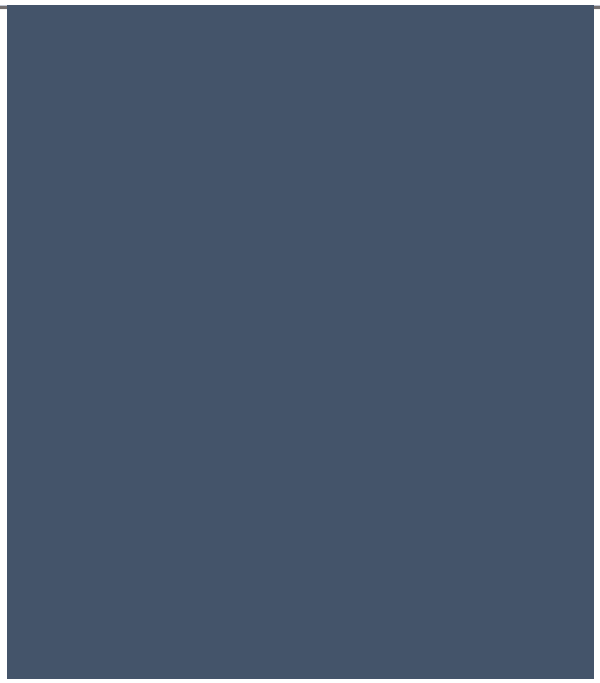


Mit Microservices

Jasmin, Aurora, Elmar, Ruben

Table of Contents

Einleitung Verteilte Systeme....2
Unser Projekt....3
Vorgehen (Projektplanung)....3
Planung und Aufgabenteilung....3
Projektphasen....4
Werkzeuge & Organisation....4
Inhalt Überblick verteiltes System....5
Grafischer Darstellung....6
Beschreibung Systemkomponente....7
Testprotokoll....13
1. User-Service Testprotokoll....13
1.1 UserTest.java....13
1.2 UserRepositoryTest.java....14
1.3 UserControllerTest.java....14
2. Grade-Service Testprotokoll....14
2.1 GradeTest.java....14
2.2 GradeRepositoryTest.java....15
2.3 GradeControllerTest.java....15
2.4 Postman-Tests....16
3. Class-Service Testprotokoll....16
3.1 ClassEntityTest.java....16
3.2 ClassServiceTest.java....17
3.3 ClassControllerTest.java....17
4. Gesamtfazit....17
Rückblick Gruppenmitglieder....18
Aurora Rückblick....18
Ruben Rückblick....18
Elmar Rückblick....19
Jasmin Rückblick....19

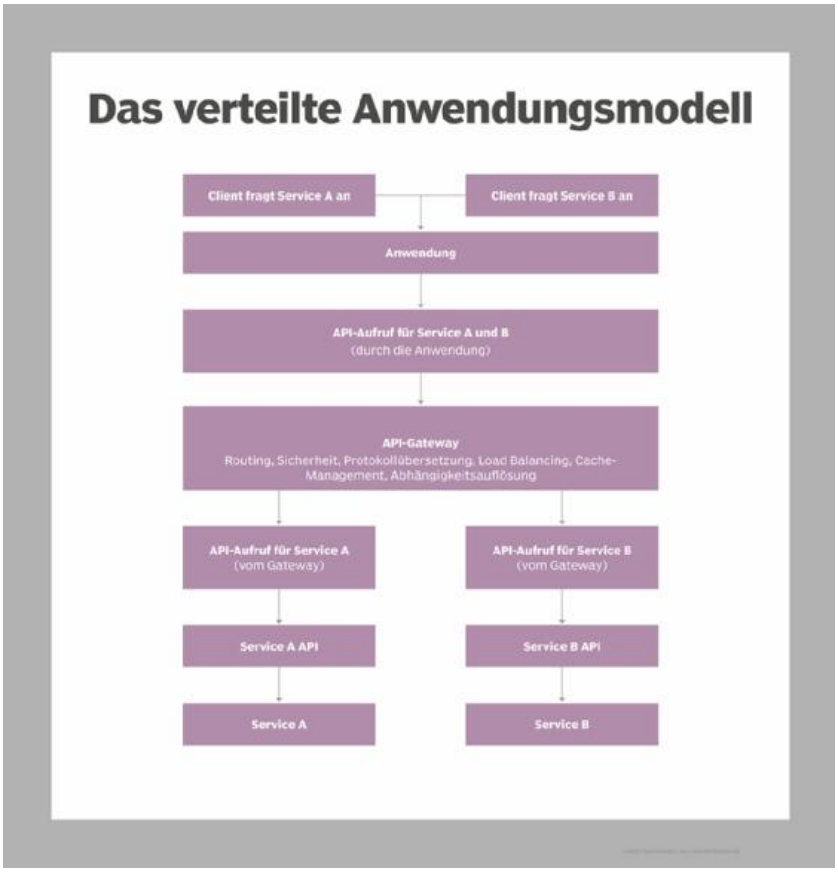
Einleitung Verteilte Systeme

Verteilte Systeme sind Computersysteme, bei denen mehrere eigenständige Rechner oder Knoten über ein Netzwerk miteinander verbunden sind und zusammenarbeiten, um ein gemeinsames Ziel zu erreichen. Diese Systeme erscheinen für den Nutzer häufig wie ein einzelnes System, obwohl die Verarbeitung, Speicherung und Kommunikation über mehrere Standorte verteilt sind.

Das Ziel verteilter Systeme ist es, Ressourcen effizient zu nutzen, Ausfallsicherheit zu gewährleisten und die Leistung zu steigern. Typische Beispiele sind Cloud-Dienste, Online-Banking, verteilte Datenbanken oder Content-Delivery-Netzwerke.

Charakteristisch für verteilte Systeme sind Aspekte wie **Kollaboration mehrerer Rechner**, **Kommunikation über Netzwerke**, **Fehler- und Ausfallsicherheit** sowie **Transparenz**, sodass Nutzer oft nicht merken, dass sie mit einem verteilten System interagieren.

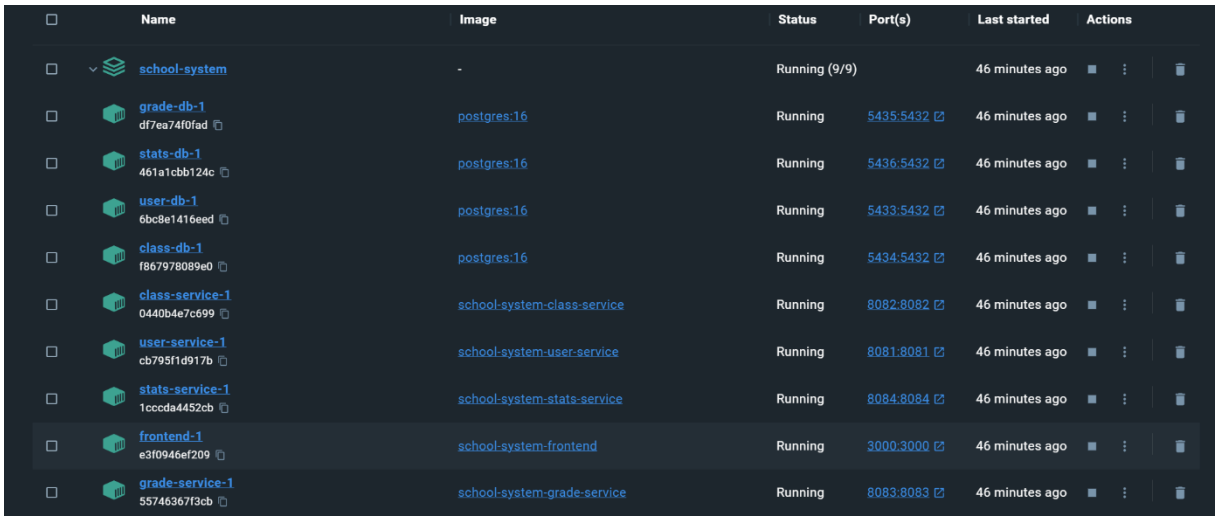
Kurz gesagt: Verteilte Systeme verbinden die Leistung vieler Rechner, um Aufgaben schneller, zuverlässiger und skalierbarer zu erledigen, als es ein einzelner Computer könnte



Quelle: [Verteilte Anwenfungssysteme]

Unser Projekt

Wir haben ein Notensystem entwickelt, das aus mehreren eigenständigen Komponenten besteht: vier Backend-Services, ein Frontend und vier separate Datenbanken. Jede Komponente ist unabhängig und kann separat betrieben, angepasst oder erweitert werden. Dieses modulare Design ermöglicht eine hohe Flexibilität und erleichtert Wartung, Skalierung sowie die Implementierung neuer Funktionen, ohne das gesamte System zu beeinflussen.



Name	Image	Status	Port(s)	Last started	Actions
school-system	-	Running (9/9)		46 minutes ago	
grade-db-1 df7ea74f0fad	postgres:16	Running	5435:5432	46 minutes ago	
stats-db-1 461a1cbb124c	postgres:16	Running	5436:5432	46 minutes ago	
user-db-1 6bc8e1416eed	postgres:16	Running	5433:5432	46 minutes ago	
class-db-1 f867978089e0	postgres:16	Running	5434:5432	46 minutes ago	
class-service-1 0440b4e7c699	school-system-class-service	Running	8082:8082	46 minutes ago	
user-service-1 cb795f1d917b	school-system-user-service	Running	8081:8081	46 minutes ago	
stats-service-1 1cccda4452cb	school-system-stats-service	Running	8084:8084	46 minutes ago	
frontend-1 e3f0946ef209	school-system-frontend	Running	3000:3000	46 minutes ago	
grade-service-1 55746367f3cb	school-system-grade-service	Running	8083:8083	46 minutes ago	

Vorgehen (Projektplanung)

Zu Beginn des Projekts wurde das Ziel definiert, ein **verteiltes Notenverwaltungssystem** zu entwickeln, das auf dem Prinzip der **Microservices-Architektur** basiert.
Jede Person im Team war für eine eigenständige Systemkomponente verantwortlich – inklusive **Backend**, **Datenbankanbindung** und **Frontend-Integration**.
Dadurch konnten alle Teammitglieder unabhängig voneinander arbeiten, was eine parallele Entwicklung und geringere Abhängigkeiten ermöglichte.

Planung und Aufgabenteilung

Nach einer kurzen Konzeptionsphase wurde festgelegt, welche Services benötigt werden und wie diese miteinander interagieren sollen.
Die Hauptkomponenten wurden wie folgt aufgeteilt:

Teammitglied	Verantwortlicher Service	Verantwortlicher Service
Ruben Schneebeli	User-Service	Verwaltung von Benutzerdaten (Lehrer, Schüler, Admins)
Elmar Kessler	Class-Service	Verwaltung von Klassen und deren Zuordnung zu Lehrern und Schülern
Jasmin Jeyakumar	Grade-Service	Verwaltung von Noten und deren Verknüpfung mit Schülern und Klassen
Aurora Gjemaj	Stats-Service	Berechnung und Bereitstellung von Statistiken aus den Notendaten

Projektphasen

1. *Konzeptionsphase*
   Definition der Services, API-Endpunkte und der Beziehungen zwischen den Komponenten (User, Class, Grade, Stats).
   Erstellung des Architekturdiagramms und Auswahl der Technologien.

2. *Implementierungsphase*
   Entwicklung der einzelnen Microservices mit **Java Spring Boot** und **PostgreSQL**.
   Aufbau der Docker-Container für Backend, Datenbanken und das React-Frontend.

3. *Integrationsphase*
   Verbindung der Services über REST-Schnittstellen, Test der Kommunikation und Validierung der IDs zwischen den Systemen.

4. *Testphase*
   Erstellung von **Unit-Tests** für Entitäten, Repositories und Controller.
   Sicherstellung der korrekten HTTP-Statuscodes, Datenvalidierung und Fehlerbehandlung.

5. *Abschlussphase / Präsentationsvorbereitung*
   Zusammenführung aller Komponenten, gemeinsame Tests im Docker-Netzwerk und Vorbereitung der Endpräsentation.

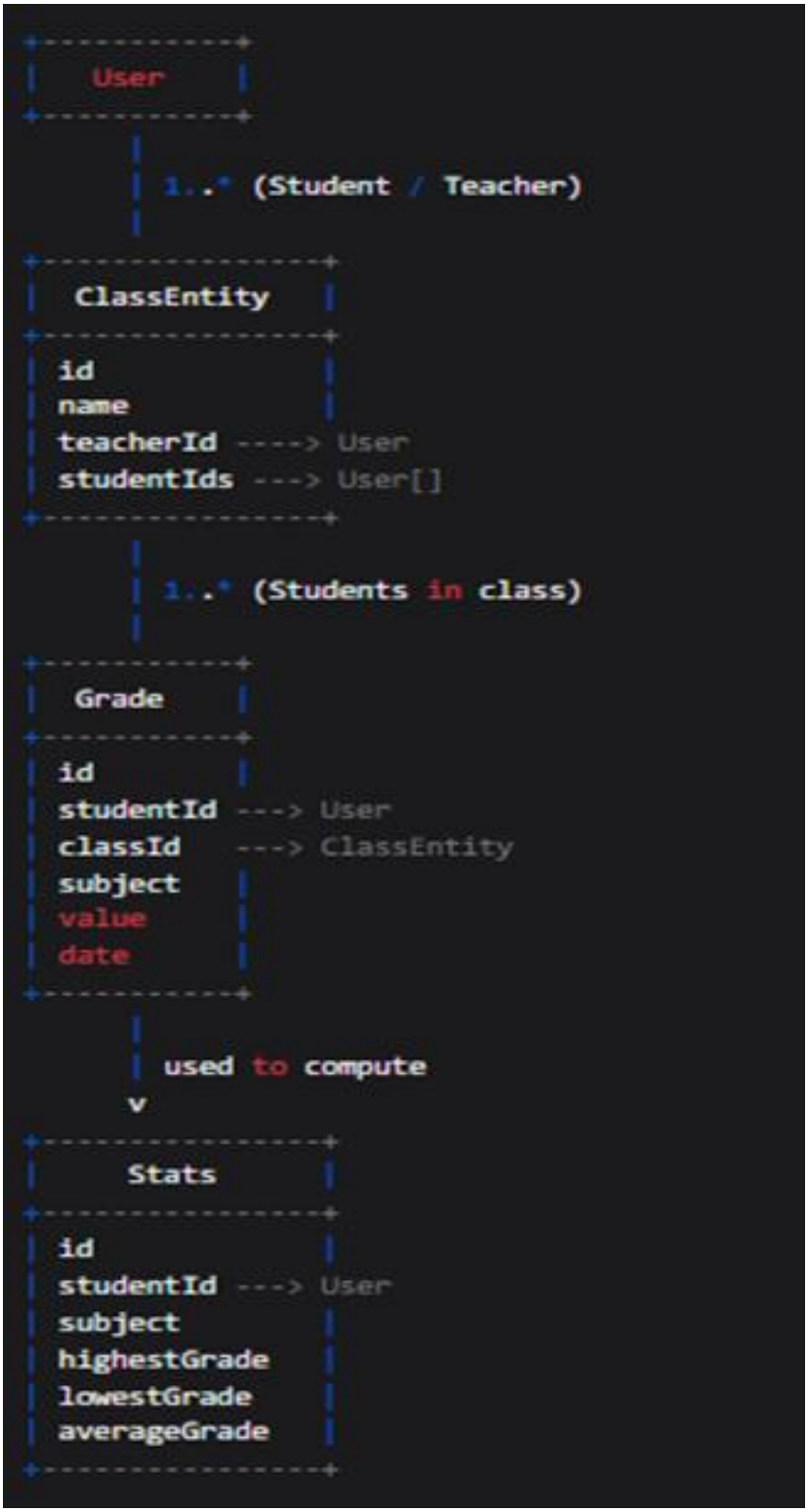
Werkzeuge & Organisation

- **Versionsverwaltung:** Git / GitHub
- **Entwicklungsumgebung:** IntelliJ IDEA, Visual Studio Code
- **Kommunikation & Aufgabenplanung:** Microsoft Teams
- **Containerisierung & Deployment:** Docker Compose

Inhalt Überblick verteiltes System

- **User-Service und Class-Service**
  - Der **User-Service** repräsentiert Benutzer, die entweder **Studenten** oder **Lehrer** sein können.
  - Der **Class-Service** enthält Klassen (**ClassEntity**), die jeweils einem Lehrer (über teacherId) und mehreren Schülern (über studentIds) zugeordnet sind.
  - **Beziehung**: Ein Lehrer kann eine oder mehrere Klassen unterrichten, und ein Schüler kann in einer oder mehreren Klassen eingeschrieben sein.
- **Class-Service und Grade-Service**
  - Der **Grade-Service** speichert Noten (**Grade**), die einem Schüler (über studentId) für eine bestimmte Klasse (über classId) und ein Fach zugeordnet sind.
  - **Beziehung**: Eine Klasse kann mehrere Noten enthalten, die den Schülern in dieser Klasse zugeordnet sind.
- **Grade-Service und Stats-Service**
  - Der **Stats-Service** berechnet Statistiken (**Stats**) basierend auf den Noten eines Schülers in einem bestimmten Fach.
  - Die Statistiken umfassen die höchste Note (highestGrade), die niedrigste Note (lowestGrade) und die Durchschnittsnote (averageGrade).
  - **Beziehung**: Die Noten aus dem **Grade-Service** werden verwendet, um die Statistiken im **Stats-Service** zu berechnen.
- **User-Service und Stats-Service**
  - Die Statistiken im **Stats-Service** sind einem bestimmten Schüler (über studentId) zugeordnet.
  - **Beziehung**: Die Leistungen eines Schülers werden in den Statistiken zusammengefasst und analysiert.

Grafischer Darstellung



Beschreibung Systemkomponente

User-Service = Ruben

1. Allgemeine Beschreibung

Der **User-Service** ist verantwortlich für die Verwaltung aller Benutzer des Systems.
Er speichert, verwaltet und liefert Benutzerdaten und Rolleninformationen.

Zweck:

- Verwaltung von Benutzern (Studenten, Lehrer, Admins)
- Bereitstellung von Benutzerdaten für andere Services (Grade-Service, Class-Service)
- Authentifizierung und Autorisierung innerhalb der Applikation

2. Funktionale Beschreibung

- **Benutzerverwaltung**
    - Anlegen, Bearbeiten und Löschen von Benutzern
    - Speicherung von Benutzerinformationen: Name, E-Mail, Passwort, Rolle
- **Rollenverwaltung**
    - Rollen: ADMIN, TEACHER, STUDENT
    - Berechtigungen werden anhand der Rolle festgelegt
- **Integration mit anderen Services**
    - Class-Service: Zuweisung von Lehrern und Schülern zu Klassen
    - Grade-Service: Referenzierung von Schülern für Noten
    - Stats-Service: Referenzierung von Schülern für Statistiken

3. Entität: User

Attribut	Typ	Beschreibung
id	Long	Eindeutige Benutzer-ID
name	String	Name des Benutzers
email	String	E-Mail-Adresse (einzigartig)
password	String	Passwort (verschlüsselt)
role	Enum	Rolle: ADMIN, TEACHER, STUDENT

4. Schnittstellen

- **REST-API-Endpunkte (Beispiele)**
    - POST /users – Benutzer anlegen
    - GET /users/{id} – Benutzerinformationen abrufen
    - PUT /users/{id} – Benutzerinformationen aktualisieren
    - DELETE /users/{id} – Benutzer löschen
- **Inter-Service-Kommunikation**
    - Bereitstellung von Benutzerinformationen für Class-Service und Grade-Service
    - Unterstützung bei der Validierung von Schüler- und Lehrer-IDs

5. Verantwortlichkeiten

- **Datenhaltung:** Speichern, Aktualisieren und Löschen von Benutzerinformationen
- **Authentifizierung/Autorisierung:** Verwaltung von Benutzerzugriffen und Rollen
- **Datenvalidierung:** Sicherstellen, dass E-Mail-Adressen eindeutig sind
- **Bereitstellung für andere Services:** Lieferung korrekter Benutzerinformationen für Noten und Klassen

Class-Service = Elmar

1. Allgemeine Beschreibung

Der **Class-Service** verwaltet Schulklassen und deren Zuordnung zu Lehrern und Schülern.
Er sorgt für die Organisation der Klassenstruktur im System und stellt die Verbindung zwischen Schülern (User) und Lehrern her.

Zweck:

- Verwaltung von Klassen (ClassEntity)
- Zuweisung von Lehrern und Schülern zu Klassen
- Bereitstellung von Klassendaten für andere Services, z. B. Grade-Service

2. Funktionale Beschreibung

- **Erstellung von Klassen**
    - Anlegen von neuen Klassen mit eindeutiger ID und Namen
    - Zuweisung eines Lehrers (teacherId)
- **Verwaltung von Schülern**
    - Hinzufügen und Entfernen von Schülern (studentIds)
    - Abfragen der Schüler einer Klasse

- **Integration mit anderen Services**
    - User-Service: Validierung der teacherId und studentIds
    - Grade-Service: Referenz auf classId

3. Entität: ClassEntity

Attribut	Typ	Beschreibung
id	Long	Eindeutige Identifikation der Klasse
name	String	Name der Klasse (z. B. „3A“, „2B“)
teacherId	Long	Referenz auf den Lehrer (User)
studentIds	List<Long>	Liste der Schüler-IDs (User)

4. Schnittstellen

- **REST-API-Endpunkte (Beispiele)**
    - POST /classes – Neue Klasse anlegen
    - GET /classes/{id} – Klasse abrufen
    - PUT /classes/{id}/students – Schüler zur Klasse hinzufügen oder entfernen
- **Inter-Service-Kommunikation**
    - **User-Service:** Validierung von Lehrer- und Schüler-IDs
    - **Grade-Service:** Referenz für Noteneintragungen

5. Verantwortlichkeiten

- Verwaltung von Klassen und deren Mitglieder
- Sicherstellen der Konsistenz von Lehrer- und Schülerzuordnungen
- Bereitstellung von Klasseninformationen für andere Services

Grade-Service= Jasmin

1. Allgemeine Beschreibung

Der **Grade-Service** ist verantwortlich für die Verwaltung von Schülernoten innerhalb des Systems. Er speichert, verwaltet und liefert Noteninformationen, die von Lehrern für Schüler eingetragen werden.

Zweck:

- Aufnahme und Verwaltung von Noten (Grade)
- Bereitstellung von Notendaten für Statistikberechnungen (Stats-Service)
- Sicherstellung der Konsistenz zwischen Schülern (User) und Klassen (ClassEntity)

2. Funktionale Beschreibung

- **Erstellung von Noten**
    - Lehrer können Noten für Schüler in einer Klasse hinzufügen
    - Daten: studentId, classId, subject, value, date
- **Abfrage von Noten**
    - Bereitstellung von Noten für einzelne Schüler oder Klassen
    - Schnittstelle für Stats-Service zur Berechnung von highestGrade, lowestGrade, averageGrade
- **Integration mit anderen Services**
    - User-Service: Validierung der studentId
    - Class-Service: Validierung der classId

3. Entität: Grade

Attribut	Typ	Beschreibung
id	Long	Eindeutige Identifikation der Note
studentId	Long	Referenz auf den Schüler (User)
classId	Long	Referenz auf die Klasse (ClassEntity)
subject	String	Fach der Note
value	Double	Notenwert
date	LocalDate	Datum der Note

4. Schnittstellen

- **REST-API-Endpunkte (Beispiele)**
    - POST /grades – Neue Note hinzufügen
    - GET /grades/student/{id} – Noten eines Schülers abrufen
    - GET /grades/class/{id} – Noten einer Klasse abrufen
- **Inter-Service-Kommunikation**
    - **User-Service:** Validierung der studentId
    - **Class-Service:** Validierung der classId
    - **Stats-Service:** Übergabe der Noten zur Berechnung der Statistiken

5. Verantwortlichkeiten

- **Datenhaltung:** Speichern, Aktualisieren und Löschen von Noten
- **Datenvalidierung:** Sicherstellen, dass Schüler und Klassen existieren
- **Bereitstellung für Statistik:** Aggregation der Daten für Stats-Service

Stats-Service

1. Allgemeine Beschreibung

Der **Stats-Service** aggregiert Noteninformationen aus dem Grade-Service und berechnet Statistiken für Schüler.
Er dient der Auswertung und Darstellung von Leistungskennzahlen.

Zweck:

- Berechnung von highestGrade, lowestGrade und averageGrade pro Schüler und Fach
- Bereitstellung von statistischen Daten für Lehrkräfte, Schüler und ggf. Administration

2. Funktionale Beschreibung

- **Aggregation von Noten**
    - Berechnet Statistiken aus allen Noten eines Schülers für jedes Fach
    - Aktualisierung der Statistik bei neuen oder geänderten Noten
- **Bereitstellung von Statistiken**
    - Endpunkte für einzelne Schüler oder Fächer
    - Schnittstelle für Frontend oder andere Services
- **Integration mit anderen Services**
    - Grade-Service: Quelle der Noten
    - User-Service: Validierung der studentId

3. Entität: Stats

Attribut	Typ	Beschreibung
id	Long	Eindeutige Identifikation der Statistik
studentId	Long	Referenz auf den Schüler (User)
subject	String	Fach
highestGrade	Double	Höchste Note
lowestGrade	Double	Niedrigste Note
averageGrade	Double	Durchschnittsnote

4. Schnittstellen

- **REST-API-Endpunkte (Beispiele)**
    - GET /stats/student/{id} – Statistiken eines Schülers abrufen
    - GET /stats/student/{id}/subject/{subject} – Statistik für ein bestimmtes Fach
- **Inter-Service-Kommunikation**
    - **Grade-Service:** Abruf von Noten zur Berechnung
    - **User-Service:** Validierung von studentId

5. Verantwortlichkeiten

- Berechnung und Speicherung von Schülerstatistiken
- Sicherstellung, dass Statistiken aktuell und konsistent mit Noten sind
- Bereitstellung von Daten für Reporting und Analyse

Testprotokoll

Projekt: Notenverwaltung

Umfang: Dieses Protokoll dokumentiert die Unit-Tests der Microservices **user-service**, **grade-service** und **class-service**. Ziel ist es, die Abdeckung und Funktionalität der Entitäten, Repositories und Controller zu überprüfen. Besonderes Augenmerk liegt auf Fehlerbehandlung und erwarteten HTTP-Statuscodes.

1. User-Service Testprotokoll

Beschreibung: Der **user-service** verwaltet Benutzerdaten. Die Tests decken die **User Entität**, das **UserRepository** und den **UserController** ab.

1.1 UserTest.java

Zweck: Überprüfung der grundlegenden Funktionalität der User-Entität **Abdeckung:**

- Standardkonstruktor: Instanziierung ohne Argumente
- Parametrisierter Konstruktor: Korrekte Initialisierung aller Felder (id, name, email, password, role)
- Getter/Setter: Korrekte Rückgabe und Aktualisierung von Feldern
- Role Enum: Korrekte Definition und Namenskonvention **Ergebnis:** Alle Methoden der User Entität funktionieren erwartungsgemäss

1.2 UserRepositoryTest.java

Zweck: Überprüfung der Datenzugriffsschicht mit H2 In-Memory-Datenbank

Testfall	Beschreibung	Ergebnis
testSaveUser()	Speichert neuen Benutzer, prüft ID und Daten	Bestanden
testFindById()	Abrufen existierender und nicht existierender Benutzer	Bestanden
testFindAllUsers()	Alle Benutzer abrufen	Bestanden
testFindByRole()	Benutzer nach Rolle filtern	Bestanden
testUpdateUser()	Benutzer aktualisieren	Bestanden
testDeleteUser()	Benutzer löschen	Bestanden

Ergebnis: CRUD-Operationen und benutzerdefinierte Abfragen funktionieren zuverlässig

1.3 UserControllerTest.java

Zweck: Überprüfung der REST-API-Endpunkte mit MockMvc **Abdeckung:**

- GET /api/users → 200 OK
- GET /api/users/{id} → 200 OK / 404 Not Found
- GET /api/users/role/{role} → 200 OK / 400 Bad Request
- POST /api/users → 200 OK
- PUT /api/users/{id} → 200 OK / 404 Not Found
- DELETE /api/users/{id} → 204 No Content / 404 Not Found

Ergebnis: Endpunkte reagieren korrekt auf Anfragen, Fehler werden robust behandelt

2. Grade-Service Testprotokoll

Beschreibung: Der **grade-service** verwaltet Notendaten. Die Tests decken **Grade Entität**, **GradeRepository** und **GradeController** ab. Zusätzlich wurden alle Endpunkte mit **Postman** getestet und alle Anfragen waren erfolgreich.

2.1 GradeTest.java

- Standard- und parametrisierter Konstruktor
- Getter/Setter, equals(), hashCode(), toString() **Ergebnis:** Methoden funktionieren erwartungsgemäss

2.2 GradeRepositoryTest.java

Testfall	Beschreibung	Ergebnis
testSaveGrade()	Note speichern	Bestanden
testFindById()	Existierende und nicht existierende Note	Bestanden
testFindAllGrades()	Alle Noten abrufen	Bestanden
testFindByStudentId()	Noten für Schüler abrufen	Bestanden
testFindByClassId()	Noten für Klasse abrufen	Bestanden
testFindBySubject()	Noten für Fach abrufen	Bestanden
testUpdateGrade()	Note aktualisieren	Bestanden
testDeleteGrade()	Note löschen	Bestanden

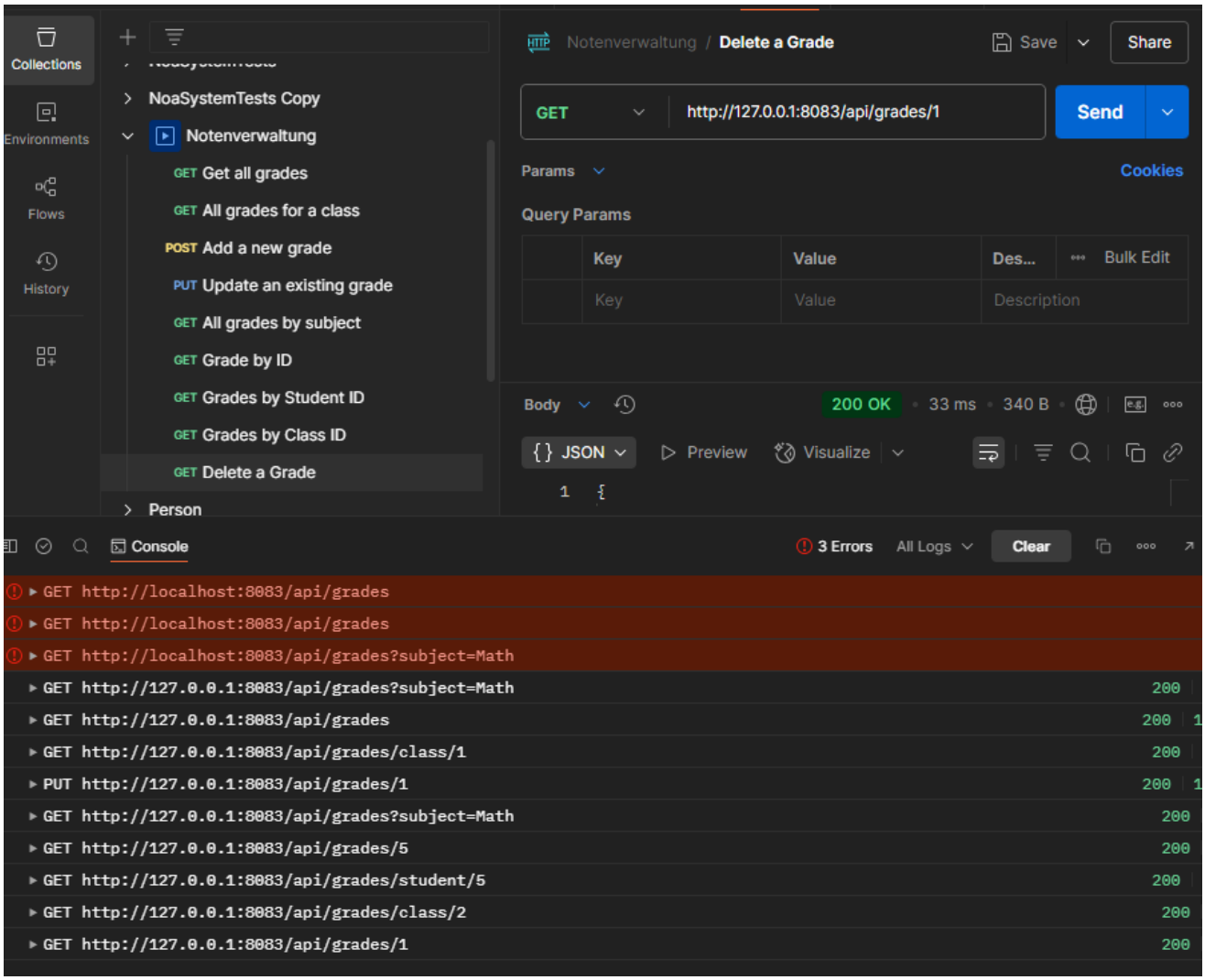
2.3 GradeControllerTest.java

- GET /api/grades → 200 OK
- GET /api/grades/{id} → 200 OK / 404 Not Found
- GET /api/grades/student/{studentId} → 200 OK
- GET /api/grades/class/{classId} → 200 OK
- POST /api/grades → 201 Created
- PUT /api/grades/{id} → 200 OK / 404 Not Found
- DELETE /api/grades/{id} → 204 No Content / 404 Not Found

Ergebnis: Endpunkte funktionieren korrekt, Fehlerfälle werden robust gehandhabt

2.4 Postman-Tests

Zusätzlich zu den Unit-Tests wurden alle Endpunkte des **Grade-Service** mit **Postman** getestet. Alle Anfragen waren erfolgreich, wie im folgenden Bild zu sehen ist:



3. Class-Service Testprotokoll

Beschreibung: Der **class-service** verwaltet Klassendaten. Die Tests decken **ClassEntity**, **ClassService** und **ClassController** ab.

3.1 ClassEntityTest.java

- Getter/Setter für id, name, teacherId, studentIds **Ergebnis:** Funktioniert korrekt

3.2 ClassServiceTest.java

Testfall	Beschreibung	Ergebnis
testGetAllClasses()	Alle Klassen abrufen	Bestanden
testGetClassByIdFound / NotFound	Klasse nach ID abrufen	Bestanden
testCreateClass()	Klasse erstellen	Bestanden
testUpdateClassSuccess / NotFound	Klasse aktualisieren	Bestanden
testDeleteClass()	Klasse löschen	Bestanden
testAddStudentToClass	Schüler hinzufügen	Bestanden
testRemoveStudentFromClass	Schüler entfernen	Bestanden
testChangeTeacher	Lehrer ändern	Bestanden

3.3 ClassControllerTest.java

- GET /api/classes → 200 OK
- GET /api/classes/{id} → 200 OK / 404 Not Found
- POST /api/classes → 200 OK
- PUT /api/classes/{id} → 200 OK / 404 Not Found
- DELETE /api/classes/{id} → 200 OK
- POST /api/classes/{classId}/students/{studentId} → 200 OK / 404 Not Found
- DELETE /api/classes/{classId}/students/{studentId} → 200 OK / 404 Not Found
- PUT /api/classes/{classId}/teacher/{teacherId} → 200 OK / 404 Not Found

Ergebnis: Endpunkte reagieren korrekt, Fehlerbehandlung über RuntimeException → HTTP 404

4. Gesamtfazit

Die Unit-Tests der Microservices **user-service**, **grade-service** und **class-service**:

- Bieten **umfassende Abdeckung** der Entitäten, Repositories und Controller
- Prüfen **CRUD-Operationen**, Filterung, Geschäftslogik und Fehlerbehandlung
- Stellen sicher, dass **API-Endpunkte die korrekten Daten und HTTP-Statuscodes** zurückgeben
- Gewährleisten die **Robustheit und Zuverlässigkeit** der Microservices

Zusätzlich wurden alle Endpunkte des **Grade-Service** mit **Postman** getestet, und alle Anfragen waren erfolgreich, wie im oben gezeigten Bild zu sehen ist.

Rückblick Gruppenmitglieder

Aurora Rückblick

Bei dem Projekt ging es für mich vor allem darum, mich mit der Stats Entity auseinanderzusetzen. Ich habe dann eigene Backends und Datenbanken aufgebaut – das war komplett neu für mich, aber wirklich interessant. Mit Microservices zu arbeiten hat mir gezeigt, wie man Services getrennt hält, aber trotzdem alles zusammen funktioniert.

Docker habe ich auch intensiv genutzt. Ich habe verstanden, wie man Services containerisiert und wie man Abhängigkeiten zwischen den einzelnen Komponenten sauber managt. Das hat mir richtig geholfen zu sehen, wie moderne Softwareentwicklung in verteilten Systemen abläuft.

Teamarbeit war manchmal ein bisschen holprig. Nicht jeder war immer gleich schnell fertig, und die Aufgaben waren nicht immer gleichmässig verteilt. Trotzdem haben wir das Projekt am Ende geschafft, und ich habe dabei gelernt, wie wichtig klare Absprachen und Organisation im Team sind.

Am Ende habe ich also nicht nur technisch etwas dazugelernt, sondern auch praktische Erfahrungen in der Zusammenarbeit im Team gesammelt.

Ruben Rückblick

In diesem Projekt konnte ich viele neue Erfahrungen sammeln und mein Wissen erheblich erweitern. Besonders spannend war die Entwicklung von unabhängig laufenden Backends und Datenbanken, da dies für mich ein neues Arbeitsfeld war. Durch die Arbeit mit Microservices konnte ich das Prinzip der modularen, servicebasierten Architektur besser verstehen und praktisch anwenden.

Darüber hinaus konnte ich mein Wissen über Docker vertiefen, insbesondere im Hinblick auf die Containerisierung von Services und die Verwaltung von Abhängigkeiten zwischen verschiedenen Komponenten. Diese technischen Erfahrungen haben mir gezeigt, wie moderne Softwareentwicklung in verteilten Systemen funktioniert und welche Vorteile eine Microservice-Architektur bietet.

Die Gruppenarbeit gestaltete sich leider als herausfordernd. Es zeigte sich, dass die Aufgabenverteilung nicht immer gleichmässig war: Einige Teammitglieder mussten mehr Arbeit übernehmen, während andere ihren Part erst sehr spät fertigstellten. Trotz dieser Schwierigkeiten konnte das Projekt erfolgreich abgeschlossen werden, und ich habe wertvolle Erfahrungen in der Zusammenarbeit in einem Team gewonnen.

Insgesamt hat mir das Projekt nicht nur technische Kompetenzen vermittelt, sondern auch gezeigt, wie wichtig klare Kommunikation und Organisation innerhalb eines Teams sind.

Elmar Rückblick

Bei dem Projekt ging es für mich vor allem darum, mich mit der Class Entity auseinanderzusetzen. Ich habe gelernt, wie man Klassen verwaltet, Lehrer und Schüler korrekt zuordnet und die Beziehungen zwischen den Entitäten abbildet. Dabei habe ich eigene Backends und Datenbanken aufgebaut, was für mich neu war und mir einen guten Einblick in die Struktur von Microservices gegeben hat.

Die Arbeit mit Microservices hat mir gezeigt, wie man Services unabhängig hält, sie aber trotzdem miteinander kommunizieren lässt. Ich habe auch intensiv Docker genutzt und verstanden, wie man Services containerisiert und die Abhängigkeiten sauber organisiert. So konnte ich sehen, wie verteilte Systeme in der Praxis funktionieren.

Die Teamarbeit verlief teilweise etwas holprig. Manche Aufgaben waren ungleich verteilt und nicht jeder konnte immer gleich schnell liefern. Trotzdem haben wir das Projekt erfolgreich abgeschlossen, und ich habe dabei erfahren, wie wichtig klare Absprachen und eine gute Organisation im Team sind.

Insgesamt habe ich nicht nur technische Kenntnisse im Umgang mit Klassen, Services und Datenbanken gewonnen, sondern auch wertvolle Erfahrungen in der Zusammenarbeit im Team gesammelt.

Jasmin Rückblick

Während des Projekts konnte ich viele wertvolle Erfahrungen sammeln, sowohl auf technischer als auch auf organisatorischer Ebene. Besonders spannend war die Arbeit mit Microservices, da ich hier viel über die Struktur unabhängiger Services, deren Kommunikation über Schnittstellen und die korrekte Dokumentation der einzelnen Komponenten lernen konnte. Themen wie Datenmapping, Beziehungen zwischen Entitäten und die Verbindung zu anderen Services haben mir gezeigt, wie wichtig eine saubere Planung und Dokumentation für die Umsetzung komplexer Systeme ist.

Während des Projekts habe ich zudem erneut erkannt, wie entscheidend es ist, frühzeitig mit der Arbeit zu beginnen. Gerade bei komplexen Projekten, wie unserem Schulprojekt, zeigt sich schnell, dass Zeitdruck sowohl die Umsetzung als auch die Koordination im Team erheblich erschwert.

Ein weiterer wichtiger Punkt war die Bedeutung klarer Absprachen im Voraus. Es ist essenziell, von Anfang an festzulegen, wer welche Aufgaben übernimmt, welche Schnittstellen zwischen den Teilprojekten existieren und wie die Kommunikation im Team funktioniert. Ohne diese Abstimmung entsteht schnell Unklarheit, was zu Verzögerungen führen kann.

Darüber hinaus habe ich gelernt, dass die aktive Mitarbeit aller Teammitglieder entscheidend für den Erfolg eines Projekts ist. Nur wenn jeder seinen Beitrag zuverlässig leistet, kann ein komplexes Projekt als Ganzes funktionieren.

Insgesamt hat das Projekt gezeigt, dass Teamarbeit, Planung, Dokumentation, klare Schnittstellen und ein frühzeitiger Projektstart entscheidend sind, um ein komplexes Projekt erfolgreich abzuschliessen. Diese Erfahrungen werde ich in zukünftigen Projekten definitiv berücksichtigen.