HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Microsoft .NET Übungen

Mini-Projekt Auto-Reservation, v10.0

Inhaltsverzeichnis

Manuel Bauer
HSR Hochschule für Technik Rapperswil ■ Oberseestrasse 10 ■ CH-8640 Rapperswil
T +41 55 222 41 11 ■ F +41 55 222 44 00 ■ office@hsr.ch ■ www.hsr.ch

1 Aufgabenstellung / Administratives

1.1 Einführung

Sie bekommen als Vorgabe eine .NET Core Solution, welche strukturell einen kompletten Service Stack abbildet. Die Applikation ist aber noch nicht implementiert. Die Server-Applikation implementiert Service-Methoden für die Verwaltung von Auto-Reservationen einer Autoverleih-Firma.

Die Aufgabe wurde mit dem WPF-Teil aus dem Modul «Mobile- and GUI-Engineering» abgestimmt, wo ein optionales Projekt in Form eines WPF User Interface durchgeführt wird. Beide Projekte bauen aufeinander auf und können nahtlos kombiniert werden.

Ziele dieser Aufgabenstellung:

- Breite Anwendung von Technologien aus der Vorlesung
- Verteilte Applikationen mit Datenbank-Zugriff konzipieren und umsetzen
- Diskussionen über Software-Architektur anregen

Teams:

Das Projekt soll in 2er Teams durchgeführt werden. Einzelarbeiten und 3er Teams sind in Ausnahmefällen möglich. Wir werden in den Übungen eine Einschreibeliste („Gruppeneinteilung Microsoft-Technologien“) auflegen, in welcher sich jede Gruppe einschreiben und den gewünschten Abnahmetermin wählen kann.

1.2 Arbeitspakete

Die Applikation besteht aus mehreren Schichten, welche innerhalb der Gruppe in folgende Deliverables aufgeteilt werden. Es ist den Studierenden grundsätzlich freigestellt, ob die Implementation bottom-up oder top-down erfolgt.

Data Access Layer

1. Implementieren Sie den DAL mit Entity Framework Code First
2. Stellen Sie sicher, dass das resultierende Datenbankschema dem der Aufgabenstellung entspricht

Business Layer

1. Implementieren Sie den Business-Layer mit den CRUD-Operationen und diversen Checks. Die Update-Operationen sollen Optimistic-Concurrency unterstützen.
2. Schreiben Sie die geforderten Tests für den Business-Layer.

Service Layer

1. Definieren Sie die Service-Schnittstelle mit gRPC.
2. Implementieren Sie die Service-Operationen.
   Der Service-Layer ist verantwortlich für das Konvertieren der DTOs in Entitäten und umgekehrt, sowie das Error Handling.
3. Schreiben Sie die geforderten Tests für den Service-Layer.

Wichtig:
Lesen Sie die nachfolgenden Kapitel sorgfältig durch. Sie erhalten dort weitere Informationen zu den einzelnen Aufgaben. Die Erläuterungen sind nicht immer in der Reihenfolge, in der Sie die Aufgaben abarbeiten. Gehen Sie daher immer das ganze Kapitel durch, bevor Sie mit der Implementation starten.

1.3 Abgabe (letzte Semesterwoche)

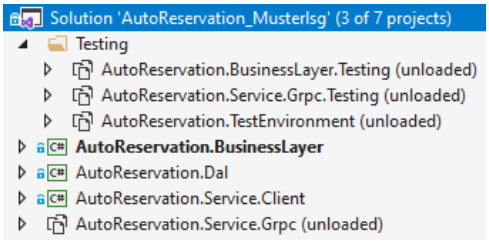
In der letzten Semesterwoche finden die Abgaben gemäss dem Plan „Gruppeneinteilung MsTe Miniprojekt“ statt. Bei der Abgabe des Miniprojektes muss jede Gruppe pünktlich zum Abnahmetermin eine lauffähige Version ihrer Implementation auf ihrem Notebook bereitstellen können, damit ein reibungsloser Ablauf und die Einhaltung des Terminplans gewährleistet werden kann. Etwas Verzögerung sollte mit eingerechnet werden.

Eine erfolgreiche Bewertung ist die Voraussetzung für die Zulassung zur Modulschlussprüfung. Es wird rechtzeitig eine Check-Liste für die Bewertung der Resultate veröffentlicht.

2 Erläuterungen

2.1 Analyse der Vorgabe

Öffnen Sie die Solution „AutoReservation.sln“ und analysieren Sie den bereits vorhandenen Code und dessen Struktur. Da bereits einiges an Funktionalität mitgeliefert wird und diese zum Teil auf Klassen aufbaut, welche erst noch entwickelt werden müssen, gibt es beim Kompilieren anfangs diverse Fehler. Die betroffenen Projekte können temporär «unloaded» werden (Beispiel: siehe Screenshot rechts).



In den meisten Entwicklungsumgebungen befindet sich diese Funktion im Kontextmenü (Rechtsklick auf entsprechendes Projekt) > «Unload Project».

Folgende Projekte sind in der abgegebenen Solution enthalten:

Projekt	**Beschreibung**
AutoReservation.BusinessLayer	Beinhaltet die Implementation der Methoden mit den CRUD-Operationen auf den Business-Entities. Dazu wird das *.Dal Projekt verwendet.
AutoReservation.Dal	Data Access Layer basierend auf Entity Framework Core.
AutoReservation.Service.Grpc	Projekt für die gRPC-Serviceschnittstelle. Implementiert die Service-Operationen, greift dazu auf den Business-Layer zu. Verantwortlich für die Konvertierung von Entities nach DTOs und zurück.
AutoReservation.Service.Client	Client-Library, in welche sämtliche Service Clients generiert werden.
AutoReservation.*.Testing	Testprojekt zur jeweiligen Schicht.

2.2 Datenbanksystem

Grundsätzlich ist die Wahl des verwendeten Datenbanksystems freigestellt. Die Vorgabe unterstützt per se zwei Systeme:

- Microsoft SQL Server
- SQLite In-Memory

Microsoft SQL Server unter Windows
Wählen Sie im Visual Studio Installer unter «Individual components» die Option «SQL Server Express 20xx LocalDB» um eine schlanke Variante einer Microsoft SQL Server Instanz zu installieren.

Der Connection String für den Zugriff auf die Datenbank befindet sich in der Datei «appsettings.json» und muss je nachdem auf die eigenen Begebenheiten angepasst werden.

Damit die Konfiguration nicht in allen Projekten einzeln gemacht werden muss, wurde die «appsettings.json» einmal im Ordner «Assets» definiert und dann in die einzelnen Projekte gelinkt.

In Bezug auf den Connection String empfiehlt es sich, dass sämtliche Gruppenmitglieder sich auf einen spezifischen Connection String einigen. Ansonsten wird der Austausch des Quellcodes schwierig. Bei Schwierigkeiten mit Suche des richtigen Connection Strings kann folgende Seite konsultiert werden: http://www.connectionstrings.com/.

Microsoft SQL Server im Container (Window und Linux)
Vor Allem unter Linux empfiehlt sich die Verwendung eines Docker Containers. Dieser kann nach Verwendung wieder relativ einfach entfernt werden.

Zur Installation können Sie sich an dieser Anleitung orientieren:
https://docs.microsoft.com/en-us/sql/linux/quickstart-install-connect-docker?view=sql-server-ver15&pivots=cs1-bash

Diese funktioniert auch unter Windows, bei den Commands muss aber jeweils «sudo» weggelassen werden.

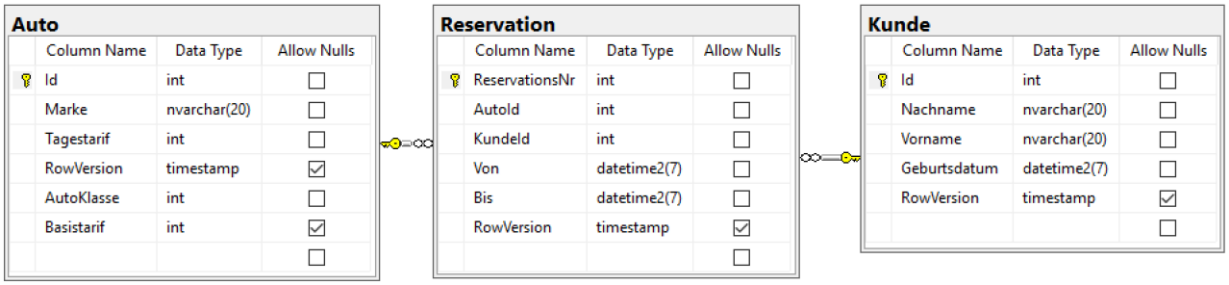
SQLite In-Memory
Hierbei ist keine zusätzliche Installation nötig.

Beachten Sie aber folgendes:
Es wird hiermit nicht möglich sein, die Requirements für «Optimistic Concurrency» zu implementieren, da es von diesem Provider nicht unterstützt wird. Der grösste Teil kann aber trotzdem entwickelt werden. Der Teil mit der Optimistic Concurrency kann mit dem Betreuer abgesprochen werden

2.3 Datenbankstruktur

Die Datenbankstruktur ist relativ einfach gehalten. Es gibt zwei Tabellen mit Stammdaten (Auto, Kunde) sowie eine Transaktionstabelle (Reservation), welche Autos und Kunden in Form einer Reservation verknüpft.

Das Erstellen der Datenbank soll nicht per Hand oder vorgegebenem Skript passieren, sondern automatisch durch Entity Framework Code First passieren. Die resultierende Struktur muss, abgesehen von der Feld-Reihenfolge, genau dem untenstehenden Diagramm entsprechen. Dies umfasst Name, Datentypen, Nullable, Schlüssel, Fremdschlüssel, etc. Abweichungen müssen mit dem Betreuer abgesprochen werden.



Von einem Auto existieren drei Ausprägungen. Diese werden über die Spalte „AutoKlasse“ unterschieden. Das Feld „Tagestarif“ muss bei allen Autos erfasst sein, der „Basistarif“ existiert nur für Wagen der Luxusklasse, ist für diese aber zwingend.

Diese Werte sind für die Unterscheidung vorgesehen:

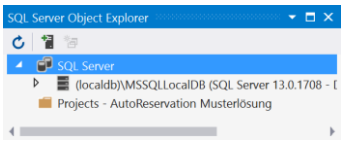
Wert	Typ
0	Luxusklasse
1	Mittelklasse
2	Standard

2.3.1 Datenbankzugriff

Es gibt mehrere Varianten, um auf die Datenbank zuzugreifen und diese zu verwalten:

Microsoft Visual Studio (Windows)

Über den Menüpunkt View > SQL Server Object Explorer. Unter dem SQL Server sollte nun mindestens (localdb)\MSSQLLocalDB erscheinen. Falls nicht gelistet, müsste dieser über das Plus-Symbol oben hinzugefügt werden.



Microsoft SQL Server Management Studio (Gratis, Windows)

https://docs.microsoft.com/en-us/sql/ssms

Dieses gehört zur Produktpalette von Microsoft SQL Server und kann in der neuesten Version als stand-alone Installer heruntergeladen werden. Sämtliche SQL Server Features werden von diesem Tool unterstützt.

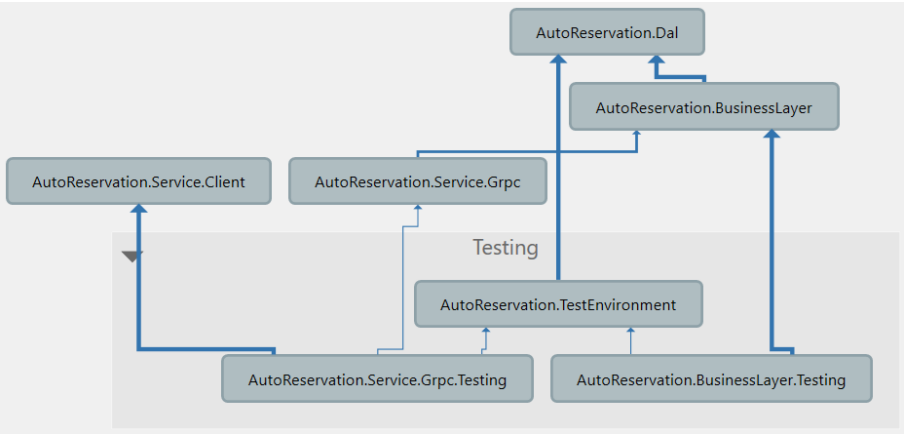
Microsoft Azure Data Studio (Gratis, Windows / Linux)
https://docs.microsoft.com/en-us/sql/azure-data-studio
Parallel zu Microsoft SQL Server Management Studio baut Microsoft eine relativ neue Electron App zur Verwaltung von SQL Server Datenbanken.

JetBrains DataGrip (Education License, Windows / Linux)
https://www.jetbrains.com/datagrip
Ein plattform-neutrales Tool zur Verwaltung diverser Datenbanken von JetBrains. Studierende können die gesamte Palette von JetBrains-Tools gratis verwenden, DataGrip inklusive.

Weitere Tools
Nebst den offiziellen Tools existiert eine Vielzahl von Drittanbieter-Software. Mit dieser kann selbstverständlich auch gearbeitet werden, Unterstützung kann aber hier nur bedingt gegeben werden.

2.4 Applikationsarchitektur

Die Architektur der Applikation ist bereits im Groben vorgegeben. Sie bewegen sich in den vorgegebenen Strukturen. In der Abbildung unten sind die vorhandenen Assemblies / Projekte der abgegebenen Solution zu finden.



2.4.1 Test-Projekte

Aus obigem Diagramm ist ersichtlich, dass für den Business- und den Service-Layer je ein Test-Projekt existiert. Die in der Aufgabenstellung verlangten Testklassen sind bereits vorgegeben.

Allgemeine Testing-Funktionalität ist im Test-Environment-Projekt vorhanden. Auf den Einsatz eines Mocking-Frameworks wurde der Übersichtlichkeit halber verzichtet.

2.5 Alternativen

In Kapitel 5 Alternativen / Weitere Möglichkeiten werden noch mögliche alternative Ansätze / weitere Möglichkeiten erwähnt, die in Betracht gezogen werden können. Diese sind aber optional und sind für Studenten gedacht, die C# / .NET bereits besser kennen. Es wird empfohlen, dass das Projekt zuerst soweit gelöst wird, dass die Bewertungskriterien erfüllt sind und erst dann Erweiterungen eingebaut werden.

3 Implementation

3.1 Optimistic Concurrency

Die in den nachfolgenden Kapiteln beschriebenen Update-Methoden müssen nach dem Prinzip Optimistic Concurrency[1] implementiert werden. Es existieren verschiedenste Ansätze, um das Problem zu lösen. Die meisten davon erfordern Anpassungen / Erweiterungen im Data Access Layer, einige sogar bis ins GUI. Die drei gängigsten Varianten sind diese:

1. Variante «Timestamp»
   Alle Tabellen erhalten einen Zeitstempel, welcher bei jedem Update automatisch von der Datenbank aktualisiert wird. Beim Speichern wird geprüft, ob der Zeitstempel in der Tabelle noch gleich dem des zu speichernden Objektes ist. Falls nicht, wurde der Datensatz in der Zwischenzeit geändert. Dieser Ansatz wird hier empfohlen.

2. Variante «Original / Modified»
   Das gelesene Objekt wird vor der ersten Änderung geklont. Später beim Speichern werden beide Versionen, die originale und die veränderte, mitgegeben. Durch das originale Objekt kann festgestellt werden, ob der Datensatz in der Zwischenzeit geändert hat. Das Klonen kann auf verschiedenen Layers passieren:
   User Interface, Service Layer oder Business Layer

3. Variante «Client-side Change Tracking»
   Änderungen werden clientseitig aufgezeichnet und dem Service-Interface in einer vollständigen Änderungsliste mitgeteilt. Undo-Redo-Funktionalität ist in dieser Variante praktisch ohne Mehraufwand dabei.

Die vorliegende Vorgabe wurde unter Verwendung von Variante 1 gebaut. Die Begründung liegt bei der Einfachheit der Lösung. Hier eine kurze Beurteilung der verschiedenen Ansätze:

Var.	Beurteilung
1.	Relativ geringer Aufwand. Es muss aber berücksichtigt werden, dass der Zeitstempel immer korrekt über die Schichten weitergegeben wird und nie verloren/vergessen geht.
2.	Ebenfalls relativ einfach. Das Klonen gibt aber etwas Aufwand bei der Implementation und auch zur Laufzeit. a. Klonen passiert erst da, wo auch effektiv Änderungen am Objekt passieren. Dafür muss beim Speichern das originale und das veränderte Objekt durchgereicht werden. b. Dem Service muss einen Cache eingebaut werden. Dies kann aber sehr schnell viel Ressourcen verbrauchen. Ausserdem muss der Cache ständig aktualisiert werden. Dies wird sehr schnell sehr kompliziert. c. Siehe b.
3.	Client-seitiges Change Tracking (Self Tracking Entities) sind zwar sehr elegant, aber auch nur mit viel Aufwand umzusetzen.

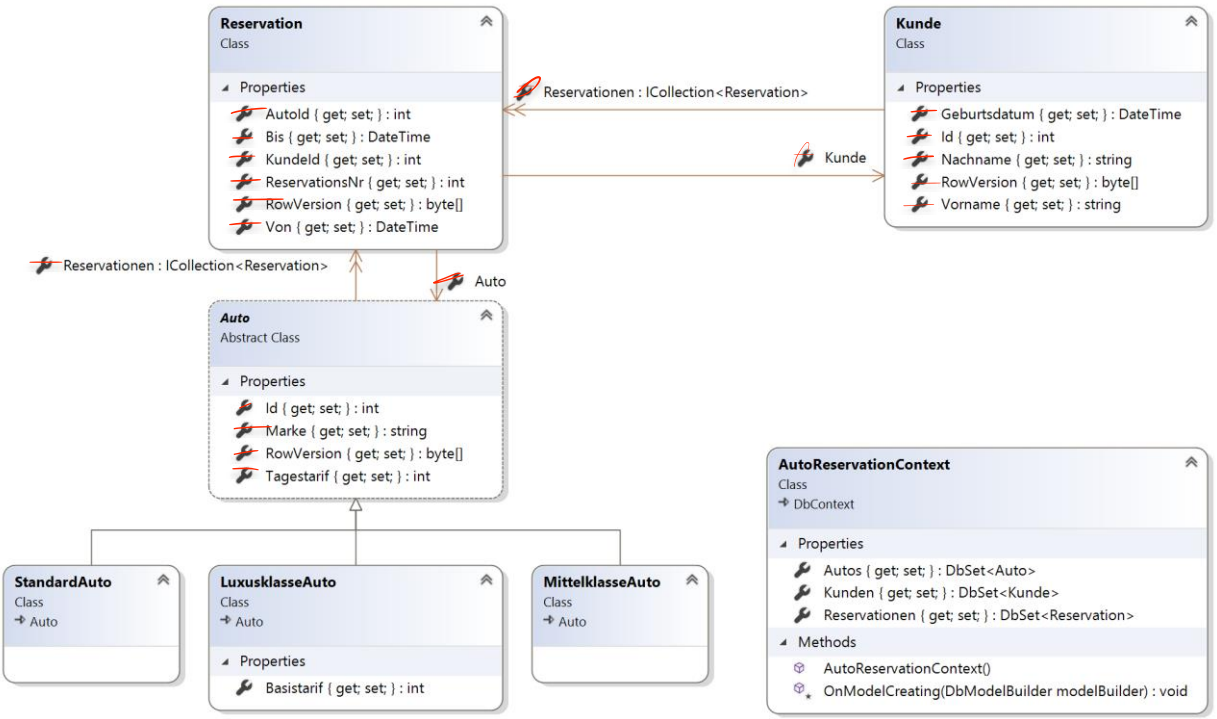
[1] Optimistic Concurrency: https://blogs.msdn.microsoft.com/marcelolr/2010/07/16/optimistic-and-pessimistic-concurrency-a-simple-explanation/

3.2 Data Access Layer (AutoReservation.Dal)

Dieser Layer beinhaltet die Datenzugriffsschicht und sollte mit Entity Framework Code First implementiert werden. Die Idee ist, das Datenmodell sowie den Database Context (Subklasse von DbContext) zu entwickeln und die Datenbank danach basierend auf dem Modell erstellen zu lassen. Vergleichen Sie die erzeugte Struktur mit dem oben vorgegebenen Datenmodell.

Das Datenmodell ist unten bereits dargestellt. Darin ist ersichtlich, dass die Unterscheidung der Auto-Klasse nicht mehr über das Feld `AutoKlasse` erfolgt. Im Modell der Business Entities wird dies durch drei von Auto abgeleitete Klassen realisiert. Die `AutoKlasse` dient implizit im Modell als Diskriminator und verschwindet somit aus der Klasse `Auto`.

Der `DbContext` namens `AutoReservationContext` dient als Zugriffspunkt auf die Datenstruktur.



Wichtige Hinweise:

- Da schon relativ viel Infrastruktur gegeben ist, können Abweichungen vom Diagramm dazu führen, dass an anderen Stellen im Code Anpassungen gemacht werden müssen
- Um zwischen der Verwendung von SQL Server und In-Memory zu wechseln können Sie in der Methode `AutoReservationContextBase.OnConfiguring` den entsprechenden Code ein-/auskommentieren

3.3 Business Layer (AutoReservation.BusinessLayer)

Im Business Layer findet der Zugriff auf den Data Access Layer (DAL) statt, sprich hier werden die Daten vom DAL geladen und verändert. Im Business Layer soll die unten definierte Business Logik implementiert werden. Folgende Operationen müssen für alle drei Entitäten zur Verfügung gestellt werden. Beachten Sie auch nachfolgende Requirements an den Business Layer in den Sub-Kapiteln.

- Alle Entitäten lesen
- Eine Entität anhand des Primärschlüssels lesen
- Einfügen
- Update
- Löschen

Hilfs-Codefragmente für das ADO.NET Entity Framework:

```
// Insert
context.Entry(auto).State = EntityState.Added;
// Update
context.Entry(auto).State = EntityState.Modified;
// Delete
context.Entry(auto).State = EntityState.Deleted;
```

3.3.1 Requirement «Optimistic Concurrency Update»

Jede Update-Methode muss nach dem Optimistic Concurrency-Prinzip implementiert werden. Entity Framework löst bei einer Optimistic Concurrency-Verletzung eine Exception vom Typ `DbUpdateConcurrencyException` aus. Im Falle des Auftretens einer solchen Exception soll eine `OptimisticConcurrencyException` (existiert bereits) geworfen werden, welche die neuen in der Datenbank vorhandenen Werte beinhaltet.

Für das Handling dieser Exception kann die bereits bestehende Methode `ManagerBase.CreateOptimisticConcurrencyException(…)` direkt im catch-Block aufgerufen und weiter geworfen werden.

3.3.2 Requirement «Date Range Check»

Eine Reservation muss 24 Stunden oder mehr dauern. Prüfen Sie dies und stellen Sie auch sicher, dass das «bis» Datum nicht vor dem «von» Datum liegt. Falls nicht muss eine `InvalidDateRangeException` ausgelöst werden (noch nicht implementiert).

Kapseln Sie die Logik in eine separate Methode, damit das Schreiben von Tests (siehe unten) einfacher bleibt.

3.3.3 Requirement «Availability Check»

Beim Erstellen / Updaten einer Reservation muss geprüft werden, ob ein Auto für den gewünschten Zeitraum zur Verfügung steht. Autos können nahtlos (Ende = Start), aber nicht überlappend gebucht werden. Falls ein Auto nicht verfügbar ist, muss eine `AutoUnavailableException` ausgelöst werden (noch nicht implementiert).

Kapseln Sie die Logik in eine separate Methode, damit das Schreiben von Tests (siehe unten) einfacher bleibt.

3.4 Service Layer (AutoReservation.Service.Grpc)

Der Service Layer ist die eigentliche gRPC-Serviceschnittstelle. Dieser Layer operiert auf den Business Layer Klassen. Der Service-Layer ist in dieser einfachen Applikation nicht viel mehr als ein «Durchlauferhitzer». In grösseren Projekten kann hier aber durchaus noch Funktionalität – z.B. Sicherheitslogik – enthalten sein. Nichts desto trotz muss er hier zwei wichtige Aufgaben erfüllen:

- Das Umwandeln von DTOs (Common) in Entities (DAL) und umgekehrt
- Bekannte Exceptions fangen und in saubere Status Codes übersetzen

3.4.1 Proto-Files allgemein

Da das Handling sowie der korrekte Einbau der Proto-Files relativ fehleranfällig ist, wurde dies bereits vorgenommen. Es soll für jede Entity ein separater Service entstehen, deshalb wurden drei Proto-Files erstellt:

- auto.proto
- kunde.proto
- reservation.proto (referenziert auto.proto / kunde.proto)

3.4.2 Proto-Files: Services

Der gRPC Service muss für jede Entität – Auto, Kunde und Reservation –folgende CRUD[2]-Operationen unterstützen:

- Alle Entitäten lesen
- Eine Entität anhand des Primärschlüssels lesen
- Einfügen
- Update
- Löschen

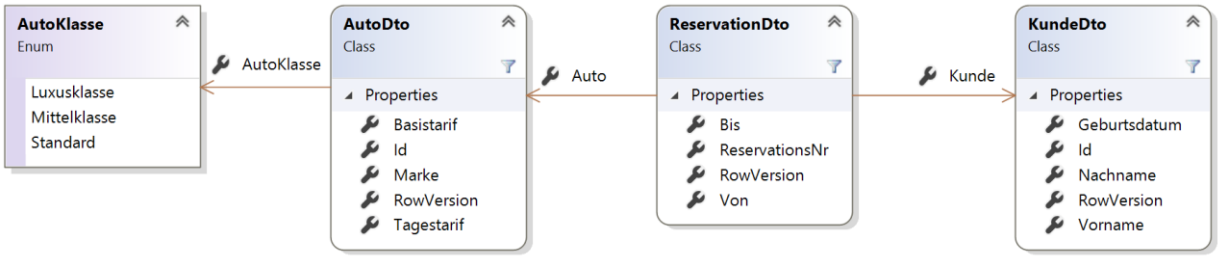
Damit die Verfügbarkeit eines Autos auch im Vorfeld abgeklärt werden kann, muss eine zusätzliche Methode auf dem Interface definiert werden. Dadurch lässt sich vermeiden, dass zu viele Fehler provoziert werden.

[2] CRUD: Create, Read, Update, Delete

3.4.3 Proto-Files: Messages

Für die in den Proto-Files definierten Messages werden nur wenige Vorgaben gemacht. Im Service muss eine Umwandlung von den Entitäten im Data Access Layer auf die Klassen, welche aus den Proto-Files generiert werden, erfolgen. Diese Logik wurde mit der Vorgabe abgegeben (DTO Converter, siehe unten). Damit diese reibungslos funktioniert, sollte ein Klassenmodell gemäss untenstehendem Screenshot angestrebt werden.

Dabei ist zu beachten, dass ein Enumerator `AutoKlasse` eingeführt wurde. Damit wird die auf dem Data Access Layer definierte Vererbungshierarchie wieder aufgehoben.



3.4.4 DTO Converter

Die im Projekt vorhandene Klasse `DtoConverter` bietet diverse Erweiterungsmethoden an, um DTOs in Entitäten und umgekehrt zu konvertieren. Die gleiche Funktionalität steht auch für Listen von DTOs respektive Listen von Entitäten zur Verfügung.

Hier ein Beispiel für die Anwendung:

```
// Entität konvertieren
Auto auto = context.Autos.First();
AutoDto autoDto = auto.ConvertToDto();
auto = autoDto.ConvertToEntity();

// Liste konvertieren
List<Auto> autoList = context.Autos.ToList();
List<AutoDto> autoDtoList = autoList.ConvertToDtos();
autoList = autoDtoList.ConvertToEntities();
```

3.4.5 Fehlerbehandung

Behandeln Sie sämtliche in den Requirements definieren Spezialfälle. Das heisst, es müssen alle auf dem Business Layer geworfenen Exceptions gefangen und eine sinvolle Service-Exception mit passendem Status Code verpackt werden.

3.5 Service Client (AutoReservation.Service.Client)

In diesem Projekt wurden die Proto-Files erneut gelinkt. An dieser Stelle werden aber nur die gRPC Client-Klassen generiert. Das Projekt soll in den Tests verwendet werden, um mit dem Service zu kommunizieren.

4 Tests

Nachfolgend werden die Anforderungen an die Tests dokumentiert. Im Projekt „AutoReservation.TestEnvironment“ befindet sich einiges an Infrastruktur um die Testumgebung zu initialisieren.

4.1 Test-Infrastruktur

Die Klasse und Methoden der minimal zu schreibenden Tests wurde bereits gegeben. Sie können sich also auf die Implementation der Tests fokussieren.

Jede Testklasse leitet von der abstrakten Basisklasse `TestBase` ab. Diese stellt sicher, dass die Datenbank initialisiert wird. Schauen Sie sich die Implementation genauer an. Diese unterscheidet, ob der `AutoReservationContext` mit einer SQL Server- oder einer In-Memory-Datenbank arbeitet und initialisiert die Datenbank entsprechend.

Sollten Sie sich für einen anderes Datenbanksystem entscheiden, werden Sie an dieser Stelle Erweiterungen vornehmen müssen.

Allgemein ist die Initialisierungslogik so aufgebaut, dass die Daten vor jedem Test gelöscht und neu erstellt werden. Der Identity Seed wird dabei immer zurückgesetzt. Das bedeutet, dass die Testdaten beim initialisieren jedes Mal mit einem inkrementellen Primärschlüssel beginnend bei 1 erstellt werden. Es darf also davon ausgegangen werden kann, dass immer ein Auto oder ein Kunde mit der ID = 1 existiert.

Als Test-Framework wurde xUnit verwendet. Studieren Sie die Dokumentation unter https://xunit.github.io um detailliertere Informationen zu erhalten. Assertions können über die Klasse `Xunit.Assert` definiert werden.

4.2 Business-Layer (AutoReservation.BusinessLayer.Testing)

Diese Tests sollen relativ früh implementiert werden und eine gewisse Sicherheit geben, dass die Applikation und vor allem die Datenbank-Verbindung und -Abfragen in ihren Grundzügen funktioniert. Mindestens folgende Tests müssen implementiert sein:

- Update Kunde
- Update Auto
- Update Reservation

Ausserdem müssen folgende im Business Layer erwähnten Requirements genau getestet werden.

4.2.1 Requirement «Date Range Check»

Analysieren Sie, welche Wertebereiche zu Problemen führen und schreiben Sie zu jedem einen sinnvollen Test. Dies sollte mindestens fünf Tests ergeben (2 gültige Kombination 3 ungültige Kombinationen von Datumswerten).

4.2.2 Requirement «Availability Check»

Analysieren Sie, was für unterschiedliche Konstellationen möglich sind und schreiben Sie zu jeder einen Test. Dies sollte mindestens fünf Tests ergeben (2+ gültige Kombinationen, 3+ ungültige Kombinationen).

4.3 Service-Layer (AutoReservation.Service.Grpc.Testing)

Studieren Sie als erstes das vorgegebene Konstrukt in der Projektvorgabe. Im Ordner «Common» befindet sich eine abstrakte Basisklasse für alle Tests (`ServiceTestBase`), welche den gRPC Service startet und ein Property namens «Channel» anbietet. Über dieses Property kann dann ein gRPC Client instanziiert werden.

Der Service Layer soll vollumfänglich getestet werden. Im Mindesten müssen folgende Operationen für alle drei Entitäten (Autos, Kunden, Reservationen) überprüft werden:

- Abfragen einer Liste
- Suche anhand des Primärschlüssels
- Suche anhand eines ungültigen Primärschlüssels
- Einfügen
- Löschen
- Updates
- Abfrage der Verfügbarkeit eines Autos (je einmal true / false als Antwort)

Ausserdem muss das Fault-Handling darauf hin geprüft werden, ob Exceptions im Service sauber behandelt werden.

- Updates mit Optimistic Concurrency Verletzung auf allen 3 Entitäten
- Insert mit ungültigem Date Range (Reservation)
- Update mit ungültigem Date Range (Reservation)
- Insert mit nicht verfügbarem Auto (Reservation)
- Update mit nicht verfügbarem Auto (Reservation)

Dies ergibt in Summe also im Mindesten 27 Tests für den Service-Layer.

5 Alternativen / Weitere Möglichkeiten

Wie zu Beginn erwähnt, können auch alternative Ansätze verfolgt oder das Projekt noch ausgebaut werden. In den nachfolgenden Kapiteln werden einige Punkte aufgegriffen. Es empfiehlt sich, Abweichungen von der Aufgabenstellung mit dem Betreuer zu diskutieren.

5.1 Allgemein

5.1.1 Dependency Injection

Bei Interesse darf Dependency Injection über das gesamte Projekt hinweg angewendet werden. Am einfachsten kann dafür der in .NET Core enthaltene Dependency Injection Container verwendet werden.

5.1.2 Repository Pattern

Die vorliegende Implementation ist nicht unbedingt sehr kompatibel mit Test-Driven Development. Das Problem ist, dass die Komponenten unter Test schlecht gemockt werden können. Damit dies möglich wäre, müsste mehr gegen Interfaces programmiert werden. Das Repository Pattern[3] ist durch seinen Aufbau an dieser Stelle eine ideale Alternative.

5.2 Datenbank

5.2.1 Entity Framework Migrations

Die Datenbank könnte auch durch den Einsatz von Entity Framework Migrations generiert werden, anstatt diese immer zu löschen und neu zu generieren.

5.2.2 Vererbung

Wie vielleicht bemerkt wurde, kann mit dem aktuellen Design der Datenbank die `AutoKlasse` nicht verändert werden. Das DTO, welches über die Service-Schnittstelle übermittelt wird, lässt dies zwar zu, aber das Entity Framework wirft eine Exception. Dies hat mit der implementierten Vererbung zu tun. Ändert beim DTO die `AutoKlasse`, wird beim Konvertieren vor dem Speichern eine andere Subklasse von Auto instanziert. Entity Framework lässt aber Updates nur dann zu, wenn das gelieferte Objekt noch dem ursprünglichen Typen entspricht (u.A. wegen möglichem Datenverlust).

Dieses Szenario müsste von Hand abgebildet werden, z.B. durch manuelles Löschen und neu erstellen des Objektes oder über das direkte Ausführen von SQL Statements / Stored Procedures auf der Datenbank.

5.3 Testing

5.3.1 Mocking

Um nicht gegen eine Datenbank testen zu müssen (unter Puristen verpönt) sollten Mocks eingesetzt werden. Dies verringert die Abhängigkeit zu Systemkomponenten wie z.B. Datenbank, Dateisystem, Netzwerkverbindungen, uva.

Es gibt diverse Mocking-Frameworks in .NET. Weit verbreitet ist Moq[4].

---

[3] Repository Pattern: http://martinfowler.com/eaaCatalog/repository.html
[4] Moq: https://github.com/Moq/moq4

5.3.2 Test-driven Development (TDD)

Es werden in diese Projekt Tests gefordert, damit Sie sich auch in C# / .NET mit dieser Disziplin auseinandergesetzt haben. Wer einen Schritt weitergehen möchte, kann sich auch gerne im Test-Driven Development (TDD) üben und diese Vorgehensweise wählen.

Die geforderten Tests sind lediglich ein Minimum und können noch ausgebaut werden. Der Einsatz von Mocks wäre mit dieser Herangehensweise empfehlenswert.