Probeklausur in POS

> Klassen: 6AAIF, 6BAIF, 6CAIF, 6AKIF, 6BKIF
> Datum: MI, 4. Juni 2025
> Arbeitszeit: 5 UE
>
> Öffnen Sie die Datei *README_en.pdf*, um die englische Fassung dieser Angabe zu sehen.

Auf Ihrem Laufwerk R befindet sich der Ordner *SPG_Fachtheorie*. Darin befindet sich die Datei *SPG_Fachtheorie.sln*. Öffnen Sie die Datei in Microsoft Visual Studio. Arbeiten Sie ausschließlich in dieser Solution. Da das Projekt auf dem Netzlaufwerk liegt, muss am Ende nicht extra auf ein Abgabelaufwerk kopiert werden. Beenden Sie am Ende Ihrer Arbeit alle Programme und lassen Sie den PC eingeschaltet.

Wichtiger Hinweis vor Arbeitsbeginn

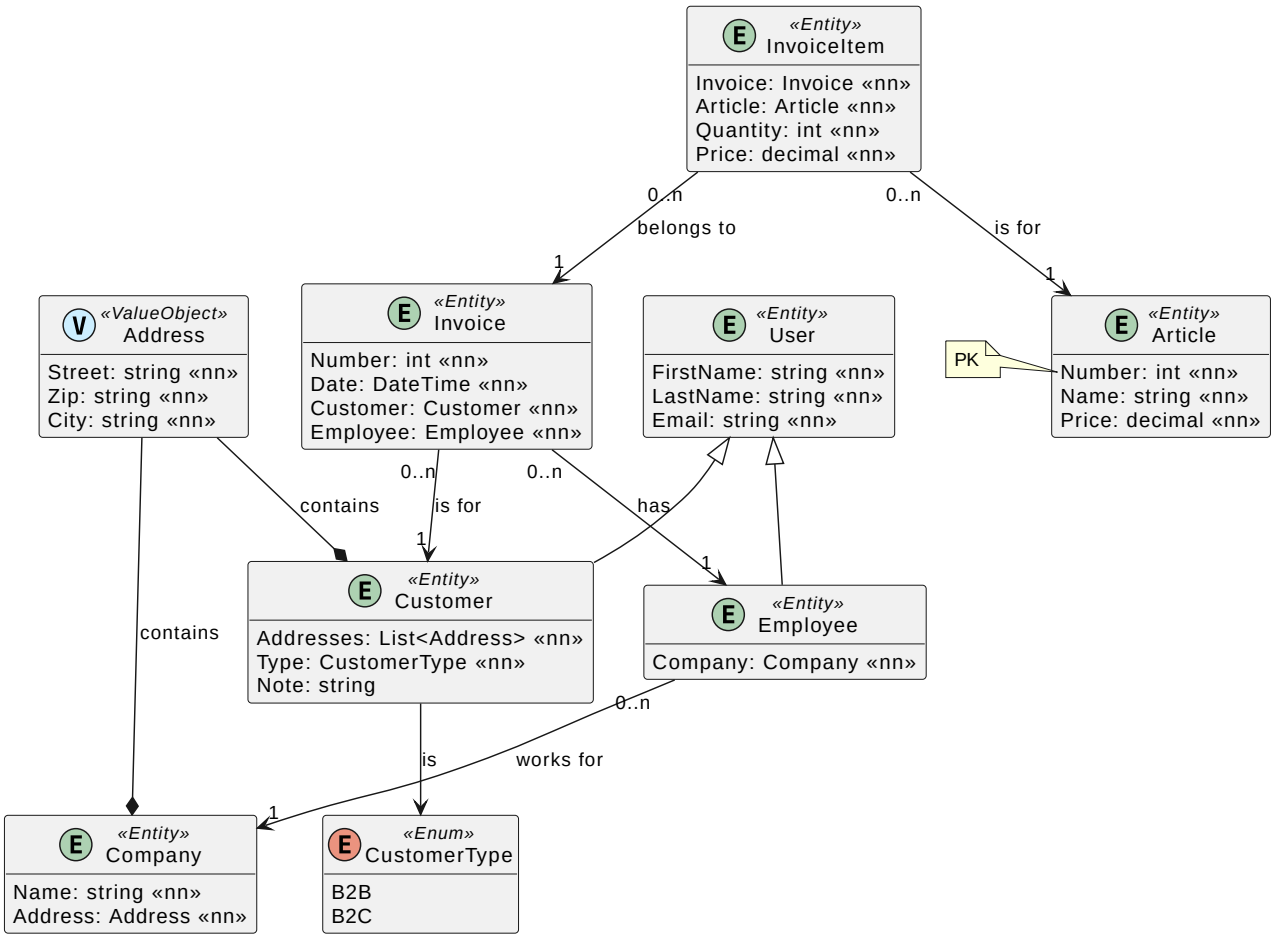
> ⚠ Füllen Sie die Datei *README.md* in *SPG_Fachtheorie/README.md* mit Ihren Daten (Klasse, Name und Accountname) aus. Sie sehen die Datei in Visual Studio unter *Solution Items* nach dem Öffnen der Solution. **Falls Sie dies nicht machen, kann Ihre Arbeit nicht bewertet werden!**

Wichtiger Hinweis nach Arbeitsende

> ⚠ Starten Sie am Ende Ihrer Arbeit - nach dem Schließen der IDE - das Skript *compile.cmd* im Ordner *SPG_Fachtheorie*. Es kontrolliert, ob Ihre Projekte kompiliert werden können. **Projekte, die nicht kompilieren, können nicht bewertet werden!**

Teilaufgabe 1: Object Relation Mapping

Ein kleines Kundenverwaltungssystem soll mit Hilfe von EF Core und .NET implementiert werden. Es soll Kunden (*Customer*) und Artikel (*Article*) erfassen können. Wenn Kunden einen oder mehrere Artikel kaufen, wird eine Rechnung (*Invoice*) angelegt. Die Rechnung hat mehrere Zeilen (*InvoiceItem*). Die Zeile beschreibt, in welcher Menge und zu welchem Preis ein konkreter Artikel gekauft wurde. Beachten Sie dabei die geforderten Anforderungen an einzelne Modellklassen.



Arbeitsauftrag

Erstellung der Modelklassen

Implementieren Sie das dargestellte Diagramm als EF Core Modelklassen. Im Projekt *SPG_Fachtheorie_Aufgabe1* befinden sich leere Klassen sowie die Klasse *InvoiceContext*, die Sie nutzen sollen. Beachten Sie bei der Umsetzung folgende Punkte:

- Legen Sie nötige Konstruktoren an. Ein *public* Konstruktor soll alle im Modell enthaltenen Properties initialisieren. Ergänzen Sie die für EF Core nötigen *protected* Konstruktoren.
- Bachten Sie Attribute Constraints wie **not null** (nn).
- Strings sollen - wenn nicht anders angegeben - mit einer Maximallänge von 255 Zeichen definiert werden.
- Der Preis soll mit 9 Stellen (5 Vorkomma- und 4 Nachkommastellen) gespeichert werden. Sie können das Attribut *[Precision(9, 4)]* verwenden.
- Das Entity *Address* soll in *Company* und *Customer* als *value object* definiert werden. Beachten Sie, dass in *Customer* eine Liste von value objects definiert werden muss.
- Verwenden sie eigene primary keys mit dem Namen *Id* (autoincrement), außer im Modell ist mit *PK* explizit ein Schlüssel angegeben.

- Speichern Sie die enum *CustomerType* in *Customer* als String in der Datenbank ab.
- Die Klassen *Employee* und *Customer* erben von der Klasse *User*.
- Legen Sie Ihre DbSets mit den folgenden Namen an: *Companies*, *Users*, *Invoices*, *InvoiceItems*, *Articles*.

Damit Sie eine Liste von value objects mit EF Core in Zusammenhang mit SQLite erstellen können, müssen Sie folgende Konfiguration verwenden:

```
modelBuilder.Entity<Customer>().OwnsMany(c => c.Addresses, c =>
{
    c.HasKey("Id");
});
```

Verfassen von Tests

In der Klasse *Aufgabe1Test* im Projekt *test/SPG_Fachtheorie.Aufgabe1.Test* sollen Testmethoden verfasst werden, die die Richtigkeit Ihrer Implementierung beweisen soll.

- *PersistEnumSuccessTest* beweist, dass die Enum *CustomerType* in *Customer* korrekt gespeichert wird.
- *PersistValueObjectInCompanySuccessTest* beweist, dass Sie ein Entity vom Typ *Company* mit einer Adresse als value object speichern können.
- *PersistValueObjectsInCustomerSuccessTest* beweist, dass Sie ein Entity vom Typ *Customer* mit einer Liste von Adressen als value object speichern können.
- *PersistInvoiceItemSuccessTest* beweist, dass Sie ein Entity vom Typ *InvoiceItem* in der Datenbank speichern können.

Bewertung

Aufgabe 1 (33 Punkte in Summe, 36 %)	**Punkte**
Article: Alle geforderten Constraints und Beschränkungen wurden umgesetzt.	1
Article: Die Klasse verwendet Number als primary key.	1
Article: Der vorgegebene Test InsertArticleTest läuft durch.	2
Address: Die Klasse enthält alle in der Angabe vorgesehenen Properties.	1
Address: Die Klasse besitzt keine ID und wird nicht als Tabelle verwendet.	1
Company: Alle geforderten Constraints und Beschränkungen wurden umgesetzt.	1
Company: Die Klasse besitzt ein korrekt konfiguriertes value object "Address".	1
Company: Der vorgegebene Test InsertCompanyTest läuft durch.	2

Aufgabe 1 (33 Punkte in Summe, 36 %)	Punkte
Employee: Alle geforderten Constraints und Beschränkungen wurden umgesetzt.	1
Employee: Der vorgegebene Test InsertEmployeeTest läuft durch.	2
Customer: Alle geforderten Constraints und Beschränkungen wurden umgesetzt.	1
Customer: Die Enumeration Type wird als String in der Datenbank gespeichert.	1
Customer: Der vorgegebene Test InsertCustomerTest läuft durch.	2
Customer: Der vorgegebene Test InsertCustomerWithAddressesTest läuft durch.	2
Invoice: Alle geforderten Constraints und Beschränkungen wurden umgesetzt.	1
Invoice: Der vorgegebene Test InsertInvoiceTest läuft durch.	2
InvoiceItem: Alle geforderten Constraints und Beschränkungen wurden umgesetzt.	1
InvoiceItem: Der vorgegebene Test InsertInvoiceItemTest läuft durch.	2
PersistEnumSuccessTest: Der Test ist richtig aufgebaut.	1
PersistEnumSuccessTest: Der Test ist läuft durch.	1
PersistValueObjectInCompanySuccessTest: Der Test ist richtig aufgebaut.	1
PersistValueObjectInCompanySuccessTest: Der Test ist läuft durch.	1
PersistValueObjectInCustomerSuccessTest: Der Test ist richtig aufgebaut.	1
PersistValueObjectInCustomerSuccessTest: Der Test ist läuft durch.	1
PersistInvoiceItemSuccessTest: Der Test ist richtig aufgebaut.	1
PersistInvoiceItemSuccessTest: Der Test ist läuft durch.	1

Teilaufgabe 2: Abfragen und Servicemethoden

Das nachfolgende Modell zeigt eine Umsetzung eines Scooter Sharing Services. Kunden können zwischen 2 Abrechnungsmethoden wählen:

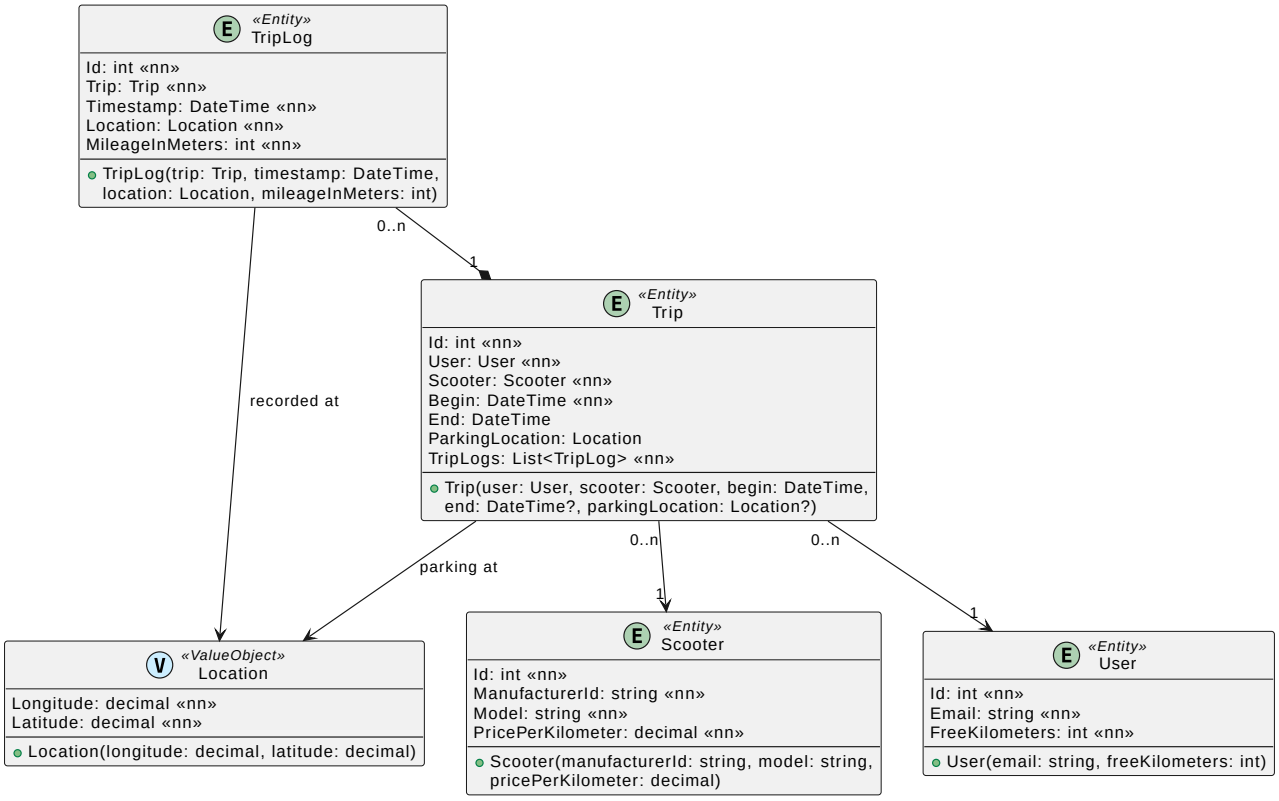
- *Pay as you go:* Der Kunde zahlt pro gefahrenen Kilometer.
- *Included Kilometers:* Der Kunde zahlt einen monatlichen Beitrag und erhält eine bestimmte Anzahl an Freikilometern. Die freien Kilometer gelten pro Fahrt. Hat ein Kunde z. B. 2 Freikilometer, so wird bei Fahrt A, die 3 km lang ist, 1 km berechnet. Bei Fahrt B, die 4 km lang ist, werden 2 km berechnet.

Beim Entsperren mit der App wird ein *Trip* gestartet. Der integrierte Tracker des Scooters übermittelt in regelmäßigen Abständen die Position und den Kilometerstand als *TripLog*. Dadurch ist eine Abrechnung auch dann möglich, wenn kein durchgehendes GPS Signal verfügbar ist (z. B.

Unterführungen, etc). Die Länge einer Fahrt (*Trip*) ist dann die Differenz zwischen niedrigsten und höchsten Kilometerstand der *TripLogs* dieser Fahrt.

Die Preise für den gefahrenen Kilometer variieren je nach Modell und sind in der *Scooter* Klasse hinterlegt. Scooter mit Sitz und hoher Reichweite haben z. B. einen höheren Preis pro Kilometer als Scooter ohne Sitz.

Wird ein Scooter abgestellt, wird der *Trip* beendet und die Felder *End* und *ParkingLocation* werden gesetzt. So ist erkennbar, welche Trips noch laufen und welche bereits beendet sind.



Arbeitsauftrag

In den Dateien *src/SPG_Fachtheorie.Aufgabe2/Infrastructure/ScooterContext.cs* steht ein vollständig implementierter DbContext zur Verfügung. Das Model ist bereits in *src/SPG_Fachtheorie.Aufgabe2/Model/Model.cs* implementiert. **Sie müssen keine Implementierung des Modelles vornehmen, sondern es in Ihren Servicemethoden verwenden.**

Implementierung von Servicemethoden

Führen Sie Ihre Implementierungen in *src/SPG_Fachtheorie.Aufgabe2/Services/ScooterService.cs* durch. Verwenden Sie den über Dependency Injection bereitgestellten *ScooterContext*.

public Trip AddTrip(int userId, int scooterId, DateTime begin)

Diese Methode einen Trip in der Datenbank anlegen. Dabei sollen folgende Randbedingungen geprüft werden:

- Wird die *userId* nicht gefunden, so ist eine *ScooterServiceException* mit dem Text *Invalid user.* zu werfen.
- Wird die *scooterId* nicht gefunden, so ist eine *ScooterServiceException* mit dem Text *Invalid scooter.* zu werfen.
- Existiert ein offener Trip für diesen Scooter in der Datenbank, so ist eine *ScooterServiceException* mit dem Text *Scooter is not parked.* zu werfen. Sie erkennen offene Trips daran, dass *Trip.End* keinen Wert (*null*) hat.

Sind alle Bedingungen erfüllt, fügen Sie den neuen Trip mit dem übergebenen User, Scooter und der Beginnzeit in die Datenbank ein.

Verwenden Sie den bereitgestellten Unittest AddTripTest in test/SPG_Fachtheorie.Aufgabe2.Test/ScooterServiceTests.cs, um die Richtigkeit Ihrer Methode zu prüfen.

public List<TripDto> GetTripInfos(DateTime beginFrom, DateTime beginTo)

Diese Methode soll Informationen zu allen Trips abfragen, die in einem gewissen Zeitraum begonnen haben. In Ihrer Servicemethode ist ein Record als Rückgabetyp definiert:

```
public record TripDto(
  int Id, DateTime Begin, DateTime? End, int ScooterId,
  string ScooterModel, int UserId, string UserEmail, bool IsParked);
```

Das Property *IsParked* hat den Wert *true* wenn *Trip.End* nicht null ist. Fragen Sie die benötigten Informationen zur Erstellung von *TripDto* aus der Datenbank ab.

Verwenden Sie den bereitgestellten Unittest GetTripInfosTest in test/SPG_Fachtheorie.Aufgabe2.Test/ScooterServiceTests.cs, um die Richtigkeit Ihrer Methode zu prüfen.

public int CalculateTripLength(int tripId)

Diese Methode soll die Länge eines Trips in Metern berechnen. Ein Trip hat mehrere *TripLogs*. Dies sind Punkte, die z. B. von einem GPS Modul am Scooter übermittelt werden. Um die Länge eines Trips in Metern zu berechnen, verwenden Sie den kleinsten und größten Kilometerstand (*TripLog.MileageInMeters*) des Logs dieses Trips. Die Differenz zwischen diesen beiden Werten ist der gesuchte Wert.

Verwenden Sie den bereitgestellten Unittest CalculateTripLengthTest in test/SPG_Fachtheorie.Aufgabe2.Test/ScooterServiceTests.cs, um die Richtigkeit Ihrer Methode zu prüfen.

public decimal CalculatePrice(int tripId)

Diese Methode soll den Preis für einen Trip berechnen. Dabei müssen folgende Werte berücksichtigt werden:

- Ein User hat freie Kilometer (*User.FreeKilometers*). Diese müssen von der Triplänge abgezo-

gen werden.

- Der Preis pro Kilometer ist in *Scooter.PricePerKilometer* gespeichert.

Wird der Trip nicht gefunden, so ist der Preis 0 zurückzugeben.

Verwenden Sie den bereitgestellten Unittest CalculatePriceTest in test/SPG_Fachtheorie.Aufgabe2.Test/ScooterServiceTests.cs, um die Richtigkeit Ihrer Methode zu prüfen.

Bewertung

Aufgabe 2 (26 Punkte in Summe, 28 %)	**Punkte**
AddTrip: Die Methode berücksichtigt alle Randbedingungen korrekt.	2
AddTrip: Die Methode fügt den neuen Trip korrekt in die Datenbank ein.	1
AddTrip: Der Unittest AddTripTest läuft durch.	3
GetTripInfos: Die Methode filtert korrekt die Daten.	1
GetTripInfos: Die Methode fragt korrekt ab und projiziert das Ergebnis auf die DTO Klasse.	2
GetTripInfos: Der Unittest GetTripInfosTest läuft durch.	3
CalculateTripLength: Die Methode filtert korrekt die Daten.	1
CalculateTripLength: Die Methode führt die Berechnung korrekt durch.	3
CalculateTripLength: Der Unittest CalculateTripLengthTest läuft durch.	3
CalculatePrice: Die Methode filtert korrekt die Daten.	1
CalculatePrice: Die Methode führt die Berechnung korrekt durch.	3
CalculatePrice: Der Unittest CalculatePriceTest läuft durch.	3

Teilaufgabe 3: REST(ful) API

Für das vorige Modell der Scooter Sharing App soll eine RESTful API implementiert werden. Wenn Sie das Projekt in *src/SPG_Fachtheorie.Aufgabe3* starten, steht Ihnen unter der URL *http://localhost:5080/swagger/index.html* ein Endpoint Explorer zur Verfügung. Die Datenbank beinhaltet Musterdaten, sodass Sie die Funktionalität Ihrer Controller damit testen können.

Arbeitsauftrag

Implementieren Sie die folgenden REST API Routen. In *src/SPG_Fachtheorie.Aufgabe3/Controllers/TripsController.cs* steht dafür ein leerer Controller bereit. Über Dependency Injection wird ein mit Musterdaten gefüllter *ScooterContext* bereitgestellt. Sie können die Abfragen direkt in den Controllermethoden implementieren. Führen Sie alle Abfragen nicht blockierend (mit *await* und *async*) durch. Achten Sie bei Fehlern auf eine RFC-9457 kompatible Antwort, indem Sie die

Methode *Problem()* mit geeignetem Statuscode verwenden.

GET /trips/{id}?includeLog=true

Diese REST API Route soll einen bestimmten Trip mitsamt der Logeinträge retournieren. Wird der optionale Parameter *includeLog* mit dem Wert *true* mitgegeben (*false* ist der Default-Wert), soll die Antwort die Logeinträge enthalten. Ist der Parameter *false*, soll ein leeres Array zurückgegeben werden.

Erstellen Sie zuerst die Felder in den Klassen *src/SPG_Fachtheorie.Aufgabe3/Dtos/TripDto.cs* und *src/SPG_Fachtheorie.Aufgabe3/Dtos/TripLogDto.cs*, sodass die Daten dem nachfolgenden Schema entsprechen.

Table 1. Erwartete HTTP-Antworten:

HTTP Status	Bedingung
200	Ein *TripDto* mit leerem Array für *logs*, wenn der Parameter *includeLog* false ist.
200	Ein *TripDto* mit einem Array von *TripLogDto*, wenn der Parameter *includeLog* true ist.
404	Für eine unbekannte Id. (inkl. RFC-9457 ProblemDetail im Body)

Response für GET trips/1?includeLog=false

```
{
  "id": 1,
  "userEmail": "alice@example.com",
  "scooterManufacturer": "Xiaomi",
  "begin": "2025-02-07T10:46:03",
  "end": null,
  "logs": []
}
```

Response für GET trips/1?includeLog=true

```
{
  "id": 1,
  "userEmail": "alice@example.com",
  "scooterManufacturer": "Xiaomi",
  "begin": "2025-02-07T10:46:03",
  "end": null,
  "logs": [
    {
      "timestamp": "2025-02-07T10:46:03",
      "logitude": 16.48055,
      "latitute": 48.16812,
      "mileageInMeters": 12397
```

```
    },
    {
      "timestamp": "2025-02-07T10:48:03",
      "logitude": 16.60784,
      "latitute": 48.30346,
      "mileageInMeters": 12663
    },
    {
      "timestamp": "2025-02-07T10:52:03",
      "logitude": 15.85092,
      "latitute": 48.21999,
      "mileageInMeters": 13510
    },
    {
      "timestamp": "2025-02-07T10:54:03",
      "logitude": 15.44802,
      "latitute": 48.26622,
      "mileageInMeters": 14396
    }
  ]
}
```


PATCH /trips/{id}

Dieser Endpunkt soll die Daten am Ende eines Trips aktualisieren. Dabei werden die Felder *End* und *ParkingLocation* auf die entsprechenden Werte gesetzt. Achten Sie darauf, dass der Wert von *Trip.End* vor der Aktualisierung *null* sein muss. Es dürfen keine schon beendeten Trips nochmals beendet werden. Ist der Trip bereits beendet, liefern Sie HTTP 400 mit der Meldung *Trip already ended.*

Verwenden Sie das Command Object in *src/SPG_Fachtheorie.Aufgabe3/Commands/UpdateTripCommand.cs*, um den Payload zu typisieren. Achten Sie darauf, dass nur gültige Werte für *Longitude* (-180 bis +180) und *Latitude* (-90 bis +90) übermittelt werden können. Stellen Sie dies durch entsprechende Annotations zur Validierung im Command Object sicher.

Der Trip mit der ID 4 ist nicht beendet, er kann zum Testen verwendet werden.

Table 2. Erwartete HTTP-Antworten:

HTTP Status	Bedingung
204	No content, wenn erfolgreich aktualisiert wurde.
400	Wenn ein Trip, der bereits beendet ist, modifiziert werden soll. (inkl. RFC-9457 ProblemDetail im Body)
400	Wenn die Werte für *Longitude* oder *Latitude* ungültig sind. (inkl. RFC-9457 ProblemDetail im Body)

HTTP Status	Bedingung
404	Für eine unbekannte Id. (inkl. RFC-9457 ProblemDetail im Body)

Payload für PATCH trips/4

```
{
  "end": "2025-05-30T10:13:29.412Z",
  "logitude": 16.4,
  "latitude": 48.1
}
```

Verfassen von Integration Tests

In *test/SPG_Fachtheorie.Aufgabe3.Test/TripsControllerTests.cs* sollen Integration Tests verfasst werden. Es soll jede Zeile in den Tabellen bei *Erwartete HTTP-Antworten* geprüft werden. Verwenden Sie dafür die in der *TestWebApplicationFactory* bereitgestellten Hilfsmethoden:

Methode	Beschreibung
InitializeDatabase	Erstellt eine leere Datenbank und fügt ggf. Werte ein.
QueryDatabase	Erlaubt das Abfragen der Datenbank.
GetHttpContent<T>	Führt einen GET Request durch und liefert das Ergebnis als Typ *T*.
PatchHttpContent<Tcmd>	Führt einen PATCH Request durch und liefert ggf. Payload als *JsonElement*.

Bewertung

Aufgabe 3 (33 Punkte in Summe, 36 %)	Punkte
GET /trip/{id} filtert die Daten korrekt.	1
GET /trip/{id} liefert das korrekte Verhalten im Fall einer nicht vorhandenen Id.	1
GET /trip/{id} verwendet den Query Parameter includeLog korrekt.	3
Die DTO Klasse TripDto wurde korrekt implementiert.	2
Die DTO Klasse TripLogDto wurde korrekt implementiert.	1
GET /trip/{id} liefert das korrekte Ergebnis im Erfolgsfall für includeLogs = false.	2
GET /trip/{id} liefert das korrekte Ergebnis im Erfolgsfall für includeLogs = true.	2
Der Integration Test beweist die korrekte Implementierung von GET /trip/{id} im Erfolgsfall.	2
Der Integration Test beweist die korrekte Implementierung von GET /trip/{id} bei Fehlerzuständen.	2

Aufgabe 3 (33 Punkte in Summe, 36 %)	**Punkte**
Der Integration Test für GET /trip/{id} läuft durch.	3
PATCH /trip/{id} fragt korrekt die Werte aus der Datenbank ab.	1
PATCH /trip/{id} liefert das korrekte Verhalten im Fall einer nicht vorhandenen Id.	1
PATCH /trip/{id} liefert das korrekte Verhalten im Fall eines bereits beendeten Trips.	1
PATCH /trip/{id} aktualisiert den Wert in der Datenbank.	2
Die Command Klasse UpdateTripCommand kann den Payload typisieren.	1
Die Command Klasse UpdateTripCommand besitzt die definierten Validierungen.	1
Der Integration Test beweist die korrekte Implementierung von PATCH /trip/{id} im Erfolgsfall.	2
Der Integration Test beweist die korrekte Implementierung von PATCH /trip/{id} bei Fehlerzuständen.	2
Der Integration Test für PATCH /trip/{id} läuft durch.	3

Beurteilung:

92 - 81 Punkte: Sehr gut (1)
80 - 70 Punkte: Gut (2)
69 - 58 Punkte: Befriedigend (3)
57 - 46 Punkte: Genügend (4)
unter 46 Punkte: Nicht genügend (5)

Viel Erfolg!