Übungsklausur für
Programmieren und Software Engineering

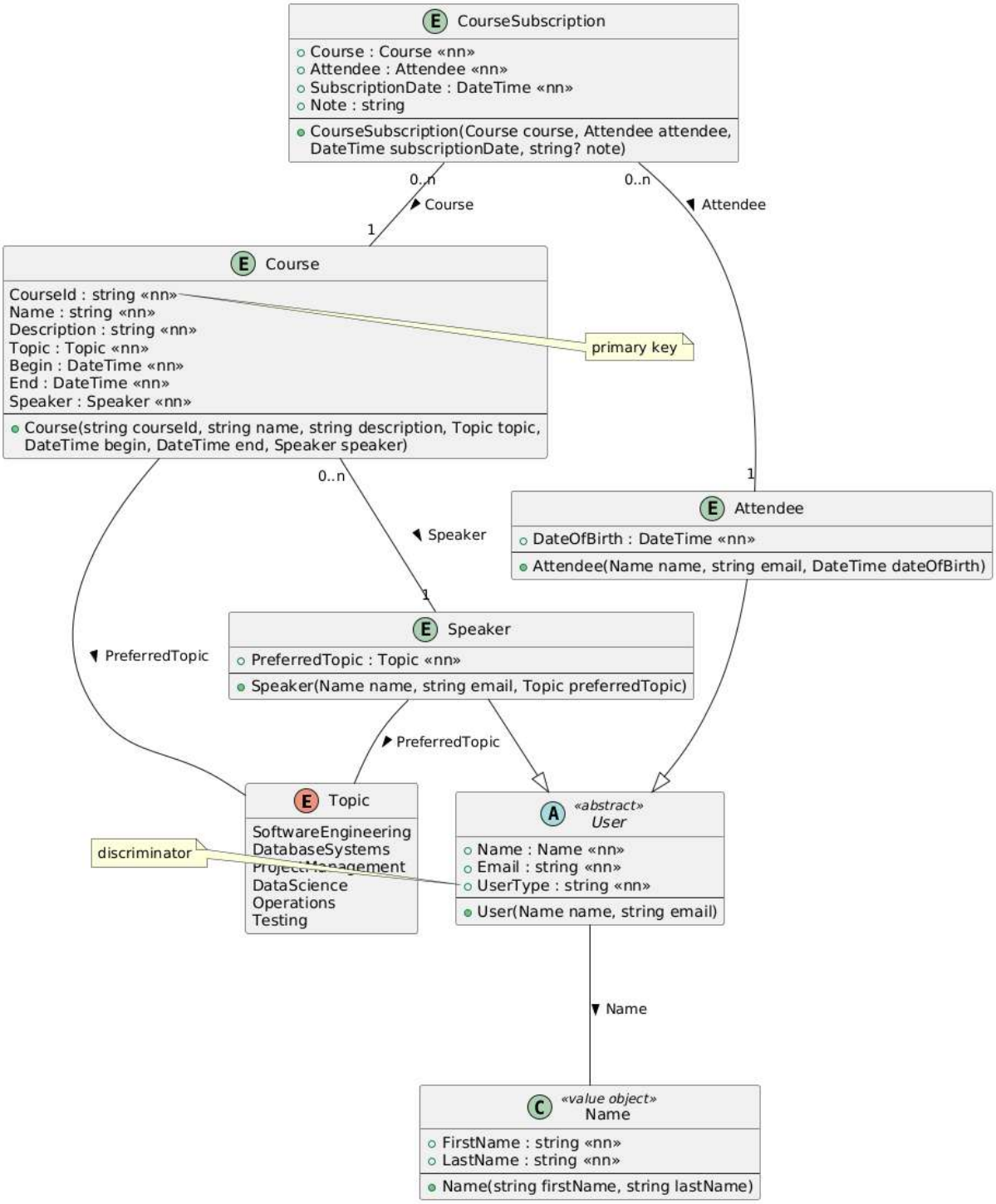
für den Aufbaulehrgang für Informatik – Tag (SFKZ 8167)
für das Kolleg für Informatik – Tag (SFKZ 8242)

Teilaufgabe 1: Erstellen von EF Core Modelklassen

Implementierung einer Kursverwaltung

Ein Lerninstitut bietet Kurse zu verschiedenen technischen Themen an. Eine Software soll die Verwaltung der Anmeldungen vereinfachen. Ein Kurs wird von einem Referenten (Speaker) gehalten. Teilnehmer:innen (Attendees) können sich vorab zum Kurs anmelden.

Modell



Beschreibung des Modelles

Felder, die nicht null sein dürfen, sind mit *<<nn>>* gekennzeichnet. In der Klasse *User* soll das Feld *UserType* automatisch vom OR Mapper mit dem Typ (*Attendee* oder *Speaker*) befüllt werden, deswegen ist es auch nicht im Konstruktor enthalten. Der Kurs (*Course*) soll einen benutzerdefinierten Primärschlüssel haben (z. B. *C100*). Bei allen anderen Klassen müssen Sie – wenn erforderlich – den Primärschlüssel selbst wählen. Beachten Sie, dass value objects in der Datenbanktabelle integriert gespeichert werden, um die Abfrageperformance zu verbessern.

Arbeitsauftrag

Erstellung der Modelklassen

Im Projekt *SPG_Fachtheorie.Aufgabe1* befinden sich im Ordner *Model* leere Klassendefinitionen. Bilden Sie jede Klasse gemäß dem UML Diagramm ab, sodass EF Core diese persistieren kann. Beachten Sie folgendes:

- Wählen Sie selbst notwendige Primary keys.
- Definieren Sie Stringfelder mit vernünftigen Maximallängen (z. B. 255 Zeichen für Namen, etc.).
- *Name* ist ein *value object*. Stellen Sie durch Ihre Definition sicher, dass kein Mapping dieser Klasse in eine eigene Datenbanktabelle durchgeführt wird.
- Das Feld *Name* in *Course* muss unique sein. Stellen Sie dies durch eine geeignete Konfiguration sicher.
- Legen Sie Konstruktoren mit allen erforderlichen Feldern an. Erstellen Sie die für EF Core notwendigen default Konstruktoren als *protected*.
- Definieren Sie das Discriminator Feld *UserType* korrekt in Ihrer Konfiguration.
- Das Feld *Topic* in *Course bzw. PreferredTopic* in *Course* ist ein enum Feld. Speichern Sie dieses Feld als String in der Datenbank. Stellen Sie dies durch geeignete Konfiguraton sicher.
- Implementieren Sie die Vererbung korrekt, sodass eine (1) Tabelle *User* entsteht.
- Legen Sie die erforderlichen DB Sets im Datenbankcontext an.

Verfassen von Tests

Im Projekt *SPG_Fachtheorie.Aufgabe1.Test* ist in *Aufgabe1Test.cs* der
Test *CreateDatabaseTest* vorgegeben. Er muss erfolgreich durchlaufen und die Datenbank erzeugen. Sie können die erzeugte Datenbank in *C:/Scratch/Aufgabe1_Test/Debug/net8.0/damages.db* in SQLite Studio öffnen.

Implementieren Sie folgende Tests selbst, indem Sie die minimalen Daten in die (leere) Datenbank schreiben. Leeren Sie immer vor dem *Assert* die nachverfolgten Objekte mittels *db.ChangeTracker.Clear()*.

- Der Test *AddSpeakerSuccessTest* beweist, dass Sie einen Referenten (*Speaker*) in die Datenbank einfügen können. Prüfen Sie im Assert, ob ein Primärschlüssel generiert wurde.
- Der Test *AddSubscriptionSuccessTest* beweist, dass Sie einen Kurs samt Anmeldung (*Subscription*) anlegen können. Legen Sie hierfür einen *Speaker*, einen *Attendee* und einen Kurs (*Course*) an. Stellen Sie im Assert sicher, dass für das gespeicherte Objekt vom Typ *CourseSubscription* einen Primärschlussel generiert wurde.
- Der Test *DiscriminatorHasCorrectTypeSuccessTest* beweist, dass der OR Mapper das Feld *UserType* in *User* korrekt befüllt. Legen Sie dafür einen Datensatz vom Typ *Speaker* an und prüfen Sie das Feld.

Bewertung (28P, 36.8% der Gesamtpunkte)

Jedes der folgenden Kriterien wird mit 1 Punkt bewertet.

1. Die Stringfelder verwenden sinnvolle Längenbegrenzungen.
2. Alle Felder, die mit „*nn*" gekennzeichnet wurden, dürfen nicht null sein.
3. Die Klasse *User* beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
4. Die Klasse *User* besitzt ein korrekt konfiguriertes value object *Name*.

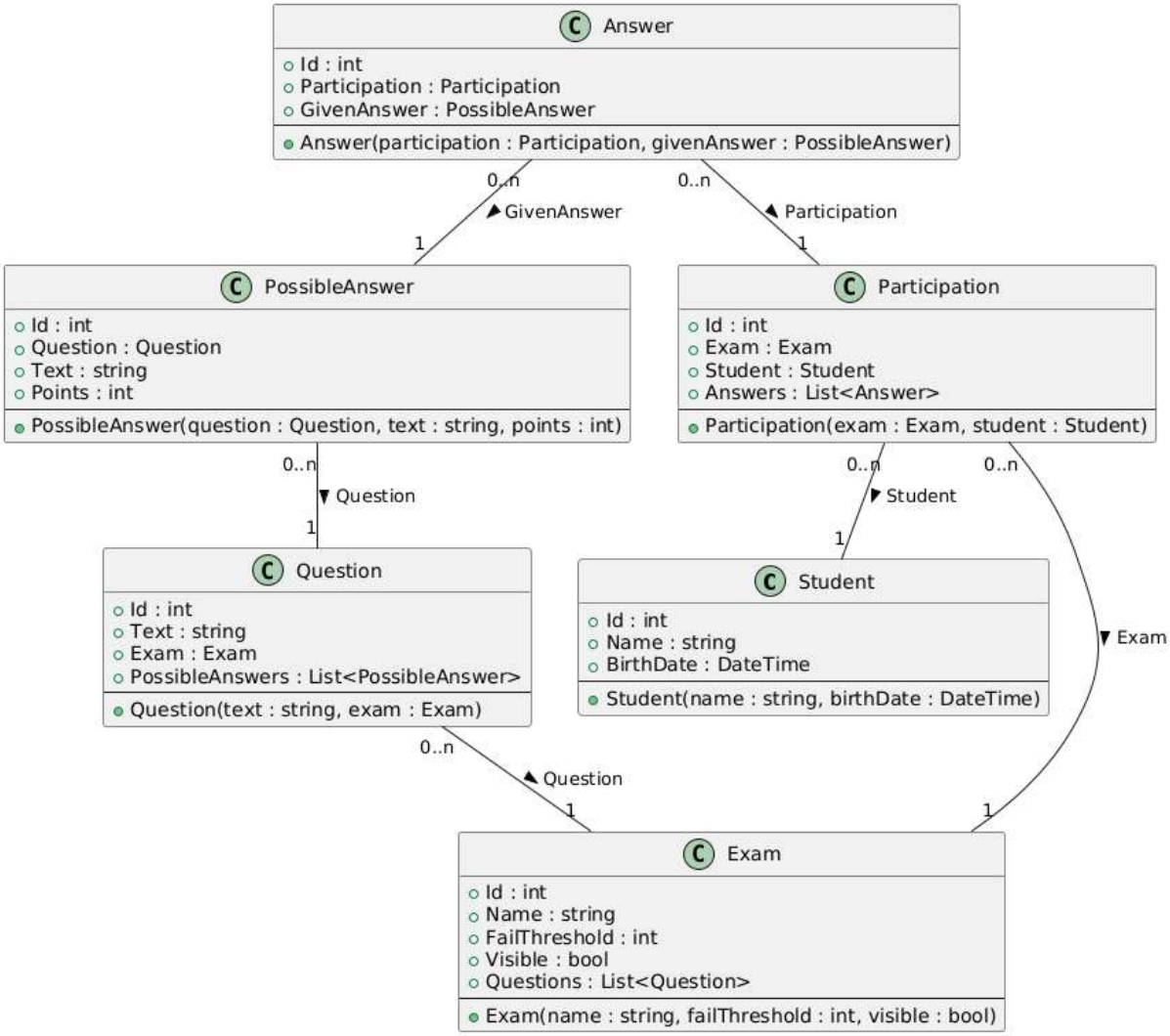
5. Die Klasse *User* besitzt ein korrekt konfiguriertes discriminator Feld *UserType*.
6. Die Klasse *User* ist als abstrakt definiert.
7. Die Klasse *User* wird korrekt im DbContext registriert.
8. Die Klasse *Attendee* beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
9. Die Klasse *Attendee* erbt korrekt von der Klasse *User*.
10. Die Klasse *Attendee* wird korrekt im DbContext registriert.
11. Die Klasse *Speaker* beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
12. Die Klasse *Speaker* erbt korrekt von der Klasse *User*.
13. Die Enumeration *PreferredTopic* in *Speaker* wird als String in der Datenbank gespeichert.
14. Die Klasse *Speaker* wird korrekt im DbContext registriert.
15. Die Klasse *Course* beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
16. Der Primary Key der Klasse *Course* ist korrekt definiert.
17. Der das Property *Name* der Klasse Course ist als unique definiert.
18. Die Enumeration *Topic* in *Course* wird als String in der Datenbank gespeichert.
19. Die Klasse *Course* wird korrekt im DbContext registriert.
20. Die Klasse *CourseSubscription* beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
21. Das Property *Note* ist als nullable Feld definiert.
22. Die Klasse *CourseSubscription* wird korrekt im DbContext registriert.
23. Der Test *AddSpeakerSuccessTest* ist korrekt aufgebaut.
24. Der Test *AddSpeakerSuccessTest* läuft erfolgreich durch.
25. Der Test *AddSubscriptionSuccessTest* ist korrekt aufgebaut.
26. Der Test *AddSubscriptionSuccessTest* läuft erfolgreich durch.
27. Der Test *DiscriminatorHasCorrectTypeSuccessTest* ist korrekt aufgebaut.
28. Der Test *DiscriminatorHasCorrectTypeSuccessTest* läuft erfolgreich durch.

Teilaufgabe 2: Services und Unittests

Eine Universität möchte Prüfungen – ähnlich wie die theoretische Führerscheinprüfung – am PC durchführen können. Eine Prüfung (*Exam*) besteht aus mehreren Fragen (*Question*). Die Frage hat mehrere Antwortmöglichkeiten zur Auswahl (*PossibleAnswer*). In der möglichen Antwort sind auch die Punkte gespeichert. Eine falsche Antwort hat 0 Punkte (oder negative Punkte, je nach Prüfungsart), eine korrekte Antwort hat z. B. bis zu 5 Punkte.

Die Studenten (*Student*) melden sich zur Prüfung an (*Participation*). Dabei wird aus den Antwortmöglichkeiten der Frage eine Antwort angekreuzt (*Answer*). Multiple Choice soll noch nicht umgesetzt werden.

Folgendes Klassendiagramm ist als Domain Model bereits vorhanden und kann verwendet werden.



Arbeitsauftrag

Implementierung von Servicemethoden

Im Projekt *SPG_Fachtheorie.Aufgabe2* befindet sich die Klasse *Services/ ExamService.cs*. Es sind 2 Methoden zu implementieren:

Question AddQuestion(string questionText, int examId)

Diese Methode soll eine neue Prüfungsfrage zu einer bestehenden Prüfung (Exam) hinzufügen. Dabei sollen folgende Regeln beachtet werden:

1. Existiert die Prüfung nicht (*examId* wird nicht gefunden), so wird eine *ArgumentException* geworfen.
2. Sind bereits 5 oder mehr Fragen der Prüfung (*Exam*) zugeordnet, so soll eine *ExamServiceException* mit dem Text „*Exam already has 5 questions*" geworfen werden. In der Realität sind natürlich mehr als 5 Fragen in der Datenbank, diese Einschränkung dient der leichteren Testbarkeit Ihrer Methode.
3. Werden die Voraussetzungen erfüllt, so wird ein *Question* Objekt erstellt und in die Datenbank eingefügt. Dieses Objekt wird zurückgegeben.

List<ExamResultDto> CalculateExamResults()

Diese Methode soll eine Statistik mit der erreichten Punkteanzahl zurückgeben. Als Rückgabetyp ist folgender C# record im Projekt definiert:

```
public record ExamResultDto(int ExamId, string ExamName, int
ExamFailThreshold, int StudentId, string StudentName, DateTime
StudentBirthDate, int Points);
```

Am einfachsten ist es, sie Fragen die Liste der *Participations* ab. Die Punkteanzahl ist durch Summierung des Feldes *Points* in der Liste der Antworten (*Answers*) über das Property *GivenAnswer* möglich.

In der Klasse *ExamServiceTests* im Projekt *SPG_Fachtheorie.Aufgabe2.Test* ist bereits ein Test *CalculateExamStatisitcsSuccessTest* implementiert. Sie können (sollen) ihn nutzen, um die Korrektheit Ihrer Implementierung prüfen zu können.

Wichtig: Laden Sie mit *Include* bzw. *ThenInclude* alle Informationen aus der Datenbank, wenn Sie auf Navigations zugreifen!

Testen der Methode AddQuestion

Schreiben Sie im Projekt *SPG_Fachtheorie.Aufgabe2.Test* in die Klasse *ExamServiceTests* Unittests, die die Korrektheit von *AddQuestion* prüfen. Mit *GetEmptyDbContext()* können Sie einen Datenbankcontext zu einer leeren Datenbank erstellen. Befüllen Sie die Datenbank selbst mit minimalen Musterdaten, sodass Sie das Methodenverhalten prüfen können. Verwenden Sie *ChangeTracker.Clear()* des Datenbankcontext, um nach dem Einfügen der Musterdaten und nach Aufrufen der Servicemethode den Changetracker zu leeren.

Es sind 3 Tests zu verfassen:

- **AddQuestionSuccessTest** prüft, ob die Methode eine neue Frage zu einer Prüfung (*Exam*) speichert, wenn alle Bedingungen eingehalten werden.
- **AddQuestionThorowsArgumentExceptionWhenExamIdIsInvalidTest** prüft, ob die Methode eine *ArgumentException* wirft, wenn die übergebene *examId* nicht gefunden wird.
- **AddQuestionThorowsExamServiceExceptionWhenQuestionNumbersIsInvalidTest** prüft, ob die Methode eine *ExamServiceException* mit dem Text „*Exam already has 5 questions*" wirft, wenn bei einer Prüfung (*Exam*), die bereits 5 oder mehr Fragen hat, eine Frage eingefügt werden soll.

Bewertung (18P, 23.7% der Gesamtpunkte)

Jedes der folgenden Kriterien wird mit 1 Punkt bewertet.

1. Die Methode *CalculateExamResults* liefert eine Liste von *ExamResultDto* Objekten.
2. Die Methode *CalculateExamResults* legt pro Teilnahme (*Participation*) eine Instanz von *ExamResultDto* an.
3. Die Methode *CalculateExamResults* ermittelt die Summe der Punkte pro Teilnahme korrekt.
4. Die Methode *CalculateExamResults* liest die anderen Felder der DTO Klasse korrekt aus der Datenbank.
5. Die Methode *CalculateExamResults* verwendet LINQ und keine imperativen Konstrukte wie Schleifen, ...
6. Die Methode *CalculateExamResults* besteht den vorhandenen Unittest.
7. Die Methode *AddQuestion* prüft korrekt, ob die übergebene Exam ID vorhanden ist.
8. Die Methode *AddQuestion* wirft eine korrekte ArgumentException.
9. Die Methode *AddQuestion* prüft korrekt, ob 5 oder mehr Fragen vorhanden sind.
10. Die Methode *AddQuestion* wirft eine korrekte ExamServiceException.
11. Die Methode *AddQuestion* erstellt korrekt ein Objekt des Typs *Question*.
12. Die Methode *AddQuestion* fügt die Frage korrekt in die Datenbank ein.
13. Der Unittest *AddQuestionSuccessTest* hat den korrekten Aufbau (arrange, act, assert).
14. Der Unittest *AddQuestionSuccessTest* läuft erfolgreich durch.
15. Der Unittest *AddQuestionThorowsArgumentExceptionWhenExamIdIsInvalidTest* hat den korrekten Aufbau (arrange, act, assert).
16. Der Unittest *AddQuestionThorowsArgumentExceptionWhenExamIdIsInvalidTest* läuft erfolgreich durch.
17. Der Unittest *AddQuestionThorowsExamServiceExceptionWhenQuestionNumbersIsInvalidTest* hat den korrekten Aufbau (arrange, act, assert).
18. Der Unittest *AddQuestionThorowsExamServiceExceptionWhenQuestionNumbersIsInvalidTest* läuft erfolgreich durch.

Teilaufgabe 3: REST API und Integrationtests

Das Datenmodell aus Aufgabe 2 soll nun herangezogen werden, um eine REST Schnittstelle zu erstellen. Die Ausgaben der nachfolgenden Layouts können abweichen, müssen aber alle geforderten Features anbieten.

Arbeitsauftrag

Implementieren Sie die folgenden Controller im Projekt *SPG_Fachtheorie.Aufgabe3*. Die Klasse ExamsController ist bereits vorgegeben, führen Sie Ihre Implementierungen in dieser Klasse durch. Es müssen keine Services verwendet werden, arbeiten Sie direkt mit dem Datenbankcontext, der über Dependency Injection im Controller zur Verfügung steht. Sie können Ihre Endpunkte mit der URL *http://localhost:5080/swagger/index.html* testen.

GET /exams/{id}?includeAnswers=(true|false)

Liefert Daten zu einer Prüfung mit der angegebenen id. Der Query Parameter includeAnswers bestimmt, ob die Antworten im Property „possibleAnswers" ausgegeben werden. Ist der Parameter false, so wird ein leeres Array in possibleAnswers zurückgegeben.

Definieren Sie im Ordner Dtos zuerst einen Record ExamDto. Er soll die Felder der obersten Ebene zum Exam bereit: Id, Name, FailThreshold. Zusätzlich beinhaltet er eine Liste von Fragen (Questions). Für die Ausgabe der Frage erstellen Sie einen Record QuestionDto. Dieser Record beinhaltet die Properties der Frage: id, Text und eine Liste der möglichen Antworten (PossibleAnswers). Für die Daten der Antwort erstellen Sie einen Record PossibleAnswerDtp mit den Feldern Text und Points.

Die Ausgabe des Requests *GET Exams/5?includeAnswers=true* sieht so aus:

```
{
  "id": 5,
  "name": "quisquam",
  "failThreshold": 46,
  "questions": [
    {
      "id": 14,
      "text": "Enim sequi voluptas quis nostrum.",
      "possibleAnswers": [
        {
          "text": "Dignissimos est soluta.",
          "points": 0
        },
        {
          "text": "Cumque ducimus officia vero illum.",
          "points": 0
        },
        {
          "text": "Ut modi sapiente excepturi esse.",
          "points": 0
        },
        {
          "text": "Et neque ut.",
          "points": 0
        }
      ]
    },
    {
      "id": 16,
      "text": "Dolorem quae consequuntur error laboriosam.",
      "possibleAnswers": [
        {
          "text": "Possimus facilis et quia quae consequatur odio veritatis amet sit.",
          "points": 0
        },
        {
          "text": "Facere ut amet.",
          "points": 5
        },
        {
          "text": "Labore quos nihil nulla est aspernatur rem sunt.",
          "points": 0
        },
        {
          "text": "Nostrum nemo sint repellendus delectus alias neque velit error.",
          "points": 0
        }
      ]
    },
    {
      "id": 20,
      "text": "Rem omnis pariatur qui et ut quis natus dolorem.",
      "possibleAnswers": [
        {
          "text": "Nesciunt iusto voluptatem sed autem animi quo.",
          "points": 5
        },
        {
          "text": "A impedit aut.",
          "points": 5
        },
        {
          "text": "Quibusdam exercitationem quia.",
          "points": 0
        },
        {
          "text": "Velit error at unde.",
          "points": 5
        }
      ]
    }
  ]
}
```

Die Ausgabe des Requests *GET Exams/5?includeAnswers=false* sieht so aus:

```
{
  "id": 5,
  "name": "quisquam",
  "failThreshold": 46,
  "questions": [
    {
      "id": 14,
      "text": "Enim sequi voluptas quis nostrum.",
      "possibleAnswers": []
    },
    {
      "id": 16,
      "text": "Dolorem quae consequuntur error laboriosam.",
      "possibleAnswers": []
    },
    {
      "id": 20,
      "text": "Rem omnis pariatur qui et ut quis natus dolorem.",
      "possibleAnswers": []
    }
  ]
}
```

Wird das Exam nicht gefunden (id ist in der Datenbank nicht vorhanden), soll HTTP 404 mit der Meldung „Exam not found" geliefert werden.

PATCH /exams/{id}

Dieser Endpunkt soll den Schwellenwert für „failed" des Exams (*failThreshold*) verändern. Der request body sieht so aus:

```
{
  "failThreshold": 10
}
```

Erstellen Sie einen Record UpdateThresholdCommand im Ordner Commands, der den request body darstellen soll. Achten Sie auf die Validierung des Wertes failThreshold. Er muss größer als 0, aber kleiner als der mögliche Punktewert des Exams sein. Sie können den möglichen Punktewert des Exams bestimmen, indem Sie über alle Questions des Exams und innerhalb der Question über alle PossibleAnswers den Wert Points summieren. Vergessen Sie nicht, diese Navigations zu inkludieren.

Wird das Exam nicht gefunden (id ist in der Datenbank nicht vorhanden), soll HTTP 404 mit der Meldung „*Exam not found.*" geliefert werden. Ist der Schwellenwert zu hoch, soll HTTP 400 mit der Meldung „*FailThreshold too high.*" Geliefert werden. Fangen Sie Fehler, die aus der Datenbankschicht kommen, ab und liefern die Meldung mit dem Statuscode 400 an den Client.

DELETE /exams/{id}

Dieser Endpunkt soll ein Exam löschen. Es wird aber nicht mit einer DELETE Anweisung aus der Datenbank gelöscht, sondern es soll lediglich das Property Visible in Exam gesetzt werden (soft delete).

Wird das Exam nicht gefunden (id ist in der Datenbank nicht vorhanden), soll HTTP 404 mit der Meldung „Exam not found." geliefert werden. Fangen Sie Fehler, die aus der Datenbankschicht kommen, ab und liefern die Meldung mit dem Statuscode 400 an den Client.

Verfassen von Integration Tests

Im Projekt SPG_Fachtheorie.Aufgabe3.Test befindet sich die Klasse ExamsControllerTests. Erstellen Sie Unittests für jeden der 3 Endpunkte, die folgendes Beweisen:

Endpunkt GET /exams/{id}?includeAnswers=(true|false)

- Der Controller liefert bei einer ungültigen ID den korrekten Statuscode (not found).
- Der Controller liefert bei einer gültigen ID die korrekte Anzahl der Datensätze.
- Der Controller liefert bei gesetztem Flag includeAnswers die korrekte Anzahl an Antworten.

Endpunkt PATCH /exams/{id}

- Der Controller liefert bei einer ungültigen ID den korrekten Statuscode (not found).
- Der Controller liefert bei zu hohem Schwellenwert den korrekten Statuscode (bad request).
- Der Controller liefert im Erfolgsfall den korrekten Statuscode. Stellen Sie durch Lesen der Datenbank sicher, dass der neue Wert für *FailThreshold* auch korrekt eingetragen wurde.

Endpunkt DELETE /exams/{id}

- Der Controller liefert bei einer ungültigen ID den korrekten Statuscode (not found).
- Der Controller liefert im Erfolgsfall den korrekten Statuscode. Stellen Sie durch Lesen der Datenbank sicher, dass der neue Wert für *Visible* auch korrekt eingetragen wurde.

Bewertung (30P, 39.5% der Gesamtpunkte)

Jedes der folgenden Kriterien wird, wenn nicht anders angegeben, mit 1 Punkt bewertet.

1. Der Record ExamDto ist korrekt.
2. Der Record QuestionDto ist korrekt.
3. Der Record PossibleAnswersDto ist korrekt.
4. Das Exam wird korrekt in der Controllermethode für den GET Request abgefragt.
5. Der Parameter includeAnswers im den GET Request wird korrekt berücksichtigt.
6. Das Fehlerverhalten für den GET Request bei einem ungültigem Exam ist korrekt.
7. Die Controllermethode für den GET Request liefert im Erfolgsfall die korrekten Daten und den korrekten Statuscode.
8. Der Record UpdateThresholdCommand ist korrekt.
9. Der Record UpdateThresholdCommand oder der Controller validiert korrekt die untere Grenze.
10. Die Controllermethode validiert korrekt die obere Grenze.
11. Das Exam wird korrekt in der Controllermethode für den PATCH Request abgefragt.
12. Das Fehlerverhalten für den PATCH Request bei einem ungültigem Exam ist korrekt.
13. Die Ermittlung der maximalen Punkte des Exams in der Controllermethode für den PATCH Request ist korrekt.
14. Die Controllermethode für den PATCH Request liefert für den Fall, dass FailThreshold zu hoch ist, den korrekten Statuscode.
15. Die Controllermetgode für den PATCH Request aktualisiert den Wert von FailThreshold
16. Die Controllermethode für den PATCH Request liefert im Erfolgsfall den korrekten Statuscode.
17. Das Exam wird korrekt in der Controllermethode für den DELETE Request abgefragt.
18. Das Fehlerverhalten für den DELETE Request bei einem ungültigem Exam ist korrekt.
19. Die Controllermethode für den DELETE Request aktualisiert den Wert für Visible.
20. Die Controllermethode für den DELETE Request liefert im Erfolgsfall den korrekten Statuscode.
21. Der Integration Test für den GET Request hat den korrekten Aufbau.
22. Der Integration Test für den GET Request prüft alle Fehlerzustände korrekt.

23. Der Integration Test für den GET Request prüft korrekt, wenn der Parameter *includeAnswers* auf false gesetzt wird.
24. Der Integration Test für den GET Request prüft korrekt, wenn der Parameter *includeAnswers* auf *true* gesetzt ist.
25. Der Integration Test für den PATCH Request hat den korrekten Aufbau.
26. Der Integration Test für den PATCH Request prüft alle Fehlerzustände korrekt.
27. Der Integration Test für den PATCH Request prüft den Erfolgsfall korrekt.
28. Der Integration Test für den DELETE Request hat den korrekten Aufbau.
29. Der Integration Test für den DELETE Request prüft alle Fehlerzustände korrekt.
30. Der Integration Test für den DELETE Request prüft den Erfolgsfall korrekt.

76 – 67 Punkte: Sehr gut, 66 – 58 Punkte: Gut, 57 – 48 Punkte: Befriedigend, 47 – 38 Punkte: Genügend, 37 – 0 Punkte: Nicht genügnd. **Für die Einrechnung der Jahresnote müssen mindestens 50% der Aufgabe 1 und 50% der Aufgabe 3 erreicht werden.**