Training exam in POS

> Classes: 6AAIF, 6BAIF, 6CAIF, 6AKIF, 6BKIF
> Date: Wed, June 4, 2025
> Working time: 5 teaching units
>
> Open the file *README.pdf* to see the German version of this document.

The folder *SPG_Fachtheorie* is located on your R drive. It contains the file *SPG_Fachtheorie.sln*. Open the file in Microsoft Visual Studio. Work exclusively in this solution. Since the project is located on the network drive, there is no need to copy it to a submission drive at the end. When you have finished your work, close all programs and leave the PC switched on.

Important note before starting your work

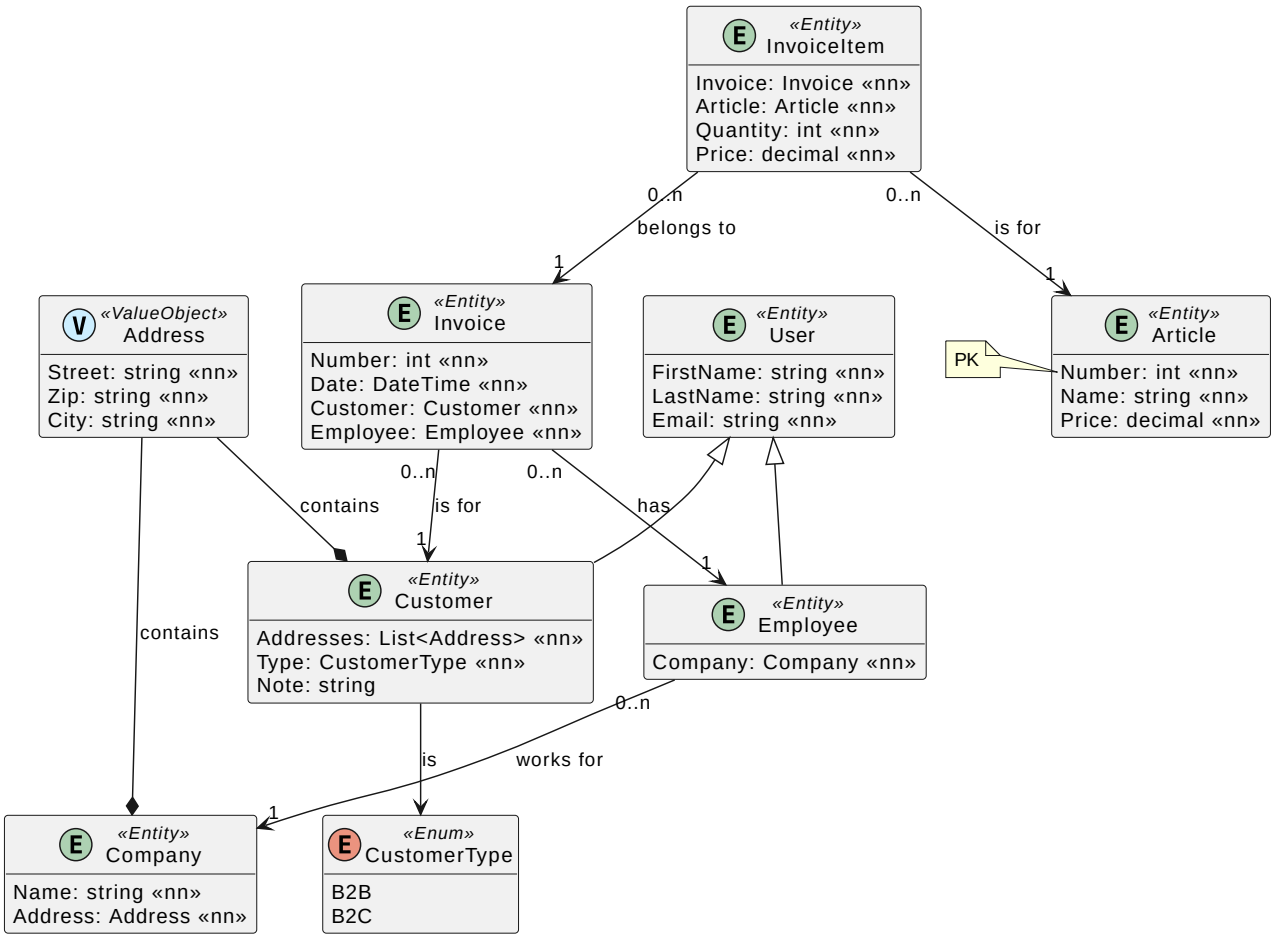
Enter your personal details (class, name, and account name) in the file *README.md* located in *SPG_Fachtheorie/README.md*. You will find the file in Visual Studio under *Solution Items* after opening the solution. **If you don't do this, your work cannot be graded!**

Important note after finishing your work

When you have finished your work, close the IDE and run the script *compile.cmd* in the folder *SPG_Fachtheorie*. This script checks whether your projects can be compiled. **Projects that doesn't compile cannot be graded!**

Subtask 1: Object Relation Mapping

A small customer management system is to be implemented using EF Core and .NET. It should be able to record customers (*Customer*) and articles (*Article*). When customers purchase one or more articles, an invoice (*Invoice*) is created. The invoice has several lines (*InvoiceItem*). The line describes the quantity and price of a specific article purchased. Please note the requirements for individual model classes.



Work assignment

Creation of model classes

Implement the diagram shown as EF Core model classes. The project *SPG_Fachtheorie_Aufgabe1* contains empty classes and the class *InvoiceContext*, which you should use. Please note the following points during implementation:

- Create the necessary constructors. A *public* constructor should initialize all properties contained in the model. Add the *protected* constructors required for EF Core.
- Pay attention to attribute constraints such as **not null** (nn).
- Unless otherwise specified, strings should be defined with a maximum length of 255 characters.
- The price should be stored with 9 digits (5 digits before the decimal point and 4 digits after the decimal point). You can use the attribute *[Precision(9, 4)]*.
- The entity *Address* should be defined in *Company* and *Customer* as a *value object*. Note that a list of value objects must be defined in *Customer*.
- Use your own primary keys with the name *Id* (autoincrement), unless a key is explicitly specified in the model with *PK*.

- Store the enum *CustomerType* in *Customer* as a string in the database.
- The classes *Employee* and *Customer* inherit from the class *User*.
- Create your DbSets with the following names: *Companies*, *Users*, *Invoices*, *InvoiceItems*, *Articles*.

To create a list of value objects with EF Core in conjunction with SQLite, you must use the following configuration:

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

Writing tests

In the *Task1Test* class in the *test/SPG_Fachtheorie.Task1.Test* project, you should write test methods that prove the correctness of your implementation.

- *PersistEnumSuccessTest* proves that the *CustomerType* enum is stored correctly in *Customer*.
- *PersistValueObjectInCompanySuccessTest* proves that you can store an entity of type *Company* with an address as a value object.
- *PersistValueObjectsInCustomerSuccessTest* proves that you can store an entity of type *Customer* with a list of addresses as a value object.
- *PersistInvoiceItemSuccessTest* proves that you can store an entity of type *InvoiceItem* in the database.

Evaluation

Task 1 (33 points in total, 36%)	Points
Article: All required constraints and restrictions have been implemented.	1
Article: The class uses Number as the primary key.	1
Article: The specified InsertArticleTest test runs successfully.	2
Address: The class contains all properties specified in the specification.	1
Address: The class has no ID and is not used as a table.	1
Company: All required constraints and restrictions have been implemented.	1
Company: The class has a correctly configured value object "Address".	1
Company: The specified test InsertCompanyTest runs successfully.	2
Employee: All required constraints and restrictions have been implemented.	1

Task 1 (33 points in total, 36%)	Points
Employee: The specified test InsertEmployeeTest runs.	2
Customer: All required constraints and restrictions have been implemented.	1
Customer: The enumeration type is stored as a string in the database.	1
Customer: The specified test InsertCustomerTest runs.	2
Customer: The specified test InsertCustomerWithAddressesTest runs successfully.	2
Invoice: All required constraints and restrictions have been implemented.	1
Invoice: The specified test InsertInvoiceTest runs successfully.	2
InvoiceItem: All required constraints and restrictions have been implemented.	1
InvoiceItem: The specified InsertInvoiceItemTest test runs successfully.	2
PersistEnumSuccessTest: The test is structured correctly.	1
PersistEnumSuccessTest: The test runs successfully.	1
PersistValueObjectInCompanySuccessTest: The test is structured correctly.	1
PersistValueObjectInCompanySuccessTest: The test runs successfully.	1
PersistValueObjectInCustomerSuccessTest: The test is structured correctly.	1
PersistValueObjectInCustomerSuccessTest: The test runs successfully.	1
PersistInvoiceItemSuccessTest: The test is structured correctly.	1
PersistInvoiceItemSuccessTest: The test runs successfully.	1

Subtask 2: Queries and service methods

The following model shows the implementation of a scooter sharing service. Customers can choose between two billing methods:

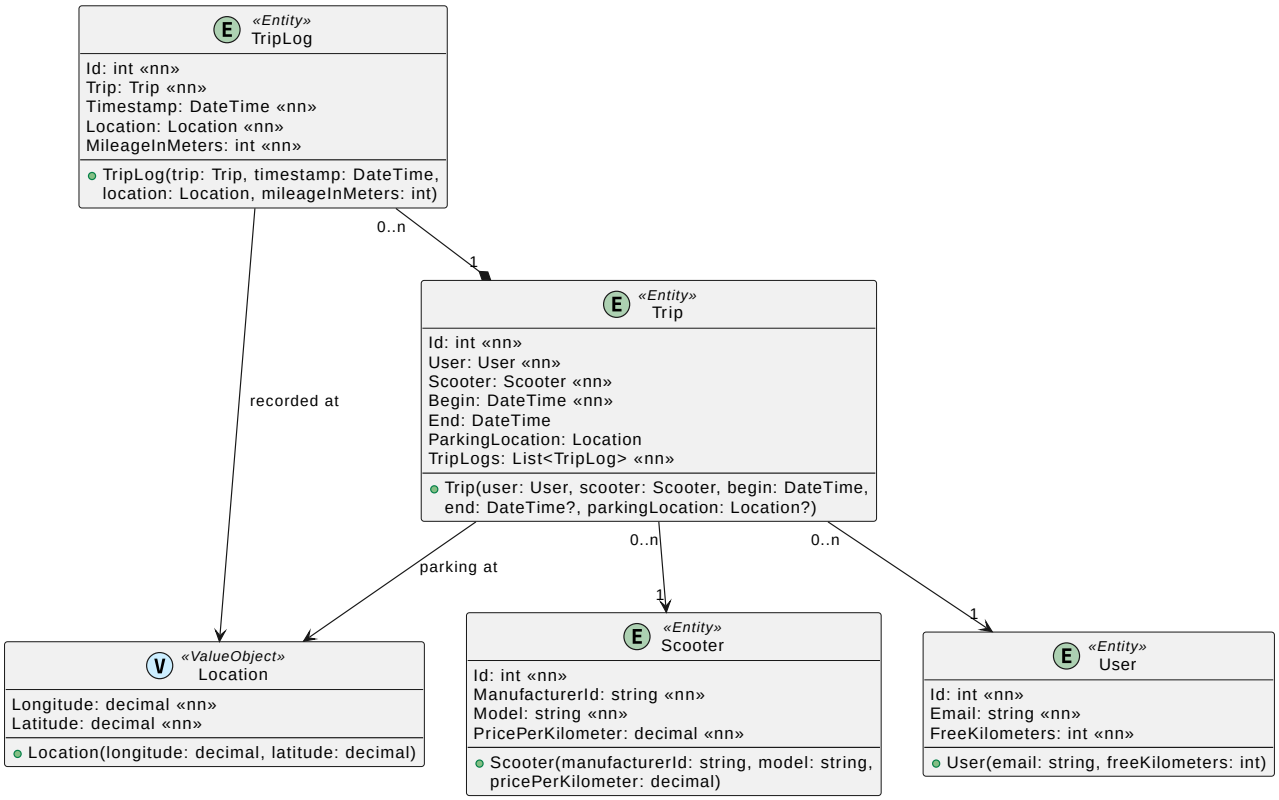
- Pay as you go: The customer pays per kilometer traveled.
- Included kilometers: The customer pays a monthly fee and receives a certain number of free kilometers. The free kilometers apply per trip. For example, if a customer has 2 free kilometers, 1 kilometer is charged for trip A, which is 3 kilometers long. For trip B, which is 4 kilometers long, 2 kilometers are charged.

A *trip* is started when the app is used to unlock the scooter. The scooter's integrated tracker transmits the position and mileage at regular intervals as a *TripLog*. This means that billing is possible even if there is no continuous GPS signal available (e.g., underpasses, etc.). The length of a trip (*Trip*) is then the difference between the lowest and highest mileage of the *TripLogs* for that trip.

The prices per kilometer traveled vary depending on the model and are stored in the *Scooter* class. Scooters with seats and a long range, for example, have a higher price per kilometer than

scooters without seats.

When a scooter is parked, the *Trip* is ended and the *End* and *ParkingLocation* fields are set. This makes it possible to see which trips are still in progress and which have already been completed.



Work assignment

A fully implemented DbContext is available in the files *src/SPG_Fachtheorie.Aufgabe2/Infrastructure/ScooterContext.cs*. The model is already implemented in *src/SPG_Fachtheorie.Aufgabe2/Model/Model.cs*. **You do not need to implement the model, but rather use it in your service methods.**

Implementation of service methods

Perform your implementations in *src/SPG_Fachtheorie.Aufgabe2/Services/ScooterService.cs*. Use the *ScooterContext* provided via dependency injection.

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

This method creates a trip in the database. The following boundary conditions should be checked:

- If the *userId* is not found, a *ScooterServiceException* with the text *Invalid user.* should be thrown.

- If the *scooterId* is not found, a *ScooterServiceException* with the text *Invalid scooter.* should be thrown.
- If there is an open trip for this scooter in the database, throw a *ScooterServiceException* with the text *Scooter is not parked.* You can recognize open trips by the fact that *Trip.End* has no value (*null*).

If all conditions are met, add the new trip to the database with the transferred user, scooter, and start time.

Use the provided unit test AddTripTest in test/SPG_Fachtheorie.Aufgabe2.Test/ScooterServiceTests.cs to check the correctness of your method.

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

This method should retrieve information about all trips that started within a certain period of time. A record is defined as the return type in your service method:

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

The *IsParked* property has the value *true* if *Trip.End* is not null. Retrieve the information required to create *TripDto* from the database.

Use the provided unit test GetTripInfosTest in test/SPG_Fachtheorie.Aufgabe2.Test/ScooterServiceTests.cs to check the correctness of your method.

public int CalculateTripLength(int tripId)

This method should calculate the length of a trip in meters. A trip has several TripLogs. These are points that are transmitted, for example, by a GPS module on the scooter. To calculate the length of a trip in meters, use the smallest and largest mileage (*TripLog.MileageInMeters*) of the log for this trip. The difference between these two values is the value you are looking for.

Use the provided unit test CalculateTripLengthTest in test/SPG_Fachtheorie.Aufgabe2.Test/ScooterServiceTests.cs to check the correctness of your method.

public decimal CalculatePrice(int tripId)

This method should calculate the price for a trip. The following values must be taken into account:

- A user has free kilometers (*User.FreeKilometers*). These must be deducted from the trip length.
- The price per kilometer is stored in *Scooter.PricePerKilometer*.

If the trip is not found, the price 0 must be returned.

Use the provided unit test CalculatePriceTest in test/SPG_Fachtheorie.Aufgabe2.Test/ScooterServiceTests.cs to check the correctness of your method.

Evaluation

Task 2 (26 points in total, 28%)	Points
AddTrip: The method correctly takes all boundary conditions into account.	2
AddTrip: The method correctly adds the new trip to the database.	1
AddTrip: The AddTripTest unit test runs successfully.	3
GetTripInfos: The method filters the data correctly.	1
GetTripInfos: The method queries correctly and projects the result onto the DTO class.	2
GetTripInfos: The unit test GetTripInfosTest runs successfully.	3
CalculateTripLength: The method filters the data correctly.	1
CalculateTripLength: The method performs the calculation correctly.	3
CalculateTripLength: The unit test CalculateTripLengthTest runs.	3
CalculatePrice: The method filters the data correctly.	1
CalculatePrice: The method performs the calculation correctly.	3
CalculatePrice: The unit test CalculatePriceTest runs.	3

Subtask 3: REST(ful) API

A RESTful API is to be implemented for the previous model of the scooter sharing app. When you start the project in *src/SPG_Fachtheorie.Aufgabe3*, an Endpoint Explorer is available at the URL *http://localhost:5080/swagger/index.html*. The database contains sample data so that you can test the functionality of your controllers.

Work assignment

Implement the following REST API routes. An empty controller is available for this purpose in *src/SPG_Fachtheorie.Aufgabe3/Controllers/TripsController.cs*. A *ScooterContext* filled with sample data is provided via dependency injection. You can implement the queries directly in the controller methods. Execute all queries non-blocking (with await and async). In case of errors, ensure an RFC-9457-compatible response by using the Problem() method with the appropriate status code.

GET /trips/{id}?includeLog=true

This REST API route should return a specific trip along with the log entries. If the optional parameter *includeLog* is specified with the value *true* (*false* is the default value), the response should contain the log entries. If the parameter is *false*, an empty array should be returned.

First, create the fields in the classes *src/SPG_Fachtheorie.Aufgabe3/Dtos/TripDto.cs* and *src/SPG_Fachtheorie.Aufgabe3/Dtos/TripLogDto.cs* so that the data corresponds to the following schema.

Expected HTTP responses:

HTTP Status	Condition
200	A *TripDto* with an empty array for *logs* if the parameter *includeLog* is false.
200	A *TripDto* with an array of *TripLogDto* if the parameter *includeLog* is true.
404	For an unknown ID. (including RFC-9457 problem details in the body)

Response for GET trips/1?includeLog=false

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

Response for 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}

This endpoint is intended to update the data at the end of a trip. The fields *End* and *ParkingLocation* are set to the corresponding values. Please note that the value of *Trip.End* must be *null* before the update. Trips that have already been completed may not be completed again. If the trip has already ended, return HTTP 400 with the message *Trip already ended.*

Use the command object in *src/SPG_Fachtheorie.Aufgabe3/Commands/UpdateTripCommand.cs* to type the payload. Make sure that only valid values for *Longitude* (-180 to +180) and *Latitude* (-90 to +90) can be transmitted. Ensure this by adding appropriate annotations for validation in the Command Object.

The trip with ID 4 is not completed; it can be used for testing.

Expected HTTP responses:

HTTP Status	Condition
204	No content if successfully updated.
400	If a trip that has already been completed is to be modified. (incl. RFC-9457 ProblemDetail in the body)
400	If the values for *Longitude* or *Latitude* are invalid. (including RFC-9457 problem details in the body)
404	For an unknown ID. (including RFC-9457 problem details in the body)

Payload for PATCH trips/4

```
{
```

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

Writing integration tests

Integration tests should be written in *test/SPG_Fachtheorie.Aufgabe3.Test/TripsControllerTests.cs*. Each line in the tables under *Expected HTTP responses* should be checked. Use the helper methods provided in *TestWebApplicationFactory* for this purpose:

Method	Description
InitializeDatabase	Creates an empty database and inserts values if necessary.
QueryDatabase	Allows queries to be made to the database.
GetHttpContent<T>	Performs a GET request and returns the result as type *T*.
PatchHttpContent<Tcmd>	Performs a PATCH request and returns the payload as *JsonElement* if applicable.

Evaluation

Task 3 (33 points in total, 36%)	Points
GET /trip/{id} filters the data correctly.	1
GET /trip/{id} delivers the correct behavior in the case of a non-existent ID.	1
GET /trip/{id} uses the query parameter includeLog correctly.	3
The DTO class TripDto was implemented correctly.	2
The DTO class TripLogDto was implemented correctly.	1
GET /trip/{id} returns the correct result in case of success for includeLogs = false.	2
GET /trip/{id} returns the correct result in case of success for includeLogs = true.	2
The integration test proves the correct implementation of GET /trip/{id} in case of success.	2
The integration test proves the correct implementation of GET /trip/{id} in error states.	2
The integration test for GET /trip/{id} runs through.	3
PATCH /trip/{id} correctly queries the values from the database.	1
PATCH /trip/{id} returns the correct behavior in the case of a non-existent ID.	1
PATCH /trip/{id} delivers the correct behavior in the case of a trip that has already been completed.	1

Task 3 (33 points in total, 36%)	Points
PATCH /trip/{id} updates the value in the database.	2
The UpdateTripCommand command class can type the payload.	1
The UpdateTripCommand command class has the defined validations.	1
The integration test proves the correct implementation of PATCH /trip/{id} in case of success.	2
The integration test proves the correct implementation of PATCH /trip/{id} in case of error states.	2
The integration test for PATCH /trip/{id} runs through.	3

Grades:

92–81 points: Sehr gut (1)
80–70 points: Gut (2)
69–58 points: Befriedigend (3)
57–46 points: Genügend (4)
Below 46 points: Nicht genügend (5)

Good luck!