutzung der Ressourcen und reduziert die Gesamtleistung des Systems.
- **Eingeschränkte Testbarkeit und fehlende Testautomatisierung:** Aufgrund der tiefen Integration in alte Systeme ist die Durchführung automatisierter Akzeptanztests (Acceptance Tests) schwierig oder unmöglich. Dies führt zu hohen Kosten für manuelle Tests und einem erhöhten Risiko von Regressionen bei Updates.

1.4 Zielsetzung

Das Hauptziel dieser Arbeit ist der Entwurf und die Implementierung eines REST-basierten Microservices auf Basis des Quarkus-Frameworks. Dieser soll eine bestehende Legacy-Bibliothek zur PIN/TAN-Verwaltung vollständig einbetten und eine nahtstörungsfreie Integration in eine bestehende Keycloak-Infrastruktur gewährleisten.

Zur Erreichung dieses Ziels werden folgende konkrete Teilaufgaben definiert:

1. **Analyse und Extraktion:** Detaillierte Untersuchung der bestehenden Bibliothek zur Identifikation der aktuellen Geschäftslogik (PIN/TAN) sowie Identifizierung und Ausschluss veralteter oder redundanter Funktionen.
2. **Neu-Implementierung:** Entwicklung eines modernen Microservices unter Verwendung von Java 17+ und des Quarkus-Frameworks.

3. Entkopplung: Ersetzung direkter Bibliotheksaufrufe durch eine Interaktion über eine standardisierte RESTful API, um die Unabhängigkeit der Systemkomponenten zu erhöhen.
4. Keycloak-Integration: Implementierung des Dienstes als Custom Credential Provider innerhalb von Keycloak, um die Logik in einen modernen OAuth2-Stack zu überführen.
5. Qualitätssicherung: Einführung einer BDD-Methodik (Behavior Driven Development) mittels Cucumber² sowie die Integration in Jira X-Ray zur Sicherstellung einer hohen Testabdeckung und Anforderungstransparenz.
6. Modernisierung des Deployments: Containerisierung des Dienstes zur Ermöglichung horizontaler Skalierung und Vorbereitung für eine automatisierte Bereitstellung (CI/CD).

1.5 Forschungsfragen

Im Rahmen dieser Arbeit soll die folgende Forschungsfrage beantwortet werden: „Wie kann eine proprietäre Legacy-Bibliothek zur PIN/TAN-Verwaltung in eine moderne, skalierbare Microservice-Architektur integriert werden, um eine sichere Authentifizierung via Keycloak und OAuth2 zu gewährleisten?“

Zur Beantwortung der Kernfrage werden folgende Unterfragen untersucht:

- Analyse: Welche Bestandteile der bestehenden Bibliothek sind für die aktuelle Geschäftslogik essenziell und welche können im Zuge der Modernisierung entfernt werden?
- Integration: Welche Auswirkung hat die Integration der Pin-Tan-Logik in Form eines Microservice als Custom Credential Provider im Vergleich zur Integration als Bibliothek?
- Testbarkeit: Welche Vor- und Nachteile bietet die Bereitstellung einer Funktionalität als Microservice im Hinblick auf die Testbarkeit im Vergleich zur Bereitstellung als integrierte Bibliothek?

1.6 Aufbau der Arbeit

Die vorliegende Bachelorarbeit besteht aus sieben Kapiteln. Nach der Einführung und Erläuterung des Problems in Kapitel 1 folgt in Kapitel 2 ein Literaturüberblick. Hier wird der aktuelle Forschungsstand im Bereich Microservice-Architekturen und Identitätsmanagementsysteme untersucht, um eine theoretische Grundlage für die ausgewählten

² <https://cucumber.io/>

Technologien zu schaffen. In Kapitel 3 werden die technischen Grundlagen erläutert, die zum Verständnis der in den folgenden Kapiteln beschriebenen Prozesse erforderlich sind. Dabei werden die verwendeten Frameworks, wie Quarkus, sowie die Protokolle OAuth 2.0 und OpenID Connect ausführlich erklärt. Kapitel 4 enthält eine Anforderungsanalyse. Zunächst wird der Ist-Zustand der PIN/TAN-Verwaltung in einer monolithischen Infrastruktur analysiert, um bestehende Mängel und ungenutzte Funktionen aufzudecken. Auf dieser Grundlage wird der Soll-Zustand definiert, der die funktionalen und nicht-funktionalen Anforderungen an den neuen Microservice festlegt. Auf Basis dieser Anforderungen werden in Kapitel 5 das Konzept sowie die Architektur des REST-Dienstes entwickelt. Die praktische Umsetzung dieses Konzepts wird in Kapitel 6 beschrieben, das sich mit der Implementierung befasst. In diesem Kapitel wird erläutert, wie das gesamte System aufgebaut ist, wie es funktioniert und wie die einzelnen Komponenten implementiert werden. Die Arbeit endet mit Kapitel 7, in dem ein Fazit gezogen und eine Zusammenfassung gegeben wird, in der die Ergebnisse kritisch bewertet und ein Ausblick auf zukünftige Entwicklungen gegeben wird.

2 Literaturrecherche

In diesem Kapitel werden die bestehenden Konzepte und Technologien analysiert, die dieser Arbeit als Grundlage dienen.

2.1 Microservice-Architekturen im Vergleich zu monolithischen Systemen

In den letzten Jahrzehnten hat sich die Architektur von Softwareprogrammen wesentlich verändert. Monolithische Systeme wurden durch verteilte Systeme ersetzt. Wie in [New19] beschrieben, ist ein Mikroservice ein kleiner autonomer Dienst, der eine bestimmte Funktion ausführt und Daten über das Netzwerk austauscht. Der Kern von Microservices besteht darin, eine Anwendung in viele kleine Systeme aufzuteilen, die miteinander kommunizieren.

Velepucha und Flores [Vel23] beschreiben einen Monolithen als einheitliche, robuste Struktur, in der alle Komponenten der Anwendung eng miteinander verbunden sind und nicht unabhängig voneinander funktionieren können. Eine solche Architektur führt häufig dazu, dass Änderungen in einem Teil des Systems nicht vorhersehbare Auswirkungen auf das gesamte System haben. Diese theoretische Definition lässt sich direkt auf die technologische Situation bei flatexDEGIRO anwenden. Hier ist die für das Geschäft entscheidende PIN/TAN-Verwaltungslogik in der Bibliothek *pitase* implementiert. Derzeit ist diese Bibliothek als harte Abhängigkeit (Dependency) in verschiedene Anwendungen integriert, was der Kritik von Velepucha und Flores an der engen Verknüpfung entspricht. Die Logik kann nicht unabhängig skaliert oder aktualisiert werden, ohne dass die verwendeten Anwendungen neu kompiliert und erneut bereitgestellt werden müssen.

In der wissenschaftlichen Literatur werden folgende Vorteile der Aufteilung des Monolithen in kleine Teile besonders herausgestellt:

- **Skalierbarkeit:** Technisch gesehen kann ein Monolith nur als Gesamtes skaliert werden. Microservices ermöglichen es, einzelne ressourcenintensive Komponenten zu skalieren[New21].
- **Wartbarkeit und Deployment:** Newman spricht über die Bedeutung der unabhängigen Bereitstellbarkeit (*Independent Deployability*) [New21]. Ein Microservice kann unabhängig von anderen Teilen des Systems bereitgestellt werden. Dies löst das Problem, mit dem die Bibliothek *pitase* konfrontiert war. In Zukunft müssen bei kleinen Änderungen an der PIN/TAN-Logik nicht mehr alle abhängigen Systeme

me neu kompiliert und bereitgestellt werden. Dies verkürzt die Durchlaufzeit des Software-Lebenszyklus.

2.2 Moderne Java-Frameworks: Quarkus vs. Spring Boot

Spring Boot¹ ist seit langer Zeit der Standard für die Entwicklung von Microservices in Java. Mit dem Aufkommen von Cloud-Architekturen und Container-Management (z. B. Kubernetes²) entstehen jedoch neue Anforderungen an die effiziente Nutzung von Ressourcen. Quarkus ist hierfür eine ausgezeichnete Alternative.

Ressourceneffizienz und Performance-Metriken

Der Hauptunterschied zwischen Quarkus und anderen Frameworks besteht darin, dass Prozesse von der Laufzeit (Runtime) in die Kompilierungszeit verschoben werden. Spring Boot führt beim Start einen langwierigen Scan durch, der viel Zeit für die Suche nach Annotationen und Konfigurationen in Anspruch nimmt, während Quarkus diese Schritte bereits während der Kompilierung ausführt [Hai21].

In der Literatur werden häufig folgende Faktoren verglichen:

- **Boot Time:** Eine klassische Spring Boot-Anwendung benötigt zum Starten mehrere Sekunden [Hai21]. Quarkus benötigt dagegen weniger als zwei Sekunden im JVM-Modus. Durch die Verwendung von *GraalVM*³ *Native Compilation* lässt sich die Startzeit verkürzen [Qua26].
- **Memory Footprint:** Vergleichende Studien zeigen, dass Quarkus-Anwendungen im Standby-Modus (RSS) deutlich weniger Arbeitsspeicher verbrauchen. Im Normalbetrieb benötigt ein einfacher REST-Dienst etwa 12 bis 30 MB RAM, während Spring Boot in der Regel mehr als 140 MB benötigt [Hai21].

2.3 Protokolle für die Identitätsverwaltung: OAuth 2.0 und OpenID Connect (OIDC)

Die Entwicklung der Softwarearchitektur hat den Ansatz zur Gewährleistung der Sicherheit wesentlich verändert. Heute wird in Wissenschaft und Industrie davon ausgegangen, dass es nicht empfehlenswert ist, wichtige Funktionen wie Authentifizierung und Autorisierung mit eigenen Bibliotheken zu realisieren, die an eine bestimmte Anwendung gebunden sind. Diese monolithischen Lösungen bringen eine Reihe von Problemen mit sich: Entdeckte Schwachstellen in der Bibliothek erfordern Updates aller abhängigen Anwendungen und schränken die Interoperabilität zwischen verschiedenen technischen Plattformen ein.

¹ <https://spring.io/projects/spring-boot>

² <https://kubernetes.io/>

³ <https://www.graalvm.org/>

Anstelle dieser isolierten Lösungen werden zunehmend standardisierte Protokolle eingesetzt. Eine der beliebtesten Lösungen ist OAuth 2.0 [Har12]. Dieses Protokoll separiert den Prozess der Identitätsprüfung des Benutzers von der Hauptlogik der Anwendung und ermöglicht eine zentralisierte Zugriffsverwaltung. Diese Lösung trägt dazu bei, die Abhängigkeit zwischen den Services zu reduzieren, vereinfacht deren Interaktion und erhöht die Sicherheit, da keine vertraulichen Daten mehr gespeichert und unnötig übertragen werden müssen.

2.3.1 Der Standard OAuth 2.0

Laut RFC 6749 ermöglicht das OAuth 2.0-Protokoll einer Drittanbieteranwendung, im Namen des Ressourcenbesitzers eingeschränkten Zugriff auf HTTP-Ressourcen zu erhalten. Außerdem überträgt das Protokoll keine Anmeldedaten direkt. Stattdessen authentifiziert es sich auf der Seite des vertrauenswürdigen Autorisierungsservers und bestätigt ausdrücklich, welche Daten oder Funktionen die Drittanbieteranwendung nutzen darf. Anschließend gibt der Autorisierungsserver der Anwendung einen speziellen Access-Token [Har12] aus.

2.3.2 Token-basierte Authentifizierung vs. klassische Sessions

Der wichtigste Aspekt der neuen Architektur ist der Ersatz von zustandsbehafteten Sessions (Session-Cookies) mit Status durch eine zustandslose (*stateless*) Authentifizierung, die auf Tokens basiert.

Die tokenbasierte Authentifizierung (mit JWT) in einer Microservice-Architektur bietet im Vergleich zu klassischen Sitzungen mehr Vorteile [RS17]:

- **Zustandslosigkeit (Statelessness):** Da alle Daten über den Benutzer bereits im Token selbst enthalten sind, muss der Microservice keine zusätzlichen Informationen über die Sitzung speichern oder diese mit einer zentralen Datenbank abgleichen. Der Microservice kann einfach das Token überprüfen und die Anfrage sofort bearbeiten, was das System deutlich vereinfacht und die horizontale Skalierung erleichtert.
- **Sicherheit und Granularität:** Tokens können bestimmte Berechtigungen (Scopes) enthalten, die genau angeben, welche Aktionen der Service ausführen darf – beispielsweise die Überprüfung der PIN oder den Zugriff auf bestimmte Daten.
- **Zentralisierung:** Durch die Verwendung eines Identitätsanbieters wie Keycloak wird die gesamte Sicherheitslogik zentralisiert und nicht in jeder einzelnen Anwendung dupliziert.

2.4 Identitäts- und Zugriffsmanagement (IAM) mit Keycloak

Ein Identitäts- und Zugriffsmanagement (IAM) dient der zentralisierten Steuerung von Identitäten sowie der Verwaltung von Zugriffsrechten in verteilten Systemen [New21]. Keycloak ist ein Open-Source-IAM-System, das als dedizierter Dienst die Authentifizierung und Autorisierung übernimmt [Red25].

Die Architektur von Keycloak basiert auf der Unterstützung etablierter Industriestandards wie OAuth 2.0 [Har12] und OpenID Connect (OIDC). Durch den Einsatz eines solchen Systems lassen sich Sicherheitsmechanismen von der eigentlichen Geschäftslogik der Microservices entkoppeln. Dies ermöglicht eine zentrale Verwaltung von Benutzern, Rollen und Sicherheitsrichtlinien, wodurch die Komplexität innerhalb der einzelnen Anwendungsmodule reduziert wird.

2.4.1 Keycloak im Vergleich zu Marktalternativen

Im Vergleich zu anderen kommerziellen Cloud-Lösungen auf dem Markt, wie Auth0 oder Okta, bietet Keycloak als Open-Source-Lösung wesentliche Vorteile:

- **Datenhoheit:** Im Gegensatz zu Auth0, das als verwalteter Cloud-Dienst angeboten wird, ermöglicht Keycloak eine vollständig autonome Bereitstellung (vor Ort oder in einer privaten Cloud). Unternehmen erhalten dadurch die vollständige Kontrolle über die Speicherung und Rückgabe von Daten. Dies ist angesichts der strengen Anforderungen an die Einhaltung normativer Anforderungen und den Datenschutz im Finanztechnologiesektor (z. B. BaFin-Regulierung) von großer Bedeutung. [TS21]
- **Flexibilität und Anpassbarkeit:** Aufgrund seiner Architektur und Erweiterbarkeit bietet Keycloak im Vergleich zu vielen SaaS-Lösungen eine tiefere Kontrolle über Protokolle und Benutzerabläufe (*User Flows*) [TS21].

2.4.2 Erweiterbarkeit durch Custom Provider (SPI)

Ein wesentliches Merkmal von Keycloak ist seine Erweiterbarkeit über das *Service Provider Interface* (SPI). Wie Thorgersen und Silva sagen, ermöglicht diese Architektur die Integration von benutzerdefinierter Logik in den Authentifizierungsfluss [TS21]. In der Fachliteratur wird dieser Ansatz als eine Methode zur Anbindung bestehender Drittsysteme oder proprietärer Identitätsquellen beschrieben.

Anstatt viel Aufwand für die Übertragung proprietärer Geschäftslogik oder Benutzerdaten in ein neues System zu betreiben, vereinfacht die Verwendung von SPI die Integration. In der offiziellen Dokumentation und Literatur wird angegeben, dass als allgemein anerkannter Standard zwei Strukturen besonders hervorzuheben sind [Red26]:

- **User Storage Provider:** Mithilfe dieser Schnittstelle ermöglicht Keycloak den

Abruf von Benutzerinformationen aus externen Quellen, als wären sie im internen Speicher gespeichert [TS21].

- **Credential Provider:** Diese Funktion ermöglicht es, bestimmte Anmeldedaten (z. B. PIN/TAN) mithilfe externer Sicherheitssysteme zu überprüfen. Außerdem ist es nicht erforderlich, vertrauliche Informationen in Keycloak zu verwalten. [Red26]

Dieser Ansatz stellt sicher, dass die für die Geschäftslogik wichtige Bibliothek in eine moderne Infrastruktur übertragen werden kann, die auf Standards basiert und die Sicherheit des Systems nicht beeinträchtigt. Darüber hinaus kann das System aktualisiert werden, da der eigene Code in Form von JAR-Dateien getrennt bleibt und nur für bestimmte SPI-Schnittstellen programmiert wird. [TS21]

3 Technische Grundlagen

Dieses Kapitel dient der detaillierten Erläuterung einiger in dieser Arbeit verwendeten Begriffe und Technologien, um das notwendige Verständnis für die in den folgenden Kapiteln dargelegten Überlegungen zu gewährleisten. Um den Fokus der Untersuchung im Rahmen der gestellten Aufgaben beizubehalten, werden im Folgenden nur die für das Verständnis wesentlichen Aspekte behandelt.

3.1 Java 17

Java 17 stellt eine bedeutende Version der Programmiersprache Java dar, die als *Long-Term-Support* (LTS) Version veröffentlicht wurde. Diese Einstufung gewährleistet eine langfristige Unterstützung sowie regelmäßige Sicherheitsupdates, was für geschäftskritische Anwendungen im Finanzsektor von hoher Relevanz ist. Mit dieser Version wurden moderne Sprachfeatures eingeführt, die die Effizienz der Softwareentwicklung steigern. Dazu gehören unter anderem Records, die den *Boilerplate-Code* für Datenübertragungsobjekte (DTOs) erheblich reduzieren, sowie Sealed Classes, die eine präzise Kontrolle über Vererbungshierarchien ermöglichen. Diese Funktionen tragen dazu bei, die Komplexität des Quellcodes zu verringern und die Wartbarkeit der Software zu erhöhen. [Ora21]

3.2 Quarkus-Framework

3.2.1 Technische Architektur von Quarkus

Quarkus ist ein Full-Stack-Java-Framework. Dieses Framework ist auf die Verwendung in Cloud-Systemen und containerisierten Plattformen ausgerichtet. Außerdem bietet es eine effektive Kompatibilität mit Orchestrierungsplattformen wie Kubernetes und gewährleistet den korrekten Betrieb von Anwendungen bei der dynamischen Erstellung, Skalierung und dem Neustart von Containern.

Im Vergleich zu klassischen Frameworks wie Spring verwendet Quarkus die Methode *Build-time Augmentation*. Aufgaben wie das Scannen von Annotationen und die Umwandlung von Abhängigkeiten werden bereits während des Build-Prozesses und nicht während der Ausführung durchgeführt. Dieser Ansatz reduziert die Startzeit und verbraucht weniger Speicher, wie bereits in Kapitel 2.2 erwähnt. Dies ist ein wichtiger Faktor bei der Skalierung in einer Cloud-Umgebung. [Hai21]

3.2.2 Wirtschaftlichkeit und Ressourceneffizienz

Wie bereits bekannt ist, hat Quarkus erhebliche Vorteile in Bezug auf Startzeit und Speicherverbrauch. Wie in der wissenschaftlichen Literatur dargelegt, wirken sich diese Vorteile direkt auf die Wirtschaftlichkeit und die Betriebskosten aus, da in modernen Cloud-Umgebungen die Kosten nach dem Prinzip *Pay-as-you-go* berechnet werden. Die Verringerung des *Memory Footprints* wirkt sich auf die Senkung der Betriebskosten aus. Varghese und Buyya betonen, dass bei einer optimierten Nutzung von Prozessor, Speicher und anderen Ressourcen durch die Anwendungen die Möglichkeit besteht, die Bereitstellungsdichte auf dem Server zu erhöhen. Dies verringert die Kosten für jede Instanz eines Microservices. [VB18]

3.3 REST-Architekturstil und der API-First-Ansatz (OpenAPI)

3.3.1 Grundlagen des REST-Architekturstils

Der Datenaustausch zwischen Keycloak-Custom-Provider und dem entwickelten Dienst erfolgt über die REST-API (*Representational State Transfer*). REST ist ein einfacher Architekturstil, der auf der Verwendung bestehender Webstandards wie dem HTTP-Protokoll und seinen Methoden basiert. Der Begriff „REST“ wurde erstmals von Roy Thomas Fielding in seiner Dissertation eingeführt, in der er REST als einen Netzwerkar-chitekturstil beschrieb, der es ermöglicht, die Netzwerkkommunikation zu minimieren und gleichzeitig die Unabhängigkeit und Skalierbarkeit der Komponenten zu erhöhen. [Fie00]

Die „Leichtigkeit“ (*Lightweight-Charakter*) solcher Schnittstellen liegt in ihrer Zustandslosigkeit (*Statelessness*) begründet: Der Server speichert zwischen den Anfragen keine Informationen über den Status des Clients, da alle erforderlichen Daten bei jeder einzelnen Anfrage übertragen werden. Dies vereinfacht die Systemarchitektur, reduziert die Speichieranforderungen und ermöglicht eine einfachere Skalierung der Dienste [All10]. REST verwendet in der Regel standardisierte und weit verbreitete Datenformate wie JSON. Dieses Format ist leicht lesbar und wird von fast allen modernen Programmiersprachen unterstützt. Im Vergleich zu komplexeren Protokollen wie SOAP, das komplexere Spezifikationen und zusätzliche Verarbeitung von XML-Nachrichten erfordert, sind REST-Schnittstellen kostengünstiger, einfacher zu implementieren und lassen sich leichter in verteilte Systeme integrieren. [RAR13]

REST ist kein konkreter Standard oder eine einzelne Technologie. Es handelt sich um einen konzeptionellen Architekturansatz, der eine Reihe von Prinzipien definiert, denen ein System entsprechen muss. Architekturen, die gemäß diesen Prinzipien und Einschränkungen aufgebaut sind, werden als REST-orientiert oder RESTful bezeichnet.

3.3.2 Kernprinzipien von RESTful Services

Ein RESTful-Webdienst ist eine über das Netzwerk zugängliche Softwareumgebung, die über URI eindeutig identifiziert werden kann und die Interaktion zwischen verschiedenen Anwendungen ermöglicht. Die Interaktion mit solchen Services erfolgt über Ressourcen, der Zugriff auf diese Services wird über eine Reihe von Operationen (HTTP-Methoden) bereitgestellt. Statusänderungen werden durch die Übertragung ihrer Darstellungen realisiert. Um als REST-kompatibel zu gelten, muss ein System bestimmte Designprinzipien und REST-Beschränkungen befolgen:

- **Ressourcenorientierung:** In REST wird alles als Ressource betrachtet, auf die über eine eindeutige Adresse (URI) zugegriffen wird. Jede Ressource ist eine grundlegende Einheit des Systems, muss identifiziert werden und mindestens eine Darstellung (z. B. im JSON-Format) haben [Til11].
- **Unterschiedliche Repräsentationen:** Die Darstellung einer Ressource ist eine Repräsentation. Eine Ressource kann verschiedene Repräsentationen haben, d. h. sie kann in verschiedenen Formaten dargestellt werden, z. B. HTML, XML oder JSON. Dabei kann der Kunde angeben, in welchem Format er die Daten erhalten möchte.
- **Hypermedia (HATEOAS):** Hypermedia (HATEOAS) ist ein REST-Prinzip, bei dem die Navigation über die API dynamisch über Links erfolgt, die in den Antworten des Servers enthalten sind. Die Ressourcen werden über Hyperlinks miteinander verknüpft. Dies ermöglicht es dem Client, auf der Grundlage der erhaltenen Informationen von einer Ressource zur nächsten zu wechseln. Auf diese Weise überträgt der Server nicht nur Daten, sondern auch mögliche weitere Aktionen. Ein einziger Einstiegspunkt kann den Zugriff auf andere Ressourcen des Systems ermöglichen, was die Flexibilität erhöht und die Abhängigkeit zwischen Client und Server verringert. [Fie00]
- **Einheitliche Schnittstelle (Uniform Interface):** Die einheitliche Schnittstelle (Uniform Interface) ist die Interaktion zwischen Client und Server, die über das HTTP-Protokoll erfolgt. Jede Ressource muss mit Standardmethoden zugänglich sein – Lesen, Erstellen, Ändern und Löschen, die in HTTP über GET, POST, PUT und DELETE realisiert werden. Die Änderung des Systemstatus erfolgt durch die Übertragung von Ressourcenansichten. Auf diese Weise wird ein einziges und verbindliches Kommunikationsformat festgelegt, das von allen Teilnehmern eingehalten werden muss.[Til11]
- **Trennung von Client und Server (Separation of Concerns:)** Im REST-Architekturstil basiert die Interaktion zwischen Server und Client auf der Abwesenheit von Status-Speicherung auf der Serverseite (Statelessness). Das heißt, der

Server speichert keine Informationen über den Benutzer zwischen den Anfragen. Anfragen müssen alle für ihre Bearbeitung erforderlichen Daten enthalten. Wenn der Client den Kontext der vorherigen Anfrage benötigt, muss er die entsprechenden Informationen erneut übermitteln. [All10]

3.3.3 Ressourcen

Die zentrale Abstraktion in der REST-Architektur ist die Ressource. Laut Fieldings Definition ist eine Ressource jede Information und jeder Datensatz. Dabei definiert Fielding eine Ressource mathematisch als eine Abbildung, die auf eine Menge von Entitäten verweist. Der konkrete Inhalt (die Darstellung) kann sich im Laufe der Zeit ändern, aber die semantische Bedeutung der Ressource bleibt stabil. Ein wichtiges Merkmal ist, dass eine Ressource durch verschiedene Quellen oder Formate (z. B. JSON, XML oder HTML) dargestellt werden kann, ohne ihre Identität zu verlieren. Um den Zugriff auf diese Ressourcen zu gewährleisten und Verbindungen zwischen ihnen herzustellen, ist eine eindeutige Adressierung erforderlich. Dies wird mit Hilfe von URIs erreicht. URIs dienen als globale Identifikatoren im Netzwerk und ermöglichen die Interaktion mit einer Ressource über standardisierte Schnittstellen. [Fie00]

3.3.4 Repräsentationen

Ressourcen in der REST-Architektur sind abstrakt und werden dem Client über Darstellungen zugänglich gemacht. Eine Darstellung ist der konkrete Zustand einer Ressource zu einem bestimmten Zeitpunkt [Fie00]. Der Hauptzweck einer Darstellung besteht darin, die interne Datenspeicherung auf dem Server von der Kommunikation mit dem Client zu trennen. Eine Darstellung besteht aus Daten und zusätzlichen Metadaten, wie z. B. dem Medientyp. Durch die Verwendung verschiedener Darstellungen kann eine Ressource in mehreren Formaten bereitgestellt werden. Die gängigsten Darstellungsformate sind XML oder JSON. In der modernen Webentwicklung hat sich JSON als Standardformat durchgesetzt, da es eine effiziente Verarbeitung und eine gute Lesbarkeit gewährleistet. [RAR13]

3.3.5 Hypermedia (HATEOAS)

Viele RESTful-Dienste basieren auf dem Prinzip der Hypermedia. Hypermedia ist eine der Einschränkungen von REST und spielt eine wichtige Rolle bei der Verwaltung des Anwendungsstatus. Der Kern dieses Prinzips besteht darin, dass zusammen mit den Hauptdaten zusätzliche Informationen übertragen werden – Metadaten, die Links zu verwandten Ressourcen enthalten. Anhand dieser Links kann der Client mögliche weitere Aktionen bestimmen, die mit ihm durchgeführt werden können. Die Darstellung einer Ressource enthält in der Regel Hyperlinks, die sie mit anderen Ressourcen verbinden. Diese Ressourcen können sich sowohl auf einem als auch auf verschiedenen Servern befin-

den. Die Verbindung zwischen ihnen wird durch zusätzliche Informationen hergestellt, die in der Darstellung der Ressource enthalten sind. Dadurch kann der Client nacheinander zwischen den verbundenen Ressourcen wechseln und die erforderlichen Informationen in der richtigen Reihenfolge abrufen. [Til+15]

Es gibt viele Formate, die für Hypermedia geeignet sind und sich in ihrer Umsetzung unterscheiden können. Die Umsetzung im JSON-Format sieht wie folgende aus:

```

1 {
2   .."account": "DE1234567890",
3   .."balance": 1500.50,
4   .."currency": "EUR",
5   .."status": "ACTIVE",
6   .."_links": {
7     .."self": { "href": "/v1/accounts/DE1234567890" },
8     .."transfer": { "href": "/v1/accounts/DE1234567890/transfer" }
9   }
10 }

```

Abbildung 3.1: Hypermedia in JSON Format

3.3.6 Einheitliche Schnittstelle

Eine einheitliche Schnittstelle vereinfacht die Systemarchitektur und macht die Interaktion zwischen verschiedenen Komponenten verständlicher und vorhersehbarer. Dank der Standardisierung der Kommunikation können Client und Server unabhängig voneinander weiterentwickelt werden.

Zwei grundlegende Eigenschaften der HTTP-Methoden sind hier entscheidend:

- **Sicherheit:** Eine Methode gilt als sicher, wenn sie den Zustand der Ressource auf dem Server nicht verändert. Sie dient nur zum Abrufen von Informationen (Lesezugriff).
- **Idempotenz:** Die Methode ist idempotent, wenn die mehrfache Ausführung derselben Anfrage denselben Effekt auf den Status des Servers hat wie die einmalige Ausführung. Dies ist im Falle von Netzwerkfehlern wichtig, da der Client die Anfrage auf sichere Weise wiederholen kann.

Tabelle 3.1: Übersicht der HTTP-Methoden und deren Eigenschaften

Methode	Beschreibung	Sicher	Idempotent
GET	Abrufen einer Ressourcen-Repräsentation	Ja	Ja
POST	Erzeugen einer Ressource oder Datenverarbeitung	Nein	Nein
PUT	Ersetzen oder Erzeugen einer Ressource	Nein	Ja
DELETE	Löschen einer Ressource	Nein	Ja

In Tabelle 3.1 sind die Verben aufgeführt, die im REST-Kontext verwendet werden, sowie ihre Bedeutung und Eigenschaften. [Til+15]

3.3.7 API-First-Ansatz und OpenAPI-Spezifikation

API-First ist ein Ansatz zur Softwareentwicklung, bei dem die API vor Beginn der Systemimplementierung entworfen und beschrieben wird. Zunächst wird festgelegt, welche Endpunkte, Methoden, Parameter und Datenformate verwendet werden sollen. Anschließend wird die Logik des Dienstes selbst entwickelt. Die Idee dahinter ist, dass die API zum zentralen Vertrag zwischen verschiedenen Systemkomponenten (z. B. zwischen Microservices, Frontend und Backend) wird. [Stu16]

Als Werkzeug für diese formale Beschreibung wird die OpenAPI-Spezifikation (OAS) verwendet. Sie hat sich zum Industriestandard für die Beschreibung von RESTful-Webdiensten in einem maschinenlesbaren und sprachunabhängigen Format entwickelt. [Ope20]

Die Verwendung von OpenAPI bietet folgende strategische Vorteile:

- **Entkopplung:** Dank der Trennung der Schnittstelle und ihrer Implementierung können die Entwicklung des Servers und der Client-Komponenten (z. B. des Keycloak-Adapters) parallel erfolgen. Die Teams können sich auf einen vorab festgelegten API-Vertrag stützen. Dies vereinfacht die Koordination der Arbeit zwischen den verschiedenen Teilen des Systems.
- **Wartbarkeit:** Die formale Beschreibung der API vereinfacht auch die Wartung des Systems. Die Struktur der Schnittstelle ist im Voraus klar in der Spezifikation beschrieben. Dadurch können Entwickler die interne Logik des Dienstes ändern, ohne die Client-Anwendungen zu beeinträchtigen, solange der API-Vertrag selbst unverändert bleibt [Stu16]. Dies ermöglicht die Weiterentwicklung und Verbesserung des Systems, ohne dass die Integration mit anderen Komponenten geändert werden muss.
- **Automatisierung:** Die OpenAPI-Spezifikation hilft bei der automatischen Erstellung von Servervorlagen, Client-Bibliotheken und interaktiver API-Dokumentation. Dies reduziert den manuellen Aufwand und trägt dazu bei, die Wahrscheinlichkeit von Fehlern zu reduzieren, die bei der manuellen Implementierung von Schnittstellen auftreten können [Ope20].

3.4 Identitätsmanagement und Autorisierung

In der modernen Microservice-Architektur ist die zentrale Verwaltung von Identitäten und Zugriffsrechten von großer Bedeutung. Für *pin-tan-service* wird Keycloak als Identitäts- und Zugriffsmanagementsystem (IAM) verwendet. Die Kommunikation basiert auf den etablierten Standards OAuth 2.0 und OpenID Connect.

3.4.1 Das OAuth 2.0 Framework

OAuth 2.0 ist ein weit verbreitetes Autorisierungsframework. Es ermöglicht Anwendungen, eingeschränkten Zugriff auf geschützte Ressourcen des Benutzers zu erhalten, ohne dessen Anmeldedaten an Dritte weiterzugeben. Die Grundidee dieses Ansatzes besteht in der Delegierung des Zugriffs. Der Benutzer erteilt der Anwendung die Berechtigung, bestimmte Aktionen auszuführen, wobei sein Benutzername und sein Passwort nur dem Authentifizierungsserver bekannt sind. Ein solcher Mechanismus erhöht die Sicherheit der Interaktion zwischen verschiedenen Diensten wesentlich und verringert das Risiko eines Verlusts von Anmeldedaten [Har12].

In der OAuth 2.0-Architektur gibt es mehrere Hauptrollen. Der Resource Owner ist der Benutzer, der die geschützten Ressourcen besitzt. Der Client ist eine Anwendung, die Zugriff auf diese Ressourcen anfordert. Der Authorization Server ist für die Authentifizierung des Benutzers und die Ausgabe von Zugriffstoken verantwortlich. Der Resource Server, in unserem Fall der pin-tan-service, speichert geschützte Daten und gewährt Zugriff darauf, wenn ein gültiger Token vorliegt. In modernen Systemen werden diese Komponenten oft als separate Dienste implementiert. [RS17] In der betrachteten Architektur fungiert beispielsweise Keycloak als Autorisierungsserver. Der entwickelte Dienst übernimmt hingegen die Funktion des Ressourcenservers.

Das zentrale Element des OAuth 2.0-Mechanismus ist das Zugriffstoken (Access Token). Nach erfolgreicher Authentifizierung und Autorisierung gibt der Autorisierungsserver ein spezielles Token an die Client-Anwendung aus. Dieses Token wird für den Zugriff auf geschützte Ressourcen verwendet. [Har12] Der Client übermittelt dieses Token in jeder Anfrage an den Ressourcenserver, in der Regel im HTTP-Header Authorization. Der Ressourcenserver überprüft die Authentizität des Tokens und entscheidet über die Gewährung des Zugriffs. In modernen Systemen werden Tokens häufig im JWT-Format implementiert, das Informationen über den Benutzer, die Zugriffsrechte und die Gültigkeitsdauer des Tokens enthält. [JBS15]

In einer Microservice-Architektur bietet die Verwendung von OAuth 2.0 eine Reihe von Vorteilen. Da alle Informationen über den Benutzer und seine Rechte in einem Token enthalten sein können, müssen einzelne Dienste den Status der Benutzersitzung nicht speichern. Dies vereinfacht die Skalierung des Systems und ermöglicht es den Diensten, unabhängig voneinander zu bleiben. Außerdem bietet ein zentraler Autorisierungsserver einen einheitlichen Mechanismus zur Zugriffsverwaltung für alle Komponenten des Systems. [New21]

Somit ist OAuth 2.0 ein wichtiger Bestandteil der modernen Architektur verteilter Systeme. Es ermöglicht eine sichere und skalierbare Zugriffsverwaltung zwischen verschiedenen Diensten und Anwendungen, was es im Kontext von Microservice- und Cloud-

Architekturen besonders relevant macht. [Har12]

3.4.2 OpenID Connect (OIDC)

OpenID Connect (OIDC) ist ein Authentifizierungsprotokoll, das auf dem OAuth 2.0-Framework basiert. OAuth 2.0 dient in erster Reihe zur Autorisierung und Verwaltung des Zugriffs auf geschützte Ressourcen. OpenID Connect erweitert diese Funktionen. Es fügt einen Mechanismus zur Benutzerauthentifizierung hinzu. Dadurch kann die Client-Anwendung nicht nur auf Ressourcen zugreifen, sondern auch die Identität des Benutzers bestätigen.

Das Hauptelement von OpenID Connect ist das ID-Token – ein spezielles Token, das Informationen über den authentifizierten Benutzer enthält. Es wird in der Regel im JWT-Format implementiert. Dieses Token wird vom Autorisierungsserver nach erfolgreicher Authentifizierung des Benutzers ausgegeben. Die Client-Anwendung kann ihn überprüfen, um die Identität des Benutzers zu bestätigen. Neben dem ID-Token verwendet OpenID Connect auch einen Zugriffstoken. Dieser Token wird für den Zugriff auf geschützte Ressourcen verwendet. Auf diese Weise erhält der Client während eines einzigen Authentifizierungs- und Autorisierungsprozesses zwei Arten von Tokens. Der ID-Token wird zur Bestätigung der Identität des Benutzers verwendet, während der Zugriffstoken für den Zugriff auf die Ressourcen des Systems dient. Diese Aufteilung ermöglicht eine klare Trennung zwischen den Aufgaben der Authentifizierung und der Autorisierung.

Darüber hinaus definiert OpenID Connect einen Standardsatz von Benutzerattributen, die als **Claims** bezeichnet werden. Diese Daten können den Benutzernamen, die E-Mail-Adresse, eine eindeutige Kennung und andere Informationen über den Benutzer umfassen. Die Client-Anwendung kann diese Daten entweder aus dem ID-Token oder über einen speziellen Endpunkt **userinfo** abrufen, der vom Autorisierungsserver bereitgestellt wird.

In modernen verteilten Systemen wird OpenID Connect häufig für die Implementierung eines einheitlichen Anmeldesystems (Single Sign-On, SSO) verwendet. Dadurch kann sich der Benutzer einmalig authentifizieren. Danach kann er auf mehrere Anwendungen zugreifen, ohne seine Anmeldedaten erneut eingeben zu müssen. In der betrachteten Architektur unterstützt Keycloak OpenID Connect. Es fungiert als Identitätsanbieter (Identity Provider) und sorgt für eine zentralisierte Authentifizierung der Benutzer. Keycloak gibt auch Token für die Interaktion mit verschiedenen Systemdiensten aus. [Ope14]

3.4.3 Keycloak als Identity and Access Management System

Keycloak ist eine Open-Source-Lösung für die Identitäts- und Zugriffsverwaltung, mit der sich Sicherheitsaspekte einfach in moderne Anwendungen integrieren lassen (IAM). Im Kontext der Infrastruktur von flatexDEGIRO fungiert Keycloak als Autorisierungsserver. Die Hauptaufgabe von Keycloak ist die Zentralisierung der Sicherheitslogik. Anstatt die

Authentifizierung und Zugriffsverwaltung in jeder einzelnen Anwendung zu implementieren, werden diese Funktionen an Keycloak delegiert. Dies vereinfacht die Wartung der Anwendungen und macht sie sicherer, da alle Zugriffs- und Authentifizierungseinstellungen an einem Ort gespeichert sind. Die folgenden Kernfunktionalitäten von Keycloak bilden die Grundlage für die Sicherheitsarchitektur dieser Arbeit [Red25]:

- **Unterstützung von Sicherheitsstandards:** Unterstützung moderner Sicherheitsstandards wie OpenID Connect, OAuth 2.0 und SAML 2.0. Aufgrund der Unterstützung dieser Protokolle kann Keycloak als zentrale Komponente für die Authentifizierung und Autorisierung in verteilten Systemen und Microservice-Architekturen eingesetzt werden.
- **Identitätsverbund (Identity Federation):** Dies ermöglicht die Integration externer Benutzerverwaltungssysteme wie LDAP oder Active Directory (AD). So können bereits vorhandene Benutzerverzeichnisse genutzt und der Zugriff auf verschiedene Anwendungen zentral verwaltet werden.
- **Multi-Faktor-Authentifizierung (MFA):** In Keycloak kann eine zusätzliche Sicherheitsstufe eingerichtet werden, beispielsweise Einmalpasswörter (OTP), die von Anwendungen wie Google Authenticator generiert werden. Dies erhöht die Sicherheit beim Anmelden der Benutzer erheblich.
- **Flexibilität und Anpassbarkeit:** Möglichkeit zur Anpassung des Erscheinungsbilds der Anmeldeseiten (Designvorlagen). Erstellung eigener Plugins und Adapter für verschiedene Frameworks.
- **Multi-Tenancy (Realms):** Eine wichtige Funktion von Keycloak ist die Unterstützung von Multi-Tenancy über den sogenannten Realm-Mechanismus. Jeder Realm ist ein isolierter Bereich, in dem Benutzer, Rollen und Sicherheitseinstellungen unabhängig von anderen Projekten verwaltet werden können. So kann eine einzige Keycloak-Installation für mehrere Anwendungen oder Organisationen genutzt werden.
- **Erweiterbarkeit (SPI):** Keycloak ist durch die Verwendung der Service Provider Interface (SPI) in hohem Maße erweiterbar. Dieser Mechanismus ermöglicht es Entwicklern, ihre eigene Authentifizierungs- und Autorisierungslogik zu implementieren und benutzerdefinierte Erweiterungen zu erstellen. Im Rahmen dieser Arbeit wird dieser Mechanismus für die Entwicklung eines Custom-Providers verwendet, der für die Implementierung des PIN/TAN-Verifizierungsprozesses vorgesehen ist [TS21].
- **Sitzungsverwaltung und Audit:** Keycloak bietet Mechanismen zur Verwaltung von Benutzersitzungen und zur Sicherheitsüberprüfung. Administratoren haben die Möglichkeit, aktive Benutzersitzungen zu verfolgen und sicherheitsrelevante Ereignisse

nisprotokolle zu analysieren. Dies vereinfacht die Überwachung des Systems und die Erkennung potenzieller Bedrohungen.

3.4.4 JSON Web Token (JWT)

JWT ist ein offener Standard gemäß RFC 7519 für den sicheren Austausch von Informationen in Form eines JSON-Objekts. In dieser Architektur dient JWT als **Access Token** zur Bestätigung der Benutzerautorisierung im *pin-tan-service* [JBS15].

JWT ist selbstbeschreibend (stateless). Das bedeutet, dass alle erforderlichen Informationen über den Benutzer und seine Rechte direkt im Token enthalten sind. Auf diese Weise kann der Microservice die Anfrage ohne zusätzliche Datenbankabfrage oder erneute Anfrage an Keycloak überprüfen [New21].

Ein Token besteht aus drei durch Punkte getrennten Bestandteilen:

1. **Header:** Enthält Metadaten, wie den verwendeten Signaturalgorithmus.
2. **Payload:** Enthält die fachlichen Daten (*Claims*), wie die Benutzer-ID und die Gültigkeitsdauer (*Expiration Time*).
3. **Signature:** Gewährleistet die Integrität des Tokens. Sie wird vom Keycloak-Server mit einem privaten Schlüssel erstellt und kann vom Service mit einem öffentlichen Schlüssel verifiziert werden.

4 Anforderungsanalyse

Im aktuellen Kapitel wird vor Beginn der technischen Umsetzung des *pin-tan-service* eine detaillierte Analyse der bestehenden Strukturen und der gestellten Aufgaben durchgeführt. Die Anforderungsanalyse ist notwendig, um die Schwachstellen des aktuellen Systems zu identifizieren. Auf der Grundlage der erzielten Ergebnisse werden funktionale und nicht-funktionale Anforderungen an die neue Microservice-Architektur formuliert.

4.1 Methodik der Anforderungserhebung

Die in diesem Kapitel definierten Anforderungen wurden mit dem Betreuer bei flatex-DEGIRO in einer Reihe von Diskussionen erarbeitet. Ziel dieser Diskussionen war es, die Schwachstellen der bestehenden Architektur aufzudecken. Dabei wurden die technischen Einschränkungen von *pitase lib* analysiert. Gleichzeitig wurden die funktionalen und nicht-funktionalen Anforderungen an den neuen Service definiert. Darüber hinaus wurden die grundlegenden architektonischen Ziele festgelegt. Dazu gehören eine höhere Entkopplung der Komponenten und die Unterstützung moderner Cloud-Infrastrukturen.

Zusätzlich wurden die bestehenden Systemabhängigkeiten und die notwendigen Funktionen aus der alten Architektur identifiziert, die im neuen Service weiterhin unterstützt werden müssen. Parallel dazu wurde eine Bestandsaufnahme der bestehenden Geschäftsprozesse durchgeführt, um veraltete oder nicht mehr genutzte Funktionen zu ermitteln. Prozesse wie die Verwaltung von TAN-Listen werden bei der weiteren Umsetzung des neuen Systems nicht berücksichtigt.

Anschließend wurden die Anforderungen nach Priorität geordnet, um eine zielgerichtete und schrittweise Umsetzung des neuen Systems zu gewährleisten.

4.2 Ist-Zustand

Das aktuelle System basiert auf einer monolithischen Struktur, in der die zentrale Logik zur Verarbeitung von PIN- und TAN-Codes in einer gemeinsamen Bibliothek namens *pitase lib* gekapselt ist. Wie in der Abbildung 4.1 dargestellt, nutzen die beiden Hauptsysteme (System A und System B) diese Bibliothek, um auf die gemeinsame *Pitase-Datenbank* zuzugreifen.

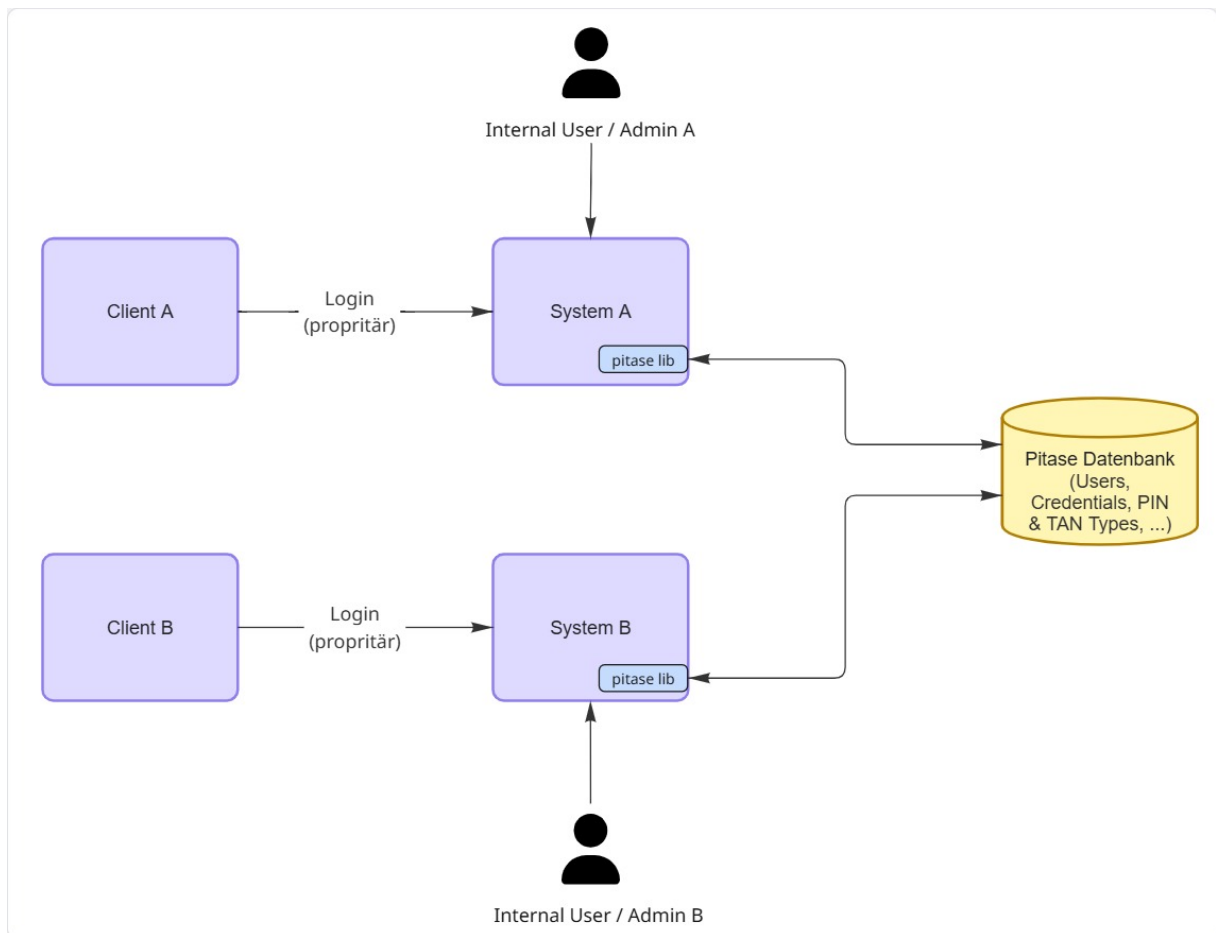


Abbildung 4.1: Ist-Zustand

Derzeit erhalten Kunden (Client A und Client B) Zugriff über systemspezifische, proprietäre Anmeldeschnittstellen. Es gibt keinen einheitlichen Sicherheitsstandard, da jedes System (System A und System B) die Authentifizierung eigenständig verwaltet. Dies erschwert die Anbindung externer Partner und erhöht das Risiko von Sicherheitslücken. Jedes System verfügt über eigene interne Benutzer und Administratoren (Internal User / Admin A und B). Diese sind direkt an das jeweilige System gebunden. Kunden interagieren mit den Systemen, die wiederum *pitase lib* aufrufen. Aufgrund der strikten Einbindung der Identitätsprüfung in diese Kette kann kein Teil der Logik aktualisiert werden, ohne den gesamten Authentifizierungsprozess in beiden Systemen zu gefährden.

Die derzeitige Architektur hat erhebliche Mängel in Bezug auf die Infrastruktur und die Wartung:

- **Enge Kopplung:** Da die Bibliothek direkt in System A und System B integriert ist, sind diese Anwendungen eng miteinander verbunden. Jede Änderung an der Bibliothek erfordert eine gleichzeitige Aktualisierung und Bereitstellung beider Systeme.
- **Wartungsaufwand und Legacy-Code:** Das Fehlen aktueller Dokumentation und Fachkenntnisse über die detaillierten internen Prozesse macht die Erweiterung der

Bibliothek schwierig und risikobehaftet.

- **Eingeschränkte Skalierbarkeit:** Die horizontale Skalierung (das Hinzufügen weiterer Instanzen) gestaltet sich schwierig, weil die Logik fest in veraltete Systeme eingebettet ist.
- **Hohe Downtime:** Bibliotheksaktualisierungen erfordern häufig, dass alle damit verbundenen Systeme angehalten werden. Derzeit führen solche Aktualisierungen zu stundenlangen Ausfallzeiten. Für den Bankensektor stellen solche langen Systemausfälle ein ernstes Problem dar und entsprechen nicht den heutigen Anforderungen.
- **Mangelnde Testbarkeit:** Es ist praktisch unmöglich, eine Bibliothek isoliert zu testen, da sie immer im Rahmen größerer Systeme ausgeführt werden muss. In den meisten Fällen lässt sie sich nur mithilfe von Unit-Tests überprüfen, während eine umfassende Integrationstestung schwierig ist.

4.3 Soll-Zustand

Ziel der vorliegenden Arbeit ist die Transformation der monolithischen Bibliothek in einen eigenständigen, modernisierten Microservice – den *pin-tan-service*. In der neuen Architektur (siehe Abbildung 4.2) wird die Logik vollständig entkoppelt.

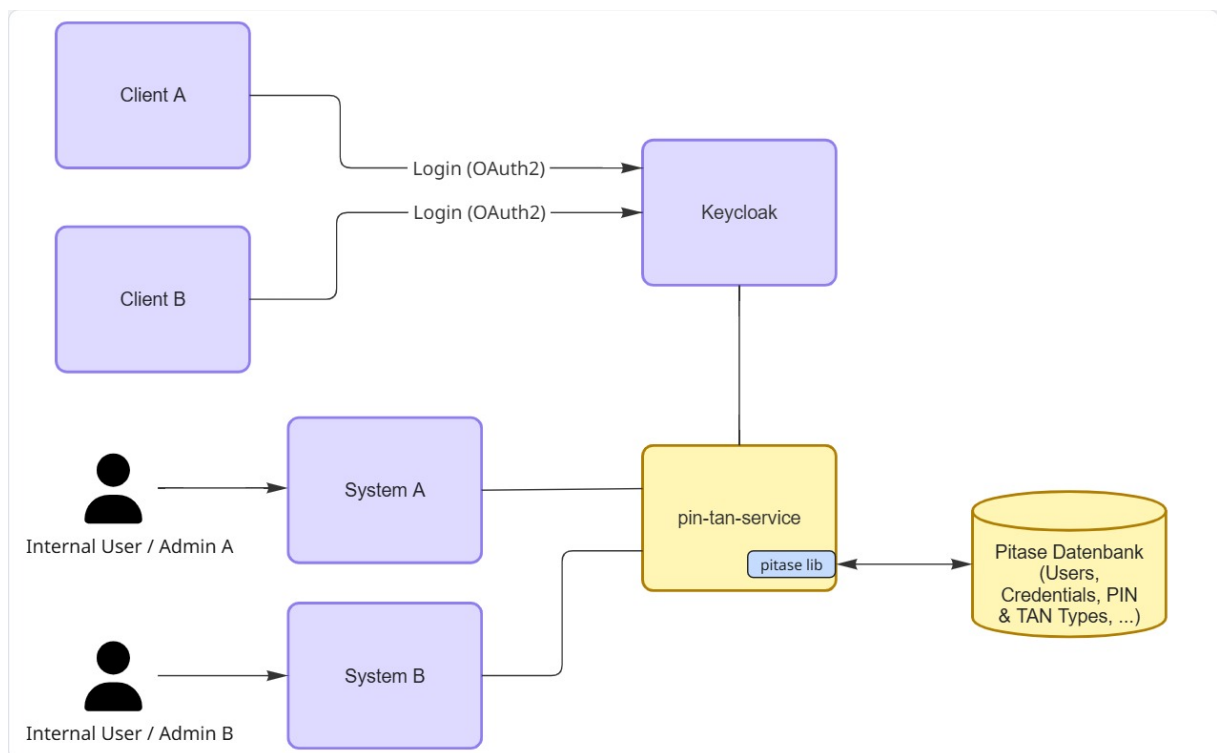


Abbildung 4.2: Soll-Zustand

Die Soll-Architektur (siehe Abbildung 4.2) definiert folgende Kommunikationswege:

- **Zentralisierte Authentifizierung via Keycloak:** Die Endkunden (Client A und Client B) nutzen nun ein standardisiertes Login-Verfahren über Keycloak. Anstelle des alten proprietären Logins kommt der moderne Standard OAuth 2.0 (bzw. OpenID Connect) zum Einsatz. Keycloak übernimmt die Identitätsprüfung und kommuniziert bei Bedarf mit dem *pin-tan-service*, um die Benutzerdaten (PIN/TAN) zu validieren.
- **Anbindung der Altsysteme:** Die Altsysteme (System A und System B) bleiben bestehen, greifen aber nicht mehr direkt auf die Datenbank zu. Stattdessen kommunizieren sie über eine definierte REST-Schnittstelle mit dem PIN-TAN-Service. Auch die Internal User und Admins von System A und B werden auf diesem Weg autorisiert.
- **Der pin-tan-service als Kernkomponente:** Der neue Microservice kapselt die ursprüngliche Pitase-Bibliothek. Er ist die einzige Komponente, die eine direkte Verbindung zur Pitase-Datenbank hält. Dadurch wird sichergestellt, dass Datenbankänderungen nur an einer Stelle (im Service) gepflegt werden müssen.
- **API-First und Versionierung:** Durch die Einführung des *pin-tan-service-api*-Moduls wird ein klarer Kontrakt (OpenAPI-Spezifikation) definiert. Dadurch können System A und System B unabhängig voneinander auf neue Versionen des Dienstes migrieren, ohne dass eine gleichzeitige Aktualisierung aller Komponenten erforderlich ist.

4.4 Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die konkreten Aufgaben und die Geschäftslogik des Service.

- **F1 Validierung von Credentials (PIN/TAN):** Der Service überprüft die eingegebenen PIN- und TAN-Codes anhand der vorhandenen Datenbank. Außerdem übernimmt der Service die Geschäftslogik der ursprünglichen Bibliothek *pitase lib*.
- **F2 Integration in Keycloak (Custom Provider):** Die Implementierung einer Schnittstelle zur Einbindung in Keycloak (User Federation / Custom User Storage). Dies ermöglicht die Nutzung des *pin-tan-service* als Credential Provider für den OAuth2-Authentifizierungsprozess.
- **F3: Bereitstellung einer REST-API:** Anwendung des *Contract-First*-Ansatzes (OpenAPI) zur Entwicklung einer Schnittstelle für die Interaktion zwischen bestehenden und neuen Systemen.
- **F4: Bereitstellung von Client-Bibliotheken:** Entwicklung geeigneter Client-Bibliotheken (SDKs) auf Basis der neuen API-Spezifikation zur Anbindung von

System A und System B.

- **F5: Entfernung nicht mehr genutzter Funktionen:** Die Bereinigung der alten Logik erfolgt durch die Entfernung veralteter Prozesse, beispielsweise der Verwaltung von TAN-Listen. Dabei bleiben die geschäftskritischen Bankfunktionen erhalten.

4.5 Nicht-funktionale Anforderungen

Nicht-funktionale Anforderungen beschreiben qualitative Merkmale wie Sicherheit, Leistungsfähigkeit und Wartungsfreundlichkeit.

- **NF1 Entkopplung und Isolation:** Komplette Trennung der PIN/TAN-Logik von veralteten Systemen. Der Dienst muss als eigenständige Einheit funktionieren können, unabhängig von den Systemen A und B.
- **NF2 Horizontale Skalierbarkeit:** Die Architektur soll es ermöglichen, bei steigender Last zusätzliche Instanzen des Dienstes (z. B. durch Pod-Replikation) zu starten, um die Verarbeitung der Anfragen zu verteilen.
- **NF3 Automatisiertes Deployment und CI/CD:** Einführung einer automatisierten Deployment-Pipeline zur schnelleren Auslieferung (Time-to-Market) und Reduzierung manueller Fehler.
- **NF4 Modernisierung des Technologiestacks:** Verwendung aktueller, unternehmensweit zugelassener Standards (Approved Technology), konkret Java 17+ und das Quarkus-Framework.
- **NF5 Zielplattform (Containerisierung):** Der Service muss für den Einsatz in einer Cloud-Umgebung (Docker, Kubernetes) optimiert sein.
- **NF6 Testbarkeit und Qualitätssicherung:** Verbesserung der Testabdeckung durch automatisierte Akzeptanztests (Behavior Driven Development mit Cucumber) und die Integration in das JIRA-Ticketsystem via X-Ray.
- **NF7 Sicherheit (Standardisierung):** Ablösung proprietärer Logins durch den Industriestandard OAuth 2.0 / OpenID Connect.

4.6 Priorisierung der Anforderungen

Die folgende Tabelle fasst alle identifizierten Anforderungen zusammen und unterteilt sie nach ihrer Priorität.

ID	Anforderung	Typ	Priorität
F1	Validierung von Credentials (PIN/TAN)	Funktional	Muss
F2	Integration in Keycloak (SPI Custom Provider)	Funktional	Muss
F3	Bereitstellung einer REST-API (Open-API)	Funktional	Muss
F4	Bereitstellung von Client-Bibliotheken (SDKs)	Funktional	Muss
F5	Entfernung nicht mehr genutzter Funktionen	Funktional	Kann
NF1	Entkopplung und Isolation vom Monolithen	Nicht-Funktional	Muss
NF2	Horizontale Skalierbarkeit (Pod-Replikation)	Nicht-Funktional	Kann
NF3	Automatisiertes Deployment (CI/CD)	Nicht-Funktional	Kann
NF4	Modernisierung (Java 17+, Quarkus)	Nicht-Funktional	Muss
NF5	Zielform (Docker, Kubernetes)	Nicht-Funktional	Muss
NF6	Testbarkeit (Cucumber, X-Ray Integration)	Nicht-Funktional	Kann
NF7	Sicherheit (Standard OAuth 2.0 / OIDC)	Nicht-Funktional	Muss

Tabelle 4.1: Priorisierungsmatrix der Anforderungen

5 Entwurf und Konzeption

5.1 Architekturüberblick

Die Architektur des Gesamtsystems ist als moderne, cloud-native Microservice-Landschaft konzipiert. Sie basiert auf einer strikten Trennung der Verantwortlichkeiten und einer netzwerktechnischen Segmentierung in verschiedene Namespaces.

5.1.1 Infrastruktur und Netzwerksegmentierung

Das System ist in zwei primäre logische Bereiche unterteilt, um die Sicherheit und Skalierbarkeit zu maximieren:

1. **Banking-API Namespace:** Hier befinden sich die fachlichen Komponenten wie das API-Gateway (KrakenD¹), welches als zentraler Einstiegspunkt fungiert und Aufgaben wie Routing, Authentifizierung sowie Rate Limiting übernimmt, die verschiedenen Backend-Services (Service A, Service B) sowie der neu entwickelte *pin-tan-service*.
2. **Banking-API-IAM Namespace:** Dieser isolierte Bereich ist für das Identity and Access Management zuständig. Er beherbergt den *Keycloak*-Server, die dazugehörige Datenbank sowie einen *Infinispan*-Cache zur Sitzungsverwaltung.

5.1.2 Komponenten und Datenfluss

Der Architekturüberblick (siehe Abbildung 5.1) verdeutlicht den Weg eines Requests und die Rolle der einzelnen Komponenten:

- **API-Gateway (KrakenD):** Fungiert als zentraler Einstiegspunkt (*Ingress*). Alle externen Anfragen treffen hier ein. KrakenD ist für das Routing und die Prüfung der Token-Authentizität zuständig. Die Validierung erfolgt dabei zustandslos (*stateless*): KrakenD bezieht über eine konfigurierte *JWK_URL* die öffentlichen Schlüssel (*JWKS*) von Keycloak. Dadurch ist das Gateway in der Lage, die Signatur des im *Authorization Header* mitgesendeten JWT eigenständig zu verifizieren. Eine erneute Kontaktierung von Keycloak ist somit bei jeder einzelnen Anfrage nicht mehr erforderlich.
- **Keycloak und Pitase-Client:** Bei einem Login-Request leitet das Gateway den Aufruf an Keycloak weiter. Innerhalb von Keycloak agiert der speziell entwickelte *Pitase-Client* (Custom User Storage Provider). Dieser kommuniziert über eine

¹ <https://www.krakend.io/docs/>

gesicherte Verbindung mit dem *pin-tan-service*, um die PIN-Validierung gegen die *Pitase DB* durchzuführen.

- **Token-basierte Autorisierung:** Nach erfolgreicher Prüfung stellt Keycloak ein JWT aus. Bei jedem folgenden Zugriff prüft KrakenD die Gültigkeit dieses Tokens. Nur valide Anfragen werden an die Backend-Services A oder B weitergereicht.
- **Backend-Validierung:** Optional können die Backend-Services das empfangene JWT eigenständig gegenüber Keycloak verifizieren, um eine feingranulare Autorisierung auf Service-Ebene sicherzustellen.

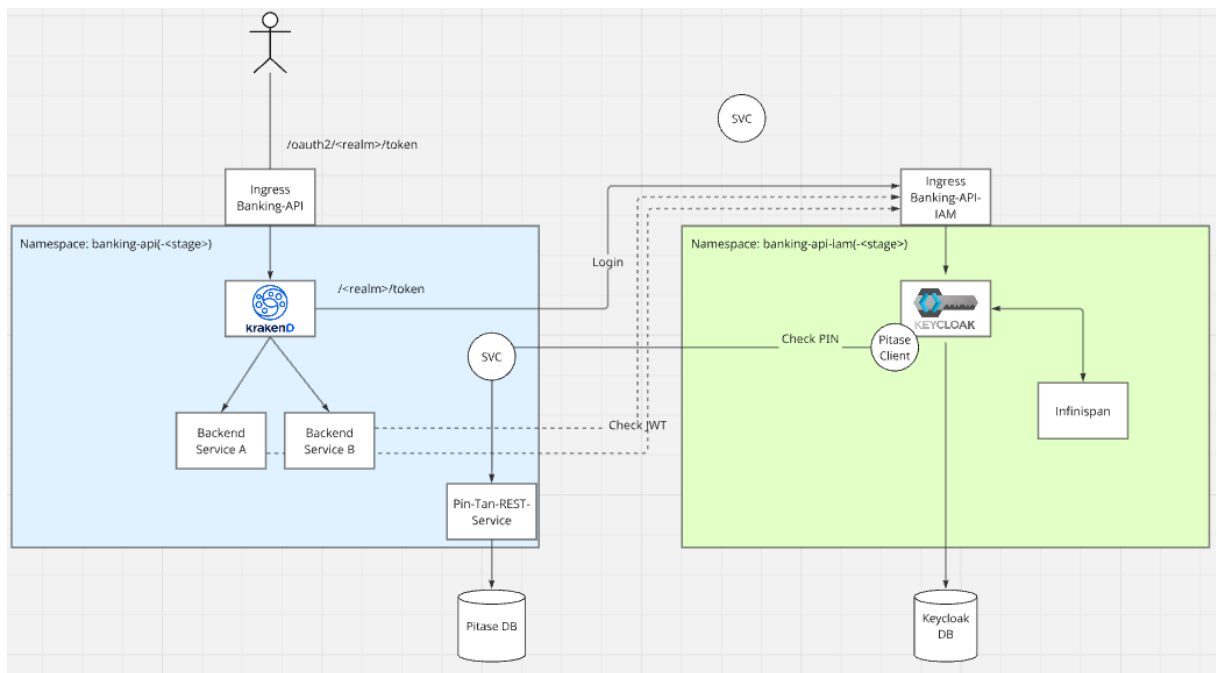


Abbildung 5.1: Detaillierter Architekturüberblick

5.2 Technologiewahl

5.2.1 Java 17

Für den *pin-tan-service* wurde Java 17 als primäre Programmiersprache ausgewählt. Neben den in Kapitel 3.1 beschriebenen technischen Vorteilen (Records, Sealed Classes) spielten strategische Gründe eine entscheidende Rolle:

Homogenität der Umgebung: Die IT-Infrastruktur der flatexDEGIRO basiert standardmäßig auf Java 17. Die Wahl dieser Version stellt sicher, dass der neue Dienst nahtlos in bestehende CI/CD-Pipelines und bankinterne Standardbibliotheken integriert werden kann.

Integrationsaufwand: Die Verwendung einer neueren Java-Version würde zum aktuellen Zeitpunkt umfangreiche Anpassungen in abhängigen Projekten erfordern und somit den

Bereitstellungsprozess (Time-to-Market) verzögern.

5.2.2 Anwendungsframework Quarkus

Für die Implementierung des *pin-tan-service* wurde das Quarkus-Framework ausgewählt. Basierend auf den in Kapitel 4 definierten Anforderungen erfüllt Quarkus alle Kriterien, die das Unternehmen flatexDEGIRO an einen modernen, sicherheitskritischen Microservice stellt. Die Entscheidung begründet sich durch folgende Faktoren:

- **Kubernetes-Native und Cloud-Optimierung:** Da die Zielplattform eine Kubernetes-Infrastruktur ist, bietet Quarkus durch seinen *Container-First*-Ansatz entscheidende Vorteile. Ein geringer Speicherverbrauch (*Memory Footprint*) und extrem kurze Startzeiten ermöglichen eine effiziente Ressourcennutzung innerhalb der Cluster-Umgebung.
- **Native Kompilierung:** Quarkus unterstützt die Erstellung nativer Executables (via GraalVM). Dies führt zu einer drastischen Reduzierung der Boot-Zeit, was insbesondere bei automatischen Skalierungsvorgängen (*Auto-Scaling*) im Bankbetrieb essenziell ist, um Lastspitzen verzögerungsfrei abzufangen.
- **Nahtlose Integration von Standards:** Quarkus bietet eine hervorragende Unterstützung für Java-Standards wie *Jakarta EE* und *MicroProfile*. Dies erleichtert die Anbindung an die bestehende *pitase lib* (Bibliothek) und ermöglicht eine effiziente Implementierung der REST-API. Durch die Nutzung dieser Standards wird sichergestellt, dass die bewährte Geschäftslogik stabil in die neue Architektur integriert werden kann.
- **Homogenität der Infrastruktur:** Ein entscheidender Faktor ist die bestehende Technologiestrategie der flatexDEGIRO. Da bereits zahlreiche Projekte innerhalb der Bank erfolgreich auf Quarkus basieren, profitiert die Entwicklung von vorhandenen Erfahrungswerten, standardisierten CI/CD-Templates und internen Bibliotheken. Dies reduziert die Einarbeitungszeit und gewährleistet eine nahtlose Integration in das bestehende Ökosystem.
- **Sicherheit und Protokollunterstützung:** Da der Service eine zentrale Rolle im Authentifizierungsprozess einnimmt, bietet Quarkus mit nativen Erweiterungen für *OpenID Connect* (OIDC) und der Integration in Keycloak eine robuste und einfach konfigurierbare Sicherheitsarchitektur.

Der ausschlaggebende Faktor für die Entscheidung zugunsten von Quarkus war die Notwendigkeit, einen leistungsstarken und ressourceneffizienten Vermittlungsdienst zu entwickeln, der den strengen Anforderungen des Unternehmens in Bezug auf Compliance und Infrastruktur gerecht wird.

5.2.3 Auswahl des IAM-Systems: Keycloak

Die Entscheidung für Keycloak basiert auf den in Kapitel 3.4.3 beschriebenen Kernfunktionalitäten. Für die Anforderungen der flatexDEGIRO sind dabei folgende Aspekte ausschlaggebend:

- **Standardisierung:** Die Unterstützung von OIDC und OAuth 2.0 (siehe Kap. 3.4.3) ermöglicht die im Ist-Zustand fehlende Trennung von Sicherheits- und Geschäftslogik.
- **Integration der Legacy-Systeme:** Der entscheidende Faktor ist die Erweiterbarkeit über SPI. Dies wird in dieser Arbeit genutzt, um den spezifischen PIN/TAN-Prozess über einen Custom-Provider anzubinden, ohne die *pitase lib* zu verändern.
- **Mandantenfähigkeit:** Durch den Einsatz von Realms wird die geforderte Isolation der Sicherheitskonfigurationen für den *pin-tan-service* technisch umgesetzt.

5.3 Build-Management und Deployment-Infrastruktur

Um einen automatisierten und reproduzierbaren Lebenszyklus des *pin-tan-service* zu gewährleisten, wird eine moderne Toolchain eingesetzt. Diese unterstützt den Prozess von der Entwicklung bis hin zum Betrieb in der Cloud-Infrastruktur der flatexDEGIRO.

- **Build-Management mit Maven:** Als primäres Build-Werkzeug für den *pin-tan-service* wird *Apache Maven*² verwendet. Es dient der zentralen Verwaltung von Abhängigkeiten und der Standardisierung des Build-Prozesses. Durch die Definition von Projekt-Objekt-Modellen (POM) wird sichergestellt, dass alle benötigten Bibliotheken, insbesondere die bankinterne *pitase lib*, in den korrekten Versionen eingebunden werden.
- **Automatisierung mit GitLab CI/CD:** Zur Automatisierung des Build- und Deployment-Prozesses wird *GitLab*³ CI/CD eingesetzt. Dieses System ist direkt in das Versionskontrollsystem des Unternehmens integriert. Sobald neuer Code in das Repository übertragen wird, triggert dies eine Pipeline, die den *pin-tan-service* baut, testet und für die Verteilung vorbereitet.
- **Containerisierung mit Docker:** Zur Reduzierung des Aufwands für die Betriebsabteilung (IT-Operating) wird der *pin-tan-service* mithilfe von *Docker*⁴ gekapselt und als *Docker-Image* bereitgestellt. Die Anwendung wird zusammen mit allen spezifischen Abhängigkeiten in ein in sich geschlossenes Image verpackt. Dadurch muss das IT-Operating keine anwendungsspezifischen Installationsmechanismen mehr ken-

² <https://maven.apache.org/>

³ <https://about.gitlab.com/>

⁴ <https://www.docker.com/>

nen. Die Betriebsteams können sich stattdessen auf die standardisierte Auslieferung der Container in die Infrastruktur konzentrieren. Ein solches Image garantiert ein konsistentes Laufzeitverhalten über alle Umgebungen hinweg. Da diese Methodik unabhängig von der verwendeten Programmiersprache (z. B. Java oder Go) funktioniert, wird eine systemweite Standardisierung erreicht. Dies vereinfacht die Deployment-Prozesse und erleichtert die Überwachung sowie Fehleranalyse durch das Monitoring-Team im laufenden Betrieb

- **Konfigurationsmanagement mit Helm:** Für die Verwaltung der Bereitstellungen in Kubernetes wird *Helm* eingesetzt. Helm bietet einen vorlagenbasierten Ansatz (*Charts*) für die Definition und Bereitstellung komplexer Anwendungen. Dies erlaubt es, umgebungsspezifische Parameter wie Ressourcenlimits oder Datenbank-Endpunkte effizient zu verwalten.
- **Orchestrierung mit Kubernetes:** Schließlich dient *Kubernetes* als zentrale Container-Orchestrierungs-Plattform für den *pin-tan-service*. Kubernetes bietet eine hochskalierbare und zuverlässige Infrastruktur für die Bereitstellung und automatisierte Verwaltung von containerisierten Anwendungen.

Der vollständige Arbeitsablauf, von der Code-Änderung in GitLab bis hin zum Deployment im Cluster, folgt einem modernen **GitOps-Ansatz**. Dabei wird der gewünschte Zustand der Infrastruktur deklarativ beschrieben und automatisch synchronisiert, was die Fehlerrate bei manuellen Eingriffen minimiert [Gez23]. Der detaillierte Workflow ist in Abbildung 5.2 dargestellt.

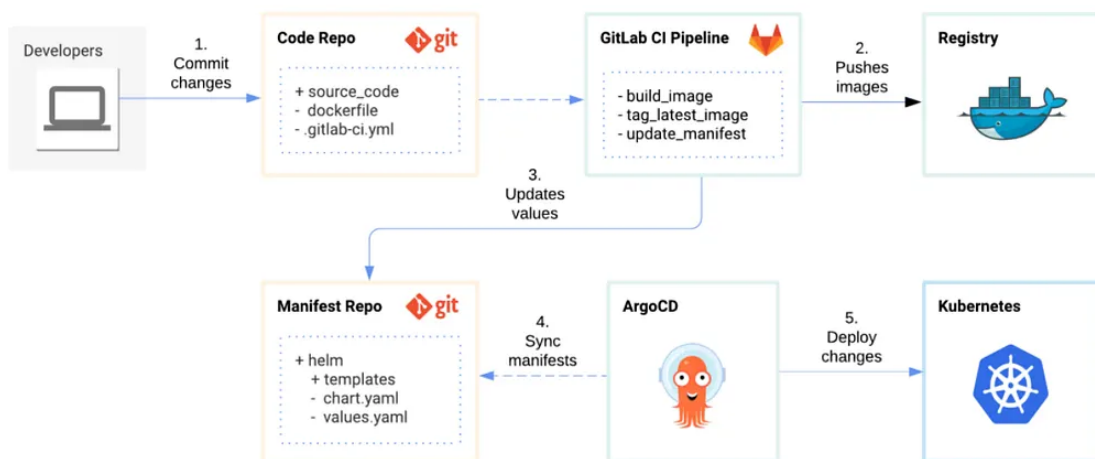


Abbildung 5.2: Vollständiger CI/CD-Arbeitsablauf und Orchestrierung mit Kubernetes

5.4 API-Design

Der Entwurf des API-Dienstes *pin-tan-service* folgt dem REST-Architekturstil und orientiert sich am API-First-Ansatz. Zur formalen Beschreibung der Schnittstelle wird die OpenAPI-Spezifikation (V3) verwendet. Dies gewährleistet, dass alle beteiligten Systeme (sowohl alte als auch neue) gemäß einem klar definierten Vertrag entwickelt werden können.

5.4.1 Strukturierung der Endpunkte

Um die Zuständigkeiten klarer abzugrenzen und die Zugriffsverwaltung zu vereinfachen, wurde die API in drei logische Bereiche unterteilt. Jeder dieser Bereiche ist für bestimmte Benutzer und deren Zugriffsrechte vorgesehen.

- **User-Operations:** Dazu gehören Funktionen, die direkt vom Endkunden (Client A und B) über Keycloak ausgelöst werden. Dazu zählen die Überprüfung von PIN-Codes, die Abfrage des PIN-Code-Status oder die Verwaltung von PTAN-Geräten (z. B. `POST/user-api/clientID/pins`).
- **Admin-Operations:** Diese Endpunkte sind für administrative Rollen vorgesehen. Sie ermöglichen es, die Sperrung von Benutzern (*pin-lock/unlock*) zu verwalten, PIN-Codes zurückzusetzen oder TAN-Typen im Kundenkontext zu initialisieren.
- **System-Operations:** Spezielle Schnittstellen für die prozessgesteuerte Kommunikation zwischen den Systemen A/B und dem Dienst, wie beispielsweise die Durchführung von TAN-Anfragen (*tan-challenge*) oder die Überprüfung von TAN-Codes (*tan-check*).

5.4.2 Ressourcen-Orientierung und Pfad-Struktur

Die Pfadstruktur wurde so entwickelt, dass sie mehrere Clients unterstützt, indem die `clientID` als fester Bestandteil des Pfads integriert wurde. Dieser Ansatz ermöglicht eine strikte Trennung der Datenkontexte bereits auf URL-Ebene:

`/api-typ/clientID/users/userID/...`

5.4.3 Beispielhafte Endpunkt-Definition

Die folgende Tabelle enthält Beispiele für wichtige API-Endpunkte:

Methode	Pfad	Beschreibung
POST	/user-api/{clientID}/users/{userID}/pin-checks	Validierung einer Benutzer-PIN
POST	/system-api/{clientID}/users/{userID}/tan-check	Verifizierung eines TAN-Codes
POST	/admin-api/{clientID}/users/{userID}/pin-unlock	Entsperren einer PIN
PATCH	/user-api/{clientID}/users/{userID}/devices	Löschen eines Geräts

Tabelle 5.1: Auszug aus der API-Endpoint-Spezifikation

5.5 Ablauf der Kommunikation

In diesem Abschnitt wird der dynamische Ablauf der Interaktion zwischen den beteiligten Komponenten beschrieben. Der Schwerpunkt liegt auf dem Prozess der Benutzerauthentifizierung, bei dem der *pin-tan-service* als Verifizierungsmechanismus für Keycloak fungiert.

Wie in Abbildung 5.3 dargestellt, ist der *pin-tan-service* bewusst von der Verwaltung der JWT getrennt. Seine einzige Aufgabe besteht darin, die Richtigkeit der eingegebenen PIN zu überprüfen (*PIN-Check*). Die Logik zur Erstellung und Verwaltung der Tokens verbleibt vollständig auf der Keycloak-Seite.

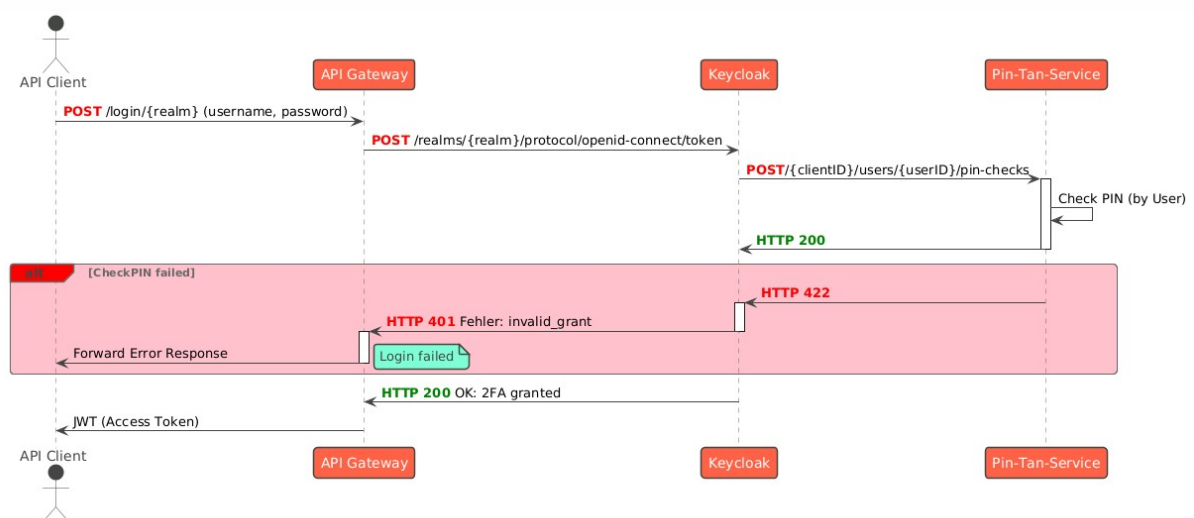


Abbildung 5.3: UML-Sequenzdiagramm des Authentifizierungsablaufs

5.5.1 Sequenzdiagramm: Authentifizierungsprozess

1. **Anmeldeanfrage:** Der API-Client sendet eine POST-Anfrage mit den Anmelde-daten (`username`, `password`) an das API-Gateway.
2. **Weiterleitung an Keycloak:** Das Gateway leitet die Anfrage an den Token-Endpunkt von Keycloak weiter (`protocol/openid-connect/token`).
3. **Validierung durch den Service:** Da Keycloak die PINs nicht selbst verwaltet, nutzt es den entwickelten Custom User Storage Provider. Dieser sendet eine Anfrage an den *pin-tan-service* (`POST/pin-checks`), um die PIN im Kontext des jeweiligen Mandanten (`clientID`) zu prüfen.
4. **Ergebnisrückmeldung:**
 - Bei Erfolg antwortet der Service mit einem `HTTP-200-OK`. Keycloak schließt die Authentifizierung ab und stellt dem Client ein JWT (Access Token) aus.
 - Schlägt die Prüfung fehl, liefert der Service einen Fehlercode (z. B. `HTTP-422`). Keycloak verarbeitet dies intern und sendet eine `HTTP-401-Unauthorized` Fehlermeldung (*invalid_grant*) zurück an den Client.

5.5.2 Trennung der Verantwortlichkeiten

Durch diese Architektur wird eine klare Trennung (*Separation of Concerns*) erreicht:

- **Keycloak:** Übernimmt die Rolle des Identity Providers, verwaltet die Realms, Rollen und die Ausstellung der JWTs.
- **pin-tan-service:** Fungiert als fachlicher Validierungsdienst, der lediglich den Status der PIN/TAN-Daten kennt, ohne Wissen über die übergeordnete OAuth2-Sitzung zu benötigen.

5.6 Teststrategie: Behavior-Driven Development (BDD)

Um eine hohe Softwarequalität und eine enge Abstimmung der Anforderungen zu gewährleisten, wird in dieser Arbeit der Ansatz des BDD verwendet. Im Gegensatz zu klassischen Testmethoden ermöglicht BDD die parallele Durchführung der Implementierung und der Entwicklung von Testszenarien. Die Spezifikationen werden dabei in natürlicher Sprache (Gherkin⁵) in Form von Textszenarien (Given/When/Then) formuliert, die mit dem Testcode verknüpft werden und als ausführbare Dokumentation dienen. Dies gewährleistet, dass die Entwicklung von *pin-tan-service* immer den festgelegten Akzeptanzkriterien entspricht.

⁵ <https://cucumber.io/docs/gherkin/>

6 Implementierung des pin-tan-service

6.1 API-Design und automatisierte Artefakt-Generierung

Die Entwicklung folgt einem strikten API-First-Ansatz, um eine saubere Trennung (Entkoppelung) zwischen der Schnittstellendefinition und der tatsächlichen Implementierung zu gewährleisten. Dieser Prozess wird vollständig durch das Build-Skript automatisiert.

- **Logische Segmentierung:** Die Endpunkte wurden bewusst in drei funktionale Bereiche unterteilt: `admin`, `system` und `user`. Dies ermöglicht eine klare Trennung der Verantwortlichkeiten und Sicherheitsrichtlinien.
- **Synchronisation mit der Bibliothek:** Die Request- und Response-Objekte in der YAML-Datei wurden so modelliert, dass sie den entsprechenden Klassen der *pitase lib* stetig angeglichen werden können.
- **Entkopplung statt 1-zu-1-Mapping:** Auf ein direktes 1-zu-1-Mapping wurde verzichtet, um die öffentliche API stabil zu halten und interne Details der Bibliothek zu verbergen. Dies erlaubt es, die API-Struktur (z. B. Bezeichner in JSON) unabhängig von den Java-Klassen der *pitase lib* zu definieren und zu erweitern.
- **Automatisierte Generierung:** Im Projekt *pin-tan-service-api* werden im Zuge des Build-Prozesses (via Maven) die Java-Interface-Klassen direkt aus der OpenAPI-Spezifikation generiert. Dies geschieht nicht manuell, sondern ist fest in den Build-Zyklus integriert.
- **Bereitstellung als API-Artefakt:** Das Ergebnis dieses Prozesses ist ein eigenständiges API-Artefakt (eine JAR-Bibliothek), welches als „Single Source of Truth“ dient – also als der einzige verbindliche Referenzpunkt, der die gesamte Schnittstellenstruktur für alle beteiligten Systeme festlegt. Andere Projekte, sowohl der Server (*pin-tan-service*) als auch potenzielle Clients, können dieses Artefakt als Abhängigkeit (*Dependency*) einbinden, ohne Kenntnis über die interne Entstehung der Bibliothek haben zu müssen.
- **Parallele Entwicklung:** Dieser Ansatz ermöglicht es verschiedenen Teams, parallel zu arbeiten. Sobald das API-Artefakt bereitsteht, kann die Implementierung der Geschäftslogik unabhängig von der Client-Entwicklung erfolgen.

Ein Auszug aus der OpenAPI-Spezifikation (siehe Listing 6.1) verdeutlicht die Struktur des `user-api` Endpunkts sowie die Definition des dazugehörigen `PinCheckRequest` Schemas. Insgesamt wurden im Rahmen der Schnittstellendefinition 31 Endpunkte spe-

zifiziert und implementiert. Diese unterteilen sich in 11 User-Operations zur Validierung und Statusabfrage von PIN/TAN-Daten, 15 Admin-Operations zur administrativen Sperrung, Entsperrung und Initialisierung von Benutzer-Credentials sowie 5 System-Operations zur prozessgesteuerten Generierung von PIN-Briefen und der Abwicklung von TAN-Challenges.

```
1 paths:
2   /user-api/{clientID}/users/{userID}/pin-checks:
3     post:
4       tags:
5         - user-operations
6       summary: Check a user PIN
7       operationId: checkPINByUser
8       parameters:
9         - name: auth
10           in: query
11           required: false
12           schema:
13             type: string
14             description: HMAC hash
15             # ... clientID und userID Parameter gekuerzt ...
16       requestBody:
17         content:
18           application/json:
19             schema:
20               $ref: '#/components/schemas/PinCheckRequest'
21       responses:
22         '200':
23           description: PIN OK
24         '422':
25           description: Error from pitase
26
27 components:
28   schemas:
29     PinCheckRequest:
30       type: object
31       required:
32         - pin
33       properties:
34         pin:
35           type: string
36         pinAlternative:
37           type: string
38         isHash:
39           type: boolean
```

```
default: true
```

Listing 6.1: Auszug aus der OpenAPI-Spezifikation (YAML)

6.2 Implementierung der Service-Ressourcen

Im Hauptprojekt *pin-tan-service* werden die generierten Interfaces als externe Abhängigkeit eingebunden.

- **Implementierung der Interfaces:** Für die verschiedenen Funktionsbereiche wurden drei zentrale Ressourcen-Klassen erstellt: `UserApiResource`, `AdminApiResource` und `SystemApiResource`. Diese Klassen implementieren die jeweils generierten Interfaces (z. B. `UserOperationApi`).
- **Methoden-Implementierung und Überschreibung** Ein wesentlicher Schritt der Implementierung ist das Überschreiben (`@Override`) der im Interface definierten Methoden. Ein Beispiel für eine solche Signatur aus dem generierten Interface `UserOperationApi` ist (siehe Listing 6.2):

```

1 @POST
2 @Path("/pin-checks")
3 @Consumes({ "application/json" })
4 @Produces({ "application/json" })
5 Response checkPINByUser(@PathParam("clientID") String clientID,
6                          @PathParam("userID") String userID,
7                          @Valid @NotNull PinCheckRequest
8                          pinCheckRequest,
9                          @QueryParam("auth") String auth);

```

Listing 6.2: Signatur der generierten Interface-Methode im API-Artefakt

- **Individuelle Logik:** Erst in diesen Implementierungsklassen wird die tatsächliche Logik (Aufruf der *pitase lib*, Tracing via OpenTelemetry¹, Fehlerbehandlung) programmiert, während die Struktur durch das API-Artefakt streng vorgegeben bleibt.

6.3 Funktionale Implementierung der Ressourcen

Nachdem das strukturelle Gerüst generiert war, wurde die Geschäftslogik in den jeweiligen Ressourcen-Klassen implementiert. Am Beispiel der Methode `checkPINByUser` (siehe 6.3) lässt sich der interne Ablauf verdeutlichen:

1. **Validierung und Tracing:** Zuerst erfolgt eine formale Prüfung des Authentifizierungsparameters durch die Hilfsklasse `Check.checkAuth(auth)`. Parallel dazu wird ein *OpenTelemetry-Span* gestartet. Dies ist essenziell für das Monitoring in

¹ <https://opentelemetry.io/>

einer verteilten Architektur, da spezifische Metadaten wie `clientID`, `userID` und `pinType` als Attribute im Span gespeichert werden.

2. **Daten-Transformation (Mapping):** Da die generierten API-Objekte (aus dem Modul *pin-tan-service-api*) und die Business-Objekte der Bibliothek technisch getrennt sind, übersetzt die Klasse `Translator` den eingehenden `PinCheckRequest` in das interne Format der *pitase lib*. Hierbei werden auch zusätzliche Felder wie der `Role.USER` gesetzt.
3. **Verarbeitung durch die Bibliothek:** Die vorbereitete Anfrage wird mittels `pitase.process(req)` an die Kernlogik delegiert.
4. **Antwort-Handling:** Nach der Verarbeitung wird die Antwort evaluiert. Bei einer erfolgreichen Prüfung (`hasNoError`) wird eine `PitaseResponse` erstellt und als HTTP 200 (OK) zurückgegeben. Im Fehlerfall oder bei Ausnahmen übernimmt ein zentraler `responseBuilder` die Erstellung einer standardisierten Fehlermeldung (z.B. HTTP 422), wobei der Tracing-Kontext für die Fehleranalyse erhalten bleibt.

```
1 @Override
2 public Response checkPINByUser(final String clientID,
3                               final String userID,
4                               @Valid @NotNull PinCheckRequest request,
5                               final String auth) {
6     // Authentifizierungs-Check des 'auth' Parameters
7     boolean isAuth = Check.checkAuth(auth);
8     LOG.info("checkPIN(){}: {}", isAuth ? " with auth" : "", request);
9     // Start eines OpenTelemetry-Spans für verteiltes Tracing
10    Span span = tracer.spanBuilder("UserOperationsApi.checkPINByUser").
11    startSpan();
12    span.setAttribute(AttributeKey.stringKey("auth"), String.valueOf(
13    isAuth));
14    span.setAttribute(AttributeKey.stringKey("clientID"), clientID);
15    span.setAttribute(AttributeKey.stringKey("userID"), userID);
16    span.setAttribute(AttributeKey.stringKey("pinType"), request.
17    getPinType());
18    span.setAttribute(AttributeKey.stringKey("request"), request.
19    getClass().getName());
20    // Ausführung der Logik im Scope des aktuellen Spans
21    try (Scope scope = span.makeCurrent()) {
22        // Initialisierung des internen Request-Objekts der pitase-
23        Bibliothek
24        de.xcom.pitase.communication.PinCheckRequest req =
25            new de.xcom.pitase.communication.PinCheckRequest(pinType(
26            request));
27        // Mapping der API-Daten auf das interne Bibliotheksformat via
28        Translator
```

```

22     Translator.userRequestToPitase(clientID, userID, request, req);
23     req.setPin(request.getPin());
24     req.setPinAlternative(request.getPinAlternative());
25     req.setStoreSuccessTime(request.getStoreSuccessTime());
26     req.setIsHash(request.getIsHash());
27     req.setRole(Role.USER); // Festlegung der Sicherheitsrolle
28     pitase.process(req);
29     // Auswertung der Antwort von pitase
30     de.xcom.pitase.communication.Response pitaseResponse = req.
getResponse();
31     // Rückgabe bei erfolgreicher Validierung (HTTP 200)
32     if (Check.hasNoError(pitaseResponse)) {
33         PitaseResponse response = new PitaseResponse();
34         return Response.ok(URI.create(""))
35             .entity(response)
36             .build();
37     }
38     // Fehlerbehandlung: Umwandlung von Bibliotheksfehlern in
standardisierte API-Antworten
39     return responseBuilder.generateErrorResponse(pitaseResponse,
span);
40 } catch (Throwable t) {
41     return responseBuilder.generateErrorResponse(t, span);
42 } finally {
43     span.end();
44 }
45 }

```

Listing 6.3: Implementierung der PIN-Verifizierungslogik in der Ressourcen-Klasse

Analog zu der oben beschriebenen Methode `checkPINByUser` wurden auch die übrigen Endpunkte in der Klasse `UserApiResource` umgesetzt.

6.4 Qualitätssicherung

Die Tests wurden unter Verwendung des Cucumber-Frameworks in Kombination mit JUnit 5 implementiert. Da Entwicklung und Testen parallel erfolgten, konnten Abweichungen von der API-Spezifikation bereits in frühen Phasen erkannt werden. Ein weiterer Vorteil dieses Ansatzes war die Möglichkeit, Tests unter Verwendung von Schnittstellenimitationen durchzuführen, noch bevor die vollständige Integration mit der *pitase lib* erfolgte. Die Testergebnisse wurden über das X-Ray-Plugin automatisch an das Aufgabensystem JIRA übermittelt, wodurch die Nachverfolgbarkeit der Anforderungen gewährleistet war.

6.4.1 Unit-Tests mit JUnit

Die isolierte Überprüfung der Geschäftslogik erfolgt mithilfe des JUnit 5 (Jupiter) Frameworks. Durch den Einsatz von JUnit 5 profitiert das Test-Setup von modernen Features wie der strengen Typisierung von Assertions und einer verbesserten Test-Struktur.

Methodik und Test-Setup:

- **Mocking-Framework:** Um den *pin-tan-service* ohne eine aktive Datenbank oder echte Banken-Infrastruktur zu testen, wird die *pitase lib* mittels eines Mocks (*pitaseMock*) simuliert. Dies erlaubt die gezielte Injektion von Fehlerszenarien.
- **Lifecycle-Management:** Unter Verwendung von JUnit 5 Annotationen wird vor jedem Testlauf ein definierter Ausgangszustand hergestellt.
- **Assertions:** Neben Standard-Checks werden spezifische JUnit 5 Assertions genutzt, um die korrekte Struktur der Antwort-Objekte (DTOs) sicherzustellen.

Der folgende Auszug in Listing 6.4 illustriert die Umsetzung dieser Teststrategie. Dabei werden nicht nur erfolgreiche Validierungen (Happy Path) geprüft, sondern auch gezielt Fehlerszenarien der *pitase lib* simuliert, um die Robustheit der Fehler-Transformation und die Korrektheit des Monitorings (OpenTelemetry) zu verifizieren.

```
1 class CheckPinByUserTest extends UserApiResourceTestBase {
2
3     @Test
4     void checkPIN_OK_withAuth() {
5         Response response = sut.checkPINByUser(CLIENT_ID, USER_ID,
6         createValidRequest(), "auth");
7
8         TestBase.assertHttpStatus(response, 200);
9         // Verifikation der Datenübergabe an pitase
10        assertTrue(TestBase.toString(pitaseMock.lastRequest()).contains(
11        "ClientId=1"));
12    }
13
14    @Test
15    void checkPIN_wrongClientId() {
16        Response response = sut.checkPINByUser("wrongClient", USER_ID,
17        createValidRequest(), null);
18
19        TestBase.assertHttpStatus(response, 422);
20        TestBase.assertCaughtException(response, "java.lang.
21        IllegalArgumentException", "clientId does not match");
22    }
23
24    @Test
```

```

21     void checkPin_faultyPitaseErrorResponse() {
22         pitaseMock.nextError(null); // Verursacht absichtlich eine
Exception
23
24         Response response = sut.checkPINByUser(CLIENT_ID, USER_ID,
createValidRequest(), null);
25
26         // Prüfung, ob die Exception im OpenTelemetry-Span erfasst wurde
27         String spanContent = TestBase.checkAndGetSpanContent("
UserOperationsApi.checkPINByUser", spanExporter);
28         assertTrue(spanContent.contains("NullPointerException"), "
Exception fehlt im Monitoring");
29         assertTrue(spanContent.contains("clientId=" + CLIENT_ID), "
Metadaten fehlen im Span");
30     }
31 }

```

Listing 6.4: JUnit 5 Testbeispiele für die PIN-Verifizierung

6.4.2 Integrationstests mit Cucumber

Während Unit-Tests die Korrektheit einzelner Methoden prüfen, dienen die Integrations-tests der Validierung des gesamten Systemverhaltens aus der Sicht eines Endanwenders oder eines externen Dienstes. Hierzu wird das Framework Cucumber eingesetzt, welches Anforderungen im Gherkin-Format (Given-When-Then) spezifiziert.

Das Listing 6.5 zeigt einen Auszug aus der Datei `pin_check.feature`. Hier wird simuliert, wie ein Administrator eine PIN generiert und der Benutzer diese anschließend verifiziert. Ein wichtiger Teil der Tests sind die Fehlerfälle. Ein Beispiel ist der Fehler `ERROR_PIN_NOT_USABLE`. Dieser Test prüft, ob das System richtig reagiert, wenn ein Nutzer eine neue PIN anfordert, aber noch die alte benutzen möchte. So werden logische Fehler im Ablauf verhindert.

```

1 @pin-check
2 Feature: PIN Check
3     As a banking service user
4     I want to check my PIN status and validate my PIN
5
6     Background:
7         Given I have a valid admin ID
8         And I have a client ID "1"
9         And I have a user ID
10        And I initialize new user with json file "
initializeUserRequest_admin"
11
12 @NBA-270

```

```

13 Scenario Outline: PIN check with different scenarios - <scenario>
14   Given <precondition>
15   And <setup>
16   When <action>
17   Then <result>
18
19   Examples: Successful PIN check
20 | scenario                | precondition                | action
21 | Pin check successful | new pin generated by admin | check pin via JSON
22 | Pin state is VALID   | new pin generated by admin | check pin state by
23 | user | PIN state is "VALID" |
24
25   Examples: Failed PIN checks
26 | scenario                | setup                | action
27 | Wrong User ID          | generate pin by admin | check with wrong userID
28 | response: "userID" mismatch |
29 | PIN not usable yet | user requested new PIN | check pin via JSON
30 | error: "ERROR_PIN_NOT_USABLE" |

```

Listing 6.5: Cucumber-Szenarien für die PIN-Prüfung

Durch die Kombination aus JUnit 5 Unit-Tests und Cucumber Integrationstests wird eine Testabdeckung erreicht, die sowohl die technische Stabilität als auch die korrekte Umsetzung der bankfachlichen Geschäftsprozesse garantiert.

6.5 Bewertung der Zielerreichung

Nach der Implementierung und erfolgreichen Testphase (siehe Tabelle 6.1) erfolgt eine abschließende Bewertung der zu Beginn der Arbeit festgelegten funktionalen (F) und nicht-funktionalen (NF) Anforderungen. In der folgenden Tabelle wird detailliert dargestellt, inwieweit die in dieser Arbeit gestellten Aufgaben umgesetzt wurden.

Zusammenfassend lässt sich feststellen, dass der entwickelte *pin-tan-service* alle gestellten Anforderungen erfüllt. Die Kombination aus einer klaren Trennung der Architekturebenen, einer hohen Testabdeckung und der Standardisierung von Sicherheitsverfahren bildet eine solide Grundlage für die Einführung in der Produktionsumgebung des Bankensektors. Damit wurde die entscheidende Grundlage für die Ablösung weiterer Altsysteme geschaffen.

ID	Anforderung	Status	Technische Umsetzung / Validierung
Funktionale Anforderungen (F)			
F1	Validierung von Credentials	Erfüllt	Prüfung gegen Pitase-DB; vollständige Übernahme der Geschäftslogik der <i>pitase_lib</i> .
F2	Integration in Keycloak	Erfüllt	Implementierung als <i>Custom User Storage Provider</i> zur Nutzung in OAuth2-Prozessen.
F3	Bereitstellung REST-API	Erfüllt	Konsequenter <i>Contract-First</i> -Ansatz mit OpenAPI zur Systementkopplung.
F4	Client-Bibliotheken (SDKs)	Erfüllt	Automatisierte Generierung von JAR-Artefakten zur Anbindung von System A und B.
F5	Bereinigung alter Logik	Erfüllt	Entfernung veralteter Prozesse (z. B. TAN-Listen) bei Erhalt der Kernfunktionen.
Nicht-funktionale Anforderungen (NF)			
NF1	Entkopplung und Isolation	Erfüllt	Eigenständiger Dienst; Kommunikation über definierte Schnittstellen statt direkter Kopplung.
NF2	Horizontale Skalierbarkeit	Erfüllt	<i>Stateless</i> -Architektur ermöglicht problemlose Replikation der Service-Instanzen.
NF3	CI/CD und Deployment	Erfüllt	Automatisierte Builds und Bereitstellung (Pipeline-Integration).
NF4	Moderner Tech-Stack	Erfüllt	Einsatz von Java 17 und Quarkus gemäß den <i>Approved Technology</i> Standards.
NF5	Containerisierung	Erfüllt	Bereitstellung als Docker-Image für Cloud-Umgebungen (Kubernetes-ready).
NF6	Testbarkeit	Erfüllt	Hohe Qualitätssicherung durch JUnit 5 und <i>Cucumber</i> -Tests (X-Ray kompatibel).
NF7	Sicherheit (OAuth 2.0)	Erfüllt	Ablösung proprietärer Logins durch Keycloak und standardisierte OpenID Connect Flows.

Tabelle 6.1: Detaillierter Soll-Ist-Vergleich der Anforderungen

7 Fazit und Ausblick

Im Rahmen dieser Bachelorarbeit wurde die Frage der Integration einer proprietären, veralteten Bibliothek zur Verwaltung von PIN/TAN in Banksysteme untersucht. Es wurde festgestellt, dass die direkte Integration der Bibliothek *pitase lib* zu einer hohen Abhängigkeit vom Technologie-Stack (Java) führte, die Release-Zyklen bei der Fehlerbehebung erschwerte und die Testmöglichkeiten für Anwendungen, die diese Bibliothek nutzen, einschränkte. Darüber hinaus entsprach die bestehende Authentifizierungslogik nicht mehr den modernen Sicherheitsanforderungen für verteilte Systeme.

Zur Lösung dieser Aufgaben wurde eine moderne Microservice-Architektur auf Basis von Java 17 und dem Quarkus-Framework entwickelt und implementiert. Die geschäftskritische PIN/TAN-Logik wurde in einen separaten Container-Service (*pin-tan-service*) ausgelagert und ist nun über eine standardisierte REST-Schnittstelle (OpenAPI) verfügbar. Die Integration in die zentrale Identitätsmanagement-Lösung Keycloak erfolgte durch die Entwicklung von *Custom User Storage Providers*, wodurch eine sichere Authentifizierung über OAuth 2.0 und OpenID Connect realisiert werden konnte.

Die Auswertung der Ergebnisse zeigte, dass die auf Microservices basierende Lösung gegenüber der bisherigen Bibliotheksintegration erhebliche Vorteile bietet. Die Kapselung der veralteten Logik in einen Microservice in Kombination mit Keycloak beseitigt erfolgreich die festgestellten Wartungs- und Integrationsprobleme. Die neue Architektur dient als zuverlässige Grundlage für die weitere Modernisierung der Bankensysteme. Darüber hinaus setzt sie einen zukunftsweisenden Standard für den Umgang mit veralteter Software im Unternehmen.

7.1 Beantwortung der Forschungsfragen

Die Beantwortung der zentralen Forschungsfrage dieser Arbeit erfolgt durch die Analyse der im Vorfeld definierten Unterfragen:

1. **Analyse der essenziellen Bestandteile (F1 und F5):** Die Analyse der veralteten Bibliothek hat gezeigt, dass die Kernlogik zur Überprüfung der Anmeldedaten nach wie vor eine zentrale Komponente darstellt. Veraltete Prozesse wie die manuelle Verwaltung von physischen TAN-Listen konnten jedoch beseitigt werden, was zu einer Vereinfachung des Systems führte. Somit konzentriert sich der neue Service ausschließlich auf die kritischen Geschäftsprozesse der Überprüfung.
2. **Integration: Microservice vs. Bibliothek:** Die Überführung der Logik in einen Microservice bietet wesentliche Vorteile:

- *Plattformunabhängigkeit der Service-Nutzung*: Während die Bibliothek auf Java beschränkt war, ermöglicht die REST-Schnittstelle (HTTP) die Nutzung durch beliebige Konsumenten (z. B. Go oder JavaScript).
 - *Wartbarkeit*: Bei Bugfixes muss nur der *pin-tan-service* aktualisiert werden. Bei einer Bibliothek müssten alle Konsumenten (System A, B) ihren Code anpassen, neu kompilieren und einen eigenen Release-Zyklus durchlaufen. Dies reduziert den Release-Aufwand erheblich.
3. **Testbarkeit und Qualitätssicherung**: Der Wechsel zum Microservice verändert den Testzyklus:
- *Entkopplung der Testphasen*: Konsumenten können isoliert gegen die OpenAPI-Spezifikation testen, was eine parallele Entwicklung ermöglicht.
 - *Optimiertes Mocking*: Ein Microservice lässt sich problemlos mocken, was die Testgeschwindigkeit erhöht und die Fehlerisolierung verbessert.
 - *Zentralisierter Testaufwand*: Die Logik muss nur einmal im Service validiert werden, statt in jedem einzelnen konsumierenden System.

7.2 Ausblick und zukünftige Entwicklung

Obwohl die technische Implementierung im Rahmen dieser Arbeit erfolgreich abgeschlossen wurde, stehen für den produktiven Einsatz weitere Phasen an. Aktuell befindet sich der *pin-tan-service* in der Phase des System Integration Testing (SIT), in der das Zusammenspiel mit der Infrastruktur geprüft wird. Im nächsten Schritt folgt das User Acceptance Testing (UAT) zur fachlichen Abnahme. Nach Abschluss dieser Tests ist die schrittweise Umstellung der Bestandssysteme (System A und B) von der Bibliotheksnutzung auf die neue REST-API geplant.

Langfristig trägt diese Modernisierung dazu bei, die Nutzung der internen Plattform zu erhöhen. Durch die Bereitstellung standardisierter Schnittstellen wird es für zukünftige Kunden und Drittsysteme deutlich einfacher, sich an die Plattform anzubinden. Dies erhöht nicht nur die Skalierbarkeit und Wartungsfreundlichkeit, sondern beschleunigt auch die Markteinführung neuer digitaler Bankprodukte. Auf diese Weise dient die im Rahmen dieser Arbeit entwickelte Architektur als skalierbares Referenzmodell für die weitere digitale Transformation des Unternehmens.

Quellenverzeichnis

- [All10] Subbu Allamaraju. *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*. Sebastopol, CA: O'Reilly Media, 2010. ISBN: 978-0-596-80168-7.
- [Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD Dissertation. University of California, Irvine, USA, 2000.
- [Gez23] S. Gezer: *How to Use GitOps with ArgoCD*, 2023, Online im Internet: <https://medium.com/trendyol-tech/how-to-use-gitops-with-argocd-1782b8493cc3>, Stand: 18.02.2026.
- [Hai21] Serhii Hainulin: *SpringBoot vs Quarkus: a Real-Life Experiment*, 2021, Online im Internet: <https://medium.com/swlh/springboot-vs-quarkus-a-real-life-experiment-be70c021634e>, Stand: 14.02.2026.
- [Har12] Dick Hardt: *The OAuth 2.0 Authorization Framework*, 2012, Online im Internet: <https://www.rfc-editor.org/rfc/rfc6749>, Stand: 20.02.2026.
- [JBS15] Michael B. Jones, John Bradley und Nat Sakimura: *JSON Web Token (JWT)*, 2015, Online im Internet: <https://www.rfc-editor.org/rfc/rfc7519>, Stand: 21.02.2026.
- [New19] Sam Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. Sebastopol, CA: O'Reilly Media, 2019. ISBN: 9781492047841.
- [New21] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. 2nd. Sebastopol, CA: O'Reilly Media, 2021. ISBN: 9781492034025.
- [Ope14] OpenID Foundation: *OpenID Connect Core 1.0*, 2014, Online im Internet: https://openid.net/specs/openid-connect-core-1_0.html, Stand: 18.02.2026.
- [Ope20] OpenAPI Initiative: *OpenAPI Specification v3.0.3*, 2020, Online im Internet: <https://spec.openapis.org/oas/v3.0.3>, Stand: 16.02.2026.
- [Ora21] Oracle Corporation: *Java 17 LTS Interactive Release Notes*, 2021, Online im Internet: <https://www.oracle.com/java/technologies/javase/17-relnote-issues.html>, Stand: 16.02.2026.
- [Qua26] Quarkus Community: *Quarkus Insights: Measuring Performance*, 2026, Online im Internet: <https://quarkus.io/guides/performance-measure>, Stand: 06.02.2026.

- [RAR13] Leonard Richardson, Mike Amundsen und Sam Ruby. *RESTful Web APIs: Services for a Changing World*. Sebastopol, CA: O'Reilly Media, 2013. ISBN: 978-1-4493-5806-8.
- [Red25] Red Hat: *Keycloak Server Administration Guide*, 2025, Online im Internet: https://www.keycloak.org/docs/latest/server_admin/index.html, Stand: 15.02.2026.
- [Red26] Red Hat: *Keycloak Server Developer Guide: User Storage SPI*, 2026, Online im Internet: https://www.keycloak.org/docs/latest/server_development/#_user_storage_spi, Stand: 14.02.2026.
- [RS17] Justin Richer und Antonio Sanso. *OAuth 2 in Action*. Manning Publications, 2017. ISBN: 9781617293276.
- [Stu16] Phil Sturgeon. *Build APIs You Won't Hate*. Leanpub, 2016.
- [Til+15] Stefan Tilkov u. a. *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*. dpunkt.verlag, 2015. ISBN: 978-3-86490-120-1.
- [Til11] Stefan Tilkov. *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*. 2., aktualisierte und erweiterte Auflage. Heidelberg: dpunkt.verlag, 2011. ISBN: 978-3-89864-732-8.
- [TS21] Stian Thorgersen und Pedro Igor Silva. *Keycloak - Identity and Access Management for Modern Applications*. Packt Publishing Limited, 2021. ISBN: 978-1-80056-470-1.
- [VB18] Blesson Varghese und Rajkumar Buyya. „Next Generation Cloud Computing: New Trends and Research Directions“. In: *Future Generation Computer Systems* 79 (2018), S. 849–861. DOI: 10.1016/j.future.2017.09.020.
- [Vel23] P. Velepucha V. und Flores. „A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges“. In: *IEEE Access* 11 (2023), S. 88339–88358. DOI: 10.1109/ACCESS.2023.3305612.

Übersicht verwendeter Hilfsmittel:

- ChatGPT (<https://chatgpt.com/>): Wurde zur Generierung von Ideen und zur präzisieren Formulierung von Sätzen auf Basis des vom Autor bereitgestellten Textes verwendet. Außerdem wurde das Tool zur Erstellung von Diagrammen und Grafiken auf Grundlage der Anweisungen des Autors sowie zur Identifikation von Grammatikfehlern im Text genutzt.
- Gemini (<https://gemini.google.com/>): Wurde für die gleichen Aufgaben wie ChatGPT eingesetzt, also zur Ideenfindung, Formulierungshilfe, Diagrammerstellung und Grammatiküberprüfung.

- DeepL Translator (<https://www.deepl.com/translator>): Genutzt für Übersetzungen und zur sprachlichen Überarbeitung von Textpassagen.



Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit in allen Teilen selbstständig angefertigt und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt habe, und dass die Arbeit in gleicher oder ähnlicher Form in noch keiner anderen Prüfung vorgelegen hat. Mir ist bewusst, dass ich Autor/in der vorliegenden Arbeit bin und volle Verantwortung für den Text trage. ☒

Ich erkläre, dass ich wörtlich oder sinngemäß aus anderen Werken – dazu gehören auch Internetquellen – übernommene Inhalte als solche kenntlich gemacht und die entsprechenden Quellen angegeben habe. ☒

Mir ist bewusst, dass meine Arbeit auf Plagiate überprüft werden kann. Mir ist bekannt, dass es sich bei der Abgabe eines Plagiats um ein schweres akademisches Fehlverhalten handelt und dass Täuschungen nach der für mich gültigen Prüfungsordnung geahndet werden. ☒

Zusätzlich versichere ich, dass ich auf künstlicher Intelligenz (KI) basierende Werkzeuge nur in Absprache mit den Prüfern verwendet habe. Dabei stand meine eigene geistige Leistung im Vordergrund, und ich habe jederzeit den Prozess steuernd bearbeitet. ☒

Diese Werkzeuge habe ich im Quellenverzeichnis in der Rubrik „Übersicht verwendeter Hilfsmittel“ mit ihrem Produktnamen und einer Übersicht des im Rahmen dieser Prüfungs-/Studienarbeit genutzten Funktionsumfangs unter Angabe der Textstelle in der Arbeit vollständig aufgeführt. ☒

Ich versichere, dass ich keine KI-basierten Tools verwendet habe, deren Nutzung die Prüfer explizit schriftlich ausgeschlossen haben. Ich bin mir bewusst, dass die Verwendung von Texten oder anderen Inhalten und Produkten, die durch KI-basierte Tools generiert wurden, keine Garantie für deren Qualität darstellt. ☒

Ich verantworte die Übernahme jeglicher von mir verwendeter maschinell generierter Passagen vollumfänglich selbst und trage die Verantwortung für eventuell durch die KI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate. ☒

Zwickau, 03.04.2026

Ort, Datum

Unterschrift