

## Komplexe Datentypen – von Owned und COMPLEX Types über Kollektionen zu Json

Dozent: Dr. Thomas Hager, Berlin,  
im Auftrag der Firma GFU Cyrus AG  
20.10.2025 – 22.10.2025

Neben Assoziation und Vererbung als wesentlichen Konzepten der Objektorientierung gibt es noch einen weiteren Aspekt, der bisher nicht beachtet wurde:

Es geht um Entitäten, die **Beziehungen zu untergeordneten Klassen** haben (z.B. Kunde-Kundendetail-Kundenadresse – Kunde ist die Hauptklasse). Solche Beziehungen lassen sich mit den Konzepten der Owned Types, der Complex Types oder durch Abbildung in JSON-Felder in Datenbankstrukturen abbilden.

Zu diesem Themenbereich gehört auch die Untersuchung von Objekten, die komplexe Attribute haben, z.B. Listen von einfachen oder komplexen Daten.



Im Folgenden:

# **KOMPLEXE DATENTYPEN – VON OWNED UND COMPLEX TYPES ÜBER KOLLEKTIONEN ZU JSON**

# Hintergrund: Komplexe Datentypen und ihre Umsetzung in der Datenbankstruktur

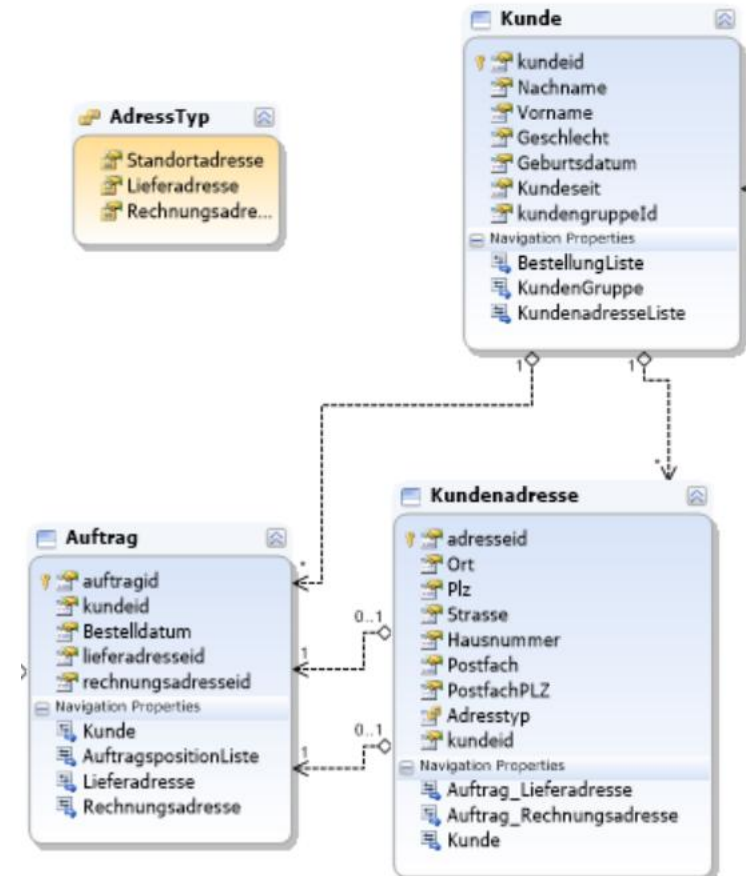
- Der Konflikt zwischen dem objektorientierten Paradigma (Klassen, Entities) und dem relationalen Paradigma (Tabellen, Datenbank-Struktur) kann auf einfachste Weise gelöst werden, indem jeder Klasse eine Tabelle zugeordnet wird. Das führt jedoch bei großen Datenmengen und vielfältigen Verknüpfungen zu Performance-Problemen. Klassisch wird das Problem durch Denormalisierung gelöst
- Objekte, die in einer Datenbank gespeichert werden sollen, können in drei Kategorien unterteilt werden:
  - Unstrukturierte Objekte, die nur einen Wert haben. Beispiele für Datentypen sind int, GUID, string, IPAdress → so genannte „primitive Typen“
  - Objekte, die strukturiert sind und so mehrere unterschiedliche Merkmale eines realen Objekts abbilden können: Kunde (Name, Vorname, Telefon usw.), Auftrag (Kunde, Auftragsdatum usw.), Auftragsposition, Artikel. Die einzelnen Objekte sind durch einen Schlüsselwert eindeutig identifizierbar. Solche Objekte werden Entity-Typen (entity types) genannt
  - Objekte, die strukturiert sind und so mehrere unterschiedliche Merkmale eines realen Objekts abbilden, aber über keinen Schlüssel verfügen: Kundendetails, Adressen, Koordinaten
- Es gibt Fälle, in denen eine bestimmte Klasse nicht für sich allein existiert, sondern zusätzliche Informationen in Bezug auf ein anderes Hauptmodell enthält. Diese Funktionalität geht davon aus, dass wir mehrere Typen haben, aber einer davon ist der Eigentümer. Andere abhängige oder eigene Entitätstypen sind Teil des Eigentümertyps (Owned Type) und können ohne diesen nicht existieren.
- Bis EFC 8 wurde mit so genannten „Owned Types“ der zweite Typ abgebildet. Ab EFC 8 können auch „Complex Types“ definiert werden, die alle drei Fälle umfassen

# Owned Entity Type (Eigentümer-Entity-Typ)

- Im Beispiel gehören zu einem Auftrag die Lieferadresse und die Rechnungsadresse. Die Entity Auftrag ist „Eigentümer“ der Lieferadresse und der Rechnungsadresse

```
private void OwnedTypeMapping(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Auftrag>()
        .OwnsOne(p => p.Lieferadresse);
    modelBuilder.Entity<Auftrag>()
        .OwnsOne(p => p.Rechnungsadresse);

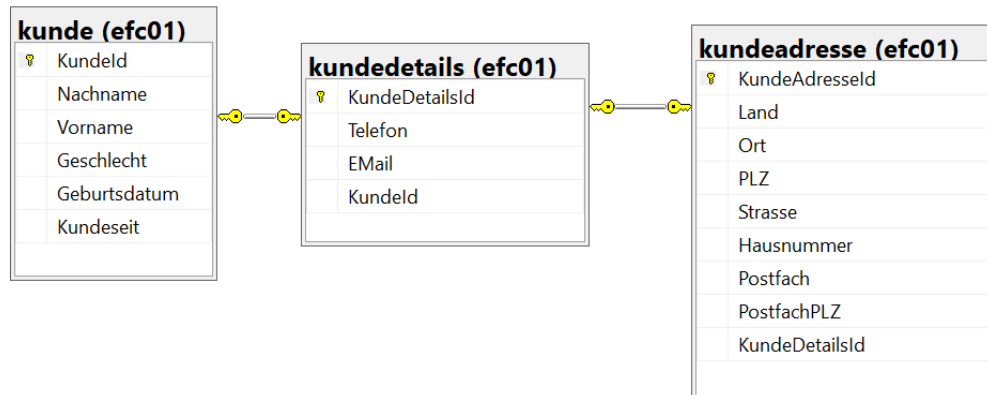
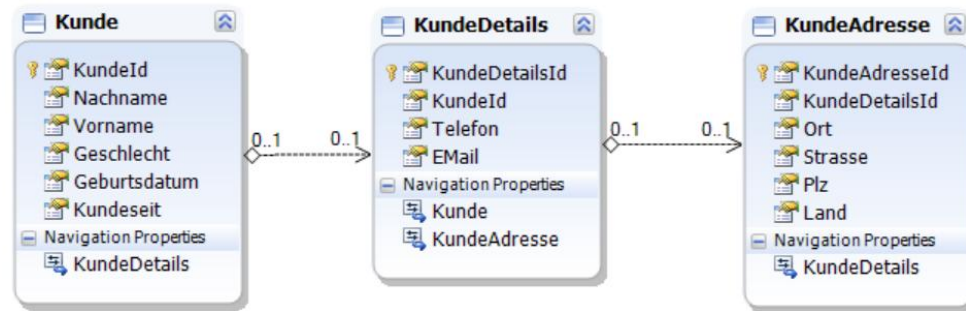
    modelBuilder.Entity<Auftrag>()
        .Navigation(p => p.Lieferadresse)
        .IsRequired();
}
```



Die Verwendung von Owned Types kann helfen, die Datenbank zu organisieren, indem gemeinsame Datengruppen in eigene Entity-Typen umgewandelt werden, was den Umgang mit gemeinsamen Datengruppen, wie z. B. Adresse und so weiter, im Code erleichtert. Die Navigationseigenschaften des eigenen Typs, wie z. B. Rechnungsadresse, werden automatisch erstellt und mit Daten gefüllt, wenn die Entität gelesen wird. Es besteht keine Notwendigkeit für eine Include-Methode oder eine andere Form des Ladens von Beziehungen.

## Von einer 1:1:1 Beziehung zu einem komplexen Datentyp

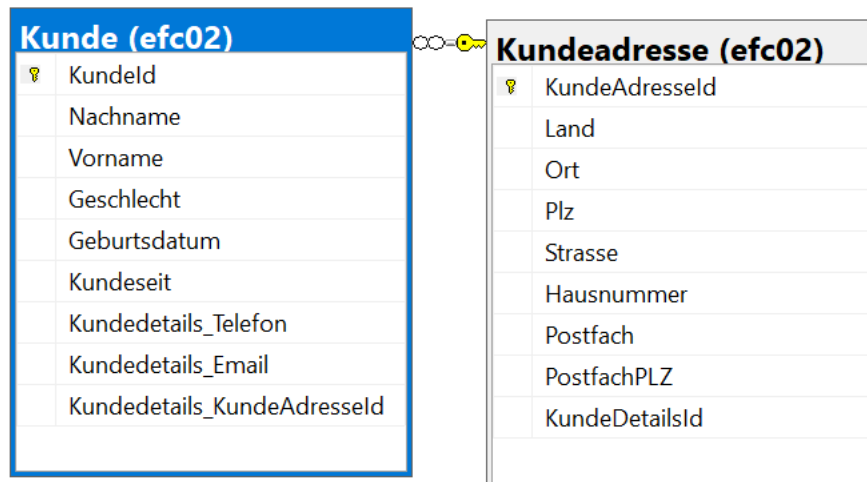
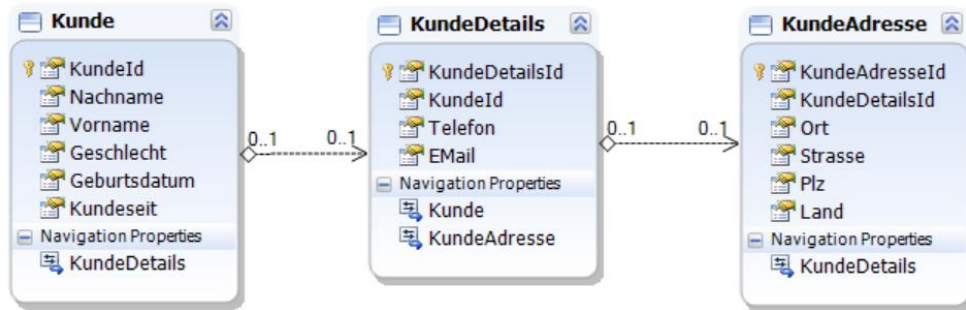
- Jeder Kunde hat bestimmte Basisdaten und gewisse Detaildaten
- Im einfachsten Fall können wir alle Entities in 3 Tabellen abbilden



```
private void RelationshipsMapping(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<KundeDetails>()
        .HasOne(x => x.Kunde).WithOne(op => op.Kundedetails)
        .OnDelete(DeleteBehavior.Cascade)
        .HasPrincipalKey(typeof(KundeDetails), @"KundeId")
        .HasForeignKey(typeof(Kunde), @"KundeId")
        .IsRequired(false);
    modelBuilder.Entity<KundeAdresse>()
        .HasOne(x => x.Kundedetails).WithOne(op => op.KundeAdresse)
        .OnDelete(DeleteBehavior.Cascade)
        .HasForeignKey(typeof(KundeAdresse), @"KundeDetailsId")
        .IsRequired(false);
}
```

# Etwas kompakter durch Owned Entity Types

Die Entity Kunde eignet die Entity Kundedetails:



```

[Owned()]
[Table("Kunde", Schema = "efc02")]
public partial class KundeDetails
{
    [StringLength(20)]
    [Unicode(false)]
    public string Telefon { get; set; }

    [StringLength(20)]
    [Unicode(false)]
    public string Email { get; set; }

    public int? KundeAdresseId { get; set; }
    public virtual KundeAdresse KundeAdresse { get; set; } = null;
}
  
```

```

private void OwnedTypeMapping(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Kunde>()
        .OwnsOne(p => p.Kundedetails);

    modelBuilder.Entity<Kunde>()
        .Navigation(p => p.Kundedetails)
        .IsRequired();
}
  
```

# Nur noch eine Tabelle mit einem Feld für die Aufnahme von JSON-Daten

- Die drei Entity-Klassen

```
[Table("Kunde", Schema = "efc03")]
public partial class Kunde
{
    public virtual int KundeId { get; set; }
    [Required]
    [StringLength(20)]
    public virtual string Nachname { get; set; }
    [StringLength(20)]
    public virtual string Vorname { get; set; }
    [StringLength(1)]
    [Unicode(false)]
    public virtual string Geschlecht { get; set; }
    [Precision(0)]
    public virtual DateOnly? Geburtsdatum { get; set; }
    [Precision(0)]
    public virtual DateOnly? Kundeseit { get; set; }
    public virtual KundeDetails Kundedetails { get; set; } = null;
}
```

```
[Table("Kunde", Schema = "efc03")]
public partial class KundeDetails
{
    [StringLength(20)]
    [Unicode(false)]
    public string Telefon { get; set; }
    [StringLength(20)]
    [Unicode(false)]
    public string Email { get; set; }
    public Adresse Anschrift { get; set; }
}
```

```
[Table("Kunde", Schema = "efc03")]
public partial class Adresse
{
    [StringLength(50)]
    public virtual string Land { get; set; }
    [StringLength(50)]
    public virtual string Ort { get; set; }
    [StringLength(10)]
    [Unicode(false)]
    public virtual string Plz { get; set; }
    [StringLength(50)]
    public virtual string Strasse { get; set; }
    [StringLength(10)]
    public virtual string Hausnummer { get; set; }
}
```

# Konfigurierung der Generierung von JSON-Daten

- Jetzt werden alle Detaildaten in einem Feld Kundedetails (string → nvarchar(max)) zusammengefasst.

```
private void OwnedTypeMapping(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Kunde>().OwnsOne(
        owner => owner.Kundedetails, ownedNavigationBuilder =>
        {
            ownedNavigationBuilder.ToJson();
            ownedNavigationBuilder.OwnsOne(contactDetails =>
                contactDetails.Anschrift);
        }
    );
}
```

Kunde (efc03)			
	Spaltenname	Datentyp	NULL-Werte zulassen
🔑	Kundeld	int	<input type="checkbox"/>
	Nachname	nvarchar(20)	<input type="checkbox"/>
	Vorname	nvarchar(20)	<input checked="" type="checkbox"/>
	Geschlecht	varchar(1)	<input checked="" type="checkbox"/>
	Geburtsdatum	datetime2(0)	<input checked="" type="checkbox"/>
	Kundeseit	datetime2(0)	<input checked="" type="checkbox"/>
	Kundedetails	nvarchar(MAX)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>



# Beispiel: Die Detaildaten für eine ausgewählte Kundin im JSON-Format

- Generierung eines Kundenobjekts

```
Kunde k1 = new Kunde()
{
    Nachname = "Paul",
    Vorname = "Gabriele",
    Geschlecht = "w",
    Kundeseit = new DateOnly(2016, 2, 1),
    Geburtsdatum = new DateOnly(1978, 5, 1),
    Kundedetails = new KundeDetails()
    {
        Email = "gpaul@projektdb.com",
        Telefon = "030 234567",
        Anschrift = new KundeAdresse() { Strasse = "Hausvogteiplatz",
            Hausnummer = "27b", Plz = "10765", Ort = "Berlin", Land = "Deutschland" }
    }
};
```

- Nach der Übertragung in die Datenbank:

	Kundeld	Nachna...	Vorname	Geschlecht	Geburtsdat...	Kundeseit	Kundedetails
▶	1	Paul	Gabriele	w	1978-05-01...	2016-02-01...	{"Email":"gpaul@projektdb.com","Telefon":"0...

```
{"Email":"gpaul@projektdb.com", "Telefon":"030 234567",
"Anschrift": {"Hausnummer":"27b", "Land":"Deutschland", "Ort":"Berlin", "Plz":"10765,
"Strasse":"Hausvogteiplatz"}}
```

# Aufruf von Daten einschließlich des JSON-Feldes

- Die Daten werden in ein komplexes Objekt übertragen:

```
using (var dbctx = new EFCTestDBContext())
{
    var kundin = dbctx.KundeListe.Find(1);
    Display($"{kundin?.Nachname}, {kundin?.Vorname}");
    Display($"Beispiele für Details:");
    Display($"... Telefon: {kundin?.Kundedetails.Telefon}");
    Display($"... Ort: {kundin?.Kundedetails.Anschrift.Ort}");
    Display($"... Straße: {kundin?.Kundedetails.Anschrift.Strasse}");
}
```

```
exec sp_executesql N'SELECT TOP(1) [k].[Kundeld], [k].[Geburtsdatum], [k].[Geschlecht],
[k].[Kundeseit], [k].[Nachname], [k].[Vorname], JSON_QUERY([k].[Kundedetails],'$')
FROM [efc03].[Kunde] AS [k]
WHERE [k].[Kundeld] = @__p_0,N'@__p_0 int',@__p_0=1
```

Paul, Gabriele  
Beispiele für Details:  
... Telefon: 030 234567  
... Ort: Berlin  
... Straße: Hausvogteiplatz

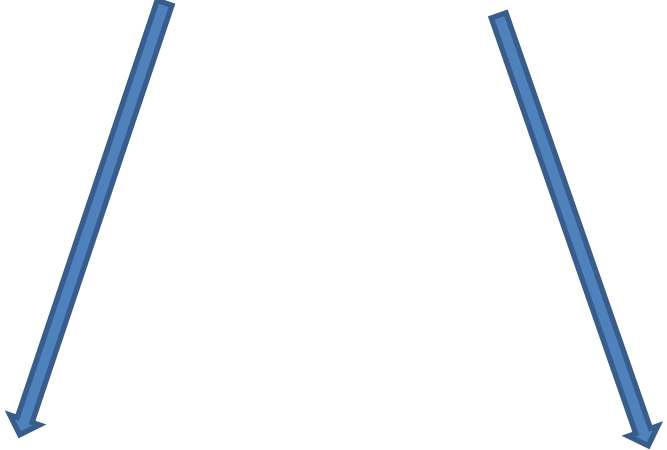
# Complex Types (ab EFC 8)

- Komplexe Typen erweitern das Spektrum möglicher Objekte
- Ein komplexer Typ kann durch eine Annotation oder per Fluid API festgelegt werden:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

public class Kunde
{
    public int Kundeld { get; set; }
    public required string Nachname { get; set; }
    //... Weitere Eigenschaften
    public List<Auftrag> AuftragListe { get; } = new();
}

public class Auftrag
{
    public int AuftragId { get; set; }
    public DateTime Auftragsdatum { get; set; }
    [Required]
    public Kunde Kunde { get; set; }
    [Required]
    public Adresse LieferAdresse { get; set; }
    [Required]
    public Adresse RechnungsAdresse { get; set; }
    //... Andere Eigenschaften
}
```

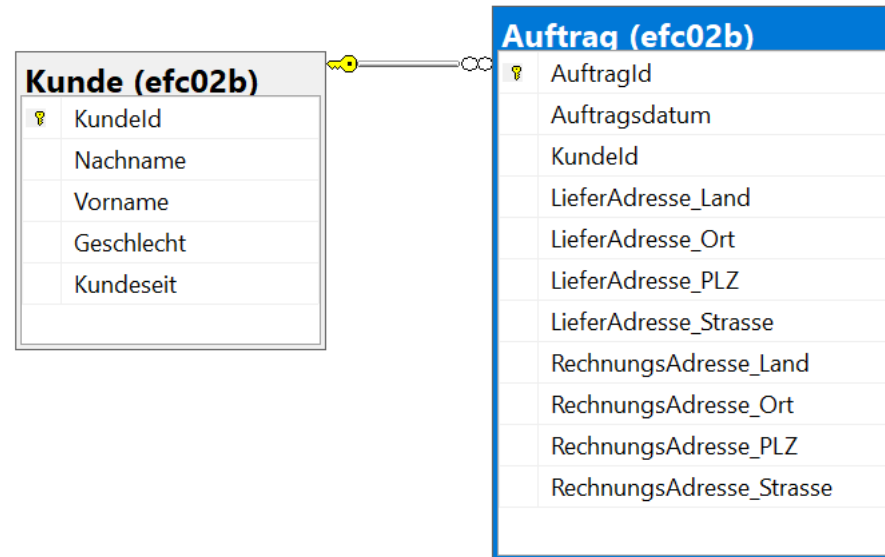


```
[ComplexType]
public class Adresse
{
    public string Strasse { get; set; }
    public string Ort { get; set; }
    public string Land { get; set; }
    public string PLZ { get; set; }
}
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Auftrag>(x => {
        x.ComplexProperty(y => y.Lieferadresse, y => { y.IsRequired(); });
        x.ComplexProperty(y => y.Rechnungsadresse, y => { y.IsRequired(); });
    });
}
```

## Ergebnis der Nutzung eines „Complex Type“

- Wir erhalten zwei Tabellen. Die Auftragsdaten werden jetzt ergänzt durch die Lieferadress- und die Rechnungsadressdaten. Dadurch können zwar Redundanzen entstehen, aber der Zugriff auf diese Daten kann beschleunigt werden.



# Primitive Kollektionen

- Primitive Kollektionen (primitive collections) erlauben es, zu einer Entity eine Liste primitiver Daten (Daten mit einem Datentyp (int, string, Boolean, DateOnly etc.)) in einem Tabellenfeld zu speichern, ohne dass eine zusätzliche Tabelle generiert werden muss.
- Beispiel: Zu jedem Mitarbeiter soll eine Liste seiner Qualifikationen (Skills) hinzugefügt werden:

```
[Table("Mitarbeiter", Schema = "efc02c")]
[Index(nameof(Prsnr), IsUnique = true)]
public class Mitarbeiter
{
    public int MitarbeiterId { get; set; }

    [StringLength(4)]
    public string Prsnr { get; set; }
    [Required]
    [StringLength(50)]
    public string Nachname { get; set; }
    [StringLength(50)]
    public string? Vorname { get; set; }
    [Precision(0)]
    public DateOnly? Geburtsdatum { get; set; }
    [StringLength(1)]
    [Unicode(false)]
    [AllowedValues("f", "m", "d", "x")]
    public string Geschlecht { get; set; }

    public List<string> Skills { get; set; }
```

Ab EFC 8! Wir benötigen weder eine Annotation noch einen Eintrag in der Methode OnModelCreating

Ergebnis:

	MitarbeiterId	Nachname	Vorname	Skills
1	1	Paul	Gabriele	["T-SQL","C#","Python"]
2	2	O'Reilly	John	["Java","C\u002B\u002B"]

## Erweiterung: JSON

- Die meisten relationalen Datenbanken unterstützen Spalten, die JSON-Dokumente enthalten. Das JSON in diesen Spalten kann mit Abfragen durchforstet werden. Dies ermöglicht z. B. das Filtern und Sortieren nach den Elementen der Dokumente sowie die Projektion von Elementen aus den Dokumenten in die Ergebnisse. JSON-Spalten ermöglichen es relationalen Datenbanken, einige der Eigenschaften von Dokumentendatenbanken anzunehmen, wodurch eine nützliche Mischung aus beiden entsteht.
- Im SQL Server gibt es keinen expliziten Datentyp JSON (im Unterschied zum Beispiel zu PostgreSQL), Daten im JSON-Format werden in einem Textfeld `nvarchar(max)` gespeichert. Jedoch kann mit spezifischen Befehlen auf diese Daten zugegriffen werden (siehe folgendes Beispiel)

## Noch eine JSON-Möglichkeit: Umsetzung einer Entity mit einer Liste komplexer Objekte

- Es wurde gezeigt, dass eine Liste primitiver Datentypen (List<string> Skills) in einer Entity ohne Probleme in ein Datenbankfeld überführt werden kann.
- Wenn aber statt des primitiven Datentyps eine Liste von Objekten (List<Skill> Skills) in einer Entity implementiert ist, ist die Lösung komplexer. In diesem Fall wird die Liste serialisiert und in ein Datenbankfeld überführt.

```
[Table("Mitarbeiter", Schema = "efc03b")]
public class Mitarbeiter
{
    public int MitarbeiterId { get; set; }
    [Required]
    [StringLength(50)]
    public string Nachname { get; set; }
    [StringLength(50)]
    public string? Vorname { get; set; }
    [Precision(0)]
    public DateOnly? Geburtsdatum { get; set; }
    [StringLength(1)]
    [Unicode(false)]
    [AllowedValues("w", "m", "d", "x")]
    public string Geschlecht { get; set; }

    public List<Skill> Skills { get; set; }
}
```

```
public class Skill
{
    public required string SkillName { get; set; }
    [AllowedValues("gering", "mittel", "hoch")]
    public string Grad { get; set; }
}
```

# Umsetzung einer Entity mit einer Liste komplexer Objekte - Ergebnis

- Die Umsetzung führt jetzt zu einer erweiterten Skill-Liste:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<Mitarbeiter>()
        .OwnsMany(e => e.Skills, builder => builder.ToJson());
}
```

	Nachname	Skills
1	Paul	[{"Grad": "mittel", "SkillName": "T-SQL"}, {"Grad": "hoch", "SkillName": "C#"}, {"Grad": "hoch", "SkillName": "Python"}]
2	O'Reilly	[{"Grad": "gering", "SkillName": "Java"}, {"Grad": "mittel", "SkillName": "C\u002B\u002B"}]

```
SELECT [m].[Nachname], [m].[Skills], [m].[MitarbeiterId] FROM [efc03b].[Mitarbeiter] AS [m]
WHERE EXISTS (
    SELECT 1
    FROM OPENJSON([m].[Skills], '$') WITH (
        [Grad] nvarchar(max) '$.Grad',
        [SkillName] nvarchar(max) '$.SkillName'
    ) AS [s]
    WHERE [s].[Grad] = N'hoch')
ORDER BY [m].[MitarbeiterId]
```

Nachname	Skills	MitarbeiterId
Paul	[{"Grad": "mittel", "SkillName": "T-SQL"}, {"Grad": "hoch", "SkillName": "C#"}, {"Grad": "hoch", "SkillName": "Python"}]	1