10 Objektbeziehungen und Ladestrategien

Entity Framework Core unterstützt im Gegensatz zum bisherigen Entity Framework nur drei statt vier Ladestrategien. Und auch die Realisierung des Eager Loading entspricht nicht dem aus dem Vorgänger gewohnten Verhalten.

10.1 Standardverhalten

Entity Framework Core beschränkt sich in der Standardeinstellung bei einer Abfrage (wie das bisherige Entity Framework auch) darauf, die tatsächlich angeforderten Objekte zu laden und lädt verbundene Objekte nicht automatisch mit. Eine LINQ-Abfrage wie

```
List<Flug> liste = (from x in ctx.FlugSet
                    where x.Abflugort == ort &&
                          x.FreiePlaetze > 0
                    orderby x.Datum, x.Abflugort
                    select x).ToList();
```

lädt in Bezug auf das in der nächsten Abbildung dargestellte Objektmodell also wirklich nur Instanzen der Klasse Flug. Damit in der Datenbank verbundene Piloten-, Buchungen- oder Flugzeugtypen-Datensätze werden nicht automatisch mitgeladen. Das Mitladen verbundener Datensätze (in der Fachsprache „Eager Loading“ genannt) wäre auch keine gute Idee für die Standardeinstellung, denn hier würden dann ggf. Daten geladen, die später gar nicht gebraucht werden. Zudem haben die verbundenen Datensätze bekanntlich selbst wieder Beziehungen, z. B. Buchungen zu Passagieren. Passagiere haben aber auch Buchungen auf anderen Flügen. Wenn man rekursiv alle diese in Beziehungen stehenden Datensätze mitladen würde, dann würde man im Beispiel des Objektmodells in der nächsten Abbildung mit großer Wahrscheinlichkeit fast alle Datensätze ins RAM laden, denn viele Passagiere sind über gemeinsame Flüge mit anderen Passagieren verbunden. Eager Loading wäre als Standardeinstellung also nicht gut.

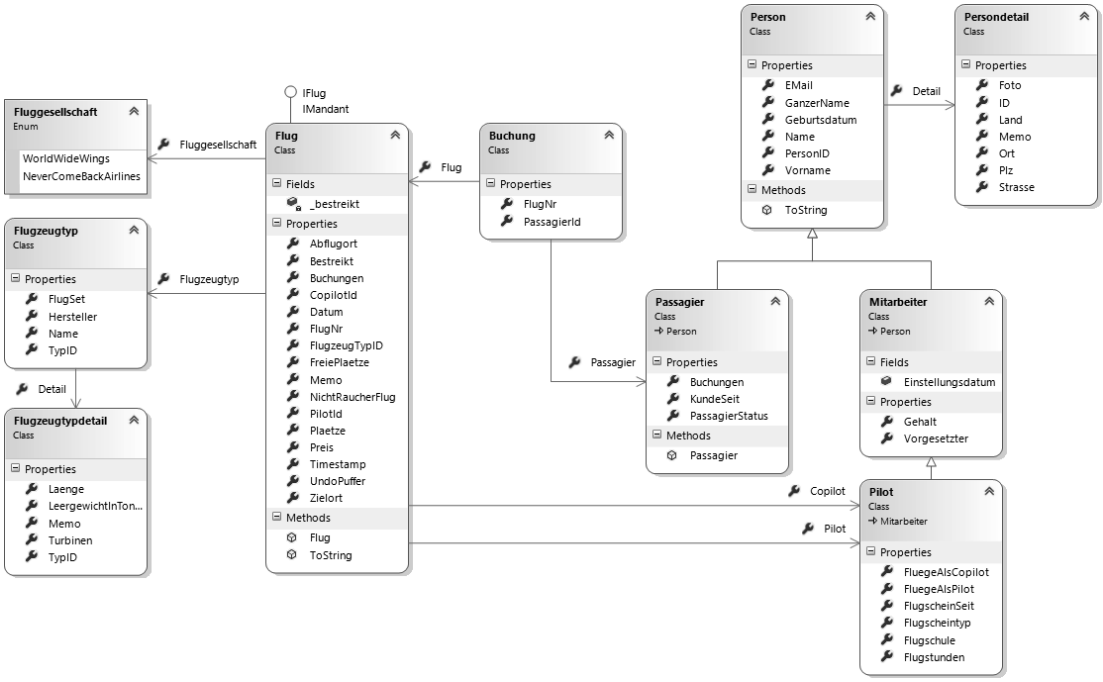

Bild 10.1 Objektmodell für die Verwaltung von Flügen, Passagieren und Piloten

10.2 Kein Lazy Loading

Das bisherige Entity Framework unterstützt vier Strategien für das Laden verbundener Objekte: Lazy Loading automatisch, explizites Laden (Explicit Loading), Eager Loading und Preloading mit Relationship Fixup (siehe Abbildung). In Entity Framework Core 1.0 standen die beiden Varianten des Lazy Loading nicht zur Verfügung. In Entity Framework Core Version 1.1 gibt es zumindest das explizite Laden, nicht aber das automatische Lazy Loading. Automatisches Lazy Loading ist die Standardeinstellung im bisherigen Entity Framework.

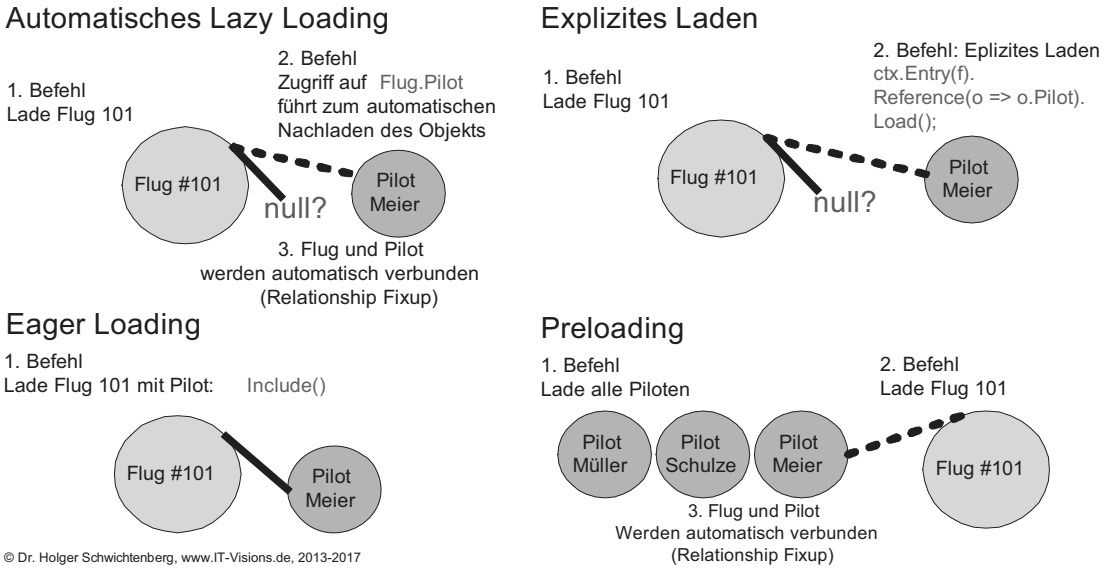

Bild 10.2 Ladestrategien im Entity Framework 1.0 bis 6.x. Entity Framework Core unterstützt auch in Version 1.1 nur drei Strategien; Lazy Loading fehlt.

Das einleitende Beispiel (siehe Unterkapitel „10.1 Standardverhalten“) würde beim bisherigen Entity Framework im ersten Schritt nur den explizit angeforderten Flug laden, dann aber in den folgenden Programmcodezeilen via Lazy Loading auch die Piloten- und Copiloteninformation (mit jeweils auch deren anderen Flügen) sowie die Buchungen mit den Passagierdaten laden. Entity Framework würde hier nacheinander eine Vielzahl von SELECT-Befehlen zur Datenbank senden. Wie viele es genau sind, hängt von der Anzahl der Passagiere auf diesem Flug ab.

Bei Entity Framework Core liefert obiger Programmcode aber weder Pilot, Copilot noch Passagiere. Es wird bei „Anzahl Passagiere auf diesem Flug“ eine 0 angezeigt. Microsoft hat das Lazy Loading für Entity Framework Core noch nicht implementiert.

Lazy Loading beinhaltet eine besondere Implementierungsherausforderung, denn der OR-Mapper muss jeglichen Zugriff auf alle Objektreferenzen abfangen, um hier bei Bedarf die verbundenen Objekte nachladen zu können. Dieses Abfangen erfolgt durch die Verwendung bestimmter Klassen für Einzelreferenzen und Mengenklassen.

Im klassischen Entity Framework gibt es sowohl die Möglichkeit, Lazy Loading durch Verwendung bestimmter Lazy Loading-fähiger Klassen als auch durch meist unsichtbare Runtime Proxy-Objekte zu realisieren. Beides ist in Entity Framework Core noch nicht vorgesehen.

Lazy Loading wird man in Entity Framework Core aber gerade dann vermissen, wenn es keinen Sinn macht, verbundene Datensätze schon vorab zu laden. Typisches Beispiel ist eine Master-Detail-Ansicht auf dem Bildschirm. Wenn es viele Master-Datensätze gibt, wäre es verschwendete Zeit, zu jedem Master-Datensatz auch schon die Detail-Datensätze zu laden. Vielmehr wird man immer nur die Detail-Datensätze zu dem Master-Datensatz, den der Benutzer gerade angeklickt hat, anfordern. Während man an dieser Stelle im bisherigen

Entity Framework Core die Master-Detail-Darstellung via Lazy Loading ohne weiteren Programmcode beim Anklicken des Master-Datensatzes realisieren konnte, muss man in Entity Framework Core das Klicken abfangen und die Detail-Datensätze explizit laden.

Listing 10.1 Vergeblicher Versuch des Zugriffs auf verbundene Objekte in Entity Framework Core

```
/// <summary>
  /// Liefert keine Pilot-, Buchungs- und Passagierinformationen
  /// </summary>
  public static void Demo_LazyLoading()
  {
   CUI.Headline(nameof(Demo_LazyLoading));

   using (var ctx = new WWWingsContext())
   {
    // Lade nur den Flug
    var f = ctx.FlugSet.SingleOrDefault(x => x.FlugNr == 101);

    Console.WriteLine($"Flug Nr {f.FlugNr} von {f.Abflugort} nach {f.Zielort} hat
{f.FreiePlaetze} freie Plätze!");
    if (f.Pilot != null) Console.WriteLine($"Pilot: {f.Pilot.Name} hat {f.Pilot.
FluegeAlsPilot.Count} Flüge als Pilot!");
    else Console.WriteLine("Kein Pilot zugewiesen!");
    if (f.Copilot != null) Console.WriteLine($"Copilot: {f.Copilot.Name} hat
{f.Copilot.FluegeAlsCopilot.Count} Flüge als Copilot!");
    else Console.WriteLine("Kein Copilot zugewiesen!");

    Console.WriteLine("Anzahl Passagiere auf diesem Flug: " + f.Buchungen.Count);

    Console.WriteLine("Passagiere auf diesem Flug:");
    foreach (var b in f.Buchungen)
    {
     Console.WriteLine("- Passagier #{0}: {1} {2}", b.Passagier.PersonID,
b.Passagier.Vorname, b.Passagier.Name);
    }
   }
```

10.3 Eager Loading

Genau wie das bisherige Entity Framework unterstützt auch Entity Framework Core das Eager Loading. Die Syntax hat sich jedoch ein wenig geändert.

Im bisherigen Entity Framework konnte man bei Include() in der ersten Version nur eine Zeichenkette mit dem Namen einer Navigationseigenschaften angeben; die Zeichenkette wurde nicht vom Compiler geprüft. Ab der dritten Version (Versionsnummer 4.1) kam dann die Möglichkeit hinzu, eines Lambda-Audrucks für die Navigationseigenschaften anstelle der Zeichenkette anzugeben. Für Ladepfade über mehrere Ebenen musste man die Lambda-Ausdrücke verschachteln und auch die Select()-Methode verwenden.

In Entity Framework Core gibt es weiterhin beide Varianten, jedoch wurde die Syntax für die Lambda-Ausdrücke etwas modifiziert: Anstelle der Verschachtelung mit Select() tritt ThenInclude() analog zu OrderBy() und ThenOrderBy() für die Sortierung über mehrere Spalten. Das nächste Listing zeigt das Eager Loading eines Flugs mit folgenden verbundenen Daten:

- Buchungen und zu jeder Buchung den Passagierinformationen: Include(b => b.Buchungen).ThenInclude(p => p.Passagier)
- Pilot und zu dem Pilot die Liste seiner weiteren Flüge als Pilot: Include(b => b.Pilot).ThenInclude(p => p.FluegeAlsPilot)
- Copilot und zu dem Copilot die Liste seiner weiteren Flüge als Copilot: Include(b => b.Copilot).ThenInclude(p => p.FluegeAlsCopilot)

Listing 10.2 Mit Eager Loading kann man in Entity Framework Core die verbundenen Objekte nutzen.

```
/// <summary>
  /// Liefert Pilot-, Buchungs- und Passagierinformationen
  /// </summary>
  public static void Demo_EagerLoading()
  {
   CUI.Headline("Demo_EagerLoading");

   using (var ctx = new WWWingsContext())
   {
    // Lade den Flug und einige verbundene Objekte via Eager Loading
    var f = ctx.FlugSet
          .Include(b => b.Buchungen).ThenInclude(p => p.Passagier)
          .Include(b => b.Pilot).ThenInclude(p => p.FluegeAlsPilot)
          .Include(b => b.Copilot).ThenInclude(p => p.FluegeAlsCopilot)
          .SingleOrDefault(x => x.FlugNr == 101);

    Console.WriteLine($"Flug Nr {f.FlugNr} von {f.Abflugort} nach {f.Zielort} hat
{f.FreiePlaetze} freie Plätze!");
    if (f.Pilot != null) Console.WriteLine($"Pilot: {f.Pilot.Name} hat {f.Pilot.
FluegeAlsPilot.Count} Flüge als Pilot!");
    else Console.WriteLine("Kein Pilot zugewiesen!");
    if (f.Copilot != null) Console.WriteLine($"Copilot: {f.Copilot.Name} hat
{f.Copilot.FluegeAlsCopilot.Count} Flüge als Copilot!");
    else Console.WriteLine("Kein Copilot zugewiesen!");

    Console.WriteLine("Anzahl Passagiere auf diesem Flug: " + f.Buchungen.Count);
    Console.WriteLine("Passagiere auf diesem Flug:");
    foreach (var b in f.Buchungen)
    {
     Console.WriteLine("- Passagier #{0}: {1} {2}", b.Passagier.PersonID,
b.Passagier.Vorname, b.Passagier.Name);
    }
   }
  }
```

Das Listing liefert die Ausgabe in der nächsten Bildschirmabbildung: Sowohl die Informationen zu Pilot und Copilot als auch die Liste der gebuchten Passagiere steht zur Verfügung.

ACHTUNG: Der Compiler prüft bei Include() und ThenInclude() nur, ob die Klasse ein entsprechendes Property oder Field besitzt. Er prüft nicht, ob es sich dabei auch um eine Navigationseigenschaft zu einer anderen Entitätsklasse handelt. Wenn es keine Navigationseigenschaft ist, kommt es erst zur Laufzeit zum Fehler: *The property xy is not a navigation property of entity type ‚ab'. The ‚Include(string)' method can only be used with a ‚.' separated list of navigation property names.*

```
Demo_EagerLoading
WWWingsContext: OnConfiguring
WWWingsContext: OnModelCreating
Flug Nr 101 von Seattle nach Moskau hat 129 freie Plätze!
Pilot: Koch hat 10 Flüge als Pilot!
Copilot: Stoiber hat 6 Flüge als Copilot!
Anzahl Passagiere auf diesem Flug: 8
Passagiere auf diesem Flug:
- Passagier #12: Niklas Bauer
- Passagier #15: Jan Schäfer
- Passagier #17: Leon Klein
- Passagier #47: Lukas Schneider
- Passagier #59: Laura Wagner
- Passagier #67: Marie Weber
- Passagier #87: Leonie Schäfer
- Passagier #98: Anna Schmidt
```

Bild 10.3 Ausgabe des Listings

Allerdings gibt es noch einen entscheidenden Unterschied zum bisherigen Entity Framework: Während Entity Framework in den Versionen 1.0 bis 6.x hier nur einen einzigen, sehr großen SELECT-Befehl zum Datenbankmanagementsystem gesendet hätte, entscheidet sich Entity Framework Core, die Abfrage in vier Teile zu teilen (siehe nächste Abbildung):

- Erst wird der Flug geladen mit Join auf die Mitarbeiter-Tabelle, in der sich auch die Pilotinformation befindet (ein Table per Hierarchy-Mapping).
- Im zweiten Schritt lädt Entity Framework Core die sechs anderen Flüge des Copiloten.
- Im dritten Schritt lädt Entity Framework Core die zehn anderen Flüge des Piloten.
- Im letzten Schritt lädt Entity Framework Core die achten gebuchten Passagiere.

Diese Strategie kann schneller sein, als einen großen SELECT-Befehl auszuführen, der ein großes Resultset, in dem Datensätze doppelt vorkommen, liefert und das der OR-Mapper dann auseinandernehmen und von den Duplikaten bereinigen muss. Die Strategie getrennter SELECT-Befehle von Entity Framework Core kann aber auch langsamer sein, da jeder Rundgang zum Datenbankmanagementsystem Zeit kostet. Im bisherigen Entity Framework hatte der Softwareentwickler die freie Wahl, wie groß er eine Eager Loading-Anweisung zuschneiden will und wo er getrennt laden will. In Entity Framework Core wird der Softwareentwickler an dieser Stelle bevormundet und verliert dabei die Kontrolle über die Anzahl der Rundgänge zum Datenbankmanagementsystem.

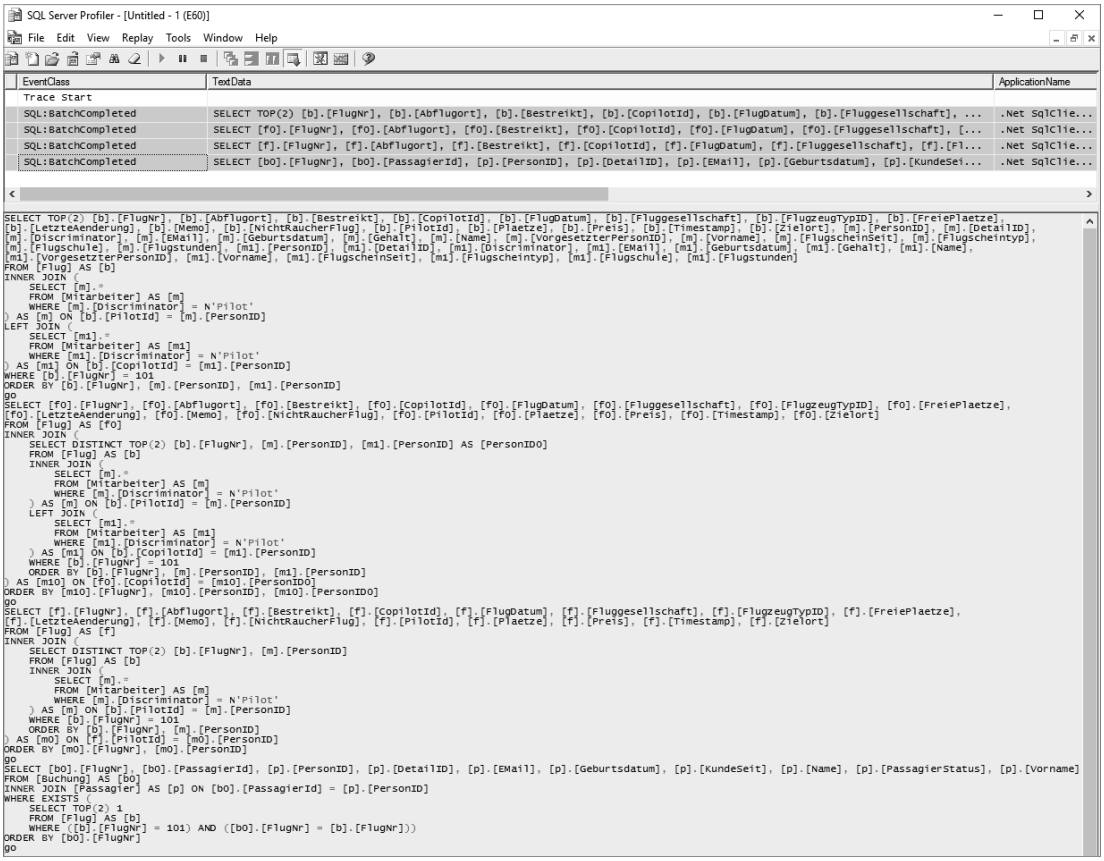

Bild 10.4 Der SQL Server-Profiler zeigt die vier SQL-Befehle, die das Easger Loading-Beispiel in Entity Framework Core auslöst.

■ 10.4 Explizites Nachladen (Explicit Loading)

In Entity Framework Core Version 1.1, das am 16.11.2016 erschienen ist, hat Microsoft das explizite Nachladen nachgerüstet. Hier gibt der Softwareentwickler mit Hilfe der Methoden Reference() (für Einzelobjekte), Collection() (für Mengen) und einem danach folgenden Load() an, dass in Beziehung stehende Objekte jetzt zu laden sind. Diese Methoden stehen aber nicht auf dem Entitätsobjekt selbst zur Verfügung, sondern sind Teil der Klasse EntityEntry<T>, die man durch die Methode Entry() in der Klasse DbContext erhält (siehe Listing). Mit IsLoaded() kann man prüfen, ob das Objekt schon geladen wurde. IsLoaded() liefert auch dann true, wenn es kein passendes Objekt in der Datenbank gab. Es zeigt also nicht an, ob eine Navigationsbeziehung ein Gegenobjekt hat, sondern ob in der aktuellen Kontextinstanz schon einmal versucht wurde, ein passendes Objekt dafür zu laden. Wenn also im nächsten Listing der Flug 101 zwar bereits einen zugewiesenen Piloten (Herrn

Koch), aber noch keinen Copiloten hat, führt dies zur Ausgabe in der nächsten Abbildung. Man muss sich bewusst sein, dass jede Ausführung von Load() zu einem expliziten Absenden eines SQL-Befehls zum Datenbankmanagementsystem führt.

Listing 10.3 Durch das explizite Nachladen sendet Entity Framework Core sehr viele einzelne SQL-Befehle zur Datenbank

```
/// <summary>
  /// Liefert Pilot-, Buchungs- und Passagierinformationen via explizitem Laden
  /// </summary>
  public static void Demo_ExplizitLoading_v11()
  {
   CUI.Headline(nameof(Demo_ExplizitLoading_v11));

   using (var ctx = new WWWingsContext())
   {
    // Lade nur den Flug
    var f = ctx.FlugSet
          .SingleOrDefault(x => x.FlugNr == 101);

    Console.WriteLine($"Flug Nr {f.FlugNr} von {f.Abflugort} nach {f.Zielort} hat
{f.FreiePlaetze} freie Plätze!");

    // Lade nur den Pilot und Copilot nach
    if (!ctx.Entry(f).Reference(x => x.Pilot).IsLoaded)
     ctx.Entry(f).Reference(x => x.Pilot).Load();
    if (!ctx.Entry(f).Reference(x => x.Copilot).IsLoaded)
     ctx.Entry(f).Reference(x => x.Copilot).Load();

    // Prüfung, ob geladen
    if (ctx.Entry(f).Reference(x => x.Pilot).IsLoaded) Console.
WriteLine("Pilot ist geladen!");
    if (ctx.Entry(f).Reference(x => x.Copilot).IsLoaded) Console.WriteLine("Co-Pilot
ist geladen!");


    if (f.Pilot != null) Console.WriteLine($"Pilot: {f.Pilot.Name} hat {f.Pilot.
FluegeAlsPilot.Count} Flüge als Pilot!");
    else Console.WriteLine("Kein Pilot zugewiesen!");
    if (f.Copilot != null) Console.WriteLine($"Copilot: {f.Copilot.Name} hat
{f.Copilot.FluegeAlsCopilot.Count} Flüge als Copilot!");
    else Console.WriteLine("Kein Copilot zugewiesen!");

    // Lade nur die Buchungsliste nach
    if (!ctx.Entry(f).Collection(x => x.Buchungen).IsLoaded)
     ctx.Entry(f).Collection(x => x.Buchungen).Load();

    Console.WriteLine("Anzahl Passagiere auf diesem Flug: " + f.Buchungen.Count);
    Console.WriteLine("Passagiere auf diesem Flug:");
    foreach (var b in f.Buchungen)
    {

     // Lade nur den Passagier für diese Buchung nach
     if (!ctx.Entry(b).Reference(x => x.Passagier).IsLoaded)
      ctx.Entry(b).Reference(x => x.Passagier).Load();

     Console.WriteLine("- Passagier #{0}: {1} {2}", b.Passagier.PersonID,
b.Passagier.Vorname, b.Passagier.Name);
```

```
    }
   }
  }
```

```
Demo_EagerLoading
WWWingsContext: OnConfiguring
WWWingsContext: OnModelCreating
Flug Nr 101 von Seattle nach Moskau hat 129 freie Plätze!
Pilot: Koch hat 10 Flüge als Pilot!
Copilot: Stoiber hat 6 Flüge als Copilot!
Anzahl Passagiere auf diesem Flug: 8
Passagiere auf diesem Flug:
- Passagier #12: Niklas Bauer
- Passagier #15: Jan Schäfer
- Passagier #17: Leon Klein
- Passagier #47: Lukas Schneider
- Passagier #59: Laura Wagner
- Passagier #67: Marie Weber
- Passagier #87: Leonie Schäfer
- Passagier #98: Anna Schmidt
```

Bild 10.5 Ausgabe zum obigem Listing

10.5 Preloading mit Relationship Fixup

Entity Framework Core unterstützt wie das bisherige Entity Framework eine weitere Ladestrategie: das Preloading in Verbindung mit dem Relationship Fixup im RAM. Dabei schickt der Softwareentwickler mehrere LINQ-Befehle für die verbundenen Objekte explizit ab, und der OR-Mapper setzt die jeweils neu hinzukommenden Objekte bei ihrer Materialisierung mit denjenigen Objekten zusammen, die sich bereits im RAM befinden. Nach der Befehlsfolge

```
var flug = ctx.FlugSet
      .SingleOrDefault(x => x.FlugNr == 101);
ctx.PilotSet.Where(p => p.FluegeAlsPilot.Any(x => x.FlugNr == 101) ||
p.FluegeAlsCopilot.Any(x => x.FlugNr == 101)).ToList();
```

findet man im RAM beim Zugriff auf flug.Pilot und flug.Copilot tatsächlich das entsprechende Pilot- und Copilotobjekt von Flug 101. Entity Framework Core erkennt beim Laden der beiden Piloten, dass es im RAM bereits ein Flug-Objekt gibt, das diese beiden Piloten als Pilot bzw. Copilot braucht. Es setzt dann im RAM das Flug-Objekt mit den beiden Piloten-Objekten zusammen. Diese Funktion nennt man Relationship Fixup.

Während in den beiden obigen Zeilen gezielt Pilot und Copilot für den Flug 101 geladen wurden, kann der Softwareentwickler das Relationship Fixup auch für die Optimierung durch Caching verwenden. Das nächste Listing zeigt, dass alle Piloten geladen werden und danach einige Flüge. Zu jedem geladenen Flug stehen danach Pilot- und Copilot-Objekt zur

Verfügung. Wie immer beim Caching braucht man hier natürlich etwas mehr RAM, da auch Pilot-Objekte geladen werden, die niemals gebraucht werden. Außerdem muss man sich bewusst sein, dass man ein Aktualitätsproblem haben kann, weil die abhängigen Daten auf dem gleichen Stand sind wie die Hauptdaten. Aber das verhält sich beim Caching bekanntlich stets auf diese Weise. Allerdings spart man Rundgänge zum Datenbankmanagementsystem, und damit verbessert man die Geschwindigkeit.

Bemerkenswert ist, dass weder beim obigen Laden der beiden Piloten noch beim Laden aller Piloten in dem Beispiel das Ergebnis der LINQ-Abfrage einer Variablen zugewiesen wird. Dies ist tatsächlich nicht notwendig, denn Entity Framework Core enthält (genau wie das bisherige Entity Framework) in seinem First-Level-Cache einen Verweis im RAM auf alle Objekte, die jemals in eine bestimmte Instanz der Kontextklasse geladen wurden. Das Relationship Fixup funktioniert daher auch ohne Speicherung in einer Variablen. Die Zuweisung zu einer Variablen (List<Pilot> allePiloten = ctx.PilotSet.ToList();) ist freilich nicht schädlich, sondern kann sinnvoll sein, wenn man im Programmablauf eine Liste aller Piloten braucht. Zu beachten ist auch, dass das Relationship Fixup nicht kontexinstanzübergreifend funktioniert. Ein dafür notwendiger Second-Level-Cache ist in Entity Framework Core bisher nicht vorhanden.

Das obige Codefragment zeigt beim Laden der beiden Piloteninformationen auch sehr schön, dass man den Join-Operator in Entity Framework Core vermeiden kann, wenn man die Navigationseigenschaften und die Any()-Methode verwendet. Any() prüft, ob es mindestens einen Datensatz gibt, der eine Bedingung erfüllt oder nicht erfüllt. Im obigen Fall reicht es, dass der Pilot einmal für den gesuchten Flug als Pilot oder Copilot zugeteilt wurde. In anderen Fällen kann man die LINQ-Methode All() einsetzen: Wenn man eine Menge von Datensätzen ansprechen will, die alle eine Bedingung erfüllen oder nicht erfüllen.

Listing 10.4 Caching der Piloten. Egal, welchen Flug man danach lädt, Pilot- und Copilot-Objekt sind vorhanden.

```
/// <summary>
  /// Liefert Pilot-, Buchungs- und Passagierinformationen via Preloading /
RelationshipFixup
  /// </summary>
  public static void Demo_PreLoading()
  {
   CUI.Headline(nameof(Demo_PreLoading));

   using (var ctx = new WWWingsContext())
   {

    int flugNr = 101;

    // 1. Lade nur den Flug selbst
    var f = ctx.FlugSet
          .SingleOrDefault(x => x.FlugNr == flugNr);

    // 2. Lade beide Piloten für obigen Flug
    ctx.PilotSet.Where(p => p.FluegeAlsPilot.Any(x => x.FlugNr == flugNr) ||
p.FluegeAlsCopilot.Any(x => x.FlugNr == flugNr)).ToList();

    // 3. Lade andere Flüge dieser beiden Piloten
    ctx.FlugSet.Where(x => x.PilotId == f.PilotId || x.CopilotId == f.CopilotId).ToList();
```

```
    // 4. Lade Buchungen für obigen Flug
    ctx.BuchungSet.Where(x => x.FlugNr == flugNr).ToList();

    // 5. Lade Passagiere für obigen Flug
    ctx.PassagierSet.Where(p => p.Buchungen.Any(x => x.FlugNr == flugNr)).ToList();

    // nicht notwendig: ctx.ChangeTracker.DetectChanges();

    Console.WriteLine($"Flug Nr {f.FlugNr} von {f.Abflugort} nach {f.Zielort} hat
{f.FreiePlaetze} freie Plätze!");
    if (f.Pilot != null) Console.WriteLine($"Pilot: {f.Pilot.Name} hat {f.Pilot.
FluegeAlsPilot.Count} Flüge als Pilot!");
    else Console.WriteLine("Kein Pilot zugewiesen!");
    if (f.Copilot != null) Console.WriteLine($"Copilot: {f.Copilot.Name} hat
{f.Copilot.FluegeAlsCopilot.Count} Flüge als Copilot!");
    else Console.WriteLine("Kein Copilot zugewiesen!");

    Console.WriteLine("Anzahl Passagiere auf diesem Flug: " + f.Buchungen.Count);
    Console.WriteLine("Passagiere auf diesem Flug:");
    foreach (var b in f.Buchungen)
    {
     Console.WriteLine("- Passagier #{0}: {1} {2}", b.Passagier.PersonID,
b.Passagier.Vorname, b.Passagier.Name);
    }
   }
  }
```

Das nächste Listing zeigt die Neufassung der Aufgabe, dieses Mal mit Preloading und Relationship Fixup statt Eager Loading. Hier werden Flug, Piloten, deren andere Flüge, Buchungen und Passagiere einzeln geladen. Der Programmcode sendet also fünf SELECT-Befehle zum Datenbankmanagementsystem (im Gegensatz zu den vier SELECT-Befehlen, die die Lösung mit Eager Loading sendet), vermeidet dabei aber einige Joins.

Der Relationship Fixup-Trick wirkt sich positiv aus, wenn eine oder mehrere der folgenden Bedingungen erfüllt sind:

- Die Ergebnismenge der Hauptdaten ist groß und die Menge der abhängigen Daten ist klein.
- Es gibt mehrere verschiedene abhängige Datenmengen, die vorab geladen werden können.
- Die vorab geladenen Objekte sind selten veränderliche (Stamm-)Daten.
- Man führt in einer einzigen Kontextinstanz mehrere Abfragen aus, die die gleichen abhängigen Daten besitzen.

Listing 10.5 Laden von Flug, Piloten, Buchungen und Passagieren in getrennten LINQ-Befehlen. Entity Framework Core setzt die getrennt geladenen Objekte im RAM zusammen.

```
/// <summary>
  /// Liefert Pilot-, Buchungs- und Passagierinformationen via Preloading /
RelationshipFixup
  /// </summary>
  public static void Demo_PreLoading()
  {
   CUI.Headline("Demo_PreLoading");
```

```
   using (var ctx = new WWWingsContext())
   {

    int flugNr = 101;

    // 1. Lade nur den Flug
    var f = ctx.FlugSet
         .SingleOrDefault(x => x.FlugNr == flugNr);

    // 2. Lade beide Piloten
    ctx.PilotSet.Where(p => p.FluegeAlsPilot.Any(x => x.FlugNr == flugNr) ||
p.FluegeAlsCopilot.Any(x => x.FlugNr == flugNr)).ToList();

    // 3. Lade andere Flüge der Piloten
    ctx.FlugSet.Where(x => x.PilotId == f.PilotId || x.CopilotId == f.CopilotId).ToList();

    // 4. Lade Buchungen
    ctx.BuchungSet.Where(x => x.FlugNr == flugNr).ToList();

    // 5. Lade Passagiere
    ctx.PassagierSet.Where(p => p.Buchungen.Any(x => x.FlugNr == flugNr)).ToList();

    // nicht notwendig: ctx.ChangeTracker.DetectChanges();

    Console.WriteLine($"Flug Nr {f.FlugNr} von {f.Abflugort} nach {f.Zielort} hat
{f.FreiePlaetze} freie Plätze!");
    if (f.Pilot != null) Console.WriteLine($"Pilot: {f.Pilot.Name} hat {f.Pilot.
FluegeAlsPilot.Count} Flüge als Pilot!");
    else Console.WriteLine("Kein Pilot zugewiesen!");
    if (f.Copilot != null) Console.WriteLine($"Copilot: {f.Copilot.Name} hat
{f.Copilot.FluegeAlsCopilot.Count} Flüge als Copilot!");
    else Console.WriteLine("Kein Copilot zugewiesen!");

    Console.WriteLine("Anzahl Passagiere auf diesem Flug: " + f.Buchungen.Count);
    Console.WriteLine("Passagiere auf diesem Flug:");
    foreach (var b in f.Buchungen)
    {
     Console.WriteLine("- Passagier #{0}: {1} {2}", b.Passagier.PersonID,
b.Passagier.Vorname, b.Passagier.Name);
    }
   }
  }
```

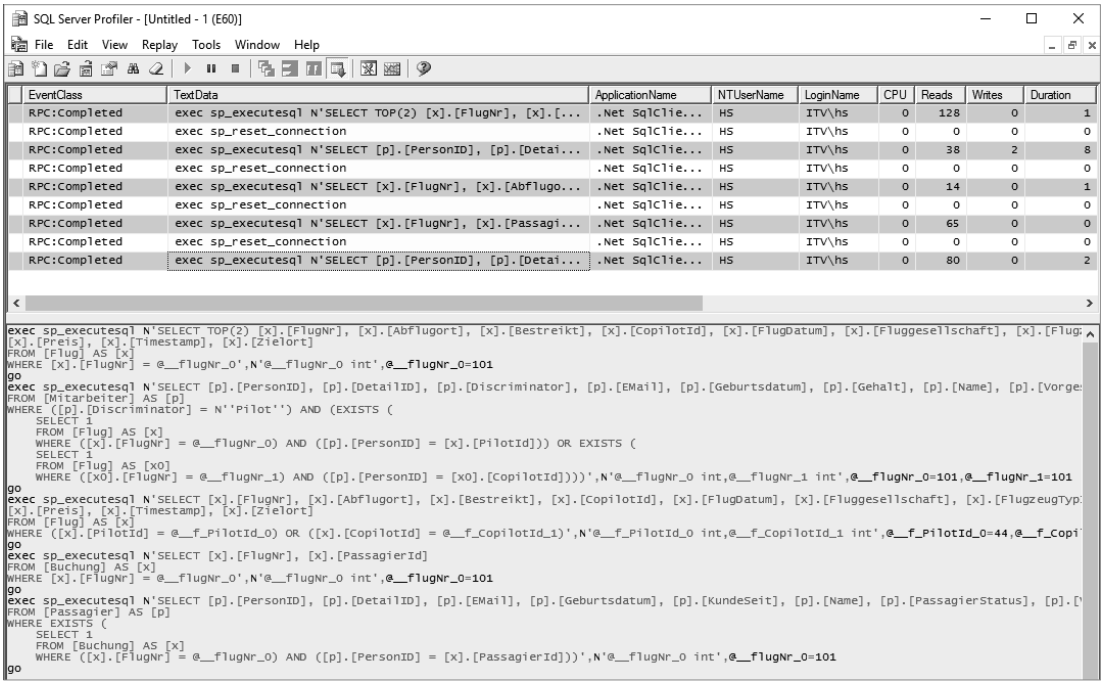

Bild 10.6 Der SQL Server-Profiler zeigt die fünf SQL-Befehle, die das obige Listing

Im Geschwindigkeitsvergleich zeigt sich auch bei dem hier besprochenen Szenario des Ladens eines Flugs mit Piloten und Passagieren bereits ein Geschwindigkeitsvorteil für das Preloading. Die Messung, die in nachstehenden Abbildung dargestellt ist, wurde zur Vermeidung von Messabweichungen mit 51 Durchläufen ermittelt, wobei der erste Durchlauf (Kaltstart für den Entity Framework Core-Kontext und ggf. auch die Datenbank) jeweils nicht berücksichtigt wurde. Zudem wurden alle Bildschirmausgaben ausgebaut.

Selbstverständlich kann man Eager Loading und Preloading beliebig mischen. In der Praxis muss man jedoch für jeden einzelnen Fall das optimale Verhältnis finden.

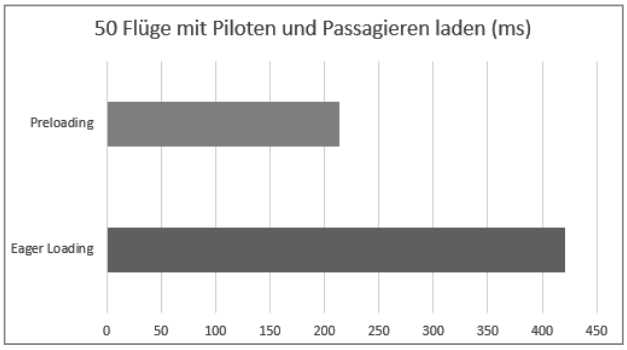

Bild 10.7 Geschwindigkeitsvergleich von Eager Loading und Preloading

10.6 Details zum Relationship Fixup

Wie beim klassischen Entity Framework erfolgt auch bei Entity Framework Core im Rahmen des Relationship Fixup eine Zuweisung der Instanz eines Mengentyps zu der Navigationseigenschaft, wenn die Navigationseigenschaft den Wert null hat.

Dieser Automatismus ist gegeben, unabhängig davon, ob der Softwareentwickler bei der Deklaration der Navigationseigenschaft als Typ eine Schnittstelle oder eine Klasse verwendet hat. Entity Framework Core instanziiert die Mengenklasse selbst. Im Fall der Deklaration der Navigationseigenschaft mit einem Schnittstellentyp wählt Entity Framework Core selbst eine geeignete Mengenklasse. Bei ICollection<T> wird HashSet<T> gewählt, bei IList<T> eine List<T>.