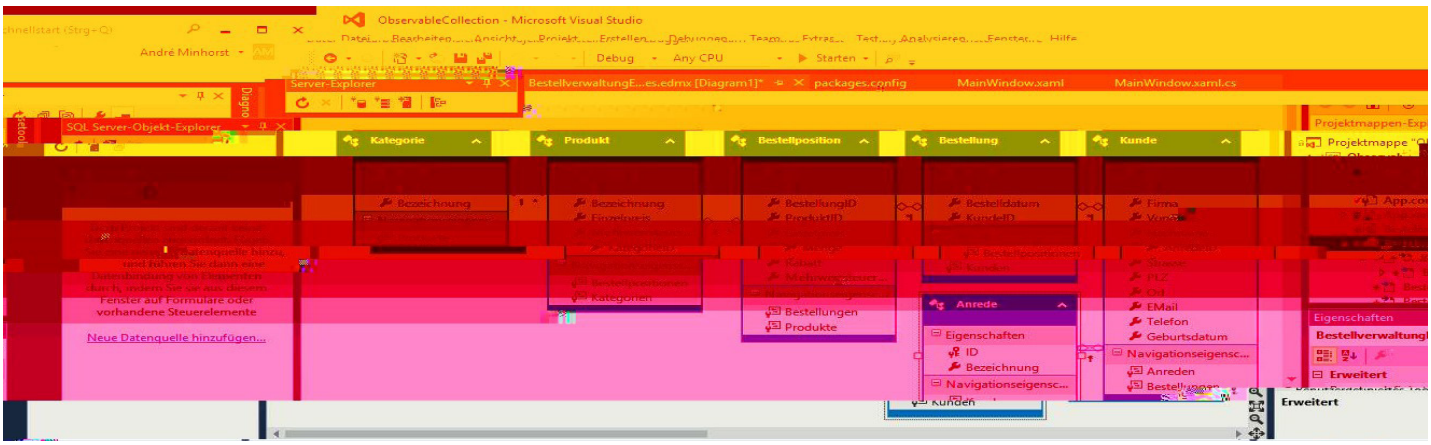
AUSGABE 04/2018

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

André Minhorst Verlag

Impressum

DATENBANKENTWICKLER
© 2018 André Minhorst Verlag
Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

.NET Core auf Version 2.1 aktualisieren

Der Vorteil eines Access-Entwicklers gegenüber einem .NET-Entwickler ist, dass er sich in den letzten zehn Jahren kaum Sorgen über Updates machen muss. Bei .NET und auch .NET Core ist das anders: Hier gibt es alle paar Monate Neuigkeiten. In diesem Fall steht ein Update auf .NET Core 2.1 an. Dieser kurze Artikel zeigt, welche Schritte notwendig sind, um die in dieser Ausgabe vorgestellten Artikel nachvollziehen zu können.

.NET Core ist noch sehr frisch, daher liefert Microsoft in sehr kurzen Abständen neue Versionen. Diese neuen Versionen enthalten oft tolle Features, die wir auch in diesem Magazin vorstellen wollen.

Daher informieren wir Sie hier über Updates, die notwendig sind, damit die in den Artikel beschriebenen Beispiele lauffähig sind und nachvollzogen werden können.

In diesem Fall geht es um die neue Version von .NET Core mit der Nummer 2.1. Diese installieren Sie unter Visual Studio 2017 (in unserem Beispiel mit der Community Edition).

Den Download finden Sie zum Zeitpunkt der Erstellung dieses Artikels unter der folgenden Internetadresse (siehe Bild 1):

```
https://www.microsoft.com/net/download
```

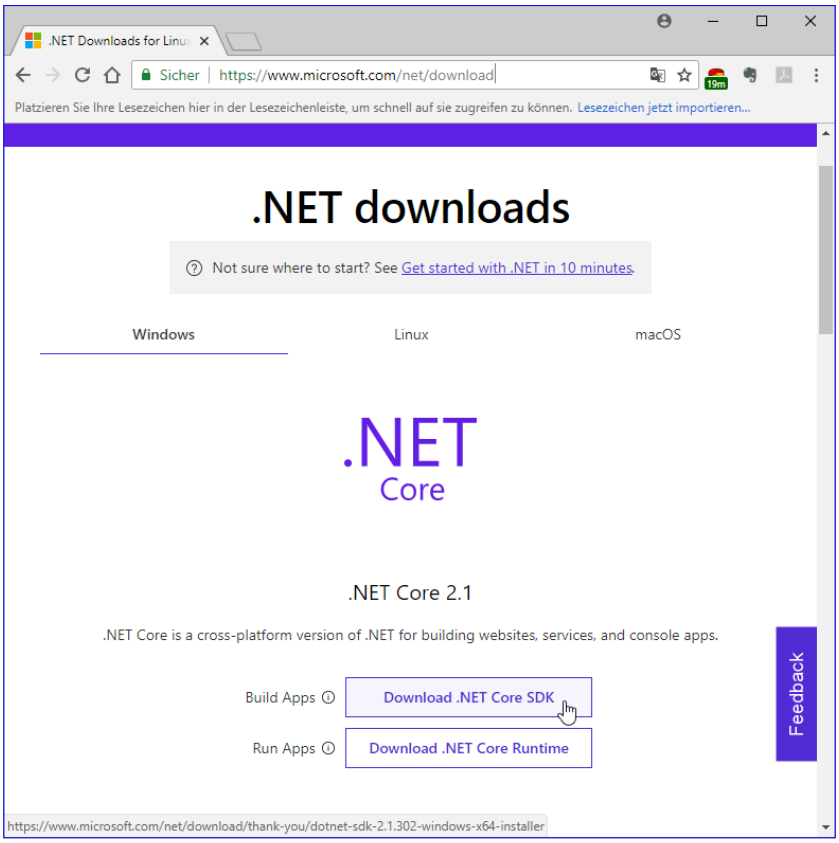


Bild 1: Download von .NET Core SDK 2.1

Hier wählen Sie die Version mit dem SDK aus. Nach dem Download installieren Sie die neue Version durch einen Doppelklick auf die Datei **dotnet-sdk-2.1.302-win-x64.exe** (siehe Bild 2).

Gegebenenfalls bittet das Setup Sie darum, geöffnete Instanzen von Visual Studio zu schließen – damit sparen Sie sich einen späteren Neustart des Systems. Nach der Installation sind Sie gut auf die in dieser Ausgabe vorgestellten Artikeln rund um das Thema Webentwicklung vorbereitet!

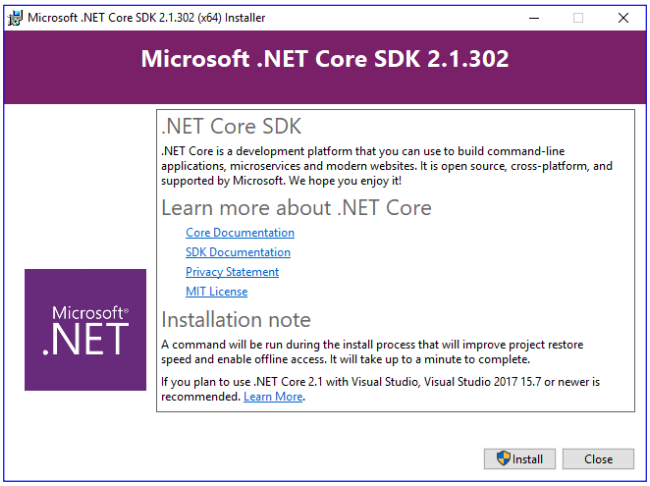


Bild 2: Installation von .NET Core SDK 2.1

Änderungen gegenüber Version 2.0

Wenn Sie bisher mit der Version 2.0 gearbeitet haben und nun ein neues Projekt auf Basis der Version 2.1 erstellen, werden Sie ein paar Unterschiede im Projektaufbau finden – wenn auch nur geringe.

In der Version ohne Authentifizierung finden Sie etwa ein neues Verzeichnis namens **Shared** im Ordner **Pages**, der unter anderem die Datei **_Layout.cshtml** enthält, welche den grundlegenden Aufbau der Seite definiert. Noch mehr Unterschiede erwarten Sie, wenn Sie die Authentifizierung direkt integrieren (siehe Bild 3).

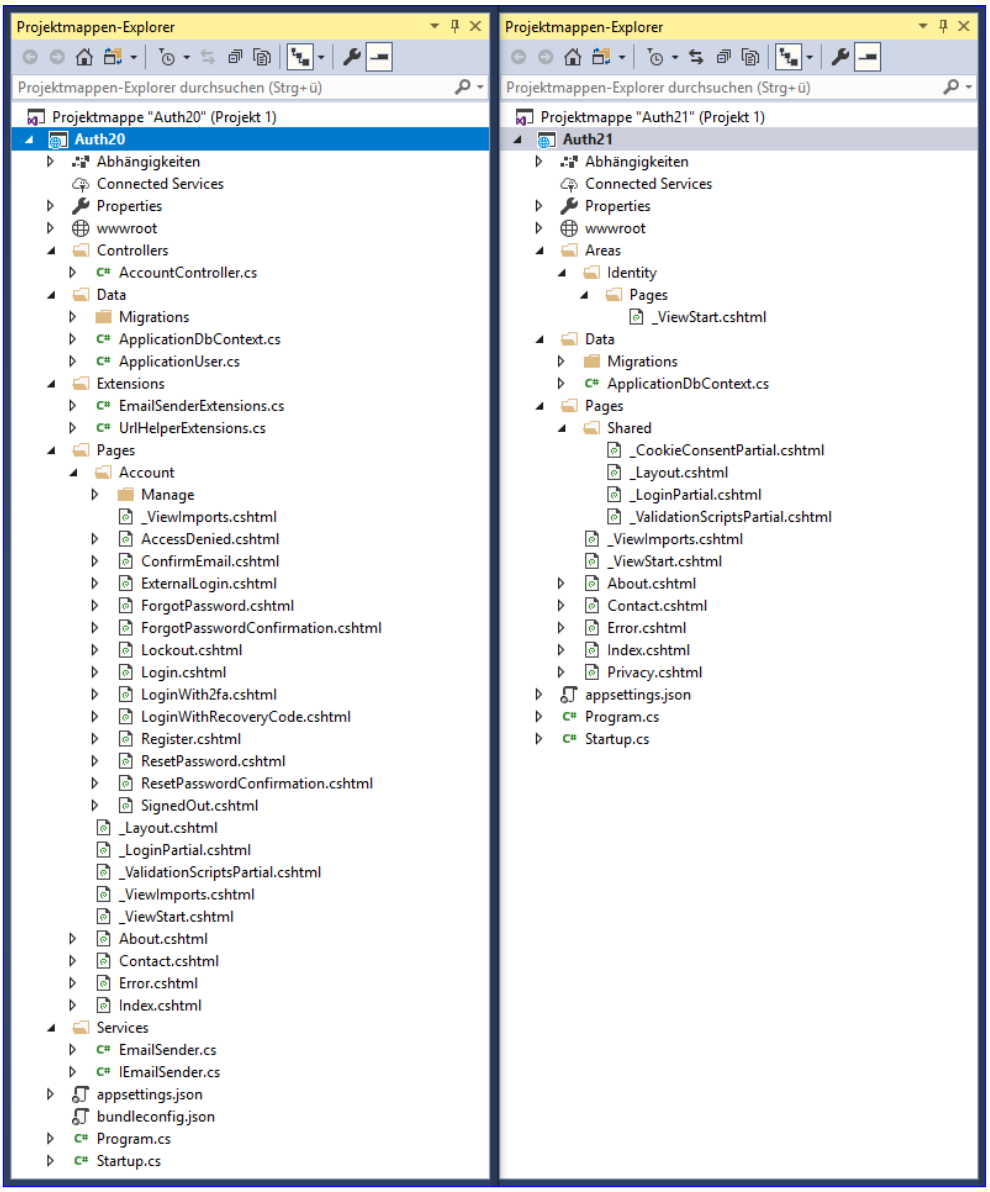


Bild 3: Unterschiedlicher Aufbau der Dateien

Hier finden Sie einen ganz neuen Bereich namens **Areas**, in dem sich die Ordner **Identity** und **Pages** befinden. Letzterer enthält allerdings lediglich eine Datei namens **_ViewStart.cshtml** und nicht mehr den ganzen Rutsch Dateien, den Sie noch in der Version 2.0 vorgefunden haben. Der Grund: Die Dateien wurden nun in einer Bibliothek zusammengefasst, die nur noch referenziert wird.

Wenn Sie eine oder mehrere Dateien manuell anpassen möchten, müssen Sie diese mit dem Menüeintrag **Neues Gerüstelement...** zum Projekt hinzufügen. Wie das im Detail funktioniert, erfahren Sie im Artikel **ASP.NET Core: Authentifizierungsseiten anpassen**.

EDM: 1:n-Beziehungen per Code First

Im Artikel »EDM: Der Code First-Ansatz« haben wir gezeigt, wie Sie unter Nutzung des Entity Frameworks ein Datenmodell per Klassenmodell programmieren und dann mit ein paar Befehlen dafür sorgen, dass die Klassen in Form eines Datenmodells in einer Datenbank umgesetzt werden. Im vorliegenden Artikel schauen wir uns nun im Detail an, wie Sie 1:n-Beziehungen mit Code First definieren, die dann in die Zieldatenbank übertragen werden.

Vorbereitungen

Für diesen Artikel nutzen wir ein Projekt auf Basis der Vorlage **Visual C#|Windows Desktop|WPF-App** namens **CustomerManagement**. Die Techniken lassen sich aber auch in Projekten auf Basis anderer Vorlagen nutzen, beispielsweise in Webanwendungen.

Fügen Sie dem Projekt ein ADO.NET Entity Data Model hinzu.

- Dazu betätigen Sie den Kontextmenü-Eintrag **Hinzufügen|Neues Element** des Projekt-Eintrags im Projektmappen-Explorer.
- Im nun erscheinenden Dialog **Neues Element hinzufügen** wählen Sie den Eintrag **ADO.NET Entity Data Model** aus. Das Element benennen wir mit **CustomerManagementContext**.
- Wählen Sie im folgenden Dialog die Option **Leeres Code First-Modell** aus.

Damit erstellt der Assistent nun eine neue Klasse namens **CustomerManagementContext.cs**, die ein paar auskommentierte Beispielanweisungen enthält. Außerdem finden Sie in der Datei **App.conf** ein Element namens **connectionStrings**. Hier passen wir den Namen der zu erstellenden Datenbank mit dem Attribut **initial catalog** auf **CustomerManagement** an:

```
01.     <connectionStrings>
02.       <add name="CustomerManagementContext" connectionString="data source=(LocalDb)\MSSQLLocalDB;
              initial catalog=CustomerManagement;integrated security=True;MultipleActiveResultSets=True;
              App=EntityFramework" providerName="System.Data.SqlClient" />
03.     </connectionStrings>
```

Dadurch wird die Datenbank beim Erstellen **CustomerManagement** benannt.

Beispiel

Unser einfaches Beispiel soll eine Klasse namens **Customer** und eine namens **Salutation** enthalten und daraus die Tabellen **Customers** und **Salutations** erzeugen, die über ein Feld **SalutationID** der Tabelle **Customers** und das gleichnamige Feld der Tabelle **Salutations** miteinander verknüpft sind.

Klassen erstellen und DbContext einrichten

Zur Erinnerung: Wir können die einzelnen Klassen direkt in der Klassendatei **CustomerManagementContext.cs** unterbringen, aber auch in jeweils einer eigenen Datei. In letzterem Fall würden wir die Klassen aus Gründen der Übersicht in einem Unterverzeichnis namens **Model** anlegen. Dieses Verzeichnis fügen wir über den Kontextmenü-Eintrag **Hinzufügen|Neuer Ordner** zum Projektordner hinzu. Dann legen wir darin eine neue Klassendatei namens **Customer.cs** an. Dadurch wird für diese Klasse automatisch der Namespace **CustomerManagement.Models** festgelegt, was wir später noch berücksichtigen müssen. Wir ändern die Sichtbarkeit der Klasse auf öffentlich (**public**) und fügen ein paar Felder hinzu – unter anderem ein Feld namens **ID**, das später als Primärschlüsselfeld verwendet werden soll und eines namens **SalutationID**, das als Fremdschlüsselfeld zur Tabelle **Salutations** zum Einsatz kommt:

```
namespace CustomerManagement.Models {
    public class Customer {
        public int ID { get; set; }
        public string Company { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int SalutationID { get; set; }
    }
}
```

Auf die gleiche Weise erzeugen wir eine Klasse für die Anreden, die wir Salutation nennen. Auch diese legen wir im Verzeichnis **Models** an und stellen die Sichtbarkeit auf öffentlich ein. Dieses Klasse soll die Felder **ID** und **Name** erhalten:

```
namespace CustomerManagement.Models {
    public class Salutation {
        public int ID { get; set; }
        public string Name { get; set; }
    }
}
```

Nun müssen wir noch die **DbSet**-Elemente in der Klasse **CustomerManagementContext** hinzufügen. Damit wir dabei auf die Klassen **Customer** und **Salutation** als Datentypen der Auflistungen zugreifen können, machen wir den Namespace **CustomerManagement.Models** mit der **using**-Anweisung verfügbar:

```
namespace CustomerManagement {
    ...
    using CustomerManagement.Models;
```

Dann folgt die eigentliche Klasse mit dem zunächst leeren Konstruktor:

```
    public class CustomerManagementContext : DbContext {
```

```
        public CustomerManagementContext() : base("name=CustomerManagementContext") {
        }
```

Diese enthält dann schließlich die **DbSet**-Elemente für die **Customers**- und die **Salutations**-Auflistungen:

```
        public virtual DbSet<Salutation> Salutations { get; set; }
        public virtual DbSet<Customer> Customers { get; set; }
    }
}
```

Damit haben wir die Voraussetzungen geschaffen, erstmalig die Datenbank zu erstellen. Das erledigen wir initial mit den drei folgenden Anweisungen, die wir in der Paket-Manager-Konsole absetzen. Die erste lautet wie folgt und aktiviert die Migration zum Datenbankserver. Dabei wird ein Verzeichnis namens **Migrations** erstellt sowie eine Datei namens **Configurations.cs**, die unter anderem die **Seed**-Methode enthält, in der Sie Anweisungen zum Füllen der erzeugten Tabellen mit Daten hinzufügen können (dieser Befehl ist ab Version 2.1 nicht mehr nötig):

```
enable-migrations
```

Die zweite erstellt den Code für die initiale Migration. Dieser wird in der Datei angelegt, die mit einer Zahl beginnt und mit **...Init.cs** endet:

```
add-migration Init
```

Nun müssen Sie die Migration nur noch aufrufen, was Sie zum Beispiel mit der folgenden Anweisung erledigen:

```
update-database
```

Alternativ können Sie dafür sorgen, dass die Migrationen beim Start der Anwendung automatisch durchgeführt werden. Dazu rufen Sie einmalig den folgenden Befehl in der Paket-Manager-Konsole auf:

```
enable-migrations -EnableAutomaticMigration:$true
```

Sollten Sie die Migration bereits einmal ausgeführt haben, müssen Sie noch den Parameter **-Force** anhängen:

```
enable-migrations -EnableAutomaticMigration:$true -Force
```

Dies passt die Klasse **MigrationsIConfiguration.cs** an, in der die Eigenschaft **AutomaticMigrationsEnabled** in der Konstruktor-Klasse auf **true** gesetzt wird:

```
public Configuration() {
    AutomaticMigrationsEnabled = true;
```

```
}
```

In der Klasse **CustomerManagementContext** fügen wir der Konstruktor-Klasse die folgende Anweisung hinzu:

```
public class CustomerManagementContext : DbContext {
    public CustomerManagementContext() : base("name=CustomerManagementContext") {
        Database.SetInitializer(new MigrateDatabaseToLatestVersion<CustomerManagementContext, Migrations.Configuration>());
    }
    public virtual DbSet<Salutation> Salutations { get; set; }
    public virtual DbSet<Customer> Customers { get; set; }
}
```

Dabei ist das erste Element in spitzen Klammern hinter dem Objekt **MigrateDatabaseToLatestVersion** der Name der Kontextklasse und das zweite die **Configuration**-Klasse im Namespace **Migrations**.

Um die Migration beim Start der Anwendung jeweils auszulösen (sofern Änderungen vorliegen), erweitern Sie noch die Konstruktor-Methode der Klasse **MainWindow**:

```
public MainWindow() {
    InitializeComponent();
    CustomerManagementContext dbContext = new CustomerManagementContext();
    dbContext.Database.Initialize(true);
}
```

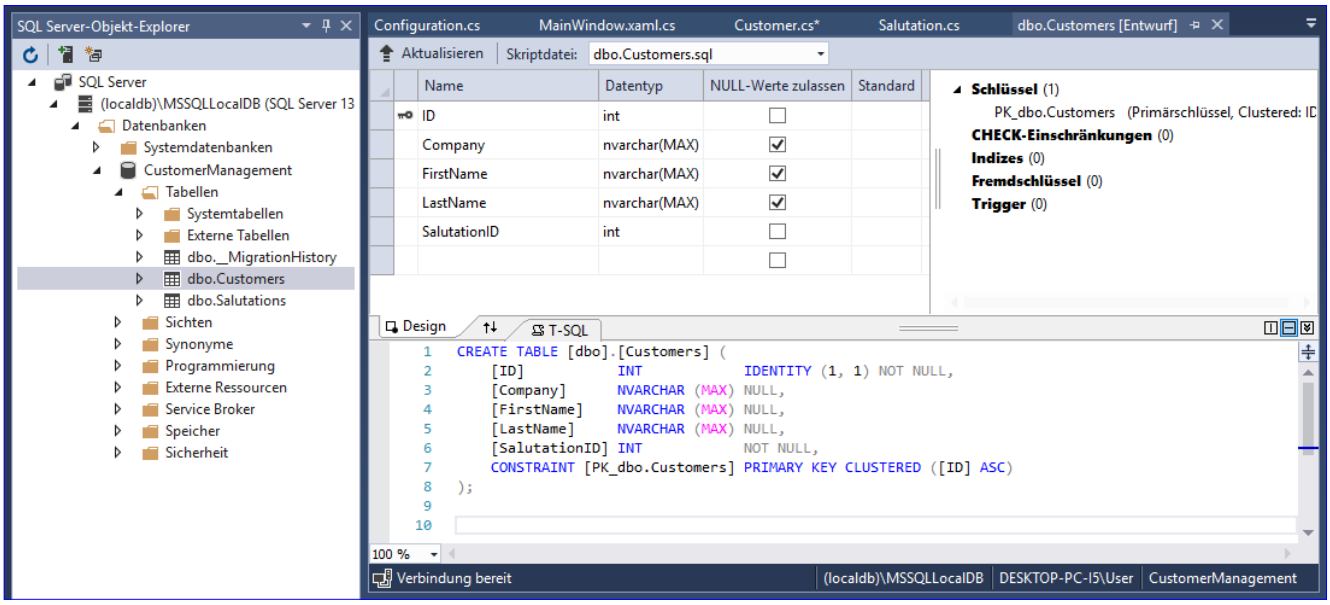


Bild 1: Definition der Tabelle **Customers**

EDM: 1:1-Beziehungen per Code First

Im Artikel »EDM: 1:n-Beziehungen per Code First« haben wir gezeigt, wie Sie unter Nutzung des Entity Frameworks zwei Klassen so gestalten, dass diese beim Migrieren in einer SQL Server-Datenbank in zwei miteinander per 1:n-Beziehung verknüpfte Tabellen umgewandelt werden. Die 1:1-Beziehung ist ein Spezialfall der 1:n-Beziehung, für die eine kleine Sonderbehandlung notwendig ist. Wie des gelingt, zeigt der vorliegende Beitrag.

Vorbereitungen

Für diesen Artikel nutzen wir ein Projekt auf Basis der Vorlage **Visual C#|Windows Desktop|WPF-App** namens **EDMConfigure11**. Die Techniken lassen sich aber auch in Projekten auf Basis anderer Vorlagen nutzen, beispielsweise in Webanwendungen. Wir fügen wie im Artikel EDM: 1:n-Beziehungen per Code First ein ADO.NET Entity Data Model hinzu, wieder namens **CustomerManagementContext**, und starten auch hier wieder mit einem leeren Code First-Modell. Das Attribut **initial catalog** in der Verbindungszeichenfolge in der Datei **App.conf** stellen wir auf **EDMConfigure11.CustomerManagement** ein.

Beispiel

Unser einfaches Beispiel soll eine Klasse namens **Customer** und eine namens **CustomerInvoiceAddress** enthalten und daraus die Tabellen **Customers** und **CustomerInvoiceAddresses** erzeugen, die über die Felder **ID** der Tabelle **Customers** und das gleichnamige Feld der Tabelle **CustomerInvoiceAddresses** miteinander verknüpft sind.

Klassen erstellen und DbContext einrichten

In einem neuen Verzeichnis namens **Data** legen wir die als öffentlich deklarierte Klasse namens **Customer.cs** mit dem folgenden Code an:

```
namespace CustomerManagement.Data {
    public class Customer {
        public int ID { get; set; }
        public string Company { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

Auf die gleiche Weise erzeugen wir eine Klasse für die Rechnungsadressen, die wir **CustomerInvoiceAddress** nennen. Auch diese legen wir im Verzeichnis **Data** an und stellen die Sichtbarkeit auf öffentlich ein:

Nun müssen wir noch die **DbSet**-Elemente in der Klasse **CustomerManagementContext** hinzufügen. Damit wir dabei auf die Klassen **Customer** und **Salutation** als Datentypen der Auflistungen zugreifen können, machen wir den Namespace **CustomerManagement.Data** mit der **using**-Anweisung verfügbar:

```
namespace EDMConfigure11.Data {
    public class CustomerInvoiceAddress {
        public int ID { get; set; }
        public string Street { get; set; }
        public string Zip { get; set; }
        public string City { get; set; }
    }
}
```

Die Datei **CustomerManagementContext.cs** passen wir nun wie folgt an. Dabei fügen wir per **using**-Anweisung den Namespace **EDMConfigure11.Data** an, damit wir auf die in dem Verzeichnis **Data** enthaltenen Klassen, die dort auch mit dem entsprechenden Namespace versehen sind, zugreifen können. Außerdem erweitern wir die Klasse um die beiden **DbSet**-Elemente namens **Customers** und **CustomersInvoiceAddresses**:

```
namespace EDMConfigure11 {
    ...
    using EDMConfigure11.Data;
    public class CustomerManagementContext : DbContext {
        public CustomerManagementContext()
            : base("name=CustomerManagementContext") {
        }
        public virtual DbSet<Customer> Customers { get; set; }
        public virtual DbSet<CustomerInvoiceAddress> CustomerInvoiceAddresses { get; set; }
    }
}
```

Datenbank erstellen

Nun migrieren wir die Klassen erstmals in die Zieldatenbank, die noch zu erstellen ist. Um das Migrieren aus dem Projekt heraus zu ermöglichen, rufen wir zunächst die folgende Anweisung in der Paket-Manager-Konsole auf, was das Verzeichnis **Migrations** mit der Datei **Configure.cs** erstellt:

```
enable-migrations
```

Dann erstellen wir das Skript für die erste Migration, was eine Datei zum Verzeichnis **Migrations** hinzufügt, die auf **_Init.cs** endet:

```
add-migration Init
```

Schließlich erstellen wir erstmalig die Datenbank:

```
update-database
```

Dies erstellt die beiden Tabellen **Customers** und **CustomerInvoiceAddresses** in der neuen Datenbank **EDMConfigure11.CustomerManagement**. Allerdings enthalten diese noch keinerlei Beziehung zueinander, wie Bild 1 zeigt.

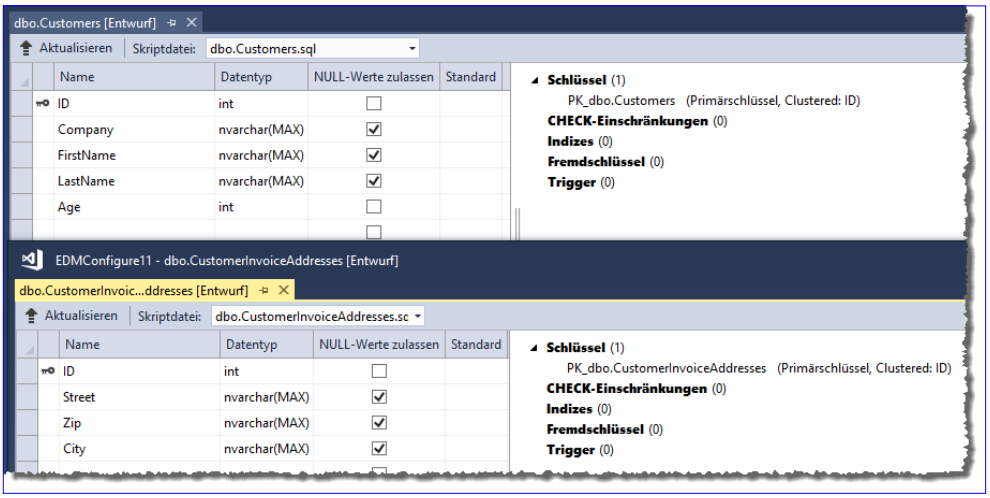


Bild 1: Definitionen der Tabellen **Customers** und **CustomerInvoiceAddresses**

1:1-Beziehung erzeugen

Wenn wir bei der 1:n-Beziehung der Klasse, welche die Seite der Beziehung mit dem Fremdschlüsselfeld der Beziehung bilden soll, eine Eigenschaft des Typs der zu verknüpfenden Klasse zugewiesen haben, kann dies auch hier nicht verkehrt sein. Also fügen wir die Eigenschaft **CustomerInvoiceAddress** mit dem entsprechenden Namen zu **Customer.cs** hinzu:

```
public class Customer {
    public int ID { get; set; }
    ...
    public CustomerInvoiceAddress CustomerInvoiceAddress { get; set; }
}
```

Bei der 1:n-Beziehung haben wir der Klasse für die Seite der Beziehung mit dem Primärschlüsselfeld, hier **Salutation**, eine **ICollection** mit den Elementen des Typs **Customer** hinzugefügt. Der Logik folgend sollten wir bei der 1:1-Beziehung dann keine **ICollection**, sondern ebenfalls eine Eigenschaft mit dem Typ des zu referenzierenden Elements hinzufügen. Das sieht dann für die Klasse **CustomerInvoiceAddress** wie folgt aus:

```
public class CustomerInvoiceAddress {
    public int ID { get; set; }
    ...
    public Customer Customer { get; set; }
}
```

Damit wollen wir nun eine erneute Migration wagen und geben die Anweisung **add-migration AddRelation** in die Paket-Manager-Konsole ein. Als Ergebnis liefert diese die folgende Fehlermeldung:

```
Das Prinzipalende einer Zuordnung zwischen den Typen 'EDMConfigure11.Data.Customer' und 'EDMConfigure11.Data.CustomerInvoiceAddress' konnte nicht ermittelt werden. Das Prinzipalende dieser Zuordnung muss mithilfe der Fluent-API für Beziehungen oder mithilfe von Datenanmerkungen explizit konfiguriert werden.
```

Mit Prinzipalende meint diese Meldung die Seite der Beziehung, welche das an der Beziehung beteiligten Primärschlüsselfeld enthält. Um das Primärschlüsselfeld festzulegen (und somit auch das Fremdschlüsselfeld), benötigen wir Data Annotations. Diese machen wir per **using**-Anweisung über den entsprechenden Namespace in der Klasse **CustomerInvoiceAddress** verfügbar:

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

Außerdem markieren wir das Feld **ID** mit der Data Annotation namens **ForeignKey** und geben für diese den Namen des zu verknüpfenden Objekts an, hier also **Customer**:

```
namespace EDMConfigure11.Data {
    public class CustomerInvoiceAddress {
        [ForeignKey("Customer")]
        public int ID { get; set; }
        public string Street { get; set; }
        public string Zip { get; set; }
        public string City { get; set; }
        public Customer Customer { get; set; }
    }
}
```

Nachdem wir erneut migrieren, erhalten wie die Entwürfe der beiden Tabellen **Customers** und **CustomerInvoiceAddresses** aus Bild 2. Hier sehen wir nun endlich, dass ein Fremdschlüssel für das Feld **ID** der Tabelle **CustomerInvoiceAddresses** angelegt wurde.

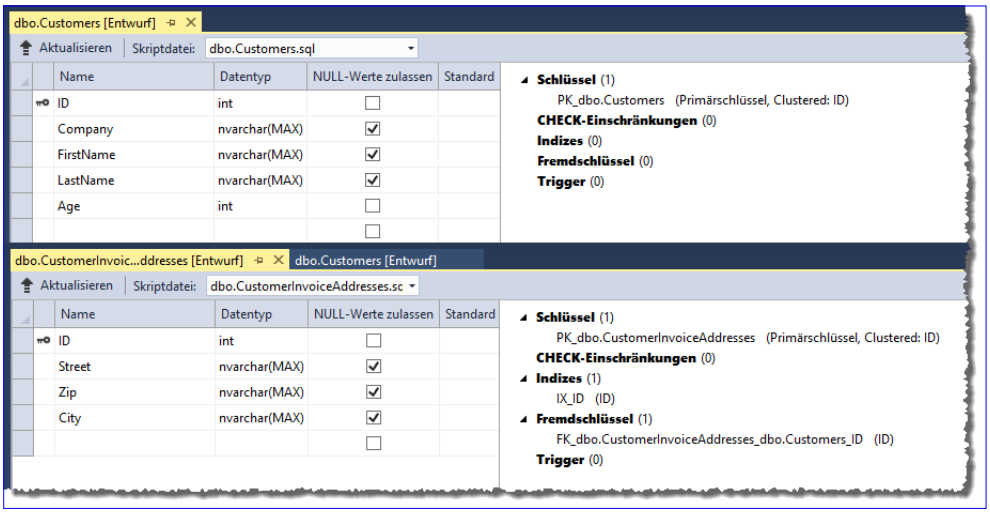


Bild 2: Tabellen mit 1:1-Beziehung

Beziehung testen

Um dies auszuprobieren, haben wir dem Fenster **Main.xaml** drei Schaltflächen hinzugefügt. In der Code behind-Datei **Main.xaml.cs** haben wir zunächst die folgende Anweisung hinterlegt, um den Datenbankkontext über die Variable **dbContext** verfügbar zu machen:

```
CustomerManagementContext dbContext = new CustomerManagementContext();
```

Die erste Schaltfläche löst den folgenden Code aus und speichert so einen neuen Datensatz in der Tabelle **Customers** – ohne einen Eintrag in der Tabelle **CustomerInvoiceAddresses** angelegt zu haben:

EF Core: Klassendiagramm anzeigen

Für die kleinen Beispiele, die wir bisher programmiert haben, benötigen Sie keine grafische Übersicht. Wenn das Projekt aber wächst und sich immer mehr Klassen ansammeln, macht es Sinn, die Klassen und ihre Beziehungen untereinander in einem Diagramm anzuzeigen. Dieser Artikel zeigt, wie Sie das mit den Bordmitteln von Visual Studio ganz einfach erledigen.

Wenn Sie ein Entity Data Model auf Basis der Vorlage **EF Designer aus Datenbank** erstellen, haben Sie keine Sorgen – das Klassendiagramm wird dann automatisch bereitgestellt. Es handelt sich um die **.edmx**-Datei, die Sie einfach zu öffnen brauchen.

Wenn Sie hingegen, wie wir es in den Entity Framework-Artikeln der laufenden Ausgaben tun, die Vorlage **Leeres Code First-Modell** nutzen, um das Modell zunächst manuell zu erstellen und erst dann die Datenbank daraus erzeugen zu lassen, finden Sie diese Datei nicht automatisch vor. Es gibt allerdings die Möglichkeit, diese Datei erzeugen zu lassen – der Einfachheit halber wollen wir das direkt beim Starten der Anwendung erledigen.

Voraussetzungen

Voraussetzung für die nachfolgend beschriebene Vorgehensweise ist, dass Ihr Projekt mindestens eine Klasse enthält, die von der Klasse **DbContext** abgeleitet ist. Eine solche Klasse wird automatisch erstellt, wenn Sie dem Projekt ein Entity Data Modell etwa auf Basis der Vorlage **Leeres Code First-Modell** hinzufügen – etwa so:

```
public class Customer {
    public int ID { get; set; }
    public string Company { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Sie benötigen ein oder mehrere Entitätsklassen, die Sie durch entsprechende Anweisungen in der **DbContext**-Klasse in **DbSet**-Auflistungen erfassen, beispielsweise wie folgt:

```
public class CustomerManagementContext : DbContext {
    public CustomerManagementContext() : base("name=CustomerManagementContext") {
    }
    public virtual DbSet<Customer> Customers { get; set; }
    }
}
```

Das unten gezeigte Klassendiagramm enthält noch weitere Objekte und Beziehungen, die wir hier aus Gründen der Übersicht nicht als Code abbilden wollen.

.edmx-Datei erstellen

Um dazu eine **.edmx**-Datei zu erstellen, brauchen Sie zwei Dinge: ein paar **using**-Anweisungen sowie ein paar Zeilen Code, die entweder beim Starten der Anwendung oder auch durch eine bestimmte Benutzeraktion ausgelöst stattfinden. Wir wollen den Code der Einfachheit halber direkt in der Konstruktor-Methode des Fensters **MainWindow.xaml** ausführen, damit wir die **.edmx**-Datei gleich beim Start der Anwendung automatisch erstellen.

Wir benötigen die folgenden beiden **using**-Anweisungen:

```
using System.Data.Entity.Infrastructure;
using System.Xml;
```

Außerdem passen wir die Konstruktor-Methode **MainWindow** wie folgt an:

```
public MainWindow() {
    InitializeComponent();
    using (var ctx = new CustomerManagementContext()) {
        using (var writer = new XmlTextWriter(AppDomain.CurrentDomain.BaseDirectory + @"\Model.edmx", Encoding.Default)) {
            EdmxWriter.WriteEdmx(ctx, writer);
        }
    }
}
```

Wenn wir die Anwendung nun starten, schreibt diese die **.edmx**-Datei in das Verzeichnis, in dem sich auch die **.exe**-Datei befindet. Diese können wir nun per Doppelklick öffnen (siehe Bild 1).

Auf diese Weise können Sie etwa prüfen, ob die per Code First, Data Annotations oder Fluent API erzeugten Klassen und Beziehungen so erstellt werden, wie Sie es sich vorstellen.

Wenn Sie Änderungen an den Klassen oder an den Fluent API-Anweisungen vornehmen, müssen Sie die Anweisungen zum Erstellen der **.edmx**-Datei erneut aufrufen.

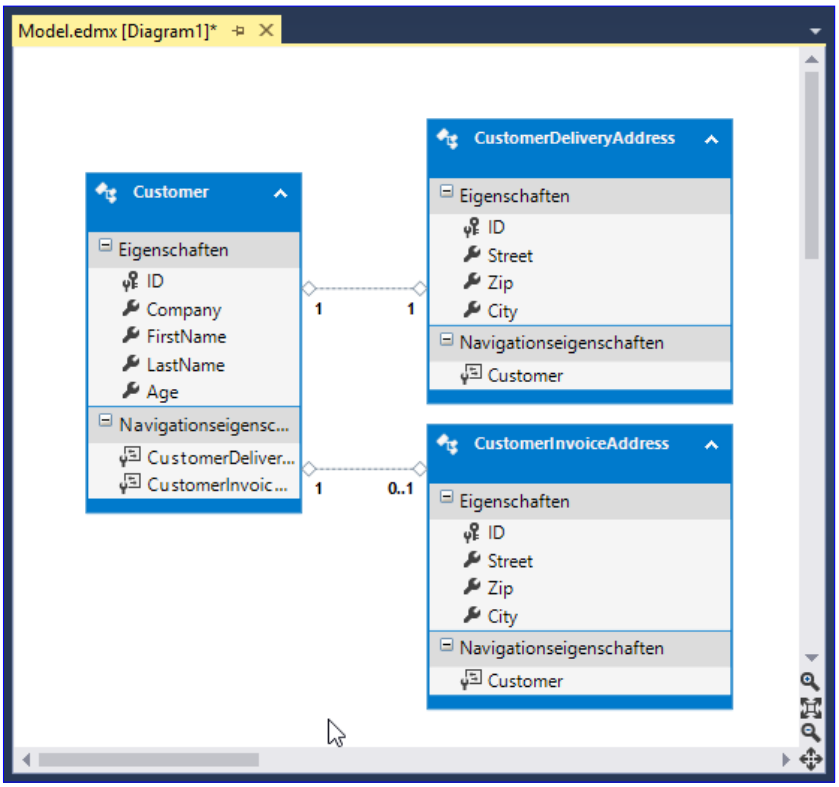


Bild 1: Per Code erzeugtes Klassendiagramm

Schnellstart mit Datenbank

In einigen Artikeln haben wir ausführlich und Schritt für Schritt erklärt, wie Sie eine Webanwendung mit Datenbankanbindung anlegen: Sie erstellen die Klassen für die einzelnen Entitäten, fügen eine Datenbankkontextklasse hinzu, legen ein paar andere notwendige Elemente an und erstellen dann mit dem Gerüstbau-Assistenten die gewünschten Seiten an, die Sie dann nach eigenen Vorstellungen erweitern können. Einige dieser Schritte können Sie sich, wenn Sie die Funktion verstanden haben, auch sparen – nach dem Anlegen der Klassen können Sie nämlich einige Schritte auf einen Schlag erledigen. Wie das gelingt, zeigt der vorliegende Beitrag.

Für die Beispiele dieses Artikels haben wir ein neues Projekt auf Basis der Vorlage Visual **C#|Web|ASP.NET Core-Webanwendung** für **ASP.NET Core 2.1** mit der Projektvorlage **Webanwendung** angelegt – ohne Hinzufügen einer Authentifizierung. Einem neuen Verzeichnis namens **Models** haben wir eine neue Klasse namens **Article.cs** hinzugefügt, welche die Artikel-Entitäten aufnehmen soll:

```
public class Article {
    public int ID { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public int CategoryID { get; set; }
    public Category Category { get; set; }
}
```

Eine weitere Klasse namens **Category** übernimmt die Kategorien:

```
public class Category {
    public int ID { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Article> Article { get; set; }
}
```

Die beiden Klassen sind über die Navigationseigenschaft **Category** beziehungsweise die **ICollection**-Auflistung **Article** miteinander verknüpft. Dann führen wir ein paar Schritte, die wir in anderen Artikeln manuell durchgeführt haben, direkt über den Gerüst-Generator aus. Dazu legen Sie ein neues Verzeichnis namens **Articles** unterhalb des Verzeichnisses **Pages** an. Wählen Sie aus dem Kontextmenü des neuen Eintrags den Befehl **Hinzufügen|Neues Gerüstelement...** aus. Im nun erscheinenden Dialog **Gerüst hinzufügen** aktivieren Sie die Option **Razor-Seiten mithilfe des Entity Frameworks (CRUD)** und klicken auf **Hinzufügen**. Da wir noch nicht, wie in anderen Beispielen, manuell eine Datenkontextklasse erstellt haben, klicken wir im Dialog aus Bild 1 auf die Schaltfläche mit dem **Plus**-Symbol neben der Eigenschaft **Datenkontextklasse**. Im nun erscheinenden Input-Fenster geben wir den Namen ein – beispielsweise wie hier vorgeschlagen **Schnellstart.Models.SchnellstartContext**. Diesen neuen Eintrag müssen

Sie dann noch für die Eigenschaft Datenkontextklasse auswählen und den Erstellungsvorgang mit einem Klick auf die Schaltfläche **Hinzufügen** starten. Danach arbeitet Visual Studio für einige Sekunden an der Erstellung der notwendigen sowie an der Anpassung vorhandener Dateien.

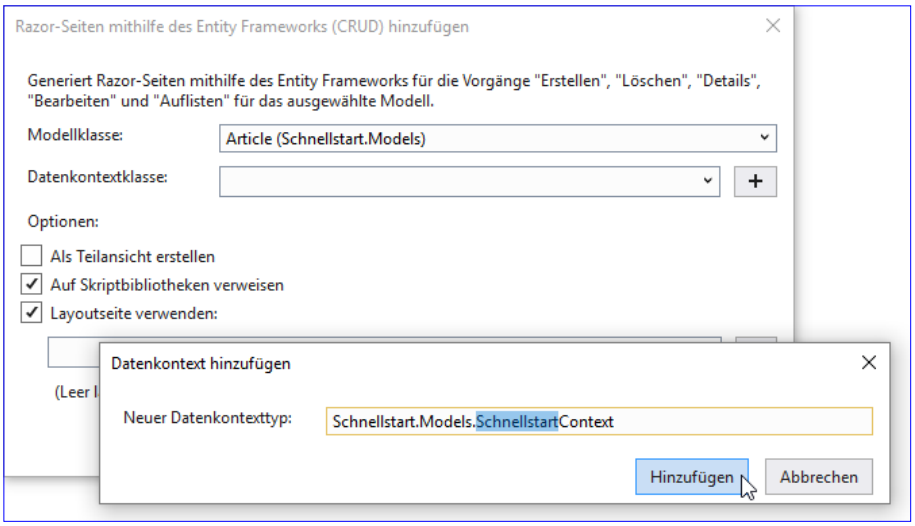


Bild 1: Anlegen einer neuen Datenkontextklasse

Was ist geschehen?

Bevor wir die Anwendung einfach einmal starten, schauen wir uns die Veränderungen in den Projektdateien an. Als Erstes fällt uns natürlich auf, dass im Verzeichnis **Pages/Articles** einige neue Dateien wie **Create.cshtml** und so weiter erscheinen – aber die waren uns ja schon von früheren Artikeln bekannt. Vorher haben wir aber noch einige Änderungen manuell durchgeführt. Diese wurden nun automatisch hinzugefügt, sofern noch nicht vorhanden. Schauen wir uns an, was noch geschehen ist. Da wäre zunächst die von uns gewünscht Datenkontextklasse. Diese wurde in einem neuen Verzeichnis namens **Data** angelegt, wurde aber dem Namespace **Schnellstart.Models** zugeordnet. Die Klasse enthält die noch leere Konstruktormethode **SchnellstartContext** sowie das **DbSet** namens **Article** (siehe Bild 2). Die Frage, die sich hier stellt, ist: Warum finden wir hier nur ein **DbSet** auf Basis der Klasse **Article** vor, aber keine auf Basis der Klasse **Category**? Der Grund ist, dass die Klasse **Article** mit der Navigationseigenschaft **Category** auf die Klasse **Category** verweist. Der Ordnung halber können Sie die Klasse **Category** aber dennoch aufnehmen. Spätestens, wenn Sie eine Klasse hinzufügen, die nicht mit **Article** oder einer anderen bereits als **DbSet** registrierten Klasse verknüpft ist, müssen Sie dafür ein eigenes **DbSet** anlegen. Außerdem wollen wir den Namen des **DbSet** noch auf **Articles** ändern. Leider müssen Sie danach auch noch die Stellen, an denen die Lösung auf das DbSet **Article** zugreift, in **Articles** umbenennen.

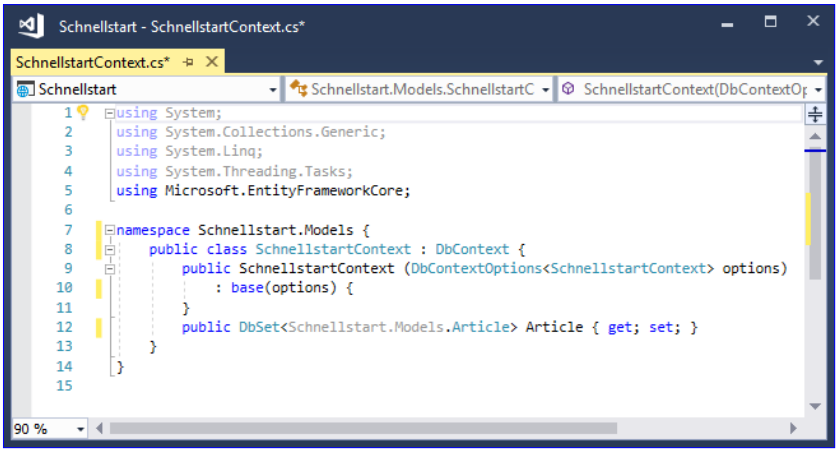


Bild 2: Die neue Datenkontextklasse

Die Datei **appsettings.json** mit den Konfigurationsdaten wurde wie folgt um ein **ConnectionString**-Element erweitert – hier haben wir den Namen der Datenbank noch etwas angepasst:

```
{
  "Logging": {
```

Authentifizierung nachrüsten

Im Artikel »Authentifizierung unter ASP.NET Core« haben wir gezeigt, wie Sie eine ASP.NET Core-Anwendung erstellen und diese direkt mit Funktionen für die Authentifizierung ausstatten. In anderen Artikeln wiederum haben wir Lösungen beschrieben, in denen wir zunächst ohne die Authentifizierung gestartet sind. Nun wollen wir beides zusammenführen. Aber was tun, wenn die Lösung noch nicht die für die Authentifizierung notwendigen Elemente enthält – nachrüsten oder neu mit Authentifizierung erstellen und dann die Lösung übertragen? Dieser Artikel zeigt den einfacheren Weg.

Voraussetzungen: .NET Core 2.1

Der Assistent zum Erstellen neuer ASP.NET Core-Anwendung liefert eine wirklich brauchbare Basis für neue Anwendungen. Noch schöner ist, dass er das Framework für die Realisierung einer Authentifizierung optional mitliefert und gleich noch die Links zur Registrierung und zur Anmeldung sowie die zur Verwaltung notwendigen hinzufügt.

Wenn Sie allerdings mit der Programmierung einer Lösung begonnen haben und nicht gleich die Option zum Hinzufügen der Authentifizierung aktiviert haben, stehen Sie erst einmal ohne Authentifizierung da. Und dazu gehören eine ganze Menge Elemente: die Datenbank zum Speichern der Benutzerdaten, die Verweise, einige Dateien mit den Seiten für Login, Registrierung und so weiter und letztlich auch noch die Einbindung der Links zum Einloggen und Registrieren in die Navigationsleiste der Startseite. Können wir diese Elemente so einfach nachrüsten und in eine bestehende ASP.NET Core-Anwendung integrieren? Oder macht es mehr Sinn, die Anwendung mit Authentifizierung neu zu erstellen und die bestehenden Elemente zu übertragen? Wenn Sie erst ein paar Seiten erstellt haben, können Sie sicher den letzteren Weg einschlagen. Je größer die Lösung allerdings schon gewachsen ist, desto geringer wird der Aufwand für das nachträgliche Einbauen der Authentifizierung gegenüber der anderen Alternative sein. Also schauen wir uns diesen Ansatz einmal genauer an.

Beispielprojekt erstellen

Als Erstes legen wir ein Beispielprojekt an. Dazu nutzen wir den Dialog **Neues Projekt**, wo wir den Eintrag **Visual C#\|Web\|ASP.NET Core-Webanwendung** auswählen (siehe Bild 1). Außerdem wollen wir unsere Lösung **AuthNachruesten** nennen.

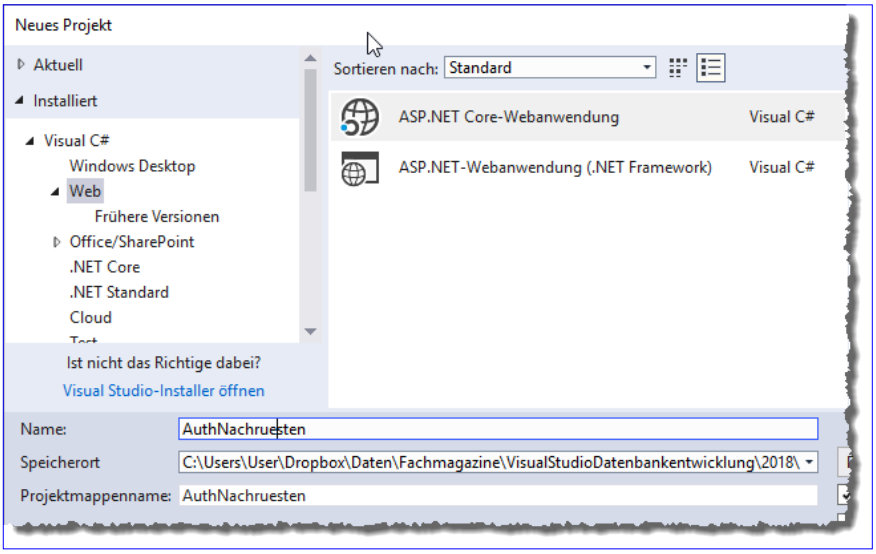


Bild 1: Projekt erstellen

Im nächsten Dialog behalten wir die Auswahl der Option **Webanwendung** bei. Hier wäre die Möglichkeit, über die Schaltfläche **Authentifizierung** ändern

direkt die Authentifizierung hinzuzufügen. Dies wollen wir aber auslassen, da das Ziel des Artikels ja gerade die manuelle Nachrüstung der Authentifizierung ist (siehe Bild 2).

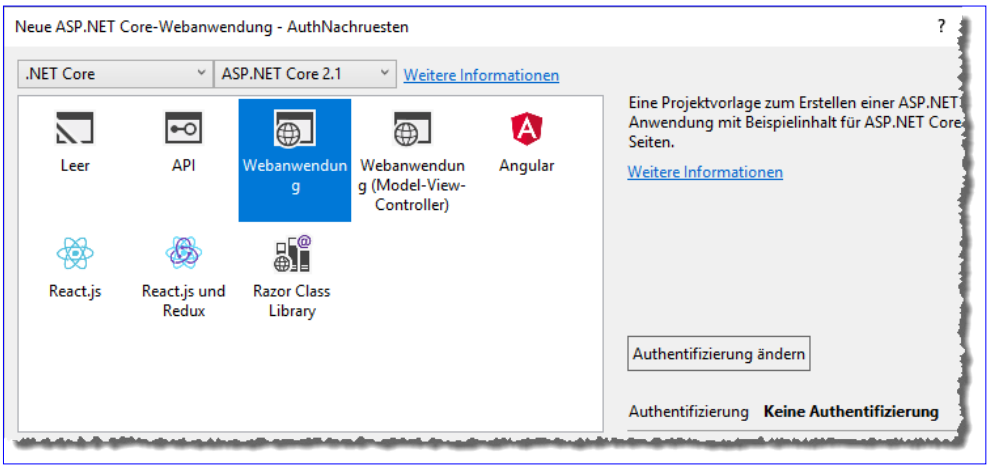


Bild 2: Auswahl ohne Authentifizierung

Wenn Sie die Anwendung nun starten, sehen Sie nur die Standardnavigation (siehe Bild 3). Wir hätten gern, dass oben rechts Navigationseinträge angezeigt werden, mit denen sich Benutzer registrieren und einloggen können. Dies rüsten wir nun nach.

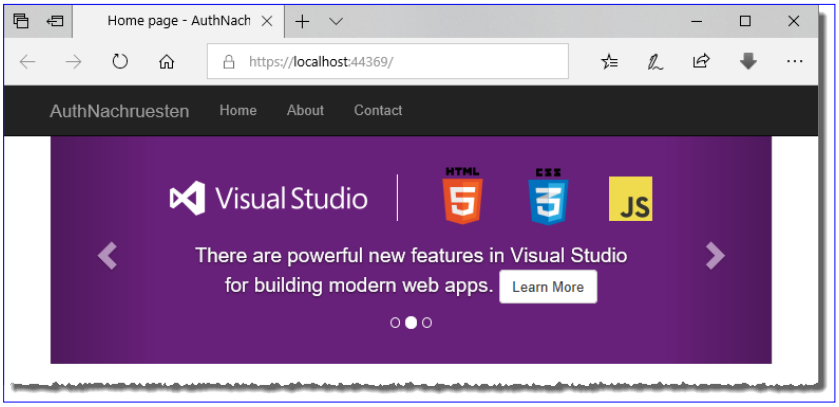


Bild 3: Start der Webanwendung ohne Authentifizierung

Notwendiges Paket nachinstallieren

Das Paket, dass die notwendigen Funktionen enthält, heißt **Microsoft.AspNetCore.Identity.EntityFrameworkCore** – es handelt sich dabei um ein NuGet-Paket. Diese können Sie leicht nachrüsten – beispielsweise über den dafür vorgesehenen NuGet-Paket-Manager.

Diesen starten Sie über den Kontextmenü-Eintrag **NuGet-Pakete verwalten...**. Es erscheint der Dialog aus Bild 4, wo Sie zum Bereich **Durchsuchen** wechseln und dort den Suchbegriff **Microsoft.AspNetCore.Identity.EntityFrameworkCore** eingeben. Haben Sie den entsprechenden Eintrag gefunden, installieren Sie diesen mit einem Klick auf die Schaltfläche **Installieren**.

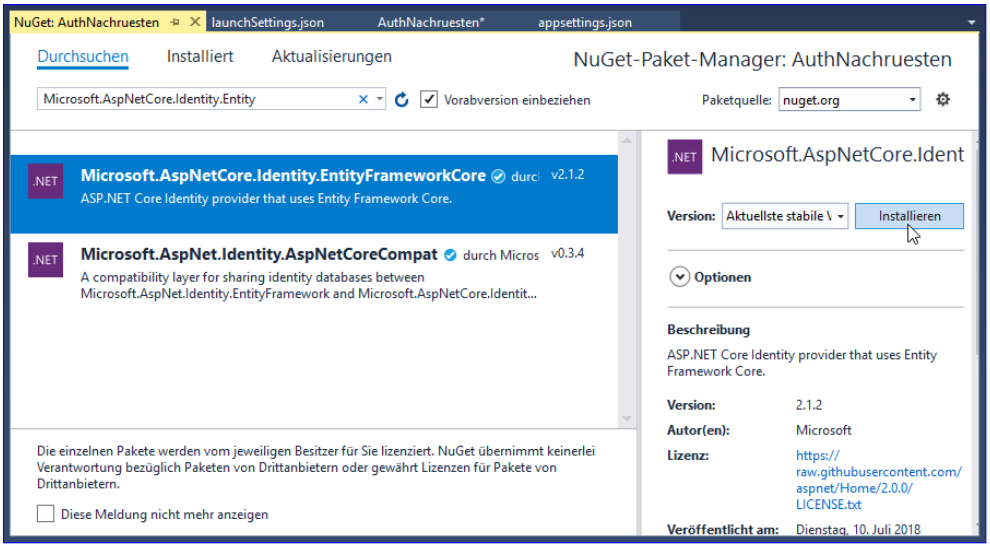


Bild 4: **Microsoft.AspNetCore.Identity.EntityFrameworkCore** nachrüsten

Microsoft holt dann noch Ihre Zustimmung zur Lizenz ein, die Sie mit einem Klick auf die Schaltfläche **Ich stimme zu** quittieren. Die entsprechende Meldung zeigt gleich noch an, welche weiteren Bibliotheken mit dieser Bibliothek installiert werden (siehe Bild 5).

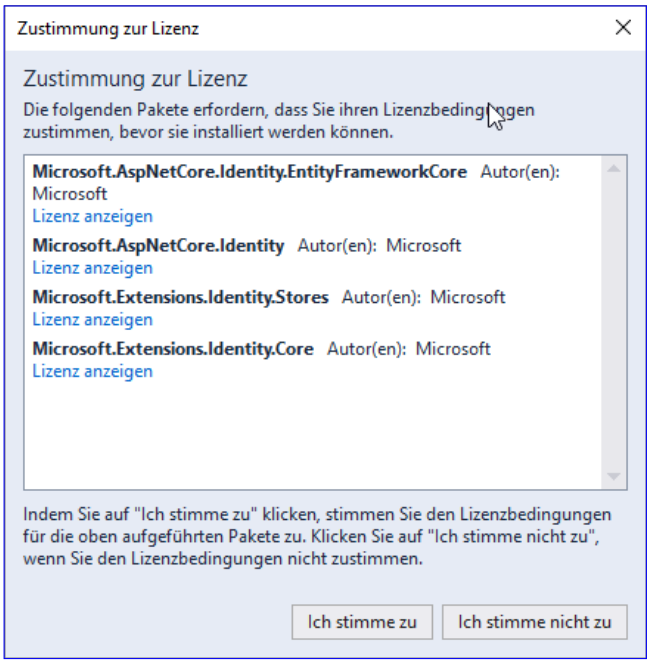


Bild 5: Microsoft.AspNetCore.Identity.EntityFrameworkCore kommt mit ein paar weiteren Bibliotheken

Elemente für CodeFirst-Datenbank hinzufügen

Wenn wir mit der Authentifizierung arbeiten wollen, wir uns die Bibliotheken zur Verfügung stellen, haben wir mehrere Möglichkeiten, was die Speicherung der damit in Zusammenhang stehenden Daten angeht. Wir wollen es klassisch machen und die Daten in einer SQL Server-Datenbank speichern. Dabei gehen wir nach der Code First-Methode vor und erstellen erst ein paar Elemente im Projekt. Danach erstellen wir auf Basis dieser Objekte die notwendige Datenbank beziehungsweise lassen diese erstellen.

Genaugenommen bauen wir ein Entity Data Model auf, dessen Elemente wir in einem Unterverzeichnis namens **Data** im Projektordner anlegen. Das Verzeichnis **Data** legen Sie an, indem Sie den Kontextmenü-Eintrag **Hinzufügen|Neuer Ordner** des Projekt-Eintrags im Projektmappen-Explorer aufrufen.

Dem Ordner **Data** fügen Sie über den Kontextmenü-Eintrag **Hinzufügen|Klasse** eine neue Klasse hinzu, die Sie **ApplicationUser** nennen. Diese Klasse füllen Sie wie folgt:

```
01.     using System;
02.     using System.Collections.Generic;
03.     using System.Linq;
04.     using System.Threading.Tasks;
05.     using Microsoft.AspNetCore.Identity;
06.
07.     namespace AuthNachruesten.Data {
08.         public class ApplicationUser : IdentityUser {
09.         }
10.     }
```

Wir fügen also den Namespace **Microsoft.AspNetCore.Identity** hinzu und stellen die Klasse **Application** so ein, dass Sie von **IdentityUser** abgeleitet wird.

Auf die gleiche Weise fügen Sie eine weitere Klasse namens **ApplicationDbContext** hinzu, welche den Datenbank-Kontext für das Entity Data Model enthalten soll. Diese passen Sie wie folgt an:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace AuthNachruesten.Data {
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser> {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options): base(options) {
        }
        protected override void OnModelCreating(ModelBuilder builder) {
            base.OnModelCreating(builder);
        }
    }
}
```

Nun nehmen wir ein paar Änderungen an der Klasse **Startup.cs** vor. Wir benötigen einen Verweis auf den Namespace **Microsoft.AspNetCore.Identity**, einen auf **Microsoft.EntityFrameworkCore** sowie einen auf unseren Namespace mit dem Entity Data Model, also **AuthNachruesten.Data**. Schließlich fügen wir vor der Anweisung **services.AddMvc()...** die Methode **services.AddDbContext...** sowie die Methode **services.AddIdentity...** ein:

```
...
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using AuthNachruesten.Data;

namespace AuthNachruesten {
    public class Startup {
        ...
        public void ConfigureServices(IServiceCollection services) {
            ...
            services.AddDbContext<ApplicationDbContext>(
                options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
            services.AddIdentity<ApplicationUser, IdentityRole>()
                .AddEntityFrameworkStores<ApplicationDbContext>()
                .AddDefaultTokenProviders();
            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
        }
    }
}
```

An dieser Stelle verwenden wir eine Verbindungszeichenfolge namens **DefaultConnection**. Diese ist allerdings noch nicht hinterlegt, was wir wie folgt in der Datei **appsettings.json** erledigen:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=AuthNachruesten;Trusted_Connection=True;
                         MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Datenbank erstellen

Um eine Datenbank auf Basis der beiden Klassen **ApplicationDbContext.cs** und **ApplicationUser.cs** zu erstellen, benötigen wir die Paket-Manager-Konsole. Diese blenden wir über den Menüeintrag **Extras|NuGet-Paket-Manager|Paket-Manager-Konsole** ein. Nun geben wir in der Paket-Manager-Konsole die folgende Anweisung ein (siehe Bild 6):

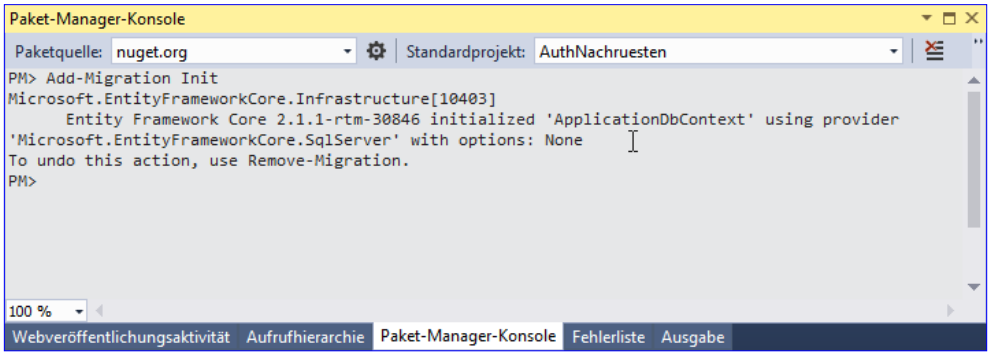


Bild 6: Initialisieren der Migration

```
Add-Migration Init
```

Danach folgt dieser Aufruf:

```
Update-Database
```

Visual Studio arbeitet nun ein paar Sekunden und meldet dann den erfolgreichen Abschluss der Aufgabe. Wenn wir nun den SQL Server-Objekt-Explorer einblenden, finden wir dort unter dem angegebenen SQL Server, hier **LocalDb**, die neue Datenbank **AuthNachruesten** vor. Klappen wir diesen Eintrag und den darin enthaltenen Eintrag **Tabellen** auf, finden wir eine Reihe Tabellen wie in Bild 8 vor. Interessanterweise ist hier keine Tabelle namens **ApplicationUser** zu sehen. Aber den haben wir in der Datei **ApplicationDbContext** ja auch gar nicht als **DbSet** definiert. Aber woher wusste Visual Studio, welche Tabellen hier erstellt werden sollten? Es handelt sich ja offensichtlich um die für die Verwaltung von Benutzern notwendigen Tabellen, ganz verkehrt liegen wir hier also nicht.

Authentifizierungsseiten anpassen

Im Artikel »Authentifizierung nachrüsten« haben wir gezeigt, wie Sie die Funktionen für die Authentifizierung von Benutzern in einer Webanwendung nachrüsten können. Dort haben wir die platzsparende Variante verwendet – also die, welche die ganzen verwendeten Elemente der Benutzeroberfläche für die Authentifizierung direkt aus einer Bibliothek verwendet. Das spart zwar eine Menge Platz, weil die ganzen Seiten nicht einzeln im Projektordner liegen, aber man kann diese nicht so einfach anpassen. Der vorliegende Artikel zeigt, wie Sie die Authentifizierung umstellen und diese so anpassbar machen.

Im Artikel **Authentifizierung nachrüsten** haben wir die unter ASP.NET Core 2.1 eingeführte Bibliothek **Microsoft.AspNetCore.Identity.UI** eingebunden, um die darin enthaltenen Elemente für die Benutzer-Authentifizierung zu nutzen. Damit haben wir den Umfang der im Projektordner enthaltenen Daten erheblich verringert, denn die Seiten für die Anzeige der Elemente der Authentifizierung und erst recht der Code für die notwendigen Funktionen sind sehr umfangreich (rund 50 Dateien). Diese Bibliothek haben wir verfügbar gemacht, in dem wir in der Klasse **Startup.cs** die Methode **AddDefaultUI()** aufrufen.

Nun wollen wir allerdings beispielsweise die Texte in deutscher Sprache präsentieren und nicht in englischer Sprache. Dazu müssen wir allerdings doch wieder den Code für die Authentifizierungsfunktionen ins Projekt holen. Hier gibt es zwei verschiedene Ansätze: Entweder Sie holen den kompletten Code an Bord oder Sie fügen nur diejenigen Dateien ein, die Sie anpassen möchten, um beispielsweise die Benutzeroberfläche in deutscher Sprache zu präsentieren. Wir zeigen Ihnen in den nächsten Abschnitten, wie das funktioniert.

Authentifizierungsfunktionen ins Projekt holen

Um die Elemente der Bibliothek **Microsoft.AspNetCore.Identity.UI** doch wieder im Projektverzeichnis unterzubringen, benötigen Sie ein neues Element. Diesmal fügen wir allerdings nicht einfach ein Element etwa wie eine Klasse hinzu, sondern verwenden einen sogenannten Scaffolded Item, zu deutsch Gerüstelement. Dazu rufen Sie zunächst den Kontextmenü-Befehl **Hinzufügen|Neues Gerüstelement...** auf. Es erscheint der Dialog **Gerüst hinzufügen**, in dem wir links **Identität** auswählen und in der Mitte anschließend auch (siehe Bild 1).

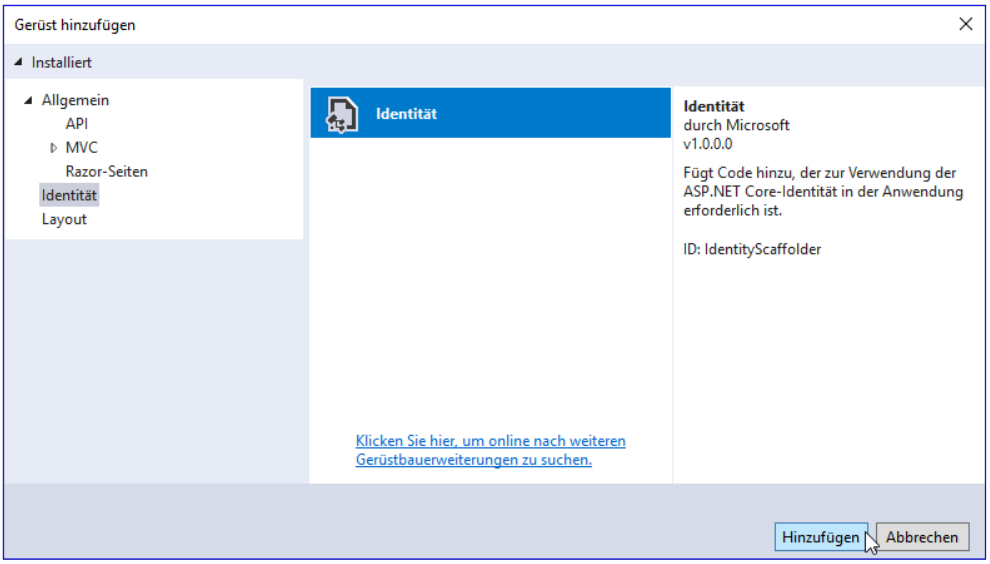


Bild 1: Hinzufügen der Authentifizierungs-Elemente per Gerüstbau

Danach erscheint der Dialog aus Bild 2. Hier haben Sie verschiedene Möglichkeiten:

- Angabe der Layout-Seite
- Auswahl der zu überschreibenden Dateien – gegebenenfalls auch die Möglichkeit, alle Dateien zu überschreiben
- Auswahl der Kontext-Klasse
- Angabe der Benutzerklasse

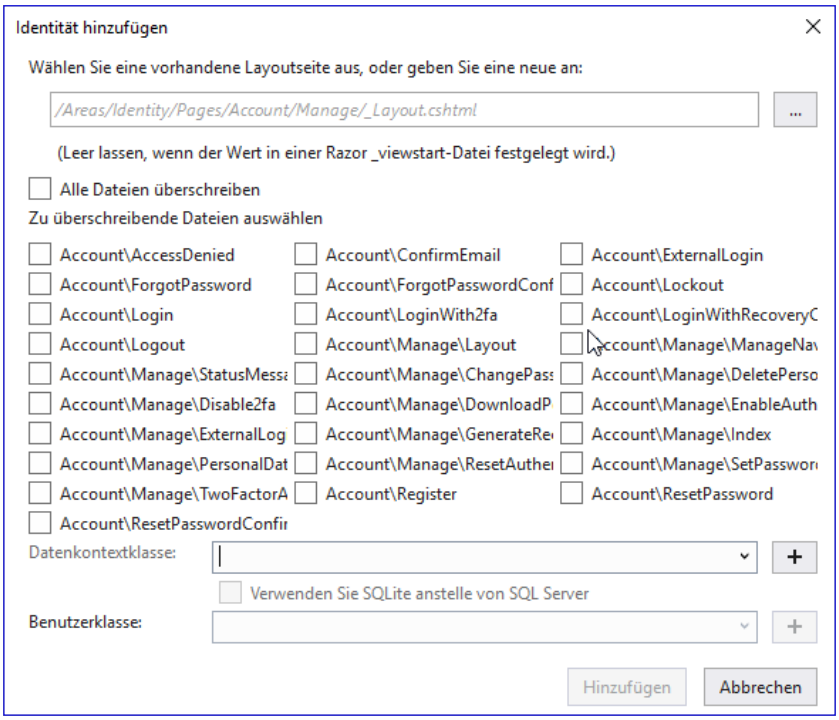


Bild 2: Konfiguration zum Hinzufügen der Elemente

Die Angabe der Layout-Seite können wir weglassen, da wir ja bereits in der Datei **Areas\Identity\Pages** eine Datei namens **_ViewStart.cshtml** angelegt haben, die einen Verweis auf die zu verwendende Layout-Datei enthält (**Pages\Shared_Layout.cshtml**).

Darunter könne Sie auswählen, welche der vielen Dateien, die sich in der Bibliothek **Microsoft.AspNetCore.Identity.UI** befinden, Sie überschreiben wollen. Überschreiben deshalb, weil ja die Dateien in der Bibliothek noch vorhanden sind, aber stattdessen dann die von Ihnen hinzugefügten Dateien verwendet werden. Wir wollen zunächst die offensichtlichen Dateien hinzufügen:

- **Account\Login**
- **Account\Register**

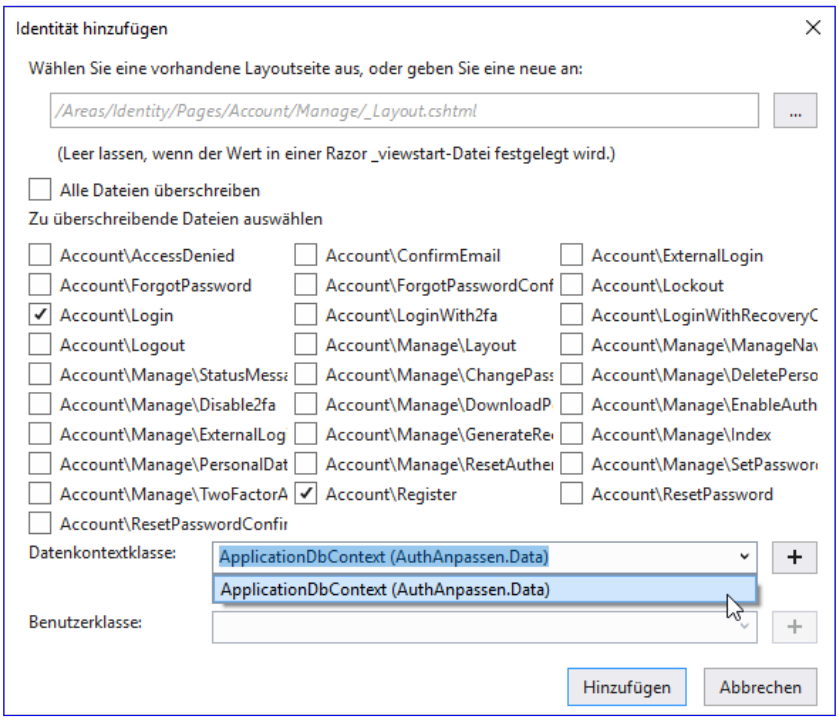


Bild 3: Auswahl der hinzuzufügenden Elemente

Erst wenn Sie hier mindestens eine Datei ausgewählt haben, wird das Auswahlfeld für die Datenkontextklasse gefüllt. Hier wählen wir dann unsere Datenkontextklasse **ApplicationDbContext** aus dem Verzeichnis **AuthAnpassen.Data** aus (siehe Bild 3). Das Textfeld Benutzerklasse wird nicht aktiviert und kann ignoriert werden.

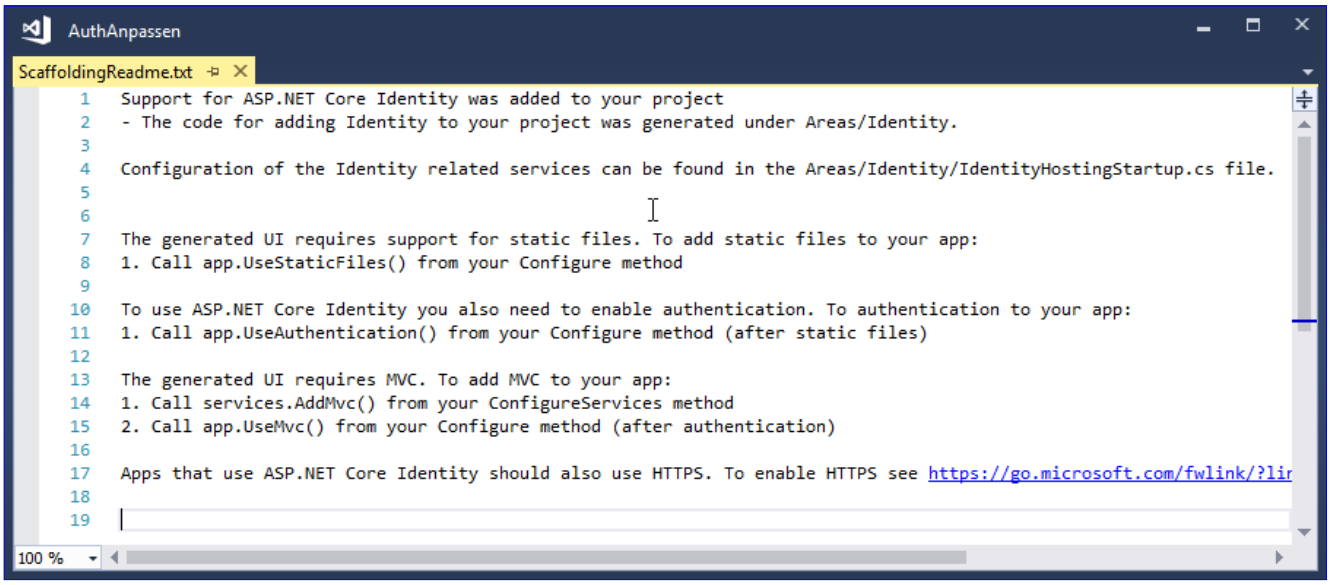


Bild 4: Hinweise nach dem Gerüstbau

Klicken Sie dann auf **OK**, erscheint ein Dialog, der den Fortschritt dokumentiert. Außerdem zeigt Visual Studio nach Abschluss des Vorgangs noch eine Readme-Datei an (siehe Bild 4). Demnach soll die **Configure**-Methode der Klasse **Startup.cs** die Methode **app.UseStaticFiles()** aufrufen. Dahinter soll die Methode **app.UseAuthentication()** aufgerufen werden sowie danach die Methode **app.UseMvc**.

Die ebenfalls in diese Klasse enthaltene Methode **ConfigureServices** soll **services.AddMvc**() aufrufen. Prüfen Sie, ob die angegebenen Methoden vorhanden sind (was der Fall sein sollte) und ergänzen Sie diese gegebenenfalls.

Dateien anpassen

Der Gerüstbauer hat außerdem ganze Arbeit geleistet, wie Bild 5 zeigt. Er hat unterhalb von **Areas\Identity\Pages** das Verzeichnis **Account** mit den gewünschten Dateien, hier **Login.cshtml** und **Register.cshtml**, und einigen weiteren Dateien angelegt. Außerdem einen Ordner namens **Services** unterhalb von **Areas\Identity** sowie die Datei **IdentityHostingStartup.cs**.

Wenn wir nun etwa die Texte der Seite **Login** eindeutschen wollen, brauchen wir nur die Datei **Login.cshtml** zu öffnen und dort die gewünschten Änderungen vorzunehmen.

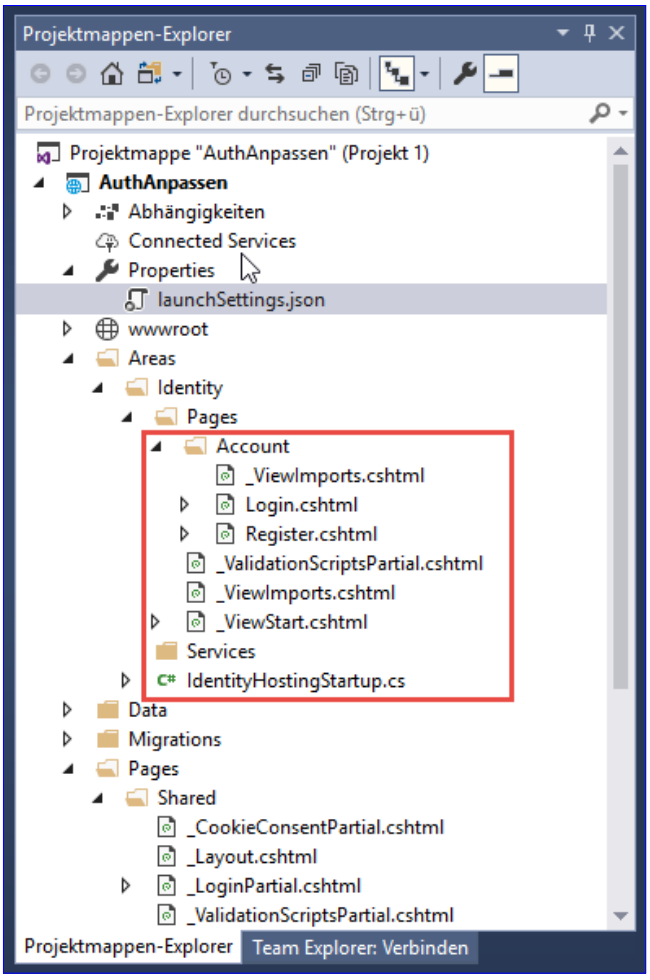


Bild 5: Ergebnis des Gerüstbaus

Hier legen wir direkt bei der Zuweisung der Eigenschaft **Title** mit dem Text **Anmelden** los:

```
@page
@model LoginModel
@{ ViewData["Title"] = "Anmelden"; }
<h2>@ViewData["Title"]</h2>
<div class="row">
    <div class="col-md-4">
        <section>
```

Dann geht es mit der Überschrift weiter:

```
            <form method="post">
                <h4>Geben Sie Ihre Anmeldedaten ein.</h4><hr />
                <div asp-validation-summary="All" class="text-danger"></div>
```

Bei den Beschriftungen gibt es zwei Möglichkeiten. Sie können wir hier einfach **asp-for="Input.Email"** überschreiben:

```
                <div class="form-group">
                    <label>E-Mail-Adresse:</label>
                    <input asp-for="Input.Email" class="form-control" />
                    <span asp-validation-for="Input.Email" class="text-danger"></span>
                </div>
```

Oder Sie machen es wie im Beispiel des Kennworts, wo wir die Beschriftung gleich in der zugrunde liegenden Klasse ändern. Hier behalten wir die **asp-for**-Eigenschaft für das **label**-Element bei:

```
                <div class="form-group">
                    <label asp-for="Input.Password"></label>
                    <input asp-for="Input.Password" class="form-control" />
                    <span asp-validation-for="Input.Password" class="text-danger"></span>
                </div>
```

Allerdings fügen wir dem Element **Password** der Klasse **InputModel** in der Code behind-Datei **Login.cshtml.cs** das **Display**-Attribut mit dem Wert **Kennwort** für die Eigenschaft **Name** hinzu:

```
public class InputModel {
    ...
    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "Kennwort:")]
```

Authentifizierung um Felder erweitern

Die Authentifizierung spielt eine wichtige Rolle bei der Programmierung von Webanwendungen, deren Inhalte nicht ausschließlich der Allgemeinheit zur Verfügung stehen sollen. Dabei kann es vorkommen, dass es nicht ausreicht, die E-Mail-Adresse und das Kennwort eines Benutzers zu speichern. Manchmal möchte man es vielleicht persönlicher gestalten und den Benutzer direkt ansprechen. Um bei der Registrierung die notwendigen Daten zu speichern, sind auch bei der ansonsten perfekt funktionierenden Registrierung von ASP NET Core noch einige Erweiterungen durchzuführen.

Voraussetzungen: Ausgangspunkt für die in diesem Artikel beschriebenen Erweiterungen ist eine Webanwendung auf Basis von ASP.NET Core 2.1 mit Authentifizierungsfunktionen. Wie Sie eine Anwendung direkt bei der Erstellung mit der Authentifizierungsbibliothek ausstatten, erfahren Sie im Artikel **Authentifizierung unter ASP.NET Core**. Wie Sie eine vorhandene Anwendung mit den Authentifizierungsfunktionen nachrüsten, lesen Sie unter dem Titel **ASP.NET Core: Authentifizierung nachrüsten**. In beiden Fällen nutzen Sie gegebenenfalls die in der Bibliothek **Microsoft.AspNetCore.Identity.EntityFrameworkCore** enthaltenen Funktionen. Da wir im vorliegenden Artikel die Benutzeroberfläche sowie den dahinter steckenden Code anpassen wollen, müssen Sie die in der Bibliothek enthaltenen Dateien gegebenenfalls überschreiben. Wie das gelingt, lesen Sie im Artikel **ASP.NET Core: Authentifizierungsseiten anpassen** nach. Wir bauen im vorliegenden Artikel auf der Beispiellösung dieses Artikels auf.

Aktueller Stand

Im Standardzustand der Authentifizierung für ASP.NET Core-Webanwendungen sieht der bereits mit deutschen Texten versehene Registrierungsdialog wie in Bild 1 aus.

Wenn wir einen Blick in die Tabelle werfen, welche nach der Registrierung die daten des neuen Benutzers aufnehmen sollen, finden wir dort auch keine weiteren Felder, die man mit zusätzlichen Daten füllen könnte. Diese Tabelle finden Sie vor, wenn Sie **AnsichtISQL Server-Objekt-Explorer** den entsprechenden Explorer einblenden und dort zur Tabelle **AspNetUsers** der passenden Datenbank navigieren und diesen Eintrag doppelt anklicken (siehe Bild 2).

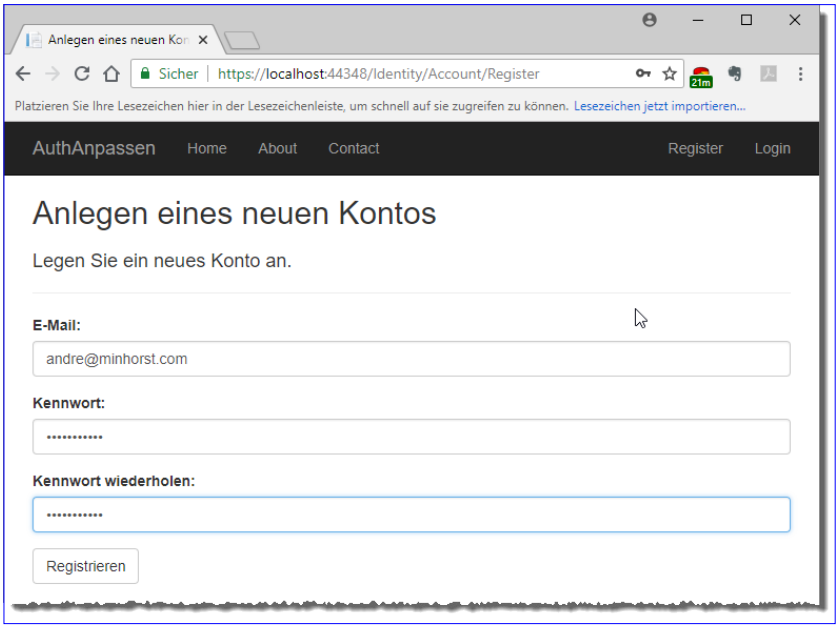


Bild 1: Dialog zum Anlegen eines Kontos vor dem Hinzufügen von Erweiterungen

Es gibt also zumindest schon einmal zwei Stellen, an denen wir Hand anlegen müssen, wenn wir beim Registrieren

eines neuen Benutzers mehr als nur die E-Mail-Adresse und das Kennwort (in verschlüsselter Form) speichern sollen.

Es gibt allerdings noch weitere Stellen – zum Beispiel das Model für den Benutzer in Form der Klasse **ApplicationUser** im Ordner **Data** sowie die passenden Elemente in der Code behind-Datei der Klasse **Register.cshtml**.

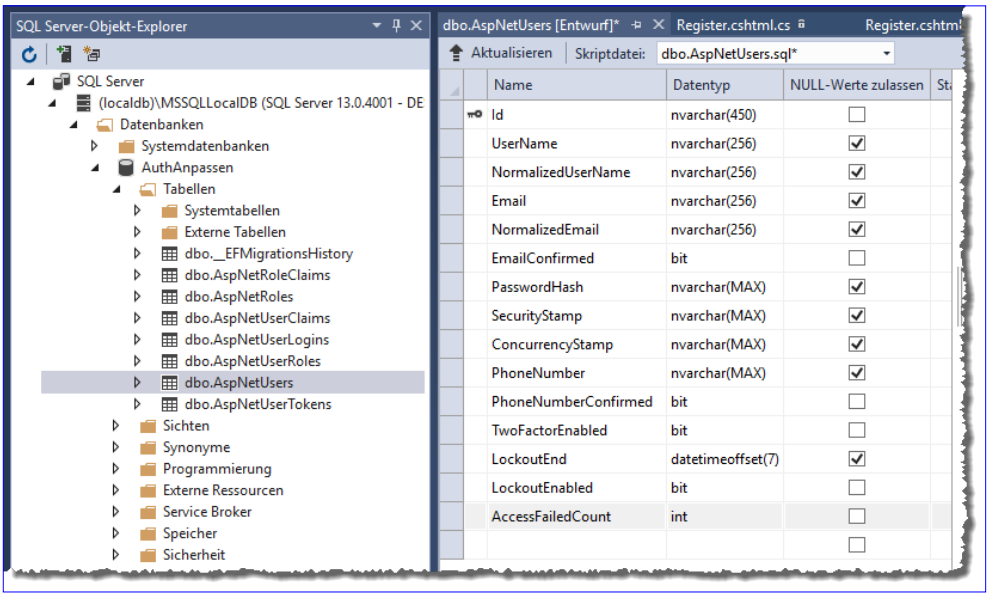


Bild 2: Datenmodell der Tabelle zum Speichern der Benutzerdaten

Schließlich wollen wir dem Benutzer auch noch die Möglichkeit geben, die bei der Registrierung angegebenen Daten zu aktualisieren, weshalb wir auch noch die Datei **ManageIndex.cshtml** erweitern müssen.

Schritt 1: Model anpassen

Damit wir bei den folgenden Schritten, also etwa beim Erweitern der Benutzeroberfläche, per IntelliSense auf die neuen Elemente zugreifen können, beginnen wir beim Model mit den Änderungen. Die dort verwendete Klasse **ApplicationUser** erbt von der Klasse **IdentityUser** und benötigte bisher noch keine eigenen Felder, da diese alle in **IdentityUser** steckten. Nun fügen wir die drei Felder **FirstName**, **LastName** und **BirthDate**hinzu:

```
01.     namespace AuthAnpassen.Data {
02.         public class ApplicationUser : IdentityUser {
03.             public string FirstName { get; set; }
04.             public string LastName { get; set; }
05.             public DateTime BirthDate{ get; set; }
06.         }
07.     }
```

Schritt 2: Vom Model zur Datenbank

Diese wollen wir nun zunächst auch in die Datenbank übertragen. Dazu benötigen wir die Paket-Manager-Konsole, die Sie, falls noch nicht sichtbar, über den Menübefehl **Extras|NuGet-Paket-Manager|Paket-Manager-Konsole** aktivieren. Wenn Sie noch keine Datenbank auf Basis der Klassen **ApplicationDbContext.cs** und **ApplicationUser.cs** erstellt haben, sind noch ein paar vorbereitende Schritte nötig. Der erste ist das Anpassen/Einfügen der Verbindungszeichenfolge in der Datei **appsettings.json**:

```
01.     {
02.       "ConnectionStrings": {
```

```
03.         "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=AuthAnpassen;Trusted_Connection=True;
                                                                    MultipleActiveResultSets=true"
04.       }, ...
05.     }
```

Dann erstellen Sie die Datenbank mit den folgenden beiden Anweisungen:

```
Add-Migration Init
Update-Database
```

Hinweis: Wenn Sie die Datenbank erst jetzt erstellen, müssen Sie die folgenden Schritte nicht mehr ausführen, denn die Felder **FirstName**, **LastName** und **Birthday** wurden nun schon hinzugefügt. Wenn die Datenbank hingegen schon vorhanden ist, sind die folgenden beiden Anweisungen noch notwendig.

Zum Aktualisieren der Datenbank setzen Sie die folgenden zwei Befehle ab. Der erste fügt eine neue Klasse mit den für die Migration der Elemente in die Datenbank notwendigen Befehlen zum Projekt hinzu und nennt diese Klasse **AddFieldsAspNetUsers**:

```
Add-Migration AddFieldsAspNetUsers
```

Die Klasse hat den folgenden Code:

```
01.     using System;
02.     using Microsoft.EntityFrameworkCore.Migrations;
03.     namespace AuthAnpassen.Migrations {
04.         public partial class AddFieldsAspNetUsers : Migration {
05.             protected override void Up(MigrationBuilder migrationBuilder) {
06.                 migrationBuilder.AddColumn<DateTime>(
07.                     name: "Birthdate",
08.                     table: "AspNetUsers",
09.                     nullable: false,
10.                     defaultValue: new DateTime(1, 1, 1, 0, 0, 0, 0, DateTimeKind.Unspecified));
11.                 migrationBuilder.AddColumn<string>(
12.                     name: "FirstName",
13.                     table: "AspNetUsers",
14.                     nullable: true);
15.                 migrationBuilder.AddColumn<string>(
16.                     name: "LastName",
17.                     table: "AspNetUsers",
18.                     nullable: true);
19.             }
```

```
20.             protected override void Down(MigrationBuilder migrationBuilder) {
21.                 migrationBuilder.DropColumn(
22.                     name: "Birthdate",
23.                     table: "AspNetUsers");
24.                 migrationBuilder.DropColumn(
25.                     name: "FirstName",
26.                     table: "AspNetUsers");
27.                 migrationBuilder.DropColumn(
28.                     name: "LastName",
29.                     table: "AspNetUsers");
30.             }
31.         }
32.     }
```

Sie löscht also eventuell vorhandene Felder namens **BirthDate**, **FirstName** und **LastName** und erstellt diese unter den angegebenen Datentypen erneut. Dabei verwendet sie die **DropColumn**- beziehungsweise **AddColumn**-Felder des **MigrationBuilder**-Objekts. Nun rufen wir in der Paket-Manager-Konsole die folgende Anweisung auf:

```
Update-Database
```

Dies durchläuft die im Verzeichnis **Migrations** enthaltenen Klassen und führt diese aus. Dadurch landen nun auch die drei neuen Felder für die Klasse **ApplicationUser.cs** in der Tabelle **AspNetUsers** – siehe Bild 3.

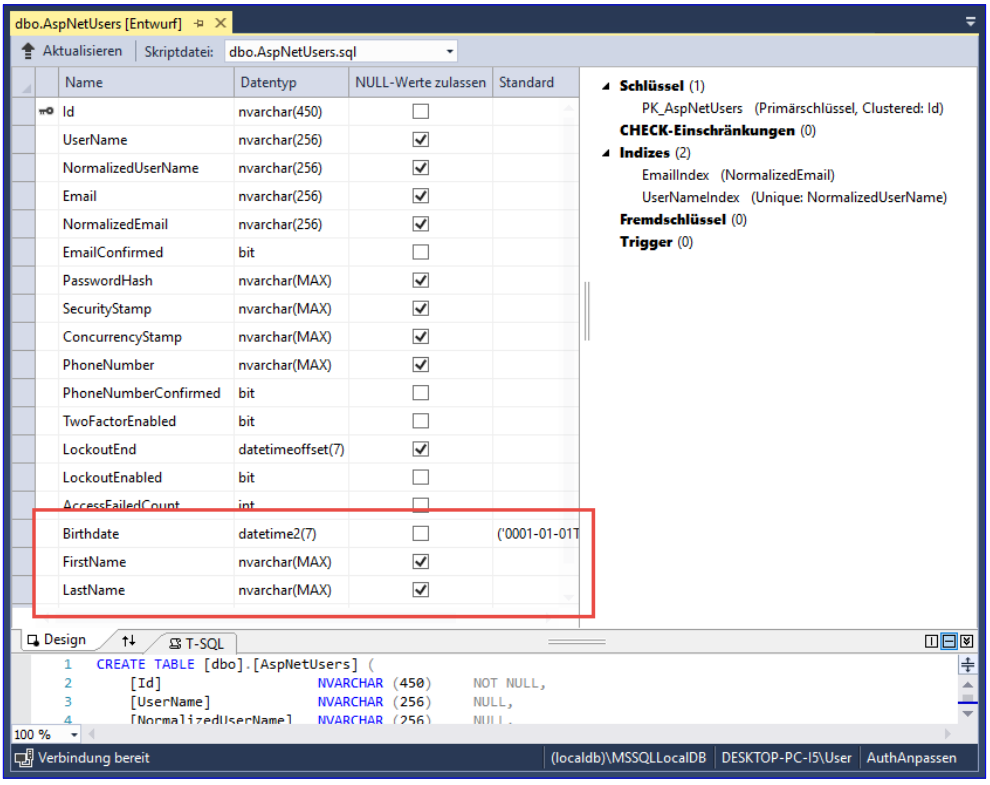


Bild 3: Die Tabelle **AspNetUsers** mit den drei neuen Feldern

Damit haben wir die Felder nun in der Klasse und in der zugrundeliegenden Tabelle angelegt. Damit diese auch über die Benutzeroberfläche verfügbar sind, sind noch einige weitere Schritte nötig.

Neue Felder zu den Webseiten hinzufügen

Wo genau benötigen wir die neuen Felder? Zunächst ein-

mal bei der Seite zum Registrieren eines neuen Benutzers, also auf **Areas|Identity|Pages|Account|Register.cshtml**. Außerdem gibt es die Möglichkeit, nach der Anmeldung das eigene Konto zu bearbeiten.

Dies geschieht auf der Seite **Areas|Identity|Pages|Account|Manage|Index.cshtml**. Natürlich können wir die vom Benutzer eingegebenen Daten auch noch an anderen Stellen anzeigen – beispielsweise im Kopfbereich nach der Anmeldung. Dies lassen wir allerdings vorerst weg.

Als Erstes passen wir in der Code behind-Datei **Register.cshtml.cs** die Klasse **InputModel** an. Diese erweitern wir um die beiden Eigenschaften **FirstName**, **LastName** und **BirthDate**:

```
01.     public class InputModel {
02.         [Required]
03.         [EmailAddress]
04.         [Display(Name = "E-Mail:")]
05.         public string Email { get; set; }
06.
07.         [Required]
08.         [StringLength(100, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.", MinimumLength = 6)]
09.         [DataType(DataType.Password)]
10.         [Display(Name = "Kennwort:")]
11.         public string Password { get; set; }
12.
13.         [DataType(DataType.Password)]
14.         [Display(Name = "Kennwort wiederholen:")]
15.         [Compare("Password", ErrorMessage = "Die beiden Kennwörter müssen identisch sein.")]
16.         public string ConfirmPassword { get; set; }
17.
18.         [Required]
19.         [Display(Name = "Vorname:")]
20.         public string FirstName { get; set; }
21.
22.         [Required]
23.         [Display(Name = "Nachname:")]
24.         public string LastName { get; set; }
25.
26.         [Required]
27.         [Display(Name = "Geburtsdatum:")]
28.         [DataType(DataType.Date)]
29.         [DisplayFormat(DataFormatString = "{0:dd.MM.yyyy}")]
30.         public DateTime Birthdate { get; set; }
31.     }
```

Razor-Pages: Daten aus Lookup-Tabellen

Im Artikel »Razor Pages mit Datenbankanbindung« haben wir gezeigt, wie Sie Daten aus einfachen Tabellen in einer Übersichtliste anzeigen, Details bearbeiten, neue Datensätze anlegen und vorhandene Datensätze ändern. In diesem Artikel nun wollen wir eine Lookup-Tabelle hinzunehmen, mit der wir die Anreden der Kunden aus der bereits vorhandenen Tabelle abbilden. Diese sollen dann in den Details per Nachschlagefeld ausgewählt werden. Eine kleine Schaltfläche neben dem Nachschlagefeld soll es dann ermöglichen, die Lookup-Daten zu bearbeiten.

Lookup-Entität hinzufügen

Im Artikel **EDM: 1:n-Beziehungen per Code First** erfahren Sie im Detail, wie Sie 1:n-Beziehungen zwischen zwei Entitäten aufbauen.

Für den vorliegenden Artikel wollen wir der bisher verwendeten Klasse **Customer** eine weitere Klasse namens **Salutation** hinzufügen, die wir dazu im Ordner **Models** anlegen:

```
public class Salutation {
    public int ID { get; set; }
    public string Name { get; set; }
    public ICollection<Customer> Customers { get; set; }
}
```

Um die Beziehung zwischen den beiden Klassen herzustellen, fügen wir der Klasse **Customer** auch noch zwei Elemente hinzu:

```
public class Customer {
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Street { get; set; }
    public string Zip { get; set; }
    public string City { get; set; }
    public int SalutationID { get; set; }
    public Salutation Salutation { get; set; }
}
```

Schließlich fügen wir die neue Klasse noch als **DbSet** zur Klasse **CustomerManagementContext** im Verzeichnis **Data** hinzu:

```
public class CustomerManagementContext : DbContext {
    ...
```

```
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Salutation> Salutations { get; set; }
    ...
}
```

Danach übernehmen Sie die Änderungen durch das Absetzen der folgenden beiden Anweisungen in der Paket-Manager-Konsole in die Datenbank, die in der Verbindungszeichenfolge der Datei **appsettings.json** angegeben ist:

```
Add-Migration Init
Update-Database
```

Um die Datenbank frisch anzulegen, löschen Sie die Datenbank zuvor aus dem SQL Server-Objekt-Explorer und entfernen auch alle Einträge im Verzeichnis **Migrations** im Projektmappen-Explorer.

Daten hinzufügen

Und damit beim Starten der Anwendung und beim Initialisieren auch noch ein paar Daten in der Tabelle **Salutations** landen und das Feld **Salutation** der Datensätze der Tabelle **Customers** mit Werten aus der Tabelle **Salutations** gefüllt wird, passen wir auch noch die Methode **Initialize** in der Klasse **DbInitializer** an:

```
public class DbInitializer {
    public static void Initialize(CustomerManagementContext dbContext) {
        dbContext.Database.EnsureCreated();
        if (dbContext.Customers.Any()) {
            return;
        }
        var salutations = new Salutation[] {
            new Salutation { Name = "Herr" },
            new Salutation { Name = "Frau" },
            new Salutation { Name = "Firma" }
        };
        foreach (Salutation s in salutations)            {
            dbContext.Salutations.Add(s);
        }
        var customers = new Customer[] {
            new Customer{FirstName="André", LastName="Minhorst", Street="Borkhofer Str. 17", Zip="47137",
                City="Duisburg", Salutation=salutations[0] }, ...
        };
        foreach (Customer c in customers) {
            dbContext.Customers.Add(c);
        }
        dbContext.SaveChanges();
```

```
    }
}
```

Diese legt nun zuerst einige Anreden an, bevor die Kunden mit dem jeweiligen Verweis auf die Anrede mit dem namens **Herr** hinzugefügt werden. Danach speichert die Methode die hinzugefügten Elemente mit der **SaveChanges**-Methode.

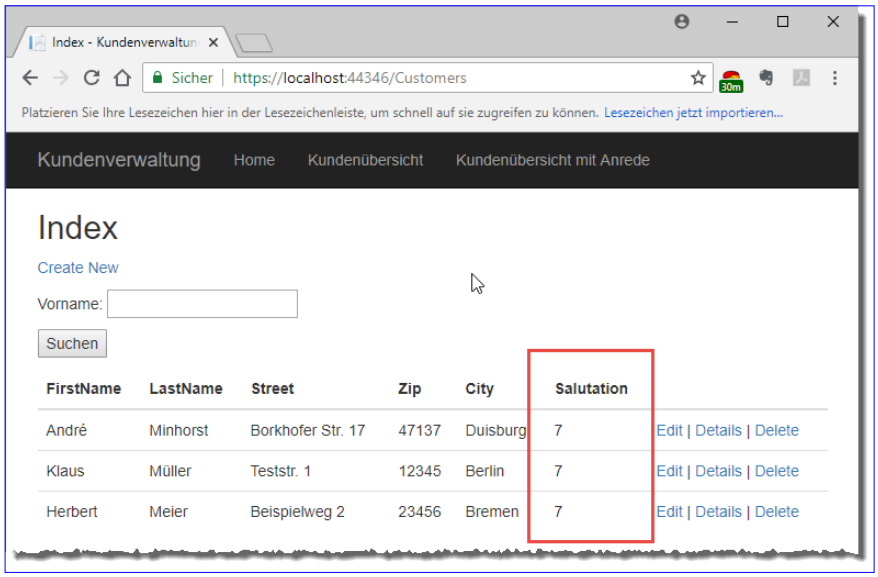


Bild 1: Ein erster Schritt – die Anzeige des Fremdschlüsselwertes

Anzeigen der Anreden in der Übersicht der Kunden

Als Erstes wollen wir die Seite **Pages\Customers\Index.cshtml** so anpassen, dass sie die Anrede der Kunden anzeigt. Dazu fügen wir der Tabelle auf der Seite **Index.cshtml** jeweils eine Spalte zum Element **theadltr** und zum Element **tbodyltr** hinzu. Dabei geben wir als Wert zunächst das Feld **SalutationID** an:

```
<table class="table">
    <thead>
        <tr>
            ... andere Überschriften ...
            <th>
                @Html.DisplayNameFor(model => model.Customer[0].Salutation)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Customer) {
        <tr>
            ... andere Feldinhalte ...
            <td>
                @Html.DisplayFor(modelItem => item.SalutationID)
            </td>
            ...
        </tr>
}
    </tbody>
</table>
```

Die Übersichtsseite zeigt nun eine neue Spalte mit dem Fremdschlüsselwert des Feldes **SalutationID** an (siehe Bild 1). Das ist zwar ein erster Schritt, aber nicht das, was wir erreichen wollen – wir möchten ja den Wert des Feldes **Name** des Datensatzes der Tabelle **Salutations**, der mit dem aktuellen Datensatz der Tabelle **Customers** verknüpft ist, anzeigen. Dazu ändern wir schlicht und einfach das Feld:

```
01.     <td>
02.         @Html.DisplayFor(modelItem => item.Salutation.Name)
03.     </td>
```

Dies führt allerdings nicht zum gewünschten Ergebnis – auf diese Weise bleibt die Spalte schlicht leer. Auch die einfache Verwendung von **item.Salutation** liefert eine leere Spalte. Allerdings wird auch keine Ausnahme ausgelöst – wo liegt also das Problem? Es liegt daran, dass wir nur die Daten der Tabelle **Customers** laden, aber nicht die der verknüpften Tabelle **Salutations** – und diese beherbergt nun einmal das Feld **Name**, mit dem wir die Anrede in der Tabelle ausgeben wollen.

Um diese Daten zu laden, gibt es verschiedene Möglichkeiten. Die einfachste ist diese hier, bei der wir einfach die Daten des **Salutation**-Elements zum jeweiligen **Customer**-Objekt auslesen, und zwar indem wir mit der **Include**-Methode arbeiten (mittlere Anweisung):

```
public async Task OnGetAsync(string firstName, string firstNamePart) {
    var customers = from m in _context.Customers select m;
    customers = customers.Include(c => c.Salutation);
    Customer = await customers.ToListAsync();
}
```

Dies liefert endlich die gewünschten Anreden in der Auflistung der Kunden (siehe Bild 2).

Anrede beim Anlegen oder Ändern auswählen

Nun gehen wir einen Schritt weiter und schauen uns den Dialog zum Bearbeiten eines Kunden an. Hier wollen wir das Steuerelement für die Auswahl der Anrede ebenfalls integrieren.

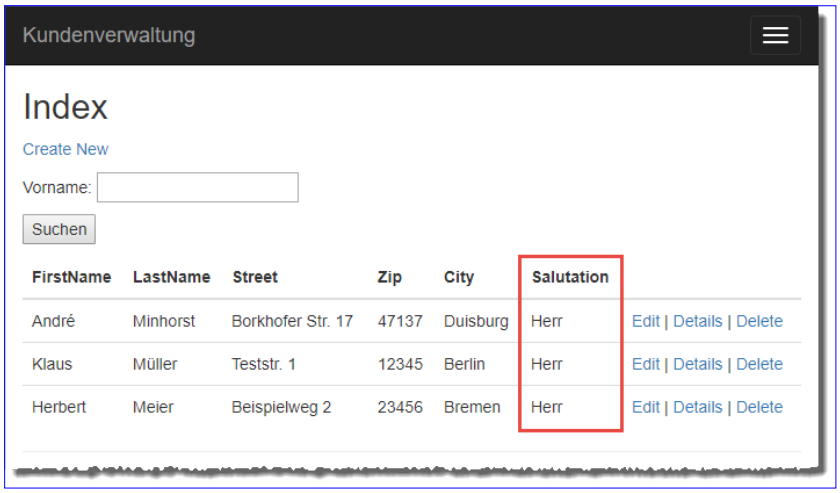


Bild 2: Kundenliste mit Anreden

In diesem Fall benötigen wir kein einfaches Textfeld, sondern ein Auswahlfeld. Unter HTML heißen diese Steuerelemente **Select**-Felder.

In diesem Fall müssen wir die Seite **Edit.cshtml** bearbeiten. Wir beginnen wieder einfach und fügen dem Formular mit den

Steuerelementen zunächst ein Steuerelement zur Eingabe des Fremdschlüsselfeldes **SalutationID** hinzu. Dieses statten wir auch mit einer Beschriftung und einer Validierung aus wie die übrigen Steuerelemente:

```
01.     <form method="post">
02.         <div asp-validation-summary="ModelOnly" class="text-danger"></div>
03.         <input type="hidden" asp-for="Customer.ID" />
04.         ...
05.         <div class="form-group">
06.             <label asp-for="Customer.SalutationID" class="control-label"></label>
07.             <input asp-for="Customer.SalutationID" class="form-control" />
08.             <span asp-validation-for="Customer.SalutationID" class="text-danger"></span>
09.         </div>
10.         <div class="form-group">
11.             <input type="submit" value="Save" class="btn btn-default" />
12.         </div>
13.     </form>
```

Damit erreichen wir unser Zwischenziel – siehe Bild 3. Allerdings wollen wir dem Benutzer nicht zumuten, dass er die Zahlenwerte für die jeweiligen Anreden von Hand in das Formular eingeben muss. Stattdessen wollen wir ein Auswahlfeld implementieren. Dazu sind nur zwei kleine Änderungen notwendig.

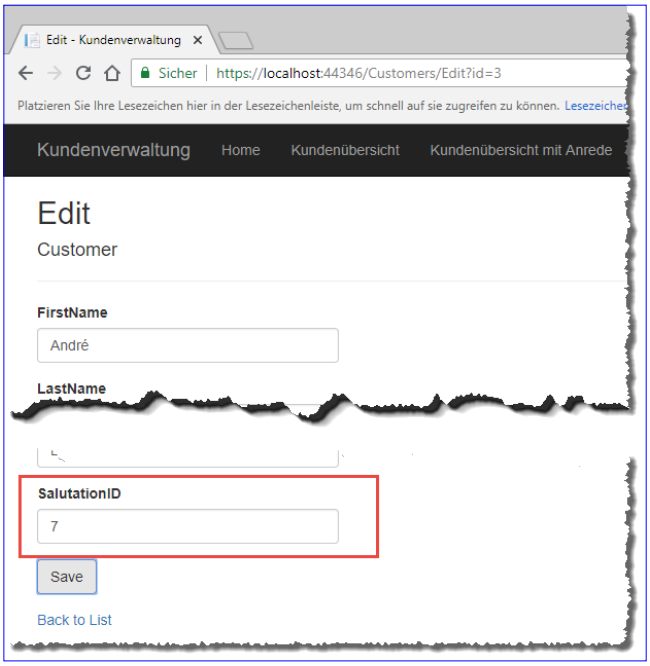


Bild 3: Eingabe des Fremdschlüsselwertes der Anrede

Als Erstes müssen wir die Daten für das Auswahlfeld in der Code behind-Datei **Edit.cshtml.cs** bereitstellen, und zwar in der Methode **OnGetAsync**. Dieser fügen wir als letzte Zeile vor der Anweisung **return Page()** eine Anweisung hinzu, welche das Objekt **ViewData["SalutationID"]** mit einer Liste füllen, die wir als neues Objekt auf Basis der Klasse **SelectList** ermitteln:

```
public async Task<IActionResult> OnGetAsync(int? id) {
    if (id == null) {
        return NotFound();
    }
    Customer = await _context.Customers.FirstOrDefaultAsync(m => m.ID == id);
    if (Customer == null) {
        return NotFound();
    }
    ViewData["SalutationID"] = new SelectList(_context.Salutations, "ID", "Name");
```