

Vertiefung:
Hinzufügen Ändern Löschen und Änderungsverfolgung

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



ENTITÄTEN ERSTELLEN, MODIFIZIEREN UND SICHERN

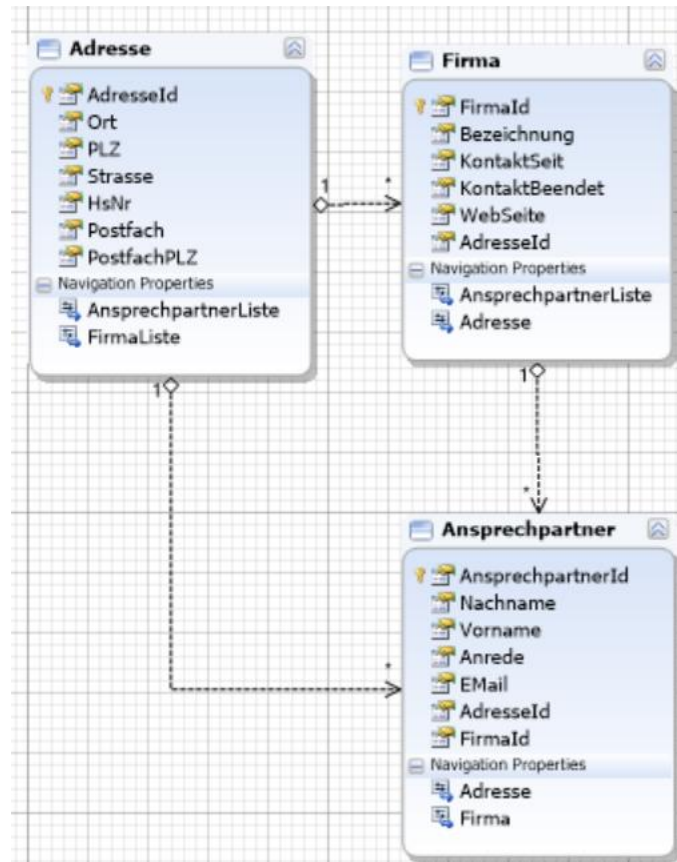
Entitäten erstellen, modifizieren und sichern

- Alle Modifikationen von Entities (d.h.: Erstellen, Ändern, Löschen) werden mit der DbContext-Methode `SaveChanges()` an die Datenbank übertragen, wobei im Hintergrund die vom EF Core - Provider zur beteiligten Datenbank erstellten SQL-Kommandos `UPDATE`, `INSERT` und `DELETE` vom Datenbankmanagementsystem (DBMS) ausgeführt werden
 - Ergänzte Entitäten werden mit dem SQL-Kommando `INSERT` in die Datenbanktabelle eingefügt.
 - Geänderte Entitäten erhalten mit dem SQL-Kommando `UPDATE` neue Werte.
 - Aus der Datenbank geladene und dann gelöschte Entitäten werden über das SQL-Kommando `DELETE` aus der Datenbanktabelle entfernt
- Intern verfolgt ein `ChangeTracker` die Modifikationen
- Mit Hilfe des SQL Profilers oder per Logging (siehe `onConfiguring` in der `DbContext`-Klasse) lassen sich die Operationen in der Datenbank verfolgen

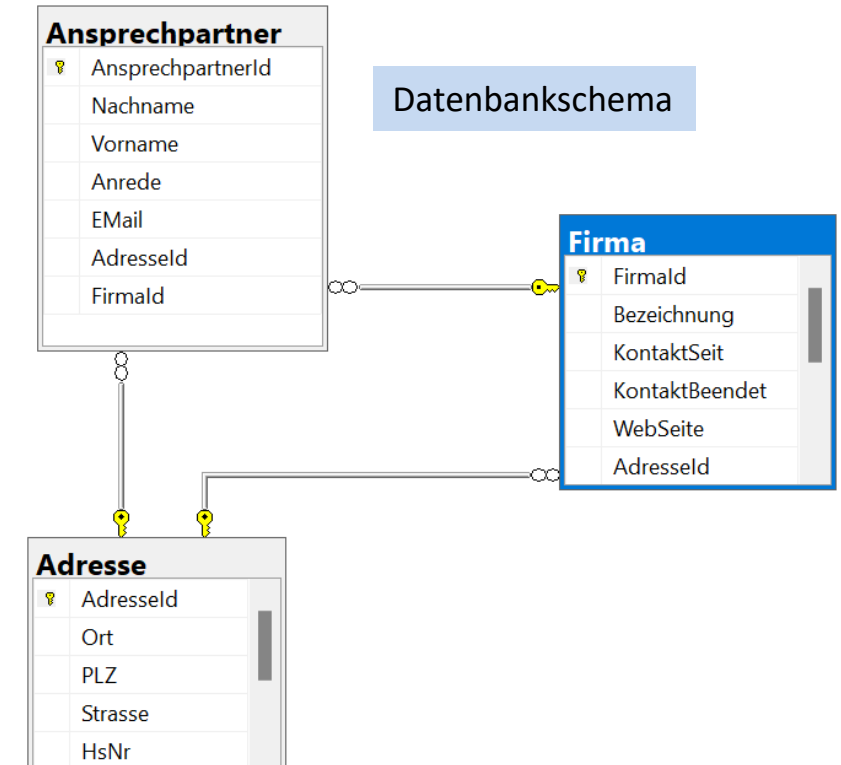
Beispiel: Eine Firma-Datenbank

- Im Folgenden demonstrieren wir die Methoden mit folgendem Beispiel: In einer Datenbank werden Firmen mit ihren jeweiligen Ansprechpartnern und Adressen gespeichert.
- Jede Firma kann genau eine Adresse haben, jedoch mehrere Ansprechpartner. Auch die Ansprechpartner haben jeweils eine Adresse.

Klassendiagramm



Datenbankschema



Hinzufügen von Daten (Insert)

- Hinzufügen eines einzelnen Datensatzes

```
using (var dbctx = new EFCTestDBContext())
{
    //Neue Adresse:
    Adresse adr = new Adresse() { Ort = "München", PLZ = "80331" };
    //Modifikation angeben
    dbctx.AdresseListe.Add(adr);
    //In der Datenbank speichern
    dbctx.SaveChanges();
}
```

- Mehrere Datensätze

```
List< Adresse> adrListe = new List<Adresse>();
adrListe.Add(new Adresse{Ort = "München", PLZ = "80333"});
adrListe.Add(new Adresse{Ort = "Berlin", PLZ = "10178"});
using (var dbctx = new EFCTestDBContext())
{
    dbctx.AdressListe.AddRange(adrListe);
    dbctx.SaveChanges();
}
```

Update

- Wir ergänzen eine einzelne Adresse durch Strasse und Hsnr

```
using (var dbctx = new EFCTestDBContext())
{
    //Ausgewählte Adresse:
    Adresse adr = dbctx.AdressListe.Where(a => a.Ort = "Berlin" && a.PLZ = "10178").First();
    //Adressmerkmale ergänzen:
    adr.Strasse = "Alexanderstraße";
    adr.HsNr = "3 ";

    //In der Datenbank speichern
    dbctx.SaveChanges();
}
```

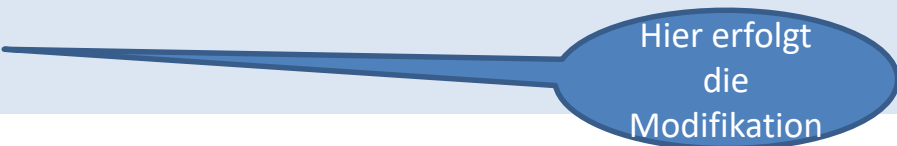
Mehrfach-Update

- In diesem Beispiel wird zunächst außerhalb des Kontext eine Liste zu modifizierender Entities generiert, anschließend wird innerhalb des DbContext die Liste durchlaufen und eine Zuordnung zu einer Entity der db.AnteilungsListe erzeugt.

```
private void UpdateMultipleRecords()
{
    List<Abteilung> abtListe = new List<Abteilung>();

    abtListe.Add(new Abteilung {AbteilungId = 1, Abteilungsbezeichnung= „Verkauf“ });
    abtListe.Add(new Abteilung {AbteilungId = 2, Abteilungsbezeichnung = „Ankauf“ });
    abtListe.Add(new Abteilung {AbteilungId = 3, Abteilungsbezeichnung = „Personal/HR“ });

    using (EFCoreContext dbctx= new EFCoreContext())
    {
        foreach (var item in abtListe)
        {
            var abt= dbctx.AnteilungsListe.Where(f => f.AnteilungsId == item. AnteilungsId).FirstOrDefault();
            if (abt != null)
                abt. Abteilungsbezeichnung = item. Abteilungsbezeichnung ;
        }
        dbctx.SaveChanges();
    }
}
```



Hier erfolgt
die
Modifikation

Löschen und mehrfaches Löschen

- Einfaches Löschen

```
Abteilung abt;  
using (var dbctx = new EFCTestDBContext())  
{  
    Abteilung abt = dbctx.AnteilungListe.Where(d => d. Anteilungsbezeichnung == "Verkauf").First();  
    dbctx.AnteilungListe.Remove(abt);  
    dbctx.SaveChanges();  
}
```

- Mit RemoveRange kann auch eine ganze Liste gleichartiger Entities entfernt werden

```
//Deleting Multiple Records  
using (var dbctx = new EFCTestDBContext())  
{  
    List<Abteilung> abtListe = db.AnteilungListe.Take(2).ToList();  
    dbctx.AnteilungListe.RemoveRange(abtListe);  
    dbctx.SaveChanges();  
}
```


MASSENAKTUALISIERUNG UND STAPELVERARBEITUNG

Massenupdate

- In der Tabelle skills soll der Zusatz „ Fortgeschritten“ eingetragen werden bei allen Datensätzen mit SkillId >= 10 und <= 20:
- Der einfache Update-Befehl führt, wenn mehrere DS gleichzeitig geändert werden sollen, zur Umsetzung der DbContext-Anweisung zur Aufspaltung in einzelne SQL-Anweisungen. Das kann zu starken Performance-Einbußen führen.

```
var skills = dbctx.SkillListe
    .Where(s => s.SkillsId >= 10 && s.SkillsId <= 20);
foreach (Skill item in skills) {
    skills.FirstOrDefault(s => s.SkillsId == item.SkillsId).Skill1 = item + " Fortgeschritten ";
}
dbctx.SaveChanges();
```

- Deshalb wurde der Befehl ExecuteUpdate eingeführt.

```
dbctx.SkillListe
    .Where(s => s.SkillsId >= 10 && s.SkillsId <= 20)
    .ExecuteUpdate(setter => setter.SetProperty(s => s.Skill1,
                                                s => s.Skill1 + " Fortgeschritten"));
```

ÄNDERUNGSVERFOLGUNG: DER CHANGETRACKER


Ein erstes Beispiel für eine Änderungsverfolgung

- Jede DbContext -Instanz verfolgt **Änderungen** nach, die an **Entitäten** vorgenommen wurden. Diese **nachverfolgten Entitäten** bestimmen wiederum die **Änderungen** an der Datenbank, wenn SaveChanges aufgerufen wird.
- Der ChangeTracker ermöglicht mittels DebugView die Anzeige der geplanten Änderungen vor dem Speichern.
- Beispiel: Einer Adresstabelle soll eine neue Adresse hinzugefügt werden:

```
using (var dbctx = new EFCTestDbContext())
{
    //Neue Adresse:
    Adresse adr = new Adresse() { Ort = "München", PLZ = "80331" };
    dbctx.AdresseListe.Add(adr);

    //Einschalten des ChangeTrackers
    dbctx.ChangeTracker.DetectChanges();
    Display(dbctx.ChangeTracker.DebugView.LongView);

    //dbctx.SaveChanges();
}
```



```
//Vor dem Speichern:
Adresse {Adresseld: -2147482647} Added
Adresseld: -2147482647 PK Temporary
HsNr: <null>
Ort: 'München'
PLZ: '80331'
Postfach: <null>
PostfachPLZ: <null>
Strasse: <null>
AnsprechpartnerListe: []
FirmaListe: []
```

Das Programm – ohne SaveChanges() – mit ChangeTracker

```
Firma? firma = dbctx.FirmaListe
    .Include(f => f.Adresse)
    .Include(f => f.AnsprechpartnerListe)
    .First(f => f.FirmaId == 1);
```

```
// Änderung von Firmendaten
firma.Bezeichnung = "Eigen GmbH & Co.KG";
firma.WebSeite = "www.eigen-elektronik.com";

//Ansprechpartner Weigen --> Eigen
Ansprechpartner a = firma.AnsprechpartnerListe.Single(x => x.Nachname=="Weigen");
if (a != null)
    a.Nachname = "Eigen";

// Einen neuen Ansprechpartner hinzufügen
Adresse ad = firma.Adresse;
firma.AnsprechpartnerListe.Add(
    new Ansprechpartner
    {
        Nachname = "Leopold",
        Vorname = "Oskar",
        Anrede = "Herr",
        EMail="oleopold@eigen.de",
        AdresseId=ad.AdresseId
    });

// Einen Anspruchspartner entfernen
var apToDelete = firma.AnsprechpartnerListe.Single(e => e.Nachname == "Müller");
dbctx.Remove(apToDelete);
```

```
dbctx.ChangeTracker.DetectChanges();
Display(dbctx.ChangeTracker.DebugView.LongView);
//Dbctx.SaveChanges();
```

Nach dem Speichern

- Nach dem Speichern der neuen Adresse kann eine neue Firma angelegt werden, der auch die neue Adressid hinzugefügt werden kann.

```
Adresse {AdressId: 2} Unchanged
  AdressId: 2 PK
  HsNr: <null>
  Ort: 'Köln'
  PLZ: '51105'
  Postfach: <null>
  PostfachPLZ: <null>
  Strasse: <null>
  AnsprechpartnerListe: [{AnsprechpartnerId: -2147482647}]
  FirmaListe: [{FirmaId: 6}]
Ansprechpartner {AnsprechpartnerId: -2147482647} Added
  AnsprechpartnerId: -2147482647 PK Temporary
  AdressId: 2 FK
  Anrede: 'Herr'
  EMail: <null>
  FirmaId: 6 FK
  Nachname: 'Meissner'
  Vorname: 'Jean'
  Adresse: {AdressId: 2}
  Firma: {FirmaId: 6}
Firma {FirmaId: 6} Modified
  FirmaId: 6 PK
  AdressId: 2 FK Modified
  Anforderungen: 'Web-Applikation mit Datenbank' Modified
  Bezeichnung: 'Bruno AG' Modified
  FirmaDiscriminator: 'Auftraggeber' Modified
  KontaktBeendet: <null> Modified
  KontaktSeit: '01.12.2021 00:00:00' Modified
  Kosten: <null> Modified
  Leistungsangebot: <null> Modified
  Volumen: 180000 Modified
  Webseite: 'www.brunoundsoehneag.com' Modified
  Adresse: {AdressId: 2}
  AnsprechpartnerListe: [{AnsprechpartnerId: -2147482647}]
```

Nach dem Speichern

- Nach dem Speichern der neuen Adresse kann eine neue Firma angelegt werden, der auch die neue Adressid hinzugefügt werden kann.

```
using (var dbctx = new EFCTestDBContext())
{
    //Neue Adresse:
    Adresse adr = new Adresse() { Ort = "München", PLZ = "80331" };
    dbctx.AdresseListe.Add(adr);
    dbctx.SaveChanges();

    //Neue Firma:
    Firma firma = new Firma() { Bezeichnung="WiGeoGis",
        Webseite= "WiGeoGis.com",
        FirmaDiscriminator="Lieferant",
        Leistungsangebot="Geodaten",
        Kosten=250456.75,
        AdressId =adr.AdressId };
    dbctx.FirmaListe.Add(firma);

    dbctx.ChangeTracker.DetectChanges();
    Display(dbctx.ChangeTracker.DebugView.LongView);

    //dbctx.SaveChanges();
}
```

```
Adresse {AdressId: 2} Unchanged
  AdressId: 2 PK
  HsNr: <null>
  Ort: 'Köln'
  PLZ: '51105'
  Postfach: <null>
  PostfachPLZ: <null>
  Strasse: <null>
  AnsprechpartnerListe: [{AnsprechpartnerId: -2147482647}]
  FirmaListe: [{FirmaId: 6}]
Ansprechpartner {AnsprechpartnerId: -2147482647} Added
  AnsprechpartnerId: -2147482647 PK Temporary
  AdressId: 2 FK
  Anrede: 'Herr'
  EMail: <null>
  FirmaId: 6 FK
  Nachname: 'Meissner'
  Vorname: 'Jean'
  Adresse: {AdressId: 2}
  Firma: {FirmaId: 6}
Firma {FirmaId: 6} Modified
  FirmaId: 6 PK
  AdressId: 2 FK Modified
  Anforderungen: 'Web-Applikation mit Datenbank' Modified
  Bezeichnung: 'Bruno AG' Modified
  FirmaDiscriminator: 'Auftraggeber' Modified
  KontaktBeendet: <null> Modified
  KontaktSeit: '01.12.2021 00:00:00' Modified
  Kosten: <null> Modified
  Leistungsangebot: <null> Modified
  Volumen: 180000 Modified
  Webseite: 'www.brunoundsoehneag.com' Modified
  Adresse: {AdressId: 2}
  AnsprechpartnerListe: [{AnsprechpartnerId: -2147482647}]
```

Ein etwas komplexeres Beispiel für eine Änderungsverfolgung

- Aufgabe: Der Name der Firma, die Webseite und der Name des Eigners wurde falsch geschrieben und soll geändert werden. Außerdem soll ein neuer Ansprechpartner (Oskar Leopold) hinzugefügt und ein anderer (Gisela Müller) entfernt werden

Daten vor der Änderung:

Firma: Weigen GmbH, www.Weigen.com
Herr Max Weigen
Frau Gisela Müller

Daten nach der Änderung:

Firma: Eigen GmbH & Co.KG, www.eigen-elektronik.com
Herr Max Eigen
Herr Oskar Leopold

Ergebnisse des ChangeTrackers: Adress-Daten und Ansprechpartner-Liste

- Für die Adressdaten wird keine Änderung angezeigt:

```
Adresse {AdressId: 1} Unchanged
  AdressId: 1 PK
  HsNr: <null>
  Ort: 'Leipzig'
  PLZ: '04103'
  Postfach: <null>
  PostfachPLZ: <null>
  Strasse: <null>
  AnsprechpartnerListe: [{AnsprechpartnerId: 1}, {AnsprechpartnerId: 2}, {AnsprechpartnerId: -2147482647}]
  FirmaListe: [{FirmaId: 1}]
```

Ergebnisse des ChangeTrackers: Ansprechpartner

Ansprechpartner {AnsprechpartnerId: -2147482647} Added

AnsprechpartnerId: -2147482647 PK Temporary

AdressId: 1 FK

Anrede: 'Herr'

EMail: 'oleopold@eigen.de'

FirmaId: 1 FK

Nachname: 'Leopold'

Vorname: 'Oskar'

Adresse: {AdressId: 1}

Firma: {FirmaId: 1}

Ansprechpartner {AnsprechpartnerId: 1} Modified

AnsprechpartnerId: 1 PK

AdressId: 1 FK

Anrede: 'Herr'

EMail: 'meigen@eigen.de'

FirmaId: 1 FK

Nachname: 'Eigen' Modified Originally 'Weigen'

Vorname: 'Max'

Adresse: {AdressId: 1}

Firma: {FirmaId: 1}

Ansprechpartner {AnsprechpartnerId: 2} Deleted

AnsprechpartnerId: 2 PK

AdressId: 1 FK

Anrede: 'Frau'

EMail: 'gmueLLer@eigen.de'

FirmaId: 1 FK

Nachname: 'Müller'

Vorname: 'Gisela'

Adresse: {AdressId: 1}

Firma: {FirmaId: 1}

Hier werden die 3 Fälle, die der ChangeTracker erkennt, angezeigt:

1. Neu einzufügen: Herr Leopold – erhält eine temporäre Id
2. Zu ändern: Herr Weigen wird zu Herr Eigen
3. Zu entfernen: Frau Müller

ChangeTracker: Firmendaten

Firma {Firmald: 1} Modified

Firmald: 1 PK

Adresseld: 1 FK

Anforderungen: 'Web-Shop mit MySQL'

Bezeichnung: 'Eigen GmbH & Co.KG' Modified Originally 'Weigen GmbH'

KontaktBeendet: <null>

KontaktSeit: '15.03.2017 00:00:00'

WebSeite: 'www.eigen-elektronik.com' Modified Originally 'www.Weigen.com'

Adresse: {Adresseld: 1}

AnsprechpartnerListe: [{AnsprechpartnerId: 1}, {AnsprechpartnerId: 2}, {AnsprechpartnerId: -2147482647}]

Nach der Kontrolle der Daten kann der eigentliche Speicherprozess gestartet werden

- Nachdem wir `dbctx.SaveChanges` aktiviert haben, werden die Modifikationen der Entities in die Datenbanktabellen Firma und Ansprechpartner übertragen. Wie man sieht, werden die Änderungen, die von `ChangeTracker` dargestellt werden, vollständig umgesetzt. Insbesondere werden die temporären Ids (bei dem neuen Ansprechpartner Herrn Leopold) in reale Ids, hier 1001, umgewandelt.

Firma: Eigen GmbH & Co.KG, www.eigen-elektronik.com
Herr Max Eigen, 1
Herr Oskar Leopold, 1001

Zusammenfassung Änderungsnachverfolgung

- Jede DbContext -Instanz verfolgt **Änderungen** nach, die an **Entitäten** vorgenommen wurden. Diese **nachverfolgten Entitäten** bestimmen wiederum die **Änderungen** an der Datenbank, wenn SaveChanges aufgerufen wird.
- EF Core besitzt eine per Voreinstellung aktive Änderungsnachverfolgung (engl.: einen change tracker) mit folgenden Leistungen:
 - Nachverfolgung von Änderungen bei Entitäten
 - Identitätsauflösung für Entitäten
- Zuständig für die Änderungsnachverfolgung ist das über die ChangeTracker-Eigenschaft der aktuellen DbContext-Instanz referenzierte DbChangeTracker-Objekt
- Falls bei einer Abfrage deren Ergebnis nicht weiter bearbeitet werden soll, kann man den Changetracker abschalten. Um für die aktuelle DbContext-Instanz die Änderungsnachverfolgung generell abzuschalten, setzt man die Eigenschaft QueryTrackingBehavior des ChangeTracker-Objekts zum Kontext auf den Wert NoTracking der Enumeration QueryTrackingBehavior:

```
dbctx.ChangeTracker.DetectChanges();  
Display(dbctx.ChangeTracker.DebugView.LongView);
```

```
dbctx.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
```

STATUSAUSSAGEN

Status-Aussagen

Welches SQL-Kommando beim Aufruf der DbContext-Methode SaveChanges() für eine Entität ausgeführt wird, hängt von ihrem Status ab, den man über die EntityEntry - Eigenschaft State mit Werten der Enumeration EntityState feststellen kann

Detached	Die Entität ist nicht in die Änderungsnachverfolgung durch das DbContext-Objekt einbezogen.
Unchanged	Die Entität ist in die Änderungsnachverfolgung durch das DbContextObjekt einbezogen. Sie existiert in der Datenbank, und die Kopie im Speicher ist im Vergleich zum Zustand in der Datenbank nicht verändert worden.
Deleted	Die Entität ist in die Änderungsnachverfolgung durch das DbContextObjekt einbezogen. Sie existiert in der Datenbank und ist vorgemerkt für das Löschen aus der Datenbank.
Modified	Die Entität ist in die Änderungsnachverfolgung durch das DbContextObjekt einbezogen. Sie existiert in der Datenbank, und die Kopie im Speicher ist im Vergleich zum Zustand in der Datenbank verändert worden.
Added	Die Entität ist per Add() in die Änderungsnachverfolgung durch das DbContext-Objekt aufgenommen worden. Sie existiert noch nicht in der Datenbank.

```
Display("Vorhandene Firma f1 (aus dbctx.FirmaListe): " + dbctx.Entry(f1).State); //Unchanged
Display("Neuer Ansprechpartner a1: " + dbctx.Entry(a1).State); //Detached
```

SaveChanges in Abhängigkeit vom Status einer Entität

- SaveChanges führt verschiedene Dinge für Entitäten in verschiedenen Zuständen aus:
 - Unchanged - Unveränderte Entitäten werden von SaveChanges nicht berührt. Aktualisierungen werden nicht an die Datenbank für Entitäten im unveränderten Zustand gesendet.
 - Attached - Hinzugefügte Entitäten werden in die Datenbank eingefügt und werden dann unverändert, wenn SaveChanges zurückgegeben wird.
 - Modified - Geänderte Entitäten werden in der Datenbank aktualisiert und werden dann unverändert, wenn SaveChanges zurückgegeben wird.
 - Deleted - Gelöschte Entitäten werden aus der Datenbank gelöscht und werden dann vom Kontext getrennt.

Status beim Ändern von Datensätzen (Update)

Unterscheidung von Verbundem und Unverbundenem Status

- Es gibt zwei Szenarien, die auftreten können, wenn Daten zu modifizieren sind

- **Verbundenes Szenarium.** In diesem Fall ist die Entity mit der DB verbunden und im Status „Modified“.

```
Display(dbctx.Entry(abt).State);  
//Modified
```

- **Nicht verbundenes Szenarium.** In diesem Fall ist die Entity nicht mehr mit der DB verbunden, sie ist im Status „Untached“.
Das kommt insbesondere vor, wenn der Nutzer zunächst Entities aus der Datenbank lädt, sie dann im Hauptspeicher bearbeitet und dann den Speichervorgang einleitet.

```
Display(dbctx.Entry(abt).State); //Modified
```

```
using (var dbctx = new EFCTestDBContext())  
{  
    var abt = dbctx.AbtListe  
        .Where(d => d.AbtBezeichnung == „Abt03“).First();  
    abt.Beschreibung = „Verkaufsabteilung – Connected Scenario“;  
    dbctx.SaveChanges();  
}
```

```
Abteilung abt;  
using (var dbctx = new EFCTestDBContext())  
{  
    abt = dbctx.AbtListe.Where(d => d.AbtBezeichnung == „Abt03“).First();  
}  
  
abt.Beschreibung = „Verkaufsabteilung - Disconnected Scenario“;  
  
using (var dbctx = new EFCTestDBContext())  
{  
    dbctx.Entry(abt).State = EntityState.Modified; //Wir erfüllen die Aufgabe des  
    //ChangeTrackers  
    dbctx.SaveChanges();  
}
```

Die Behandlung von isolierten Entities

- Hier wird geprüft, ob die Firma bereits in der Datenbank existiert und entsprechend wird der Status gesetzt.

```
public void InsertOrUpdate(Firma firma)
{
    using (var dbctx = new EFCTestDBContext())
    {
        dbctx.Entry(firma).State = firma.FirmaId == 0 ?
            EntityState.Added :
            EntityState.Modified;

        dbctx.SaveChanges();
    }
}
```

Attach und Entry

```
string fall = "Entry"; // "Attach"
Firma? firma;
using (var dbctx = new EFCTestDB09Context())
{
    firma = dbctx.FirmaListe.Include(f => f.Adresse).Where(f => f.Bezeichnung == "Bruno AG").FirstOrDefault();
}
if (firma != null)
{
    List<Ansprechpartner> apListe = new List<Ansprechpartner>();
    Ansprechpartner ap = new Ansprechpartner() { Nachname = "Kerzschner",
        Vorname = "Sybille", Anrede = "Frau", Adresse = firma.Adresse };
    apListe.Add(ap);
    firma.WebSeite = "www.brunoag.com";
    firma.AnsprechpartnerListe = apListe;
    using (var dbctx = new EFC03DBContext())
    {
        switch (fall)
        {
            case "Entry":
                var entry = dbctx.Entry(firma); // Nur die Firma wird geupdated, der Ansprechpartner wird ignoriert
                entry.State = EntityState.Modified;
                break;
            case "Attach":
                var attach = dbctx.Attach(firma); // Die Firma wird geupdated,
                // der Ansprechpartner wird eingetragen
                attach.State = EntityState.Modified;
                break;
        }
        dbctx.SaveChanges();
    }
}
```

- **Entry:** Nur das Objekt, dass durch die Entry-Methode angesprochen, wird berücksichtigt und nicht seine verbundenen Entitäten.
- **Attach:** Das Objekt wird rekursiv analysiert und alle verbundenen Entitäten werden berücksichtigt.

Mit Entry haben wir mehr Kontrolle, aber wir müssen die Untereigenschaften (zweite Ebene) selbst aktualisieren. Mit Attach können wir alles aktualisieren, aber wir müssen die Referenzen im Auge behalten.

Die SQL-Statements für die Attach-Operation

- Im SQL-Profiler lassen sich die Operationen verfolgen:

```
exec sp_executesql N'SET NOCOUNT ON;
INSERT INTO [dbo].[Ansprechpartner] ([Adresseld], [Anrede], [EMail], [Firmald], [Nachname], [Vorname])
OUTPUT INSERTED.[AnsprechpartnerId]
VALUES (@p0, @p1, @p2, @p3, @p4, @p5);
UPDATE [dbo].[Firma] SET [Adresseld] = @p6, [Anforderungen] = @p7, [Bezeichnung] = @p8, [FirmaDiscriminator] = @p9, [KontaktBeendet] = @p10,
[KontaktSeit] = @p11, [Kosten] = @p12, [Leistungsangebot] = @p13, [Volumen] = @p14, [WebSeite] = @p15
OUTPUT 1
WHERE [Firmald] = @p16;
',N'@p0 int,@p1 nvarchar(10),@p2 nvarchar(50),@p3 int,@p4 nvarchar(20),@p5 nvarchar(20),@p16 int,@p6 int,@p7 nvarchar(4000),@p8
nvarchar(100),@p9 nvarchar(4000),@p10 datetime2(7),@p11 datetime2(7),@p12 float,@p13 nvarchar(4000),@p14 float,@p15
nvarchar(100)',@p0=2,@p1=N'Frau',@p2=NULL,@p3=6,@p4=N'Fleurol',@p5=N'Francoise',@p16=6,@p6=2,@p7=N'Web-Applikation mit
Datenbank',@p8=N'Bruno AG',@p9=N'Auftraggeber',@p10=NULL,@p11='2021-12-01
00:00:00',@p12=NULL,@p13=NULL,@p14=180000,@p15=N'www.brunoundsoehneag.com'
```

Entfernen eines Datensatzes (DELETE)

- Auch in diesem Fall existieren die beiden Szenarien

- Verbundenes Szenarium:
Hier sind die Entities immer mit der Datenbank verbunden

```
public void DeleteConnected()
{
    Abteilung abt;
    //Verbundenes Szenarium
    using (var dbctx = new EFCTestDBContext())
    {
        abt = dbctx.AbteilungListe.Where(d => d. Abteilungsbezeichnung == „Verkauf“).First();
        dbctx.AbteilungListe.Remove(abt);
        dbctx.SaveChanges();
    }
}
```

- Nicht verbundenes Szenarium:
Hier ist die Entity nicht mehr verbunden

```
public void DeleteDisconnected()
{
    Abteilung abt;
    using (var dbctx = new EFCTestDBContext())
    {
        abt = dbctx.AbteilungListe.Where(d => d. Abteilungsbezeichnung == „Verkauf“).First();
    }

    using (var dbctx = new EFCoreContext())
    { //abt ist jetzt im unverbundenen Zustand
        //db.AbteilungListe.Attach(abt); - auch möglich
        dbctx.Entry(abt).State = EntityState.Deleted;
        dbctx.SaveChanges();
    }
}
```

LÖSCHVARIANTEN BEI 1:1, 1:N, N:M- RELATIONEN

Lösch-Varianten bei Entities, die mit einander in Beziehung stehen

Fall 1

- Beispiel **Abteilung – 1:n – Mitarbeiter**. Wenn eine Abteilung gelöscht werden, dürfen die zugehörigen Mitarbeiter nicht gelöscht werden, es wird ein Fehler generiert (Referentielle Integrität verletzt!) Natürlich darf beim Löschen eines Mitarbeiters nicht die zugehörige Abteilung gelöscht werden!

```
public class Mitarbeiter
{
    [Key]
    public int MitarbeiterID { get; set; }
    public string NachName { get; set; }

    [ForeignKey("AbteilungId")]
    public int? AbteilungId { get; set; }
    public virtual Abteilung Abteilung { get; set; }
}

public class Abteilung
{
    public int AbteilungId { get; set; }
    public string Bezeichnung { get; set; }
    public virtual IList<Mitarbeiter> MitarbeiterListe { get; set; }
}
```

```
modelBuilder.Entity<Mitarbeiter>()
    .HasOne(x => x.Abteilung)
    .WithMany(op => op.MitarbeiterListe)
    .HasForeignKey(@"AbteilungId")
    .IsRequired(false);
```

```
modelBuilder.Entity<Abteilung>()
    .HasMany(x => x.MitarbeiterListe)
    .WithOne(op => op.Abteilung)
    .HasForeignKey(@"AbteilungId")
    .IsRequired(false);
```

Test:

Vor dem Löschen der Abteilung „Schulung“:

... Tributsch, Ursula, MitarbeiterId: 34, AbteilungId: 15

... Becker, Ortwin, MitarbeiterId: 35, AbteilungId: 15

Nach dem Löschen der Abteilung „Schulung“:

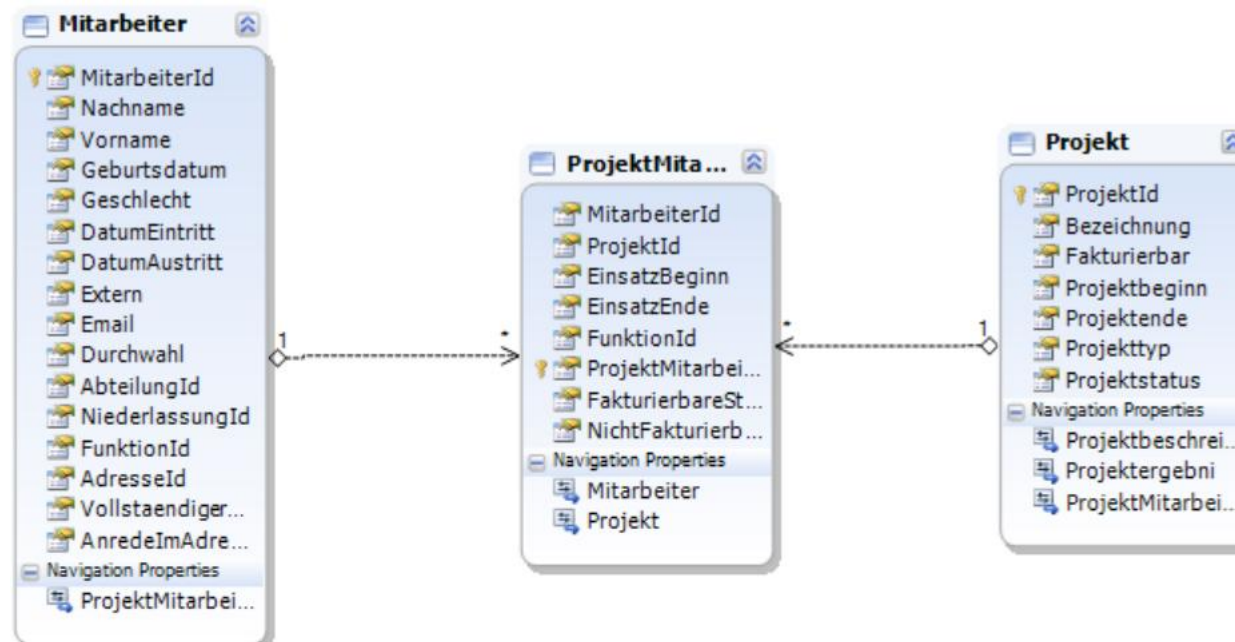
... Tributsch, Ursula, MitarbeiterId: 34, AbteilungId: null

... Becker, Ortwin, MitarbeiterId: 35, AbteilungId: null

Lösch-Varianten bei Entities, die mit einander in Beziehung stehen

Fall 2

- Beispiel **Mitarbeiter – 1:n – ProjektMitarbeiter m:1 - Projekt**. Wenn ein Mitarbeiter oder ein Projekt gelöscht wird, müssen die zugeordneten Datensätze in der Tabelle ProjektMitarbeiter gelöscht werden. (Natürlich darf beim Löschen eines Datensatzes in MitarbeiterProjekt nicht der zugehörige Mitarbeiter oder das zugehörige Projekt gelöscht werden!)



Lösch-Varianten bei Entities, die miteinander per m:n in Beziehung stehen

Fall 2

- **Mitarbeiter – 1:n – ProjektMitarbeiter m:1 - Projekt.**

```
[Table("Mitarbeiter")]
public partial class Mitarbeiter: Audit
{
    public Mitarbeiter() { }
    [Key]
    [Column(Order = 0)]
    public virtual int MitarbeiterId { get; set; }
    [Required]
    [StringLength(50)]
    [Column(Order = 1)]
    public virtual string Nachname { get; set; }
    [StringLength(50)]
    [Column(Order = 2)]
    public virtual string Vorname { get; set; }
    [Precision(0)]
    public virtual DateTime? Geburtsdatum { get; set; }

    public virtual IList<ProjektMitarbeiter> ProjektMitarbeiterListe { get; }
}
```

```
//Ist Detail-Tabelle bezüglich Mitarbeiter und Projekt
[Table("ProjektMitarbeiter")]
public partial class ProjektMitarbeiter
{
    [Key]
    public int ProjektMitarbeiterId { get; set; }
    [ForeignKey("MitarbeiterId")]
    [Required]
    public int MitarbeiterId { get; set; }
    public virtual Mitarbeiter Mitarbeiter { get; set; }
    [ForeignKey("ProjektId")]
    [Required]
    public int ProjektId { get; set; }
    public virtual Projekt Projekt { get; set; }
    [Precision(0)]
    public virtual DateTime? EinsatzBeginn { get; set; }
    [Precision(0)]
    [Column(Order = 4)]
    public virtual DateTime? EinsatzEnde { get; set; }
    //usw.
}
```

Lösch-Varianten bei Entities, die miteinander in Beziehung stehen

Fall 2

- Mitarbeiter – 1:n – ProjektMitarbeiter m:1 - Projekt.**

```
modelBuilder.Entity<Mitarbeiter>()
    .HasMany(x => x.ProjektMitarbeiterListe)
    .WithOne(op => op.Mitarbeiter)
    .OnDelete(DeleteBehavior.Cascade)
    .HasForeignKey(@"MitarbeiterId")
    .IsRequired(true);
```

```
modelBuilder.Entity<ProjektMitarbeiter>()
    .HasOne(x => x.Mitarbeiter)
    .WithMany(op => op.ProjektMitarbeiterListe)
    .OnDelete(DeleteBehavior.Cascade)
    .HasForeignKey(@"MitarbeiterId")
    .IsRequired(false);
```

```
modelBuilder.Entity<Projekt>()
    .HasMany(x => x.ProjektMitarbeiterListe)
    .WithOne(op => op.Projekt)
    .OnDelete(DeleteBehavior.Cascade)
    .HasForeignKey(@"ProjektId")
    .IsRequired(true);
```

```
modelBuilder.Entity<ProjektMitarbeiter>()
    .HasOne(x => x.Projekt)
    .WithMany(op => op.ProjektMitarbeiterListe)
    .OnDelete(DeleteBehavior.Cascade)
    .HasForeignKey(@"ProjektId").IsRequired(false);
```

Wenn ein Mitarbeiter oder ein Projekt gelöscht wird, müssen auch die entsprechenden Datensätze in Projektmitarbeiter verschwinden

Wenn aber ein Datensatz in Projektmitarbeiter gelöscht wird, dürfen weder der Mitarbeiter noch das Projekt gelöscht werden

Test der m:n-Beziehung: Löschen einer Projekt-Mitarbeiter-Verknüpfung

```
select pm.ProjektMitarbeiterId, p.ProjektId, m.MitarbeiterId, p.Bezeichnung, m.Nachname, p.Projektbeginn, pm.EinsatzBeginn
from ProjektMitarbeiter pm inner join Projekt p on p.ProjektId=pm.ProjektId
inner join Mitarbeiter m on m.MitarbeiterId=pm.MitarbeiterId
```

ProjektMitarbeiterId	ProjektId	MitarbeiterId	Bezeichnung	Nachname	Projektbeginn	EinsatzBeginn
1	1	1	1Sommerfeld	Tributsch	15.08.2022	15.08.2022
2	1	1	2Sommerfeld	Becker	15.08.2022	01.10.2022
3	2	2	2Beetz II	Becker	01.11.2022	13.09.2022
4	2	2	3Beetz II	Fleurol	01.11.2022	14.11.2022
5	2	2	4Beetz II	Olzychovski	01.11.2022	01.12.2022

1. Löschen einer Projekt-Beteiligung, z.B. ProjektMitarbeiterId = 4 (Frau Fleurol)

```
var m = dbctx.ProjektMitarbeiterListe.Where(x => x.Mitarbeiter.Nachname == searchTerm).FirstOrDefault();
if (m!=null)
{
    try
    {
        Display(dbctx.Entry(m).State);
        var entityEntry = dbctx.ProjektMitarbeiterListe.Remove(m);
        Display(entityEntry.State);
        dbctx.SaveChanges();
        Display(entityEntry.State);
    }
    catch (Exception ex)
    {
        Display(ex.Message);
    }
}
```

Funktioniert, ohne dass Projekt oder Mitarbeiter gelöscht werden

Test der m:n-Beziehung: Löschen einer Mitarbeiterin – Kaskadierendes Löschen

In diesem Fall sollte automatisch auch die Projekt-Mitarbeiter-Verknüpfung gelöscht werden, nicht aber das Projekt

MitarbeiterId	AbteilungId	Nachname	Vorname
1	1	Tributsch	Ursula
2	1	Becker	Ortwin
4	2	Olzychovski	Marek
3	2	Fleurol	Francois

ProjektMitarbeiterId	ProjektId	MitarbeiterId	Bezeichnung	Nachname	Projektbeginn	EinsatzBeginn
1	1	1	1Sommerfeld	Tributsch	15.08.2022	15.08.2022
2	1	2	2Sommerfeld	Becker	15.08.2022	01.10.2022
3	2	2	2Beetz II	Becker	01.11.2022	13.09.2022
4	2	2	3Beetz II	Fleurol	01.11.2022	14.11.2022
5	2	4	4Beetz II	Olzychovski	01.11.2022	01.12.2022

2. Löschen einer Mitarbeiterin (Frau Fleurol)

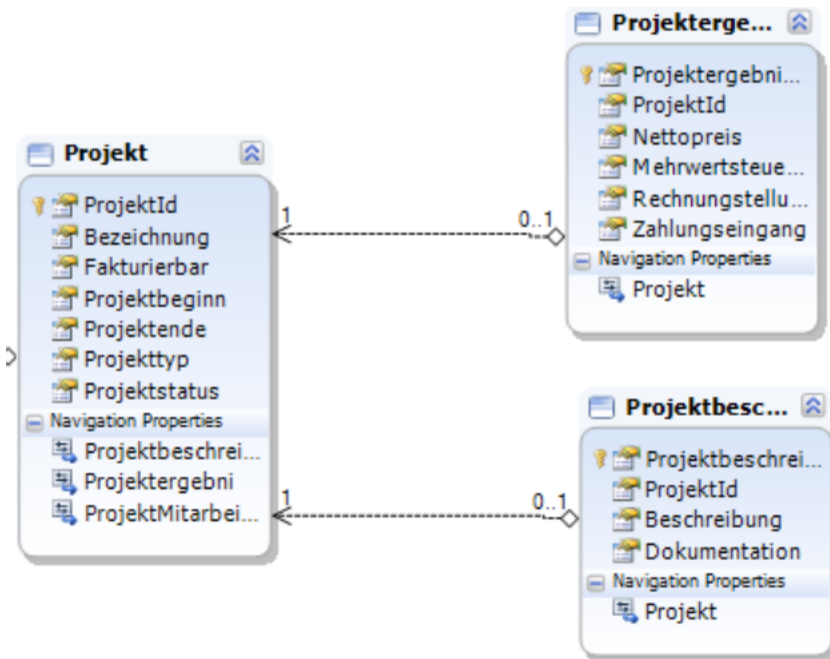
```
var m = dbctx.MitarbeiterListe.Where(x => x.Nachname == searchTerm).FirstOrDefault();
if (m!=null)
{
    try
    {
        Display(db.Entry(m).State);
        var entityEntry = dbctx.MitarbeiterListe.Remove(m);
        Display(entityEntry.State);
        dbctx.SaveChanges();
        Display(entityEntry.State);
    }
    catch (Exception ex)
    {
        Display(ex.Message);
    }
}
```

Funktioniert einschließlich
Löschung der Projektzuordnung,
ohne dass Projekte gelöscht
werden

Lösch-Varianten bei Entities, die mit einander in einer 1:1 - Beziehung stehen

Fall 3

- Beispiel **Projekt – 1:1 – ProjektErgebnis** bzw. **Projekt – 1:1 – ProjektBeschreibung**
- Wenn ein Projekt gelöscht wird, werden auch die zugeordneten Datensätze in den Tabellen Projektergebnis und Projektbeschreibung (und Projektmitarbeiter) gelöscht werden



```
modelBuilder.Entity<Projekt>()
    .HasOne(x => x.Projektbeschreibung)
    .WithOne(op => op.Projekt)
    .HasForeignKey(typeof(Projekt), @"ProjektId")
    .IsRequired(true);
```

```
modelBuilder.Entity<Projekt>()
    .HasOne(x => x.Projektergebnis)
    .WithOne(op => op.Projekt)
    .HasForeignKey(typeof(Projekt), @"ProjektId")
    .IsRequired(true);
```

Lösch-Varianten bei Entities, die mit einander in einer 1:1 - Beziehung stehen

Fall 3 -- ?

- Beispiel **Projekt – 1:1 – ProjektErgebnis** bzw. **Projekt – 1:1 – ProjektBeschreibung**
- Wenn ein Projektergebnis gelöscht wird, werden auch die zugeordneten Datensätze in den Tabellen Projekt und Projektbeschreibung sowie ProjektMitarbeiter gelöscht. Hier kommt keine automatische Warnung!

```
var projektErgebnis = dbctx.ProjektergebnisListe
    .Include(pe => pe.Projekt)
    .Where(x => x.Projekt.Bezeichnung.Contains(searchTerm)).First();
Display($"{projektErgebnis.ProjektId}: {projektErgebnis.Projekt.Bezeichnung}, {projektErgebnis.Nettopreis} ");

if (projektErgebnis != null)
{
    try
    {
        Display(dbctx.Entry(projektErgebnis).State);
        var entityEntry = dbctx.ProjektergebnisListe.Remove(projektErgebnis);
        Display(entityEntry.State);
        dbctx.SaveChanges();
        Display(entityEntry.State);
    }
    catch (Exception ex)
    {
        Display(ex.Message);
    }
}
```