AUSGABE 02-03/2018

DATENBANK ENTWICKLER

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

TOP-THEMEN:

André Minhorst Verlag

Impressum

DATENBANKENTWICKLER
© 2015-2018 André Minhorst Verlag
Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

Visual Studio 2017 Community Edition

Nachdem die Beispiele der bisherigen Ausgaben von DATENBANKENTWICKLER, also bis Ausgabe 1/2018, mit Visual Studio 2015 erstellt wurden, steigen wir mit Ausgabe 2/2018 auf Visual Studio 2017 um. Dieser Artikel beschreibt die gegenüber der Version Visual Studio 2015 leicht veränderte Installation. Außerdem fügen wir gleich noch ein paar Komponenten hinzu, mit denen wir uns auf neues Terrain begeben – der Entwicklung einfacher Webanwendungen.

Zum Download der Visual Studio 2017 Community Edition gelangen Sie über den Link **https://www.visualstudio.com/de/**. Dort wechseln Sie zum Download **Visual Studio-IDE** und wählen dort unter **Für Windows herunterladen den Eintrag Community 2017** aus (siehe Bild 1).

Bild 1: Download von **Visual Studio 2017 Community**

Dies lädt eine kleine Datei herunter, die Sie starten und die weitere Informationen nachlädt. Danach erscheint der Dialog aus Bild 2. Wer weiß, warum wir schon beim Download die Edition auswählen mussten? Vermutlich noch nicht einmal Microsoft selbst.

Jedenfalls erhalten wir nach dem Start des Setup-Programms nochmal die Auswahl der gewünschten Edition. Wiederum wählen wir den Eintrag **Visual Studio Community 2017** aus.

Damit landen wir in einem weiteren Dialog, in dem wir einfach auf **Installieren** klicken könnten (siehe Bild 3).

Wenn wir dies tun, erscheint allerdings die Meldung aus Bild 4. Ohne ein sogenanntes **Workload**-Element auszuwählen, würde das Setup nur den Kern-Editor installieren, der aber keine Tools für den Umgang mit Projekten liefert.

Also schauen wir uns doch einmal genauer an, welche Optionen uns der Bereich **Workloads** bietet.

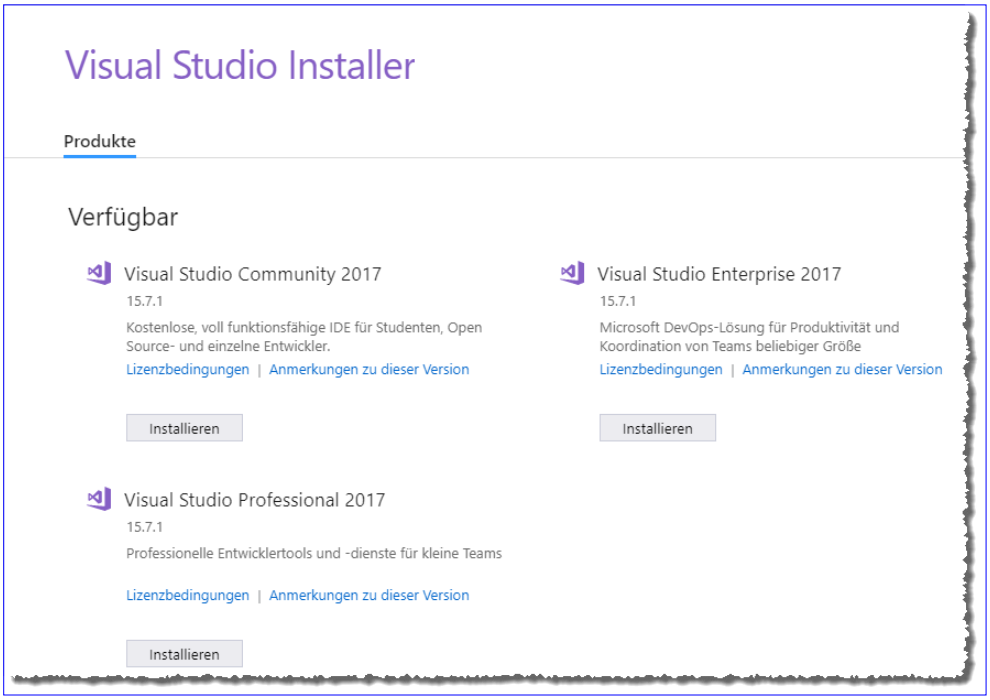

Bild 2: Start des Installers

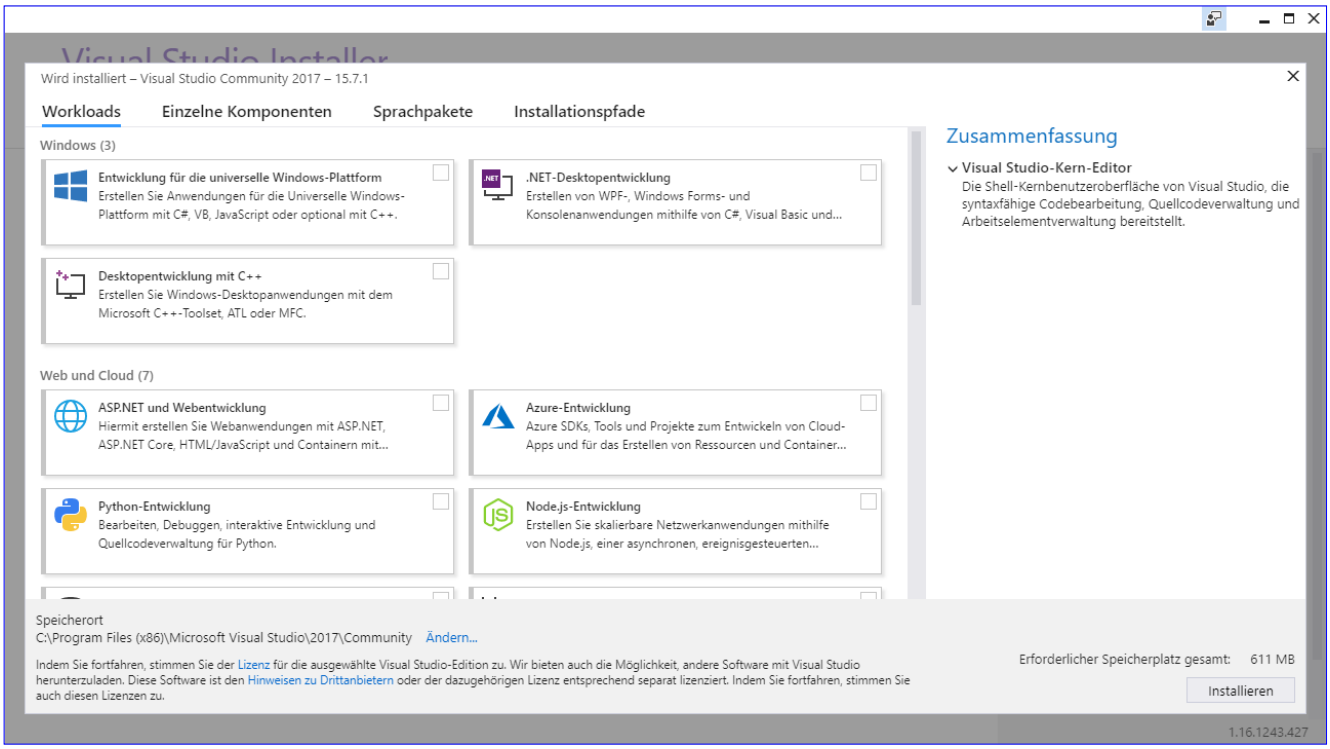

Bild 3: Auswahl der Workloads

Als Erstes aktivieren wir den Eintrag **.NET Desktopentwicklung**, denn diesen benötigen wir zur Entwicklung von WPF-Anwendungen mit den verschiedenen Sprachen. Sobald wir diese Option anhaken, erhalten wir im rechten Bereich noch eine Zusammenfassung der installierten Optionen. Einige davon sind standardmäßig installiert, weitere können Sie optional hinzufügen. Wir nehmen hier zumindest noch den Eintrag **SQL Server Express 2016 LocalDB** hinzu.

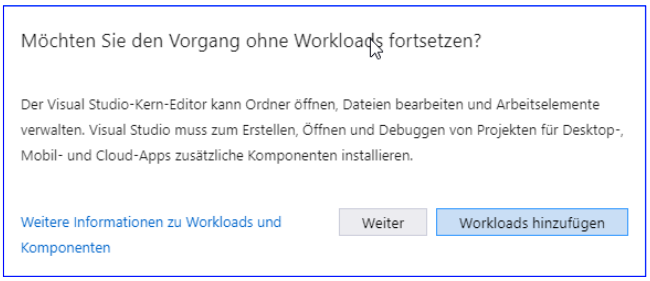

Bild 4: Meldung, wenn man keine Workloads installieren möchte

Außerdem wollen wir auch gleich noch die **.NET Core 1.0 - 1.1-Entwicklungstools** und die **.NET Core 2.0-Entwicklungstools** mitnehmen. .NET Core soll eine neue, leichtgewichtigere Implementierung des .NET-Frameworks sein, das plattformunabhängig und modular aufgebaut ist. Allerdings ist es noch nicht vollständig implementiert und liefert so beispielsweise noch keine Unterstützung für die Entwicklung von WPF-Anwendungen. Sie können damit aber Konsolen- und Webanwendungen mit den neuesten Technologien entwickeln. Das ist auch der Grund, warum wir die .NET Core-Elemente beim Setup mit hinzunehmen.

Das nächste **Workload**-Element, welches wir hinzufügen wollen, ist das Element **ASP.NET und Webentwicklung**. Auch hier gibt es wieder einige vordefinierte Unterelemente im rechten Bereich. Hier nehmen wir noch die beiden Einträge **Entwicklungszeit-IIS-Unterstützung** mit sowie **.NET Core 1.0 - 1.1-Entwicklungstools für Web** mit. Im Zusammenhang damit fügen wir auch noch das **Workload**-Element **Node.js-Entwicklung** hinzu.

Außerdem fügen wir dem Setup noch das **Workload**-Element **Datenspeicherung und -verarbeitung** hinzu. Hier brauchen wir keine weiteren Einträge auszuwählen.

Und da wir sicher auch einmal DLLs oder andere Anwendungen wie beispielsweise Add-Ins für die Zusammenarbeit mit Office erstellen wollen, nehmen wir auch noch das **Workload**-Element **Office-/SharePoint-Entwicklung** mit auf. Hier begnügen wir uns mit dem Eintrag **Visual Studio-Tools für Office (VSTO)**.

Für die Entwicklung von .NET Core-Anwendungen fügen wir noch das Element **Plattformübergreifende .NET Core-Entwicklung** hinzu.

Damit haben wir nun immerhin schon ein rund 10 Gigabyte großes Paket geschnürt, das erst einmal heruntergeladen und installiert werden will. Das ist aber auch nach einigen Minuten erledigt, sodass wir nun den erforderlichen Neustart durchführen können. Danach verbinden Sie die neue Version mit Ihrem Windows-Konto und Visual Studio bereitet alles für den ersten Start vor. Schließlich erwartet Sie die Entwicklungsumgebung von Visual Studio Community 2017 (siehe Bild 5).

ASP.NET Core

Mit dieser Version können Sie nun auch Projekte etwa des Typs **ASP.NET Core-Webanwendung** erstellen. Mit Anwendungen dieses Typs werden wir uns in dieser Ausgabe verstärkt beschäftigen.

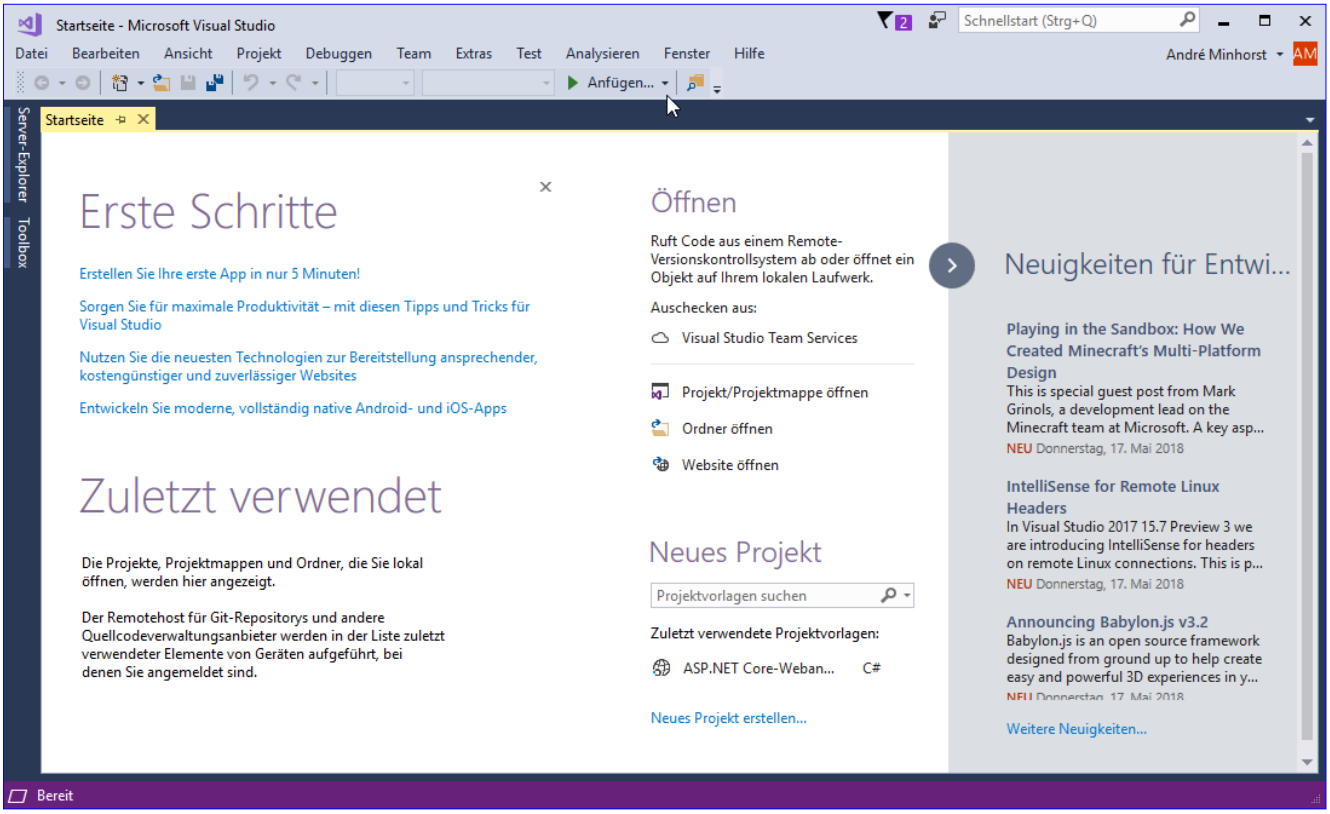

Bild 5: Start der neuen Version von Visual Studio

Einfache ASP.NET Core-Anwendung erstellen

ASP.NET-Webanwendungen gibt es schon, seit Microsoft .NET vorgestellt hat. Mit .NET Core und Bootstrap wird dies noch interessanter, vor allem für die Programmierung moderner Webanwendungen mit Responsive Webdesign – also mit einem Design, das auf die Gegebenheiten des jeweiligen Endgerätes reagiert und die Darstellung darauf optimiert. Dies wird möglich mit dem neuen ASP.NET Core und der Bootstrap-Bibliothek. Dieser Artikel zeigt, wie Sie eine einfache Anwendung auf Basis von ASP.NET Core mit Bootstrap erstellen.

Voraussetzungen

Voraussetzung für das Reproduzieren der in diesem Artikel vorgestellten Abläufe ist das Vorhandensein von Visual Studio Community 2017. Daneben stellt sich an dieser Stelle heraus, dass es gut war, in diesem Magazin zunächst die Programmiersprache C# vorzustellen. ASP.NET Core unterstützt nämlich aktuell noch kein Visual Basic, sondern nur C#.

Projekt anlegen

Um das neue Projekt anzulegen, starten Sie Visual Studio und öffnen den Dialog zum Erstellen eines neuen Projekts. Hier wählen Sie als Projektvorlage den Eintrag **Visual C#\|Web\|ASP.NET Core-Webanwendung** aus (siehe Bild 1).

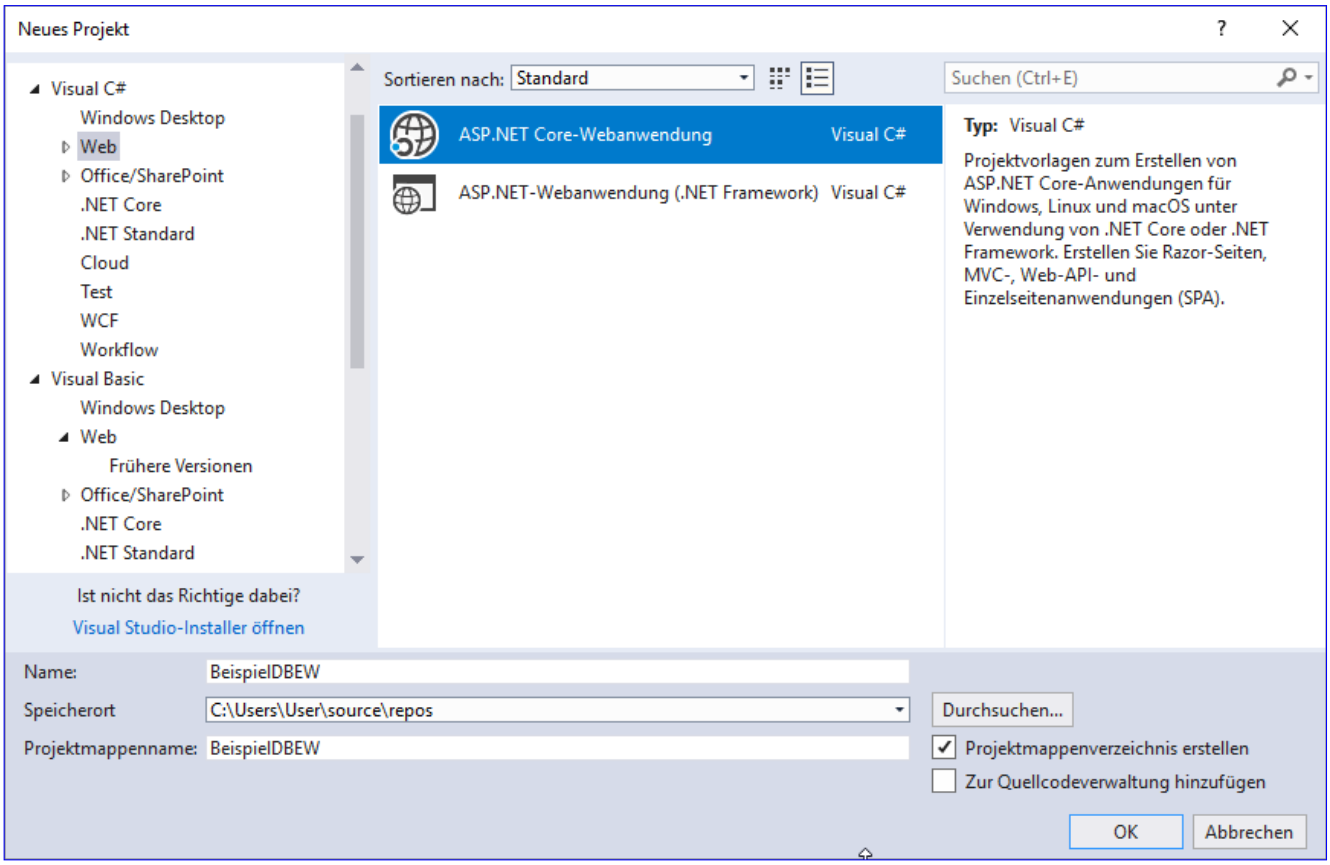

Bild 1: Erstellen einer Webanwendung für .NET Core

Nach dem Festlegen des gewünschten Projektnamens sowie des Speicherorts legt Visual Studio das Projekt allerdings noch nicht direkt an. Zuvor erscheint noch der Dialog aus Bild 2. Hier behalten wir den Eintrag **Webanwendung** sowie die übrigen Einstellungen bei und klicken auf die Schaltfläche **OK**.

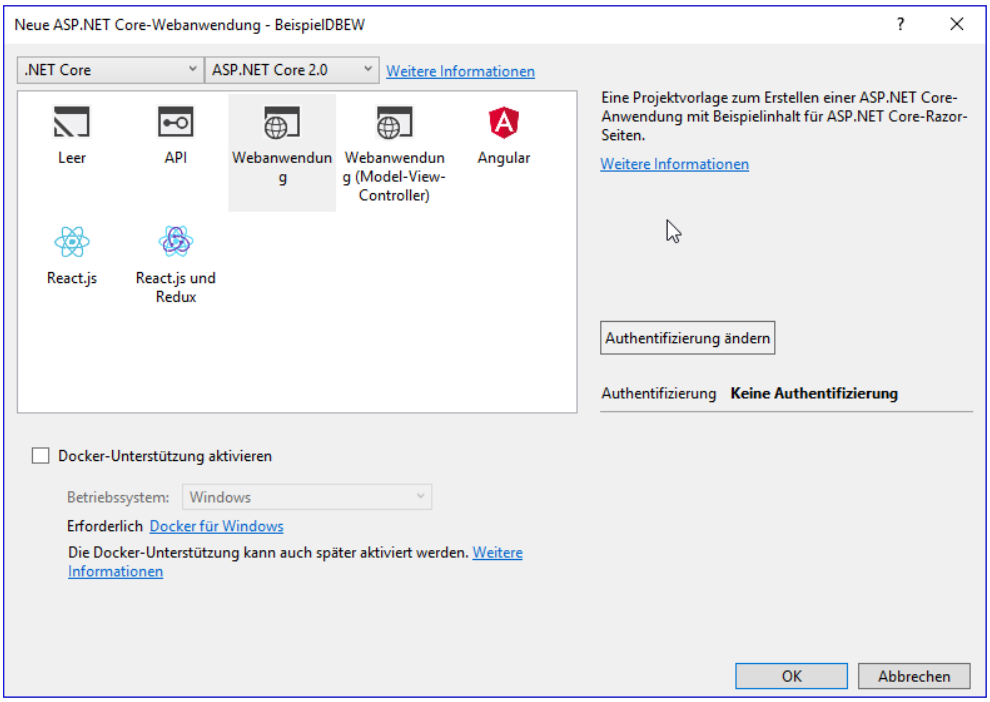

Bild 2: Festlegen weiterer Parameter für die neue Weblösung

Anschließend wird das Projekt erstellt. Ein Blick in den Projektmappen-Explorer liefert die erstellten Projektdateien, die sich – wenig überraschend – in ihrer Zusammensetzung weitgehend von denen einer Desktop-Anwendung unterscheiden (siehe Bild 3). Interessant ist für uns zunächst der Ordner **wwwroot**, der **.css**-Dateien, Bilder und weitere Dateien enthält. Die eigentlichen Internetseiten werden im Verzeichnis **Pages** gespeichert. Hier finden Sie die bekannte Bezeichnungen wie **Index**, **Contakt**, **About** und so weiter, allerdings nicht mit einer Dateiendung wie **.html**, sondern mit der Dateiendung **.cshtml**.

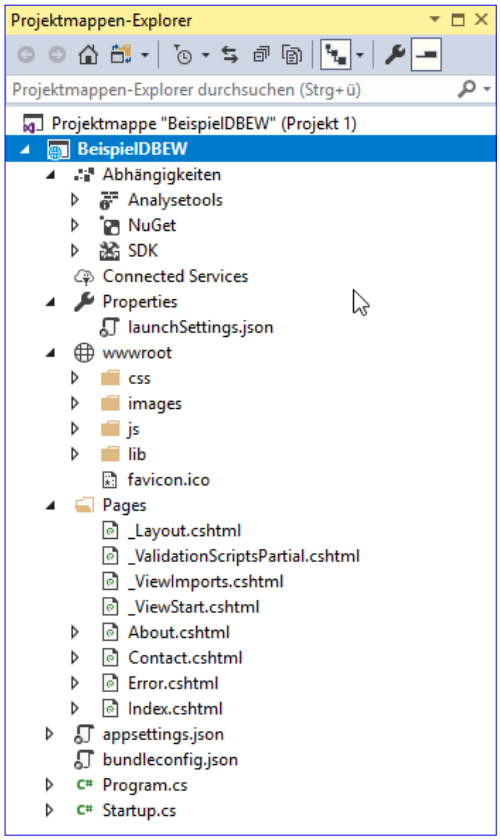

Bild 3: Projektmappen-Explorer der Webanwendung

ASP.NET Core-Projekt starten

Bevor wir näher auf die einzelnen Dateien eingehen, sind wir neugierig, was geschieht, wenn wir einfach einmal auf die **Starten**-Schaltfläche klicken. Diese enthält in diesem Fall die Beschriftung **IIS Express**, was darauf hindeutet, dass wir uns mit der neuen Version von Visual Studio auch gleich noch einen Internet Informationen Server in der Express-Variante installiert haben. Ein Klick auf die **Starten**-Schaltfläche blendet dann in Visual Studio die Diagnose-Tools ein, welche die Speicherauslastung anzeigen. Interessanter ist aber, dass die Anwendung in einem neuen Browser-Fenster angezeigt wird (sofern nicht bereits ein Browser geöffnet ist). Hier finden wir bereits einen Header-Bereich mit der Seitennavigation sowie ein Slide-Element, das verschiedene Features für Web-Anwendungen anpreist (siehe Bild 4). Die Links funktionieren wir erwartet und liefern die entsprechenden weiterführenden Seiten im Browser.

In der URL-Leiste finden wir eine Adresse wie **localhost:12345**. **localhost** ist der Standardhostname für den lokalen Computer. Hinter dem Doppelpunkt folgt die Angabe des Ports in Form einer fünfstelligen Zahl. Diese Zahl wird beim Starten des Projekts zufällig vergeben. Unter **Application uses** finden wir bereits einen Hinweis auf die verwendeten Techniken. **ASP.NET Core Razor Pages** bieten einen einfacheren Ansatz zur Programmierung von Web-Anwendungen als es mit den bisherigen ASP.NET basierten Anwendungen nach dem MVC-Pattern der Fall war. Die zweite Technik heißt **Bootstrap**. Dabei handelt es sich um eine Bibliothek, welche das Design von Anwendungen für verschiedene Endgeräte vereinfacht. Wenn Sie unter Verwendung von Bootstrap nun beispielsweise das Browser-Fenster viel schmaler machen, verschwinden die Einträge der Navigationsleiste und werden in Form eines kleinen Icons rechts oben angezeigt, welches dann per Mausklick die Menüeinträge liefert (siehe Bild 5).

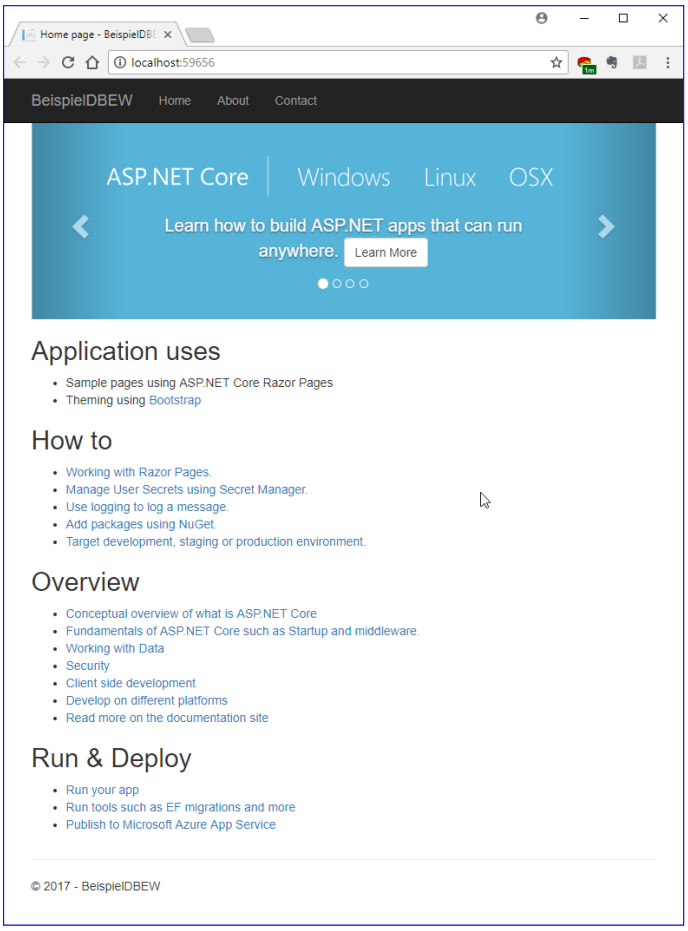

Bild 4: Erster Start der Beispielanwendung

Damit sind sowohl Bootstrap also auch **Razor Pages** bereits in der Beispielanwendung enthalten. **Razor Pages** war die Vorlage, die wir gewählt haben, also sind die entsprechenden Elemente ebenfalls bereits vorhanden. Mit der Beispielanwendung können Sie nun ein wenig experimentieren. Anschließend legen wir ein neues, leeres ASP.NET Core-Projekt an und zeigen, wie Sie selbst die benötigten Elemente hinzufügen.

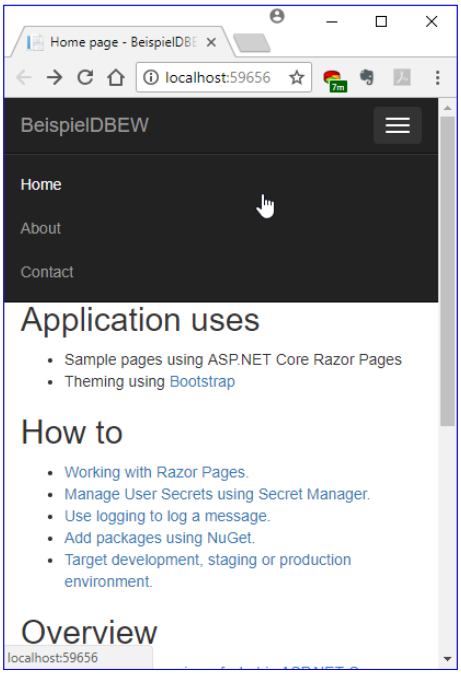

Bild 5: Vorteil mit Bootstrap: Skalierung der Größe der Anwendung

Leeres Projekt anlegen

Ein leeres Projekt legen Sie auf ähnliche Weise an. Im Dialog **Neues Projekt** wählen Sie wiederum den Eintrag **ASP.NET Core-Webanwendung** aus und geben den Projektnamen und den Pfad an – in diesem Fall verwenden wir **DBEWBlank**. Im danach erscheinenden Dialog **Neue ASP.NET Core-Webanwendung** wählen Sie die Vorlage **Leer** aus (siehe Bild 6).

Wenn Sie das Projekt schon jetzt starten, erhalten Sie im Browser die Ansicht aus Bild 7. Woher kommt diese Seite? Dazu schauen wir uns den Code der Datei **Startup.cs** an, der nach dem Entfernen der Kommentare wie folgt aussieht.

ASP.NET Core-Anwendung anpassen

Wenn Sie ein neues Projekt auf Basis der Vorlage »Visual C#|Web|ASP.NET Core-Webanwendung« mit dem Typ »Webanwendung« erstellt haben, finden Sie eine komplette Beispielanwendung vor. Im Gegensatz dazu steht der Typ »Leer«, mit dem Sie eine komplett leere Anwendung erstellen, der allerdings auch jegliche Infrastruktur wie etwa das Menü, die CSS-Dateien und vieles mehr fehlen. Wir wollen einmal von einer Anwendung des Typs »Webanwendung« ausgehend betrachten, welche Anpassungen notwendig sind, um die Beispielinhalte, -menüs und -designs in eigene Elemente umzuwandeln.

Für die Beispiele dieses Artikels legen Sie wieder ein Projekt auf Basis der Vorlage **Visual C#|Web|ASP.NET Core-Webanwendung** an. Wählen Sie im Dialog **Neue ASP.NET Core Webanwendung** den Eintrag **Webanwendung** aus. Dies liefert eine Anwendung, die nach dem Starten ein Menü und entsprechende Inhalte im Webbrowser anzeigt (siehe Bild 1). Allerdings ist es gar nicht so einfach, zu identifizieren, wo die einzelnen Bestandteile der Anwendung wie die Inhalte, das Menü und das Layout definiert sind. Somit ist auch die Anpassung etwas kompliziert. Aber genau das beleuchtet der vorliegende Artikel: Wir sehen uns an, wo die verschiedenen Elemente, die Sie nach dem Start der Beispielanwendung sehen, angepasst werden können und wie Sie eine für eigene Zwecke vorbereitete Vorlage daraus erstellen können.

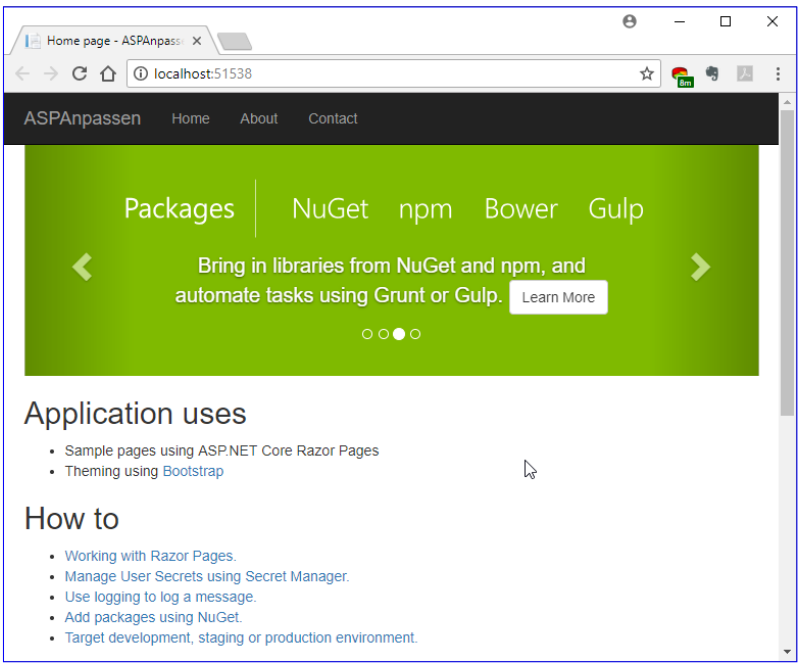

Bild 1: So sieht die Anwendung nach dem ersten Start aus.

Bild 2: Code der Seite **Index.cshtml**

Startseite anpassen

Wir beginnen mit der Seite, die zu Beginn angezeigt wird. In einem Projekt, das auf Basis der oben genannten Vorlage erstellt wurde, sind die Seiten mit den Inhalten im Ordner **Pages** untergebracht. Hier finden wir auch gleich die Seite **Index.cshtml**. Die Index-Seite ist normalerweise die zuerst angezeigte Seite einer Webanwendung,

wenn keine anderen Seite in der URL angegeben wurde. Nach einem Doppelklick auf den entsprechenden Eintrag im Projektmappen-Explorer zeigt dann auch gleich den Code der Seite an (siehe Bild 2). Interessanterweise finden wir hier ganz oben nicht etwa den Code für die Menüleiste, sondern den für den Slider, der durchlaufend verschiedene Inhalte anzeigt. Um zu prüfen, welche Inhalte nun überhaupt in der Seite **Index.cshtml** gespeichert werden, kommentieren wir einfach einmal alle HTML-Inhalte dieser Seite aus. Das erledigen Sie in **.cshtml**-Seiten auf eine der folgenden Arten: Entweder Sie verwenden die unter HTML üblichen Elemente zum Auskommentieren (**<!--...-->**) oder Sie nutzen die dafür vorgesehenen Tags der Razor Pages, nämlich **@*...*@**. Ich nutze gern immer die zum Auskommentieren vorgesehene Tastenkombination **Strg + K, Strg + C** und zum Einkommentieren **Strg + K, Strg + U**. Dies schließt den markierten Text in **@*...*@** ein und kommentiert diesen so aus.

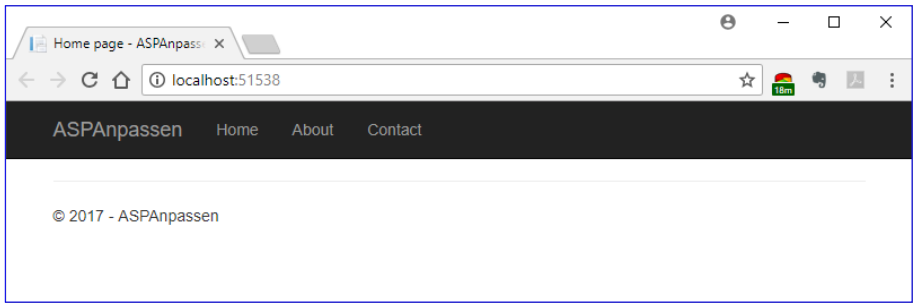

Bild 3: Was von **Index.cshtml** übrig bleibt ...

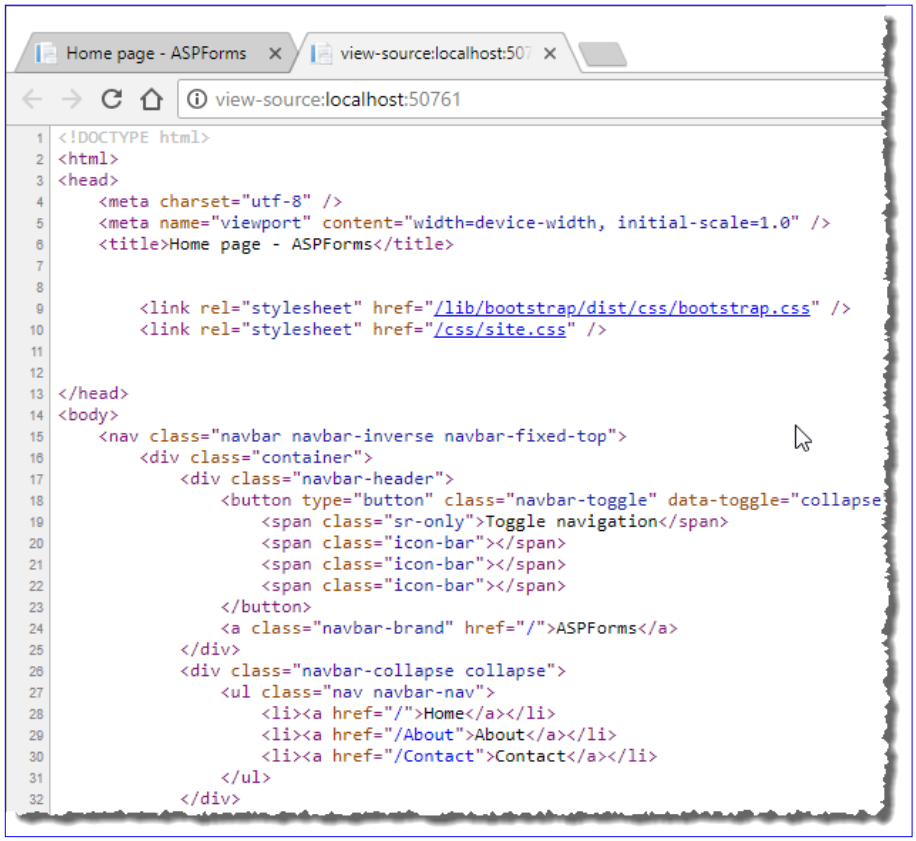

Bild 4: Quellcode der Startseite

Danach starten wir die Webanwendung erneut und schauen uns an, was von der Startseite übrig bleibt. Das Ergebnis sehen Sie in Bild 3 – es bleiben nur das Menü und die Fußzeile übrig. Damit wissen wir schon einmal, welchen Teil der Webanwendung wir über die Seite **Index.cshtml** manipulieren können. Darüber hinaus stellt sich heraus, dass **Index.cshtml** nicht den kompletten Inhalt der angezeigten Startseite enthält, sondern nur den zwischen Navigationsleiste und Fußzeile erscheinenden Inhalt. Den Rest müssen wir also an anderen Stellen im Projekt suchen.

Navigationsleiste ändern

Als Nächstes wollen wir herausfinden, wo sich der Code für die Navigationsleiste befindet. Dazu schauen wir uns den Quellcode der Webseite an. Dazu klicken Sie beispielsweise unter Google Chrome mit der rechten Maustaste auf die Webseite und wählen den Kontextmenü-Eintrag **Seitenquelltext anzeigen** aus. Dies öffnet dann eine neue Registerseite im Browser mit

dem Inhalt aus Bild 4. Die Einträge für die Navigationsleiste finden wir dann im unteren Bereich. Die relativen Links, die zu den übrigen Seiten führen und die in den **<a>**-Elementen unter dem Attribut **href** angegeben sind, heißen etwa **/About** und **/Contact**. Davon ausgehend, dass der Code der Navigationsleiste nicht dynamisch auf Basis der Daten etwa aus einer Datenbank generiert wird, müssten sich also in den Dateien des Projekts die Zeichenfolgen **/About** und **/Contact** finden lassen. Die passenden Dateien dazu befinden sich dann im gleichen Verzeichnis wie die Datei, welche diese Links enthält – darauf weist der relative Pfad mit **/About** hin.

Bild 5: Quellcode der Layout-Seite

Die gewünschte Seite mit dem Navigationsmenü ist dann auch schnell gefunden: Es handelt sich um die Seite **_Layout.cshtml**. Der Quellcode dieser Seite sieht wie in Bild 5 aus. Hier finden wir nicht nur den Code für die Navigation (obere Markierung), sondern auch noch den Bereich, in den die eigentlichen Seiteninhalte wie etwa aus den Dateien **Index.cshtml**, **About.cshtml** oder **Contact.cshtml** angezeigt werden (untere Markierung).

Das dortige Element **@RenderBody()**, welches übrigens ein Markup-Element der Razor Pages ist, sorgt offensichtlich dafür, dass der Inhalte der jeweils anzuzeigenden Seite hier eingefügt wird.

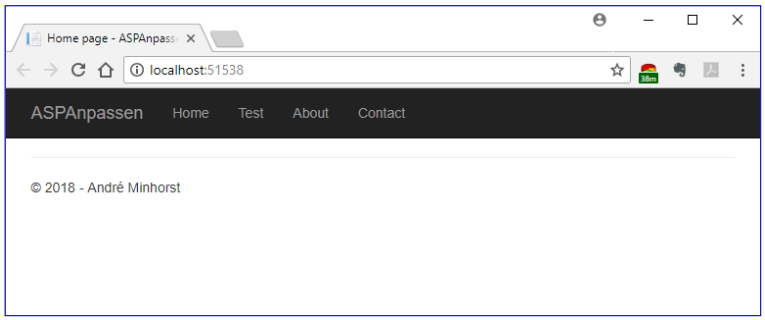

Bild 6: Die neue Index-Seite mit angepasster Datei **_Layout.cshtml**

Ändern wir also einmal den Fußbereich wie folgt:

```
<footer>
    <p>&copy; 2018 - André Minhorst</p>
</footer>
```

Und der Navigationsleiste fügen wir ein Element hinzu:

Markup mit Razor Pages

Wenn Sie Webseiten auf Basis von ASP.NET Core bauen, wollen Sie in den statischen HTML-Code gegebenenfalls auch dynamische Elemente einbauen. Dazu verwenden Sie eine sogenannte Server Side Markup Language. Eine solche Sprache ist das relativ neue Razor Pages. Dieser Artikel zeigt, wie Sie den Inhalt von HTML-Seiten über Razor Pages durch dynamische Inhalte erweitern.

Wenn Sie schon einmal eine Seite mit PHP oder mit klassischen ASP programmiert haben, kennen Sie die Code-Schnipsel, die dort in den HTML-Code eingepflegt werden. Unter PHP etwa beginnen diese mit **<?php** und enden mit **?>**. Für ASP.NET Core gibt es mit Razor Pages eine neue Technologie, die wir uns im vorliegenden Artikel ansehen wollen. Voraussetzung dafür ist, dass Sie ein einfaches Webprojekt auf Basis von ASP.NET Core angelegt haben – siehe auch im Beitrag **Einfache Web-Anwendung erstellen**. Die dort angelegte Anwendung nutzt eine statische Seite namens **index.html** als Startseite. Diese kann wegen ihrer Dateiendung keine dynamischen Elemente aufnehmen.

Model View Controller hinzufügen

Model View Controller ist ein Entwurfsmuster, mit dem eine Software in verschiedene Ebenen aufgeteilt wird. Das Model entspricht dem Datenmodell, die View ist die Präsentation der Daten, also die Benutzeroberfläche und der Controller steuert die Anwendung. An dieser Stelle steigen wir noch nicht tiefer in dieses Entwurfsmuster ein (genau genommen verwenden wir MVC noch nicht einmal im eigentlichen Sinne), aber wir benötigen einen Service, der uns bei der Ausführung sogenannter MVC-Aktionen unterstützt. In diesem Fall wird die MVC-Aktion eine HTML-Seite mit einigen Besonderheiten sein, mehr dazu weiter unten. Als Erstes schauen wir uns an, welche Änderungen im Quellcode der Klasse **Startup.cs** nötig sind, um die Verarbeitung von MVC-Aktionen zu unterstützen. Wir fügen also zunächst den Aufruf der Methode **AddMVC** zur Methode **ConfigureServices** hinzu. Danach erweitern wir die Methode **Configure** neben dem Aufruf der Methode **UseStaticFiles** um die Methode **UseMVC**:

```
public class Startup {
    public void ConfigureServices(IServiceCollection services) {
        services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
        if (env.IsDevelopment()) {
            app.UseDeveloperExceptionPage();
        }
        app.UseStaticFiles();
        app.UseMvc();
    }
}
```

Razor-Seite hinzufügen

Nun benötigen wir einen Ersatz für die bisher statisch verwendete Seite **index.html**. Dazu rufen wir wieder den Kontextmenü-Eintrag **Neues Element hinzufügen...** auf, diesmal aber für den noch hinzuzufügenden Ordner **Pages**. Im Dialog **Neues**

Element hinzufügen wählen Sie unter der Kategorie **Web** den Eintrag **Razor Seite** aus und legen den Namen **index.cshtml** für die Seite fest (siehe Bild 1).

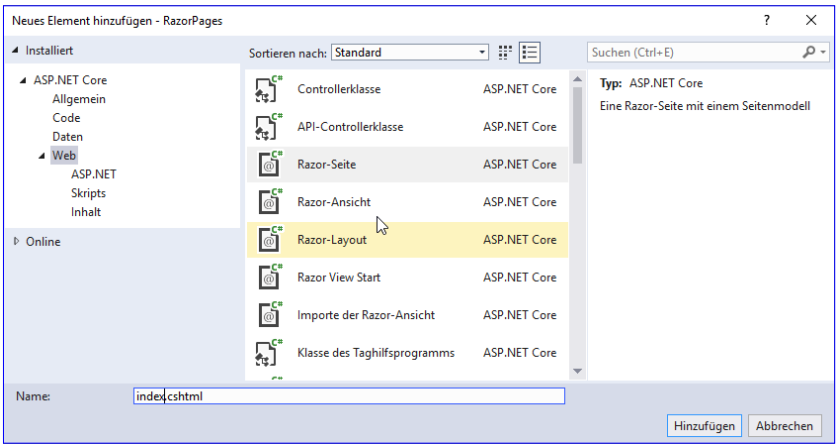

Bild 1: Hinzufügen einer Razor-Seite

Die neue Seite namens **index.cshtml** sieht wie in Bild 2 aus. Hier ist zunächst kein HTML-Code in Sicht, was ein wenig irritiert. Stattdessen erwartet uns ein als Fehler markierter Ausdruck namens **indexModel**.

Zusätzlich liefert der Projektmappen-Explorer noch eine der Seite **index.cshtml** untergeordnete Datei namens **index.cshtml.cs** (siehe Bild 3). Das Schema erinnert uns an die Code behind-Klassen von WPF-Seiten. Der Fehler hinter dem Element **indexModel** in der Datei **index.cshtml** lautet »Der Typ- oder Namespacename "indexModel" wurde nicht gefunden (möglicherweise fehlt eine using-Direktive oder ein Assemblyverweis).«.

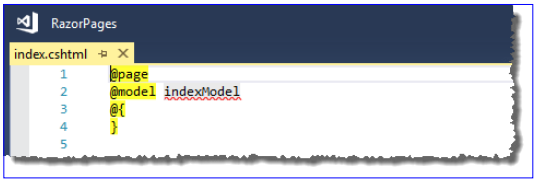

Bild 2: Code der neuen Seite **index.cshtml**

Wenn wir die Anwendung starten, erhalten wir auch tatsächlich eine Fehlermeldung im Browser (siehe Bild 4). Das ist übrigens dem Umstand geschuldet, dass wir in der Methode **Configure** der Klasse **Startup** für den Entwicklungsmodus die Methode **UseDeveloperExceptionPage** aufgerufen haben. Der Fehler liegt darin, dass wir den Namespace zur Seite **index.cshtml** nicht angegeben haben. Das holen wir nun nach, indem wir den Code der Seite **index.cshtml** wie folgt ändern und den Namespace **RazorPages** sowie den Namen des Ordners **Pages** hinzufügen:

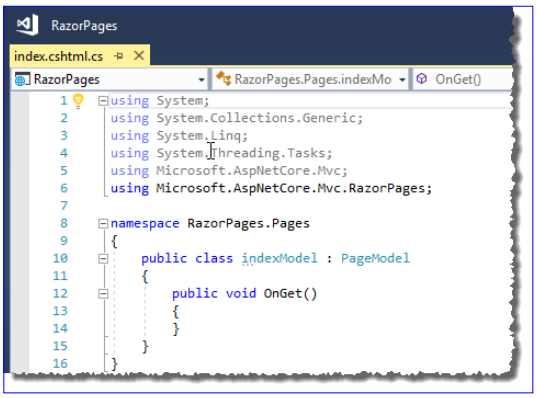

Bild 3: Die »Code behind«-Klasse von **index.cshtml**

```
@page
@model RazorPages.Pages.indexModel
@{
}
```

Danach können wir die Anwendung ohne Fehlermeldung starten, erhalten allerdings auch eine leere Seite – kein Wunder, denn wir haben ja noch keine Inhalte auf der Seite **index.cshtml** hinterlegt. Das ändern wir nun durch Hinzufügen der folgenden Zeile:

```
<h1>Hello index.cshtml!</h1>
```

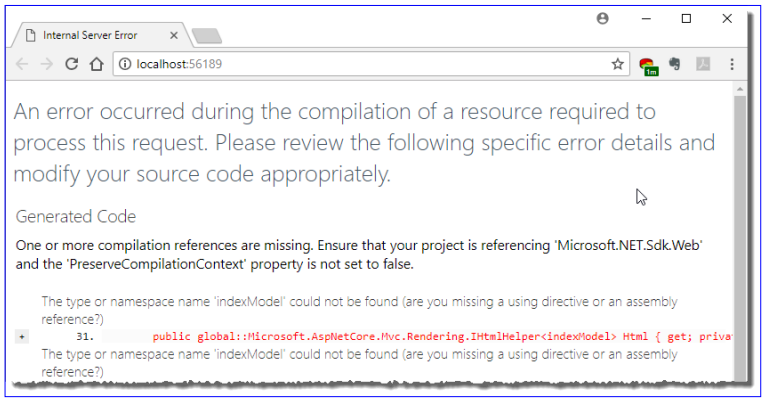

Bild 4: Fehler beim Starten der Anwendung

Beim nächsten Start der Anwendung sehen wir dann auch den gewünschten Text im Browser (siehe Bild 5).

Bild 5: Die Seite läuft!

Dynamische Inhalte hinzufügen

Nun gehen wir einen Schritt weiter und wollen einen dynamischen Inhalt hinzufügen – was der eigentliche Sinn des Artikels ist. Dazu nutzen wir das **@**-Zeichen und die dadurch per IntelliSense eingeblendeten verfügbaren Code-Elemente. In diesem Fall wollen wir das aktuelle Datum und die Uhrzeit ausgeben (siehe Bild 6). Das erledigen wir mit folgendem Code (die ersten Zeilen bleiben natürlich erhalten):

```
<h1>Hello index.cshtml!</h1>
<p>Es ist jetzt @DateTime.Now</p>
```

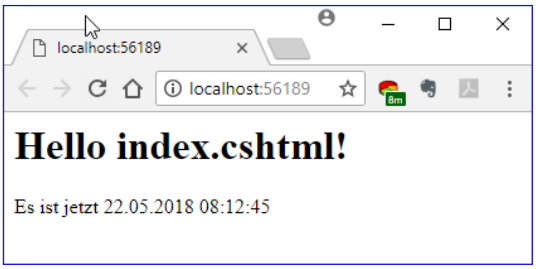

Bild 6: Anzeige von Datum und Uhrzeit

Razor-Anweisungen

Schauen wir uns nun an, was die mit **@** beginnenden Anweisungen in der Datei **index.cshtml** bedeuten.

- **@page**: Diese Anweisung kennzeichnet die Seite als sogenannte MVC-Aktion. Dies ist immer die erste Anweisung auf der Seite.

- **@model RazorPages.Pages.indexModel**: Diese Seite gibt an, wo sich das Model für diese Seite (View) befindet. Dabei handelt es sich um die Klasse **indexModel** im Ordner **Pages** des Namespaces **RazorPages**.

- **@{...}**: Hier können Sie C#-Anweisungen eingeben, die zur Laufzeit auf dem Server ausgeführt werden.

Möglichkeiten, C#-Code und HTML in Razor Pages zu kombinieren

Wenn Sie HTML-Code und C#-Code so kombinieren wollen, dass der C#-Code zur Laufzeit auf dem Server ausgeführt wird und das Ergebnis mit dem bereits vorhandenen HTML-Code zusammengestellt wird, haben Sie verschiedene Möglichkeiten:

- Sie schreiben den C#-Code in **@{...}**-Blöcke. Jede Anweisung muss dann mit einem Semikolon abgeschlossen werden.

- Sie fügen Variablen oder Funktionsaufrufe mit führendem **@** in den HTML-Code ein.

Die Dateien, die solche Kombinationen aus HTML- und C#-Code enthalten, müssen immer die Dateiendung **.cshtml** aufweisen. Auf diese Weise können Sie etwa in einem Razor Code-Block eine Variable deklarieren und füllen und dann im HTML-Code über den **@**-Operator auf die Variable zugreifen. Hier ist gleich anzumerken, dass Variablen in Razor-Blöcken immer als **var** deklariert werden. Anweisungen werden wie unter C# üblich mit dem Semikolon abgeschlossen (**;**). Zeichenketten werden in Anführungszeichen eingefasst. Achtung: Razor Code unter C# ist case sensitive, **Vorname** ist also ungleich **vorname**. Ein Beispiel für sieht etwa wie folgt aus:

```
@page
```

```
@model RazorPages.Pages.indexModel
@{
    var Seitentitel = "Startseite";
}
<h1>@Seitentitel</h1>
```

Dies liefert im Browser eine Seite, die lediglich den Text **Startseite** im Format **h1** enthält. Was geschieht nun auf dem Server? Dieser führt zunächst die Razor Code-Anweisungen aus, fügt den dadurch generierten Inhalt an den entsprechenden Stellen im HTML-Code ein und schickt dann beides zusammen zum Client.

Pfad zum Model per using

Die Anweisung **@model RazorPages.Pages.indexModel**, die das Model über den Namespace **RazorPages**, den Pfad **Pages** und den Klassennamen **indexModel** referenziert, können wir auch wie folgt auf zwei Anweisungen aufteilen:

```
@using RazorPages.Pages
@model indexModel
```

Das in der **@model**-Zeile angegebene Model können wir anschließend im HTML-Code mit **@Model** referenzieren – wobei Model groß geschrieben werden muss, **@model** mit kleinem **m** darf nur einmalig zum Zuweisen des Models für die Seite verwendet werden. Was fangen wir nun damit an? Wir öffnen einmal die Datei **index.cshtml.cs**, die wir auch als Code behind-Datei für die Razor Page bezeichnen können. Dort fügen wir eine öffentliche Eigenschaft namens **Seitentitel** wie folgt hinzu:

```
namespace RazorPages.Pages {
    public class indexModel : PageModel {
        public string Seitentitel { get; set; } = "Seitentitel aus dem Model";
        public void OnGet() {
        }
    }
}
```

Diese Eigenschaft können wir nun auch von der Seite **index.cshtml** aus referenzieren und so dafür sorgen, dass der Wert aus der Model-Klasse aus der Code behind-Datei in das Ergebnis, als die **.html**-Seite, eingearbeitet wird. Dies zeigt als Erstes den Seitentitel wie in der **.cshtml**-Datei definiert und dann den Seitentitel aus dem Model an (siehe Bild 7):

```
<h1>@Seitentitel</h1>
<h1>@Model.Seitentitel</h1>
```

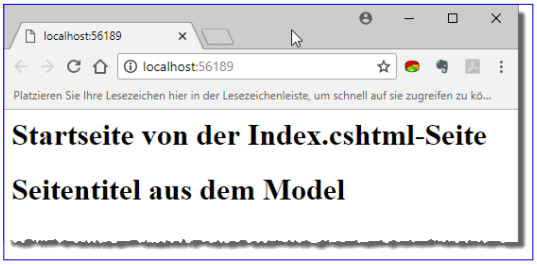

Bild 7: Inhalte von verschiedenen Quellen

Inline oder mehrzeilig?

Es gibt zwei Varianten, Razor Code im HTML-Code unterzubringen. Entweder Sie fügen einfache Ausdrücke einfach mit einem führenden

@-Zeichen in den Code ein. Oder Sie verwenden das Konstrukt **@{...}**, um komplette Anweisungen oder auch mehrzeilige Codes zu nutzen. Die einfache Variante haben Sie bereits oben gesehen:

```
<h1>@Seitentitel</h1>
```

Die Variante mit kompletten Anweisungen sieht so aus – hier zunächst mit einer Anweisung:

```
@{ var Seitentitel = "Dies ist der Seitentitel"; }
```

Hier geben wir komplette C#-Anweisungen inklusive Semikolon ein. Sie können aber auch mehrere Anweisungen eingeben:

```
@{
    var Seitentitel = "Dies ist der Seitentitel";
    var AndereInformation = "Andere Information";
}
```

Variablen im Razor-Code

Variablen im Razor-Code werden mit dem Datentyp **var** angegeben. Das unter C# am Zeilenende gängige Semikolon ist auch hier obligatorisch (hier in einer Zeile):

```
@{ var Text = "Dies ist ein Beispieltext."; }
```

Danach können wir im HTML-Code mit **@Text** auf die Variable zugreifen:

```
<p>@Text</p>
```

Wir können den HTML-Code aber interessanterweise auch in den Codeblock integrieren:

```
@{
    var TextIntegriert = "Dies ist ein in den Razor-Code integrierter Beispieltext.";
    <p>@TextIntegriert</p>
}
```

Anführungszeichen oder Backslash in Razor-Code

Wenn Sie innerhalb eines Razor-Blocks Anführungszeichen oder das Backslash-Zeichen nutzen wollen, stellen Sie der in Anführungszeichen angegebenen Zeichenkette ein @-Zeichen voran, also etwa so:

```
@{
    var TextMitSonderzeichen = @"Text mit "" und \"
}
```

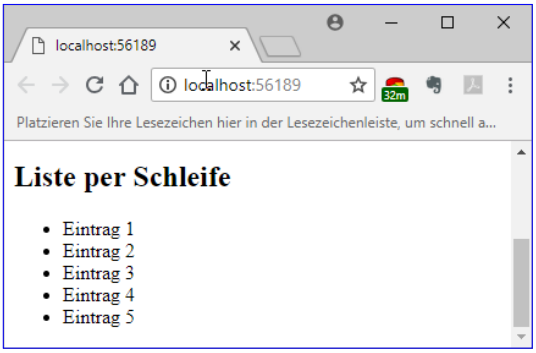

Bild 8: Schleife im **.cshtml**-Code

Webdesign mit Bootstrap Studio

Wer einmal eine Webseite mit CSS designt hat, weiß, wie aufwendig das ist. Selbst das Anpassen bestehender Seiten ist nicht gerade intuitiv, wenn man sonst eher in der Programmierung unterwegs ist. Manchmal führt allerdings kein Weg um Designarbeit herum, und dann ist es wichtig, ein gutes Tool an der Hand zu haben. Die mit ASP.NET Core Razor Pages verwendete Technik namens Bootstrap können Sie beispielsweise mit dem Tool Bootstrap Studio sehr gut in den Griff bekommen. Dieser Artikel zeigt die grundlegenden Schritte zum Designen einer Webanwendung mit Bootstrap Studio.

Das Tool **Bootstrap Studio** finden Sie unter dem Link bootstrapstudio.io. Es kostet regulär 50$, zum Zeitpunkt der Erstellung dieses Artikels gab es einen reduzierten Preis von 25$. Nachdem Sie das Tool heruntergeladen und installiert haben, können Sie es gleich starten. Es liefert dann praktischerweise gleich den **Getting Started Guide** (siehe Bild 1). Bevor wir dort einsteigen, noch ein paar Erläuterungen, warum wir neben Visual Studio noch ein weiteres Tool benötigen. Visual Studio ist eine exzellente Entwicklungsumgebung, aber für die Razor Pages bietet es eben noch keine guten Tools zur Entwicklung der Benutzeroberflächen. Dies ist gerade anspruchsvoll, wenn Sie Webanwendungen bauen wollen, die nicht nur auf einem Endgerät laufen sollen, sondern sich an verschiedene Bildschirmgrößen anpassen. Genau dabei unterstützt uns Bootstrap als ein CSS-Framework, das Vorlagen für die unterschiedlichen Design-Elemente liefert. Die Razor Pages können Bootstrap nutzen, was uns eine Menge Zeit sparen würde, wenn wir die Möglichkeit hätten, das Bootstrap-Framework zur Erstellung der Gestaltungselemente zu nutzen. Und hier setzt Bootstrap Studio an.

Bootstrap Studio

Bootstrap Studio kommt mit drei Bereichen. Der linke liefert, ähnlich wie die Toolbox in Visual Studio, die verschiedenen Elemente, die Sie in Ihre Internetseiten integrieren können. Diese ziehen Sie dann einfach auf den Bereich in der Mitte, der die ak-

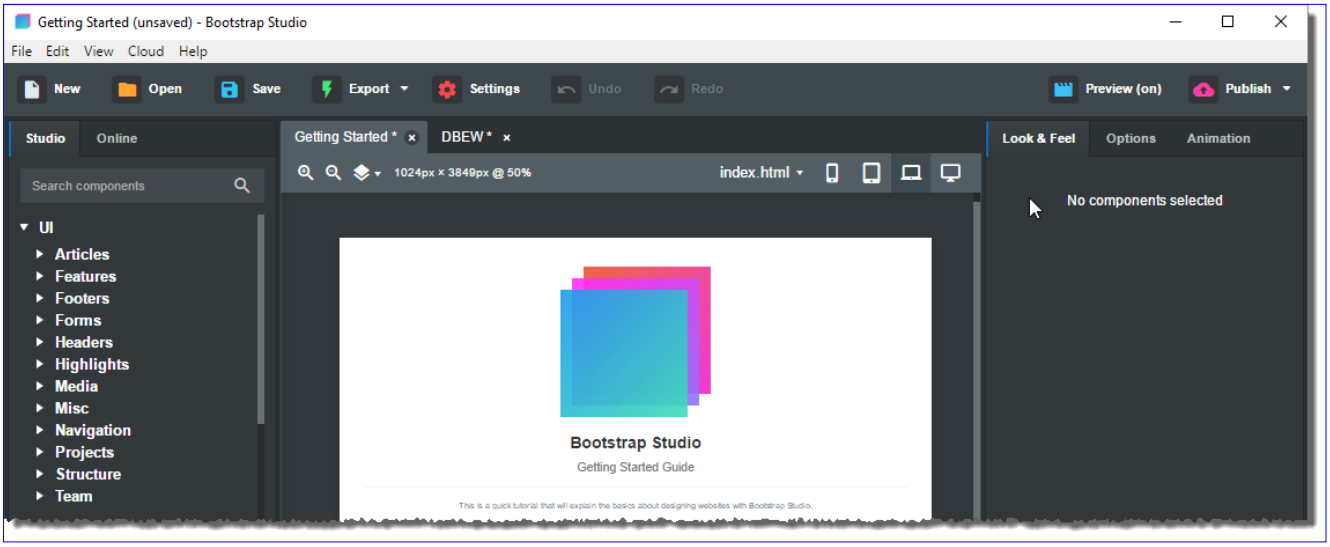

Bild 1: Startbildschirm von Bootstrap Studio

tuelle Ansicht der Internetseite liefert. Sie können hier an beliebigen Stellen Design-Elemente einfügen, wobei Sie zunächst einmal das erste Element hinzufügen müssen. Wenn Sie eine komplette Webseite wie beispielsweise die automatisch beim Anlegen eines neuen ASP.NET Core-Webprojekts generieren wollen, würden Sie beispielsweise wie folgt vorgehen:

- Als Erstes ziehen wir eine geeignete Navigation aus der Toolbox in den Entwurf. Um herauszufinden, welche Navigationen es gibt, klappen Sie den Eintrag **Navigation** auf und überfahren die einzelnen Einträge mit der Maus. Der Eintrag **Navigation with Button** entspricht etwa der Navigation, die Visual Studio anlegt, wenn Sie die Option zum Hinzufügen einer Benutzerverwaltung aktivieren (siehe Bild 2). Ziehen Sie diesen Eintrag einfach in den Entwurf der Seite im mittleren Bereich.

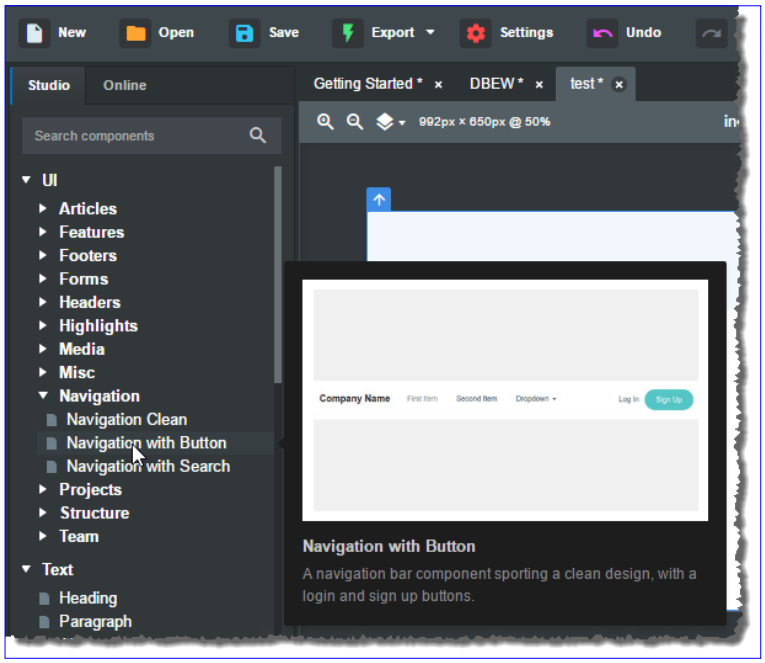

Bild 2: Ansehen der verschiedenen Navigationen

- Das Ergebnis sieht dann wie in Bild 3 aus. Bevor wir dieses Element anpassen, fügen wir zunächst weitere Elemente hinzu.

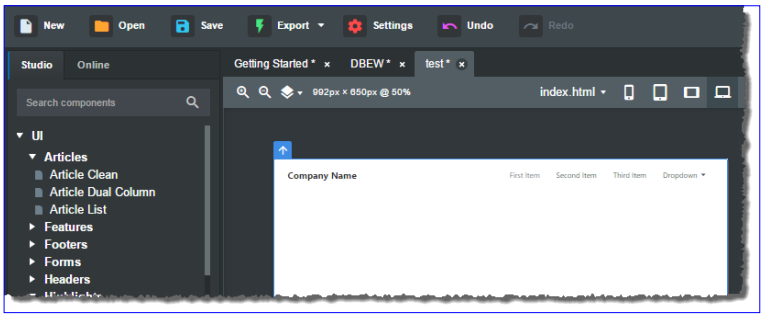

Bild 3: Die Navigation als erstes hinzugefügtes Element

- Danach benötigen wir den Inhalt. Den Slider, den wir im Beispielprojekt von Visual Studio finden, wollen wir an dieser Stelle weglassen und stattdessen eines der **Article**-Templates hinzufügen. Wir überfahren diese wieder, um das für uns passende Template zu finden, und entscheiden uns für die Vorlage **Article Clean**. Diese können wir nun per Drag and Drop an zwei Stellen einsetzen – entweder über oder unter dem bereits vorhandenen Element. Der Artikel soll natürlich darunter platziert werden. Dies fügt auch gleich noch ein paar Bilder hinzu – praktisch, wenn Sie auch Bilder in ihrem Inhalt aufnehmen möchten (siehe Bild 4).

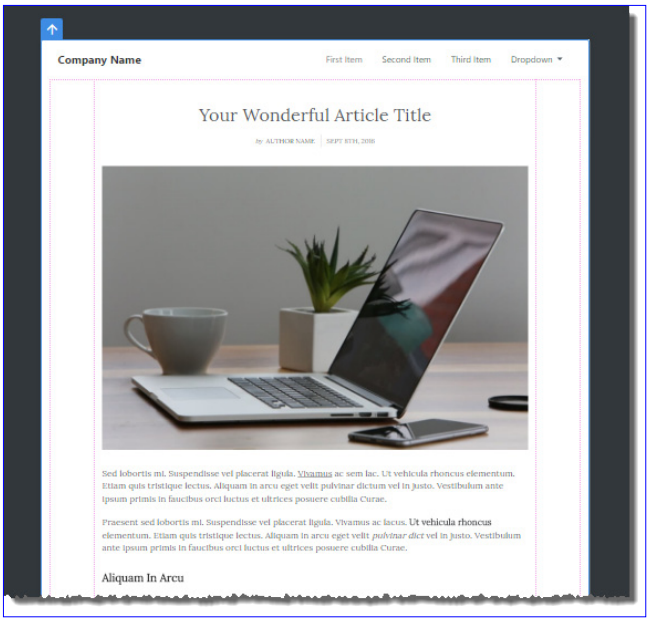

Bild 4: Die Webseite mit Navigation und einem Artikel-Inhaltselement

- Nun fehlt noch der Fußbereich. Diesen finden wir unter dem Element **Footers**. Wir ziehen den Eintrag **Footer Basic** unter dem Inhaltselement in den Entwurf und erhalten

die Ansicht aus Bild 5. Damit haben wir im Prinzip schon den Aufbau des Beispiels nachgebaut, das auch in Visual Studio verwendet wird.

Bild 5: Navigation, Inhalt und Footer – fertig ist die Webseite.

Anpassen des Entwurfs

Nun wollen Sie die Elemente natürlich nicht wie hier zu sehen übernehmen, sondern gegebenenfalls diverse Änderungen vornehmen – beispielsweise am Inhalt, an den Farben von Text und Hintergrund, an der Schriftart und an anderen Elementen der Benutzeroberfläche. Vielleicht möchten Sie auch noch Elemente entfernen oder hinzufügen.

Navigation bearbeiten

Die Navigation besteht aktuell aus dem Namen der Seite, einigen Navigationslinks und einem Dropdown-Menü. Wenn Sie zum Beispiel den Namen der Seite anpassen möchten, klicken Sie auf das Element, klicken auf das Symbol zum Bearbeiten und ändern dann den Inhalt des Elements. Auf die gleiche Art passen Sie die Beschriftungen der Navigationslinks an. Die Bearbeitung der Beschriftungen können Sie allerdings auch ganz einfach per Doppelklick auf das jeweilige Element aktivieren (siehe Bild 6).

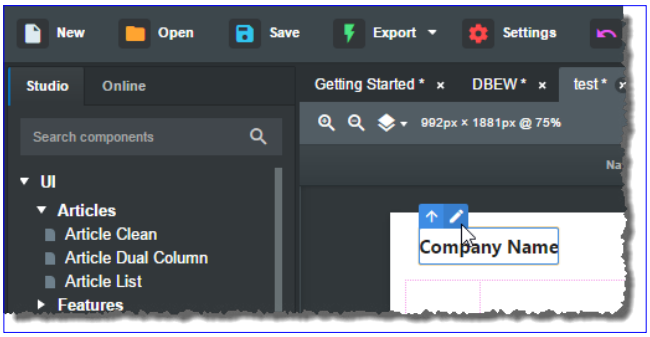

Bild 6: Anpassen der Beschriftung

Wenn Sie Navigationseinträge hinzufügen oder entfernen oder als aktiv oder inaktiv markieren möchten, erledigen Sie das über die Schaltflächen über dem Entwurf (siehe Bild 7). Im Bild haben wir etwa durch Anklicken der Schaltfläche **After** ein neues Element namens **Nav Item** rechts vom Element **First Item** hinzugefügt.

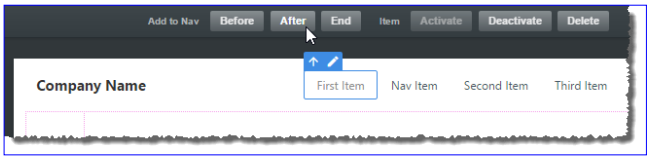

Bild 7: Bearbeiten der Navigationsleiste

Übergeordnetes Element markieren

Manchmal klicken Sie ein Element an, um es beispielsweise zu löschen, und der dann um das Element gezeichnete Rahmen bietet nur die beiden Einträge mit dem **Pfeil nach oben**- und dem **Bearbeiten**-Eintrag an. Der Grund, warum Sie das Element zu diesem Zeitpunkt nicht löschen können, ist, dass aktuell das Beschriftungs-Element markiert ist, das zu dem Objekt gehört. Zum Löschen klicken Sie einmal auf die **Select Parent**-Schaltfläche mit dem Pfeil nach oben, dann finden Sie in der Regel auch die Befehle zur weiteren Bearbeitung wie etwa die **Löschen**-Schaltfläche.

Navigationselemente bearbeiten

Wenn Sie nun die Navigationselemente bearbeiten wollen, können Sie neue Elemente an der gewünschten Stelle hinzufügen, die vorhandenen Elemente verschieben, die Beschriftungen anpassen oder Elemente löschen.

Wenn Sie das Dropdown in der Navigationsleiste bearbeiten wollen, klicken Sie dieses zunächst an. Dann finden Sie oben links neben der Beschriftung **Dropdown** eine Schaltfläche namens **Open**. Damit klappen Sie das Dropdown-Element auf, um seine Einträge zu bearbeiten.

Stellt sich noch die Frage, wie Sie ein weiteres Dropdown zur Navigationsleiste hinzufügen. Wie Sie einfache Button-Elemente zur Navigationsleiste hinzufügen, haben Sie ja schon erfahren. Und auch das Hinzufügen von weiteren Dropdowns gelingt ganz einfach: Diese finden Sie nämlich in der Toolbox unter dem Eintrag **Controls**. Das Dropdown-Element ziehen Sie dann einfach von dort an die gewünschte Stelle in der Navigationsleiste.

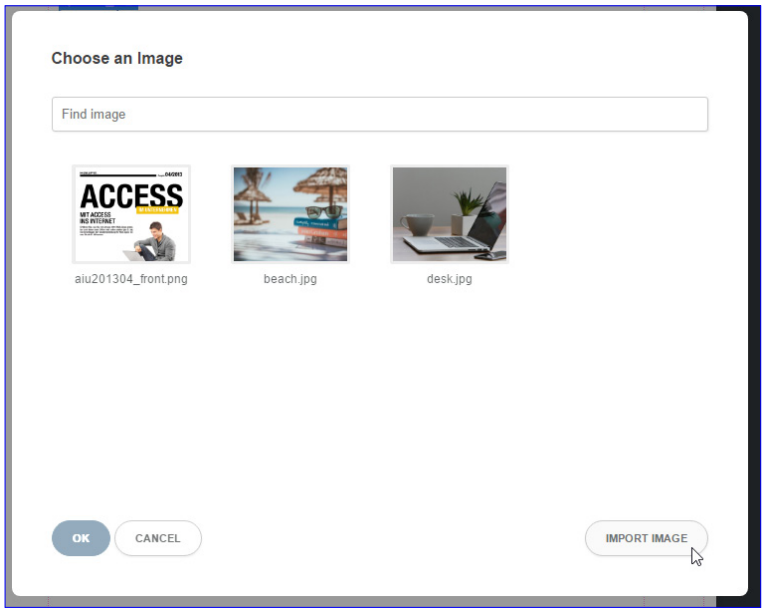

Bild 8: Hinzufügen einer Bilddatei

Inhalt bearbeiten

Den Inhalt können Sie zunächst durch Eingabe der gewünschten Texte bearbeiten. Wenn Sie Texte in verschiedenen Formatierungen benötigen, können Sie jeweils neue Absätze aus dem Bereich **Text** der Toolbox an die gewünschte Stelle ziehen – also etwa Überschriften (**Heading**) oder einfache Absätze (**Paragraph**).

Sie können auch beispielsweise neue Bilder zum Inhalt hinzufügen. Dazu ziehen Sie das Element **ImageIImage** an die gewünschte Stelle im Inhalt. Gleich nach dem Hinzufügen erscheint zunächst ein viereckiger Platzhalter. Per Doppelklick auf diesen öffnen Sie den Dialog **Choose an Image**, der die Möglichkeit bietet, bereits vorhandene Bilder einzufügen oder über die Schaltfläche **Import Image** neue Bilder zur Vorlage hinzuzufügen (siehe Bild 8). Klicken Sie dann eines der angezeigten Bilder an, erscheint dieses im Inhalt.

Fußzeile bearbeiten

Wenn Sie zum Beispiel keine Links auf Twitter, Facebook und Co. im Footer anzeigen möchten, können Sie diese einfach entfernen. Die dort angezeigten Elemente bearbeiten Sie grundsätzlich genauso wie die in der Navigationsleiste. Auch den Namen des Unternehmens ganz unten passen Sie beispielsweise per Doppelklick und Ändern des Inhalts an.

Mit Rastern arbeiten

Wenn Sie kein schnödes einzeiliges Layout wünschen, können Sie über die Grid-Elemente auch ein Raster definieren, das Ihre Inhalte anzeigt. Dazu ziehen Sie zunächst ein **Row**-Element aus dem Bereich **Grid** der Toolbox an die gewünschte Stelle. Diese trägt dann die Beschriftung **Empty Row** (siehe Bild 9).

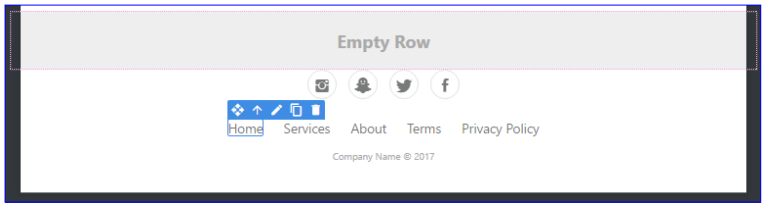

Bild 9: Hinzufügen eines **Row**-Elements als Basis für ein Grid

Dem neuen Element können Sie nun **Column**-Elemente hinzufügen. Ziehen Sie eines aus der Toolbox in ein **Empty Row**-Element, wird dieses mit dem **Empty Column**-Element gefüllt. Diesem können Sie nun ein weiteres **Column**-Element hinzufügen. Um festzulegen, ob diese links oder rechts vom bereits vorhandenen **Column**-Element eingefügt wird, lassen Sie es erst dann fallen, wenn links oder rechts nach dem Überfahren mit der Maus ein entsprechender Pfeil auftaucht. Experimentieren Sie einfach damit herum. Sie können damit dann beispielsweise solche Konstrukte wie in Bild 10 erzeugen.

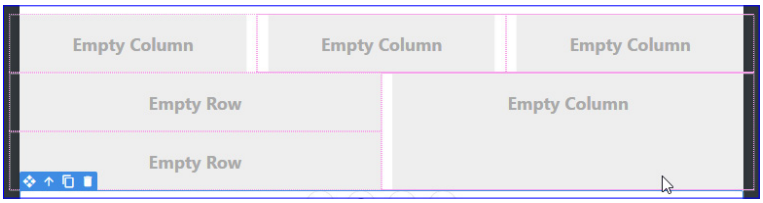

Bild 10: Zeilen und Spalten als Grid

Überführen des Entwurfs in Razor Pages

Nun wird es interessant: Wir haben jetzt eine Seite mit Seitennavigation, Inhalt und Fußzeile entworfen und eventuell bereits mit Inhalten gefüllt. Nun wollen wir die so erzeugten Inhalte in eine frisch erzeugte Webanwendung in Visual Studio auf Basis der Vorlage **ASP.NET Core Webanwendung** mit dem Typ **Webanwendung** überführen.

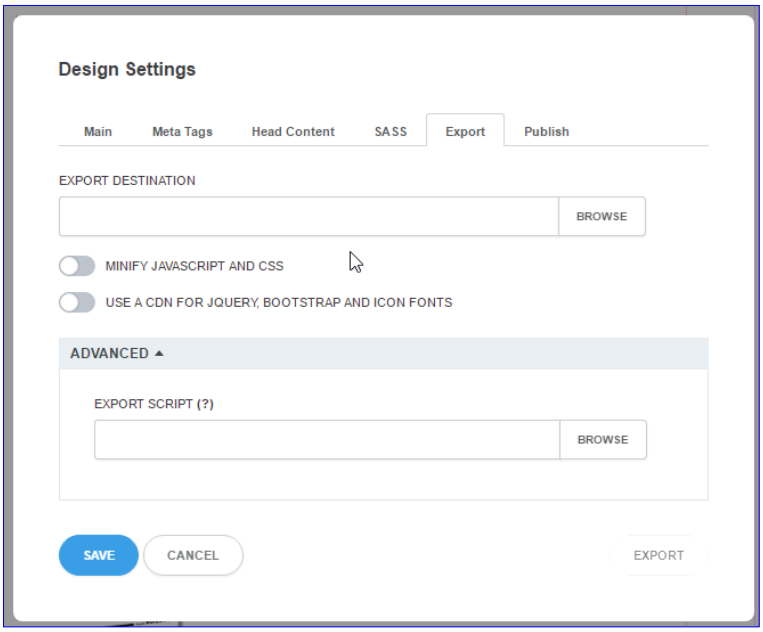

Bild 11: Export-Einstellungen

Dazu benötigen wir zuerst einmal den Code für die Elemente des frisch erzeugten Entwurfs. Bootstrap Studio bietet dazu verschiedene Export-Funktionen. Diese starten wir über den Menüeintrag **Export**. Es erscheint der Dialog aus Bild 11, dessen Einstellungen wir übernehmen können. Wichtig ist die Schaltfläche **Save**, mit der wir die Einstellungen speichern können. Mit der Schaltfläche **Export** exportieren Sie schließlich die Webseite – aber erst, nachdem Sie auf der Registerseite **Export** das Zielverzeichnis ausgewählt haben. Dies exportiert dann den HTML-Code in eine Datei namens **index.html** und die übrigen Elemente wie die **.css**-Dateien et cetera einen Ordner namens **assets**.

Was machen wir nun damit? Wie wir bereits im Beitrag **ASP.NET Core-Anwendung anpassen** beschrieben haben, teilen sich die eigentlich HTML-Daten auf die beiden Dateien **_Layout.cshtml** und in die jeweilige Datei mit dem anzuzeigenden Inhalt auf, also beispielsweise **Index.cshtml** oder **About.cshtml**. Genau auf diese Dateien müssen wir nun auch den in die Datei **index.html** exportieren HTML-Code aufteilen.

Dabei können wir den Inhalt nicht einfach aufteilen und in die beiden Dateien **_Layout.cshtml** und **Index.cshtml** schreiben, sondern wir müssen zuvor noch schauen, welche dynamischen Elemente – sprich Markup – sich in der aktuellen Version dieser Dateien befindet. So enthält die Datei **_Layout.cshtml** ja beispielsweise ein Element, für das später der Inhalt von **Index.cshtml** eingebettet wird.

Testweises Anzeigen der Datei

Übrigens brauchen wir zur Anzeige der Datei index.html kein Visual Studio und auch kein ASP.NET Core – es handelt sich um eine einfache HTML-Datei, die ein paar Erweiterungen nutzt, die aber ohne serverseitigen Code auskommen. Klicken Sie doppelt auf **index.html** und öffnen diese, erscheint diese wie in Bild 12 im Webbrowser.

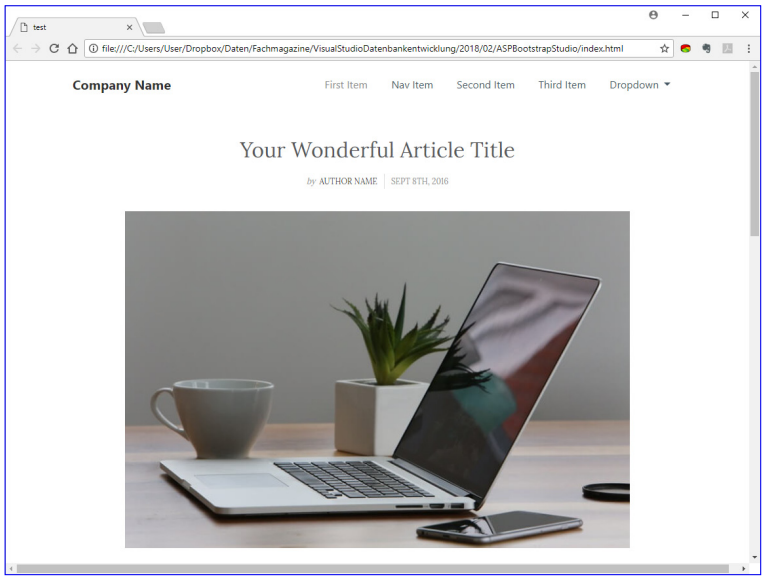

Bild 12: Die Datei **index.html** im Webbrowser

Übertragen der HTML-Teile in eine ASP.NET Core-Anwendung

Für das Übertragen der mit Bootstrap Studio erzeugten Dateien in eine ASP.NET Core-Anwendung sind grob zwei Schritte nötig:

- Kopieren des Verzeichnisses **assets** in den Ordner **wwwroot** des Projekt-Explorers
- Aufteilen und Einfügen von Navigations-, Inhalts- und Fußzeilenelementen in die beiden Dateien **_Layout.cshtml** und **Index.cshtml**

Den ersten Schritt müssen wir nicht weiter erläutern, den zweiten Schritt schon. Hier beginnen wir damit, die Datei **index.html** in Visual Studio zu öffnen.

Wenn wir uns die Datei anschauen, wird schnell deutlich, dass wir die drei Teile Kopf plus Navigation, Inhalt und Fußbereich leicht separieren können (gekürzte Form, siehe Bild 13). Wichtig ist nur, dass wir erkennen, welche Teile der Datei immer erscheinen sollen, also unabhängig vom angezeigten Inhalt, und welcher Teil abhängig vom jeweils gewählten Inhalt ausgetauscht wird.

Bild 13: Quellcode der Datei **index.html**

E-Mails per Sendgrid verschicken

Wenn Sie am Desktop arbeiten, können Sie E-Mails einfach per Outlook verschicken. Das gelingt auch ferngesteuert etwa von einer Desktop-Anwendung auf Basis von .NET aus. Wenn Sie jedoch Mails sicher über eine Webanwendung verschicken wollen, wird es interessant. Sollten Sie keinen eigenen Mailserver auf dem Internetserver betreiben, können Sie auf einen Dienstleister ausweichen, der Ihnen eine entsprechende Schnittstelle anbietet. Microsoft empfiehlt hier den Anbieter Sendgrid. Wir zeigen, wie Sie sich registrieren und für den ASP-gesteuerten Versand von E-Mails vorbereiten. Der Clou: Bis zu einer bestimmten Menge von E-Mails ist der Service kostenlos.

Sendgrid

Sendgrid ist ein US-amerikanischer Anbieter, der erstmal eines für Sie erledigt: das Versenden von E-Mails. Dafür gibt es verschiedene Pakete, die sowohl die Menge der zu versendenden E-Mails als auch die Anzahl der zu speichernden Kontakte festlegen (siehe Bild 1). Das günstigste Paket heißt **Free**, was sich auf den monatlichen Preis bezieht – es ist nämlich kostenlos. Je mehr E-Mails verschickt und Adressen verwaltet werden sollen, desto höher sind die monatlichen Kosten. Für Testzwecke reicht uns jedoch das Paket **Free** völlig aus – hier können Sie 40.000 E-Mails in den ersten 30 Tagen und danach täglich 100 E-Mails verwenden.

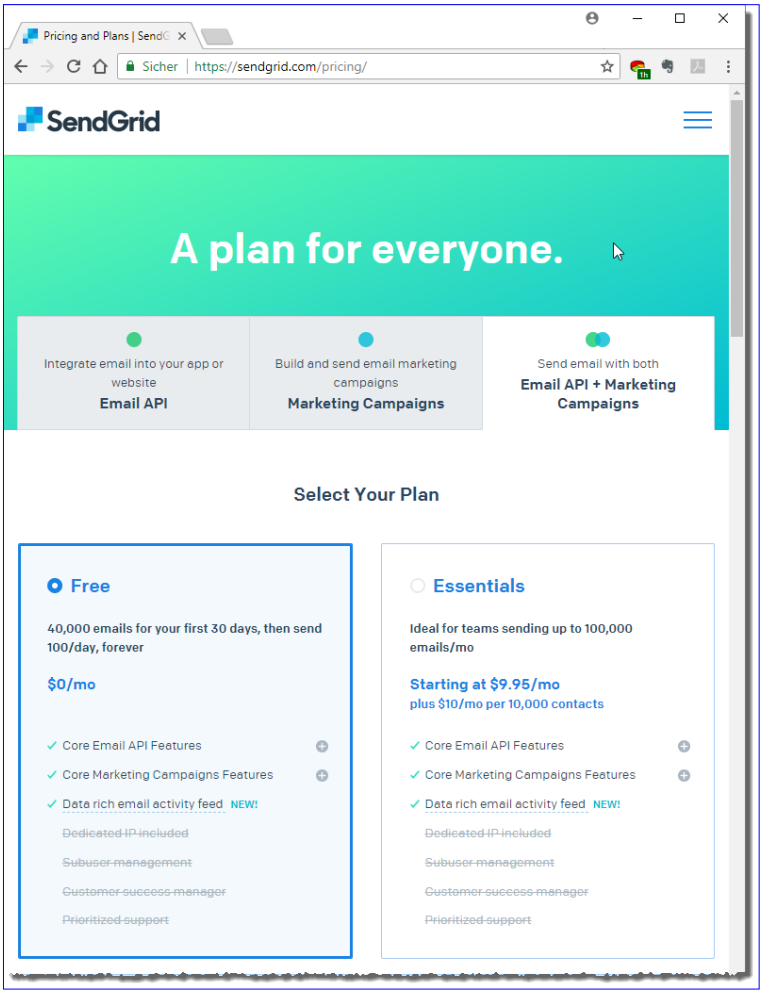

Bild 1: Der E-Mail-Versender **Sendgrid**

Für eine Webseite, die ein paar Bestätigungsmails und ein paar Kontaktanfragen versendet, reicht das locker aus. Auch das nächstgrößere Paket, das bereits den Versand von 100.000 E-Mails pro Monat erlaubt, kostet nur 9,95 $ pro Monat.

E-Mails und mehr

Neben dem über verschiedene Schnittstellen steuerbaren Versand von E-Mail liefert der Anbieter noch weitere Dienstleistungen – so können Sie etwa Newsletter darüber versenden und die dazu notwendigen Kontakte verwalten. Dabei sind typische Funktionen wie etwa das Bereitstellen eines Abmeldelink für den Newsletter und automatisches Austragen aus der Empfängerliste bereits implementiert. Wer etwa eine Newsletter-Anmeldung per Double-Opt-In realisieren möchte, muss allerdings eine eigene Funktion dafür implementieren.

Kostenloses Paket

Wir wählen für den Beginn das Paket **Free** aus. Aktuell wollen wir diesen Dienstleister dafür nutzen, die in dem Artikel **Authentifizierung unter ASP.NET Core** benötigten Bestätigungsmails zu versenden.

Nach der Auswahl des gewünschten Pakets und dem Anklicken des Links **Try for free** landen Sie auf der Registrierungsseite. Hier geben Sie den Benutzernamen, das Kennwort und Ihre E-Mail-Adresse an. Nach dem Bestätigen der obligatorischen Optionen geht es dann mit einem Klick auf **Create Account** weiter (siehe Bild 2). Im Bereich rechts können Sie die aktuell anfallenden Kosten einsehen.

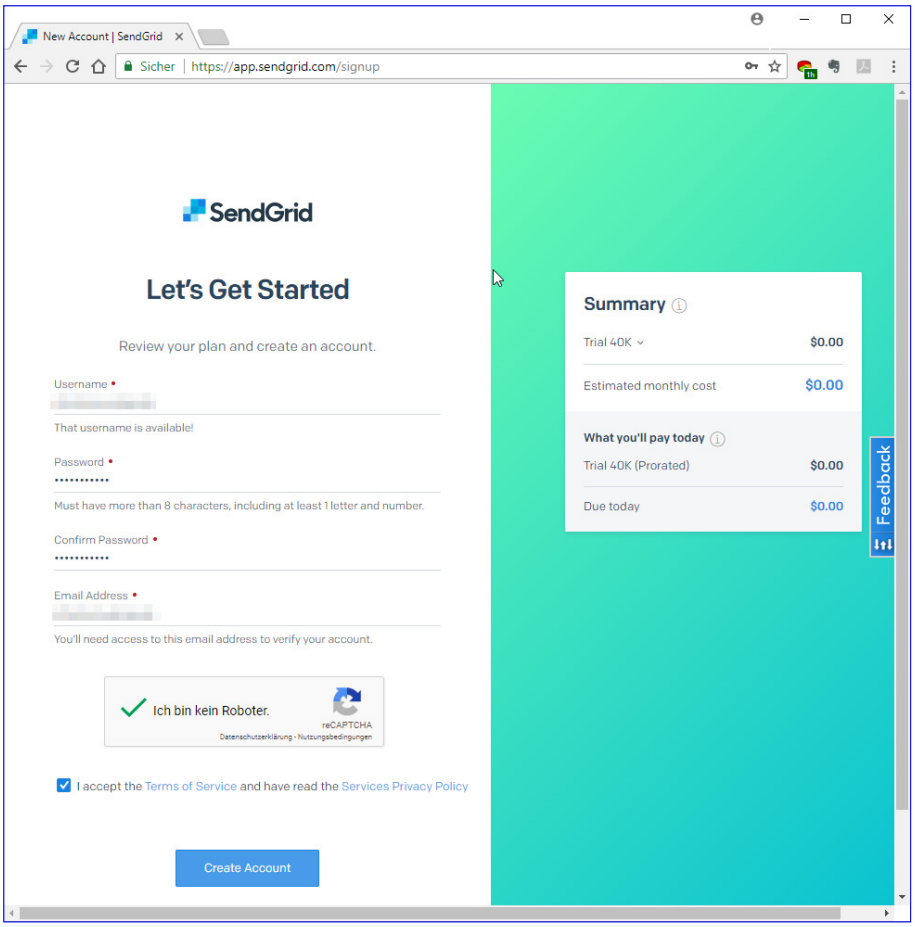

Bild 2: Beispielfenster zum Versenden von E-Mails

Die folgende Seite fragt dann noch einige persönliche Daten ab, bevor es nach einem Klick auf **Get Startet!** losgeht. Auf der Willkommen-Seite können wir nun auswählen, auf welche Weise wir mit Sendgrid arbeiten wollen. Wir möchten Mails per Web API verschicken, also wählen wir den passenden **Start**-Button (siehe Bild 3).

Der nächste Dialog fragt nach einer Setup-Methode. Hier haben Sie die Auswahl zwischen **Web API** und **SMTP Relay**. Wir wählen hier den Eintrag **Web API**, da wir den Versand per C#

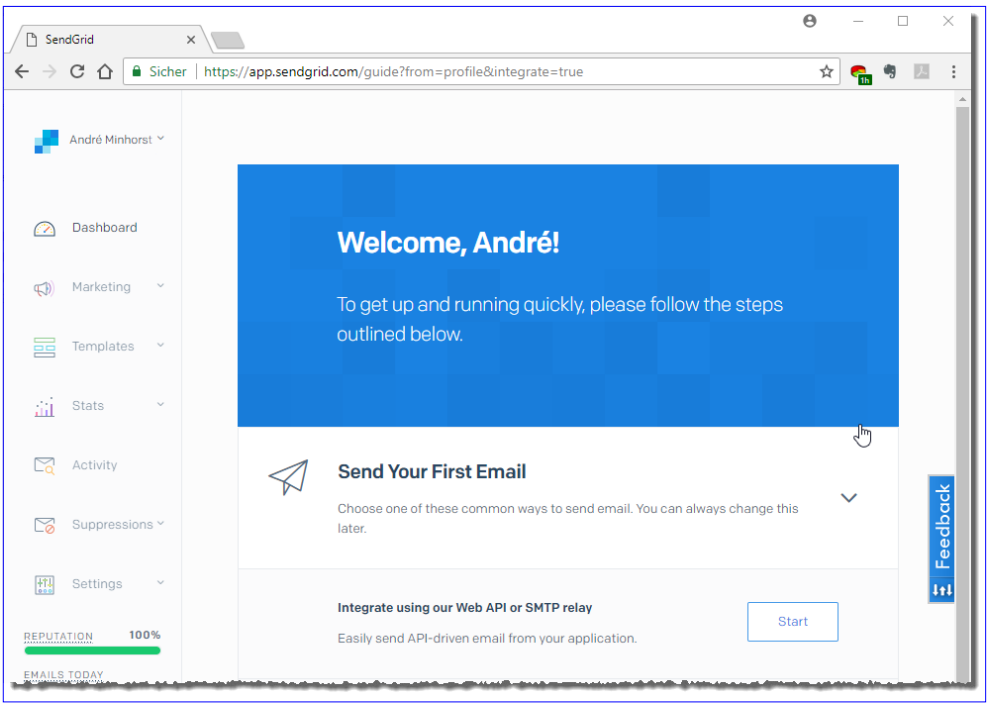

Bild 3: Auswahl des benötigten Services

programmieren wollen. Noch eine Seite später wählen Sie die Sprache aus, mit der Sie auf die Web API zugreifen wollen. Hier fällt die Wahl auf **C#**.

Danach müssen wir wieder arbeiten: Als Erstes prüfen wir die Voraussetzungen, die hier mit .NET 4.5.2 angegeben sind. Diese müssen Sie erfüllen. Als Zweites legen wir einen API-Key an, der einen sicheren Zugriff von der Web-Anwendung auf die Schnittstelle von Sendgrid erlaubt. Dazu geben Sie unter **Create an API key** einen Namen für den Key ein, beispielsweise **APIKey1**, und klicken auf **Create Key**. Dann erhalten Sie den Key wie in Bild 4. Den dritten Schritt mit dem Titel **Create an environment variable** überspringen wir zunächst.

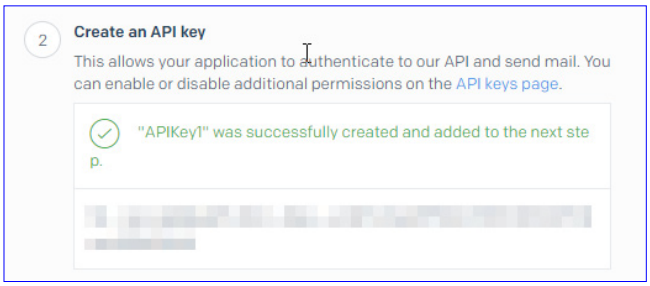

Bild 4: Erfolgreich angelegter API-Key

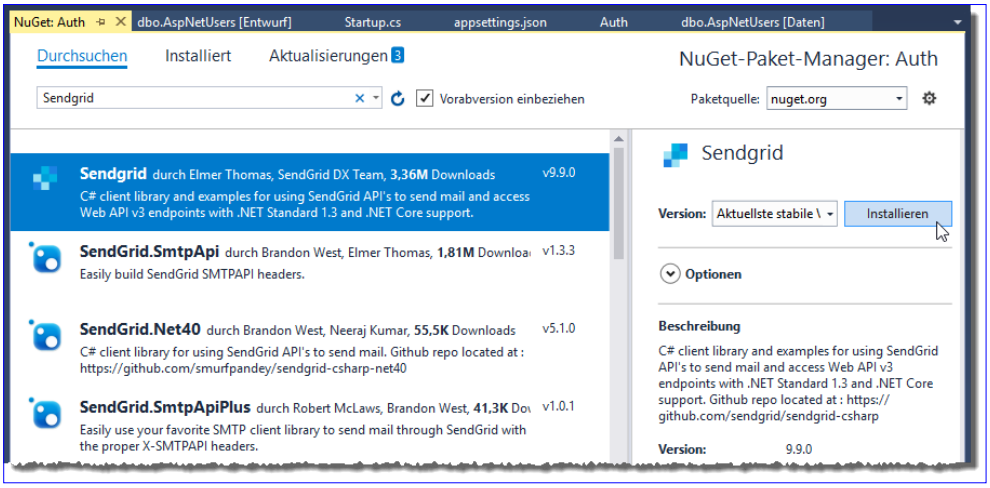

Bild 5: Installieren des NuGet-Pakets **Sendgrid**

Projekt anlegen und fortfahren

Dafür fahren wir mit dem vierten Schritt fort, für den Sie zunächst ein neues Projekt erstellen. Zu Beispielzwecken legen wir eine einfache C#-Konsolenanwendung an. Bevor wir die Anwendung nun mit dem Beispielcode versehen, müssen wir noch das Sendgrid-Paket hinzufügen. Dann folgt der vierte Schritt auf der Sendgrid-Webseite: **Install the package**. Ziel dieses Schritts ist die Installation des NuGet-Pakets für den Zugriff auf **Sendgrid**. Dazu starten Sie den NuGet-Dialog über den Kontextmenü-Eintrag **NuGet-Pakete verwalten...** des Projekts im Projektmappen-Explorer. Hier wechseln Sie zum Bereich **Durchsuchen** und suchen nach dem Eintrag **Sendgrid**. Diesen markieren Sie dann und klicken im rechten Bereich auf die Schaltfläche **Installieren** (siehe Bild 5).

Die Sendgrid-Bibliothek finden Sie dann beispielsweise im Projektmappen-Explorer unter **<Projektname>\Abhängigkeiten\NuGet** (siehe Bild 6).

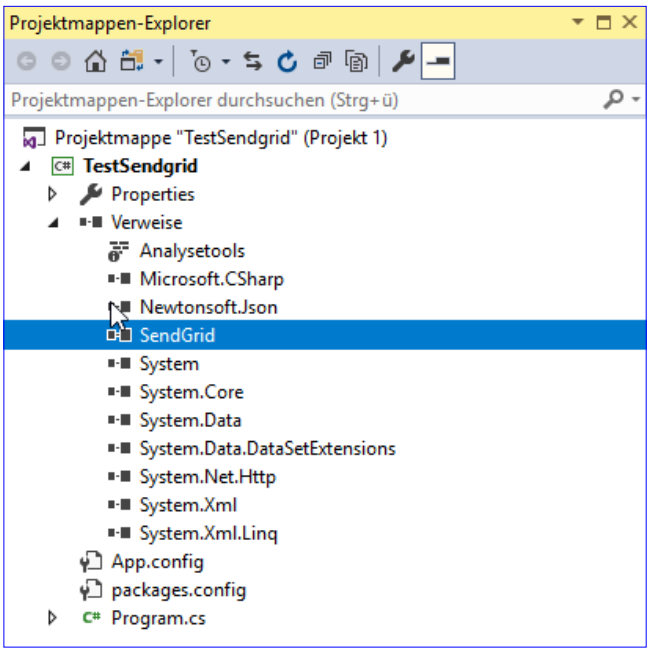

Bild 6: Die Sendgrid-Erweiterung in den Verweisen

Test-E-Mail versenden

Um eine Test-E-Mail zu versenden, fügen wir der Klasse **Program** den folgenden Code hinzu. Für die darin verwenden Objekttypen benötigen wir allerdings noch ein paar using-Anweisungen, die wir im Kopf der Klassendatei unterbringen:

```
using System.Threading.Tasks;
using Sendgrid;
using Sendgrid.Helpers.Mail;
```

Die Methode **Main** ruft dabei den asynchronen Task **SendMail** auf. Das Erstellen einer E-Mail würden wir normalerweise eher eine ASP.NET-Anwendung zuschreiben, daher programmieren wir die Methode, welche die E-Mail zusammenstellt und versendet als asynchronen Task – und es gibt noch einen weiteren Grund dafür, den wir weiter unten erläutern. Die asynchrone Methode erkennen Sie am Schlüsselwort **async** und am Methodentyp **Task**. Damit der Aufruf dennoch wartet, bis die asynchrone Methode durchgeführt wurde, rufen wir diese mit der **Wait**-Methode auf:

```
class Program {
    private static void Main() {
        SendMail().Wait();
    }
    static async Task SendMail() {
```

Den soeben ermittelten API-Key schreiben wir in die **string**-Variable **apiKey**. Normalerweise würden wir diesen an anderer Stelle speichern, wo er auch zur Laufzeit geändert werden kann, aber für Testzwecke wählen wir die einfache Möglichkeit:

```
        string apiKey = "SG.HT7pvIFvQJWK12BMBcoEq-yg.xXWtoVKw3NlpYBld7nrVY3429xKomFUL1ZUcGHe2n0";
```

Dann erstellen wir ein neues Objekt des Typs **SendgridClient** und übergeben dieser beim Erstellen den API-Key aus **apiKey**:

```
        SendgridClient SendgridClient = new SendgridClient(apiKey);
```

Dann legen wir ein neues Objekt des Typs **EmailAddress** namens **from** an und weisen diesem die E-Mail-Adresse und den Namen des Absenders zu:

```
        EmailAddress from = new EmailAddress("andre@minhorst.com", "André Minhorst");
```

Die Variablen **subject** nimmt den Betreff auf:

```
        string subject = "Beispiel per Console-App";
```

Die Empfänger-Adresse landet in der Variablen **to**:

```
        EmailAddress to = new EmailAddress("andre@minhorst.com", "André Minhorst");
```

Razor Pages: Von Seite zu Seite

Wenn Sie eine Webanwendung zur Verwaltung von Daten programmieren, wollen Sie Daten anzeigen, eingeben, löschen und so weiter. Das gelingt nicht ohne Navigation zwischen den verschiedenen Seiten. Die Hauptseiten sind über die Navigationsleisten erreichbar, aber wenn Sie etwa einen bestimmten Datensatz anzeigen, Daten von einer Seite zur nächsten übermitteln oder nach dem Löschen eines Datensatzes wieder die Übersicht einblenden wollen, benötigen Sie entsprechende Techniken. Dieser Artikel zeigt, wie Sie mit Links, Senden von Formularen und anderen Methoden von Seite zu Seite gelangen und die gewünschten Daten übermitteln. Dabei spielen auch die Ereignis-Handler einer Seite eine Rolle.

Einfache Links

Einen einfachen Link erstellen Sie in herkömmlichen HTML-Seiten mit dem **a**-Element. In Razor Pages verwenden Sie ein anderes Attribut, um die Zielseite anzugeben – nämlich **asp-page**. Wenn Sie etwa von der **Index.cshtml**-Seite im Verzeichnis **Pages** des Projekts auf die **About**-Seite verlinken wollen, können Sie das etwa mit dem folgenden Code erledigen:

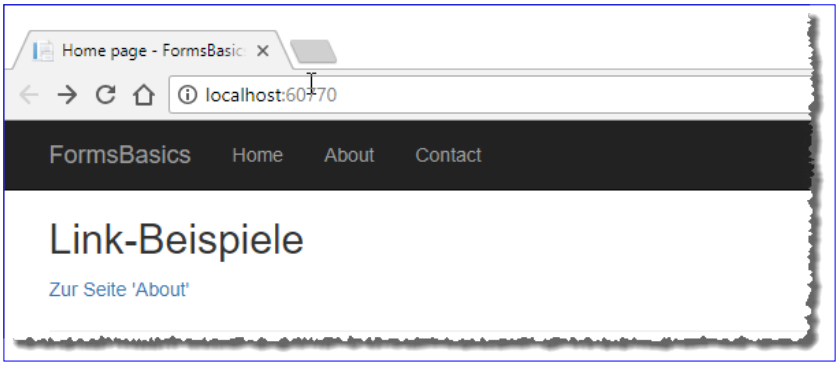

Bild 1: Einfacher Link

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}
<h2>Link-Beispiele</h2>
<p><a asp-page="About">Zur Seite 'About'</a></p>
```

Das Ergebnis sieht dann etwa wie in Bild 1 aus. Da sich die Seite **About.cshtml** im gleichen Verzeichnis wie die Seite **Index.cshtml** befindet, reicht die Angabe des Namens der Seite ohne die Dateiendung **.cshtml**. Ein Klick auf diesen Link zeigt dann die Seite **About.cshtml** im Browser an.

Handler-Methoden

Unter ASP.NET Core/Razor Pages gibt es verschiedene Handler-Methoden. Diese heißen beispielsweise wie folgt:

- **OnPost**

- **OnGet**
- **OnPostAsync**
- **OnGetAsync**

Die Code behind-Datei einer Razor Page enthält normalerweise die folgende Handler-Methode:

```
public void OnGet() {

}
```

Diese wird gleich beim Laden der Seite auf der Serverseite ausgelöst. Dies können wir prüfen, indem wir der Methode einen Haltepunkt hinzufügen und das Projekt starten. Bevor die Seite im Browser erscheint, hält die Methode an der Stelle mit dem Haltepunkt an. Auch wenn Sie die Seite mit der Taste **F5** aktualisieren, wird die Methode wieder ausgelöst.

In der durch unsere Seite **Index.cshtml** verlinkten Seite **About.cshtml** haben wir die gleiche Prozedur angelegt und mit einem Haltepunkt versehen. Klicken wir zur Laufzeit auf den Link, löst dies die Methode **OnGet** dieser Seite aus.

Methode OnPost auslösen

Nun wollen wir eine zweite Handler-Methode namens **OnPost** hinzufügen, die genau gleich aufgebaut ist wie **OnGet**. Auch diese fügen wir wieder beiden Pages hinzu, also **Index.cshtml** und **About.cshtml**:

```
public void OnPost() {

}
```

Um diese zu testen, benötigen wir auf der Seite **Index.cshtml** ein Formular. Dieses erhält die Methode **post** für das Attribut **method** und eine Schaltfläche zum Absenden des Formulars:

```
<form method="post">
    <button class="btn btn-default">Zur gleichen Seite.</button>
</form>
```

Wenn wir nun auf diese Schaltfläche klicken, wird die Methode **OnPost** der aufgerufenen Seite **Index.cshtml** aufgerufen – diese wird dann auch erneut angezeigt.

Nun fügen wir eine weitere Schaltfläche in einem neuen Formular hinzu, welche die Seite **About.cshtml** aufrufen soll. Standardmäßig wird beim Absenden eines Formulars immer die Seite aufgerufen, auf der sich das Formular befindet. Um eine andere Seite wie etwa **About.cshtml** aufzurufen, fügen Sie dem **button**-Element das Attribut **asp-page** mit dem Namen der aufzurufenden Datei hinzu:

```
<form method="post">
    <button class="btn btn-default" asp-page="About">Zur Seite 'About'.</button>
</form>
```

Dies öffnet die Seite **About.cshtml** und führt dort die Handler-Methode **OnPost** aus.

Mehrere Formulare für die gleiche Seite

Im obigen Fall haben wir nun zwei Formulare, die allerdings verschiedene Seiten aufrufen. Wenn Sie jedoch verschiedene Formulare auf einer Seite nutzen, die alle die gleiche Seite aufrufen – beispielsweise die aktuelle Seite – gelingt dies natürlich. Allerdings wird immer die gleiche Handler-Methode aufgerufen, standardmäßig **OnPost**. Wenn Sie mehrere Formulare nutzen wollen, die zur Ausführung verschiedener **OnPost**-Methoden führen sollen, legen Sie etwa die folgenden Formulare an:

```
<h3>Zwei Formulare zu verschiedenen OnPost-Methoden</h3>
<form method="post" asp-page-handler="Eins">
    <button class="btn btn-default">Zur Methode OnPostEins.</button>
</form>
<form method="post" asp-page-handler="Zwei">
    <button class="btn btn-default" >Zur Methode OnPostZwei.</button>
</form>
```

Die Seite sieht mittlerweile wie in Bild 2 aus. Wenn Sie nun nacheinander auf die beiden Schaltflächen **Zur Methode OnPostEins** und **Zur Methode OnPostZwei** anklicken, wird nun beide Male die Methode **OnPost** des Formulars **Index.cshtml** ausgeführt. Es hat sich also erstmal nichts geändert.

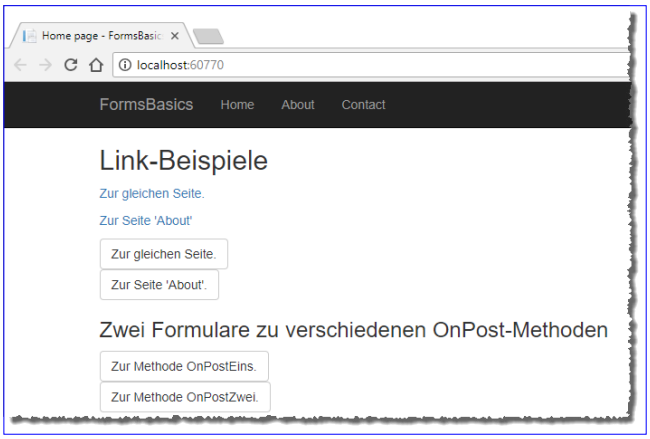

Bild 2: Beispiel-Links und -Formulare

Um tatsächlich zwei verschiedene Methoden aufzurufen, müssen Sie diese nach einem bestimmten Schema anlegen. Dieses besteht aus dem eigentlichen Namen der Handler-Methode, also **OnPost**, und dem mit **asp-page-handler** angegebenen Ausdruck. Die beiden von den beiden Formularen adressierten Handler-Methoden heißen also **OnPostEins** und **OnPostZwei** und sehen wie folgt aus:

```
public void OnPostEins() {

}

public void OnPostZwei() {

}
```

Beim Anklicken der beiden Schaltflächen werden nun diese beiden Handler-Methoden aufgerufen. Diese Technik können Sie also beispielsweise nutzen, wenn Sie die angezeigten Daten einer Datenbank bearbeiten, löschen oder anzeigen wollen.

Wie wird der **Page**-Handler nun übermittelt? Wie Bild 3 zeigt, über die Parameterliste beim Aufruf der Seite.

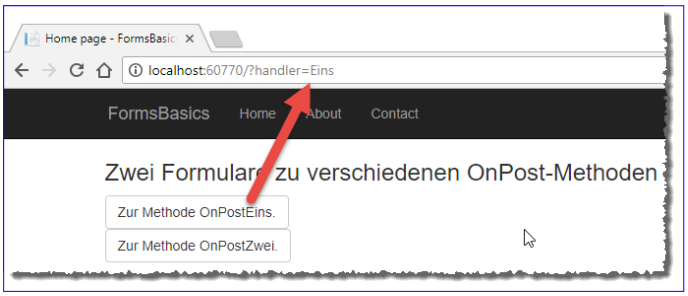

Bild 3: **Page**-Handler als Parameter beim Seitenaufruf

Ein Formular mit mehreren Schaltflächen

Sie können auch zwei verschiedene Aktionen mit zwei Schaltflächen in einem **form**-Element unterbringen. Dazu müssen Sie lediglich das Attribut **asp-page-handler** vom **form**-Element in die **button**-Elemente übertragen.

Folgendes Beispiel funktioniert genauso wie das vorherige:

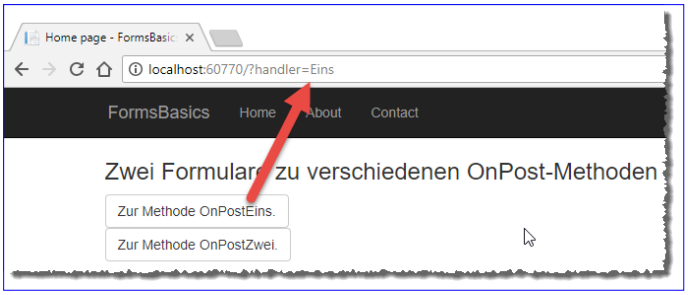

Bild 4: Kein Parameter mit **Page**-Handler bei der **Get**-Methode

```
<h3>Zwei Schaltflächen eines Formulars zu verschiedenen OnPost-Methoden</h3>
<form method="post">
    <button class="btn btn-default" asp-page-handler="Eins">Zur Methode OnPostEins.</button>
    <button class="btn btn-default" asp-page-handler="Zwei">Zur Methode OnPostZwei.</button>
</form>
```

Formulare mit Get

Wir sehen also, dass bei Formularen mit der Post-Methode die Parameter – wie hier zum Beispiel der Wert des **Page**-Handlers – per Parameter an die URL angehängt werden. In manchen Fällen macht es Sinn, einen alternativen Ansatz zu wählen, nämlich die **Get**-Methode. Der einzige Unterschied zur **Post**-Methode im HTML-Code ist, dass Sie statt dem Wert **post** den Wert **get** für das Attribut **method** des Formulars festlegen. Dazu legen wir zwei weitere Formulare an:

```
<h3>Zwei Formulare zu verschiedenen OnGet-Methoden</h3>
<form method="get" asp-page-handler="Eins">
    <button class="btn btn-default">Zur Methode OnGetEins.</button>
</form>
<form method="get" asp-page-handler="Zwei">
    <button class="btn btn-default">Zur Methode OnGetZwei.</button>
</form>
```

Starten Sie nun die Anwendung und klicken diese beiden Schaltflächen an, lösen Sie jeweils die Methode **OnGet** der Code behind-Klasse der Seite **Index.cshtml** aus. Allerdings sehen wir auch, dass der Page-Handler hier nicht als Parameter übergeben wird (siehe Bild 4). Legen wir also nun die beiden passenden Handler-Methoden namens **OnGetEins** und **OnGetZwei** an:

```
public void OnGetEins() {

}
public void OnGetZwei() {

}
```

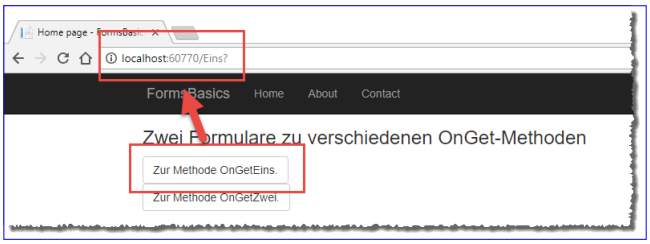

Bild 5: Auslösen einer Get-Handler-Methode

Normalerweise würde man annehmen, dass nun die beiden Methoden **OnGetEins** und **OnGetZwei** ausgelöst werden, wenn der Benutzer auf eine der beiden neuen Schaltflächen klickt. Dies ist allerdings nicht der Fall: Das Anklicken der Schaltflächen löst nach wie vor die Handler-Methode **OnGet** aus. Damit dies dennoch funktioniert, müssen wir auf der Seite **Index.cshtml** noch die Direktive **@page** anpassen:

```
@page "{Handler?}"
```

Wenn Sie nun etwa auf die Schaltfläche des Formulars mit **method="get"** und **asp-page-handler="Eins"** klicken, wird erstens die Methode **OnGetEins** aufgerufen. Zweitens wird der Wert des Attributs **asp-page-handler** in die URL integriert (siehe Bild 5).

Parameter für Handler-Methoden festlegen und verarbeiten

Wenn Sie eine Handler-Methode aufrufen, wollen Sie gegebenenfalls Informationen übergeben – wie beispielsweise die ID eines zu bearbeitenden oder zu öffnenden Datensatzes. In diesem Fall legen wir beispielsweise drei **form**-Elemente an, welche alle drei jeweils eine Schaltfläche zum Aufrufen des fiktiven Datensatzes enthalten und zusätzlich ein **input**-Element, das durch den Wert **hidden** für das Attribut **type** ausgeblendet erscheint und den Namen und den Wert des zu übermittelnden Parameters enthält:

```
<form method="post" asp-page-handler="Anzeigen">
    <button class="btn btn-default">Datensatz mit ID = 1 anzeigen</button>
    <input type="hidden" name="id" value="1" />
</form>
<form method="post" asp-page-handler="Anzeigen">
    <button class="btn btn-default">Datensatz mit ID = 2 anzeigen</button>
    <input type="hidden" name="id" value="2" />
</form>
<form method="post" asp-page-handler="Anzeigen">
    <button class="btn btn-default">Datensatz mit ID = 3 anzeigen</button>
    <input type="hidden" name="id" value="3" />
</form>
```

Diesen Elementen hängen wir schließlich noch einen als Überschrift 3 formatierten Absatz an, der die Eigenschaft **Meldung** von **@Model** liefert. **@Model** ist ein Platzhalter für die unter **@model** ganz oben angegebene Klasse. Damit können Sie auf öffentliche Eigenschaften der Code behind-Klasse zugreifen:

Kontaktformular unter ASP.NET Core

Wer Anwendungen für den Desktop entwickelt hat, kennt die Vorgehensweise: Bei einem Klick auf OK werden die eingegebenen Werte in einer Ereignisprozedur ausgewertet, gespeichert oder anderen Schritten unterzogen. Bei Internetanwendungen ist das aufgrund der Eigenarten der Browser etwas anders. Hier klicken Sie zwar nach der Eingabe von Daten auch auf eine Schaltfläche, um die Daten zu verarbeiten, aber es gibt dort keine Ereignisprozeduren im klassischen Sinne. Wie die Datenverarbeitung hier geschieht, zeigt der vorliegende Artikel.

Für die Beispiele dieses Artikels legen Sie wieder ein Projekt auf Basis der Vorlage **Visual C#|Web|ASP.NET Core-Webanwendung** an. Wählen Sie im Dialog **Neue ASP.NET Core Webanwendung** den Eintrag **Webanwendung** aus.

Wir wollen zunächst nur eine Seite namens **Index.cshtml** nutzen, die wir allerdings mit eigenen Inhalten füllen wollen. Also entfernen wir den aktuell in der Datei **/Pages/Index.cshtml** enthaltenen Code mit Ausnahme der Razor-Anweisungen zu Beginn des Codes (siehe Bild 1).

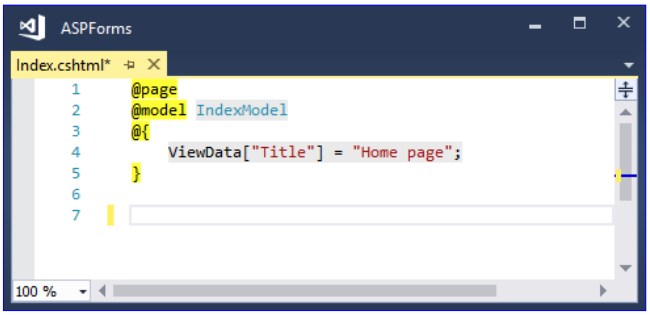

Bild 1: Basis unseres Formulars

In unserem Beispiel wollen wir ein Formular erstellen, in das der Benutzer die Daten wie in einem Kontaktformular eingibt und dessen Inhalt dann an den Betreiber der Webseite geschickt wird. Als E-Mail-Provider wollen wir dann wie im Artikel **E-Mails per SendGrid verschicken** den Anbieter **Sendgrid** nutzen.

Model anlegen

Bevor wir mit der Programmierung des eigentlichen Formulars im HTML-Format beginnen, stellen wir ein Model zusammen, oder genauer: eine Klasse mit einem Model für die E-Mail-Daten. Diese Klasse soll wie folgt aussehen und kann der Einfachheit halber direkt in der Datei **Index.cshtml.cs** eingefügt werden:

```
public class KontaktformularModel {
    [Required]
    public string Vorname { get; set; }
    [Required]
    public string Nachname { get; set; }
    [Required]
    public string Email { get; set; }
    [Required]
    public string Nachricht { get; set; }
}
```

Hier sehen Sie neben den üblichen Elementen einer Klasse jeweils das Attribut **[Required]**. Damit dieses nicht als Fehler markiert wird, fügen wir der Klassendatei mit der **using**-Anweisung den Namespace **System.ComponentModel.DataAnnotations** hinzu:

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

Damit wir ein Objekt auf Basis der Klasse **Kontaktformular-Model** im HTML-Formular nutzen können, machen wir dieses ähnlich wie unter WPF öffentlich verfügbar (siehe Bild 2):

```
[BindProperty]
public KontaktformularModel Kontakt{ get; set; }
```

Damit kommen wir nun zum HTML-Code des Formulars. Die einfachste Form dieses Formulars sieht wie folgt aus – wenn wir das Projekt nun starten, erhalten wir das Formular aus Bild 3:

```
<form method="post">
    <table>
        <tr>
            <td>Vorname:</td>
            <td><input asp-for="Kontakt.Vorname" /></td>
        </tr>
        <tr>
            <td>Nachname:</td>
            <td><input asp-for="Kontakt.Nachname" /></td>
        </tr>
        <tr>
            <td>E-Mail:</td>
            <td><input asp-for="Kontakt.Email" /></td>
        </tr>
        <tr>
            <td>Nachricht:</td>
            <td><input asp-for="Kontakt.Nachricht" /></td>
        </tr>
        <tr>
            <td colspan="2"><button type="submit">Absenden</button></td>
        </tr>
```

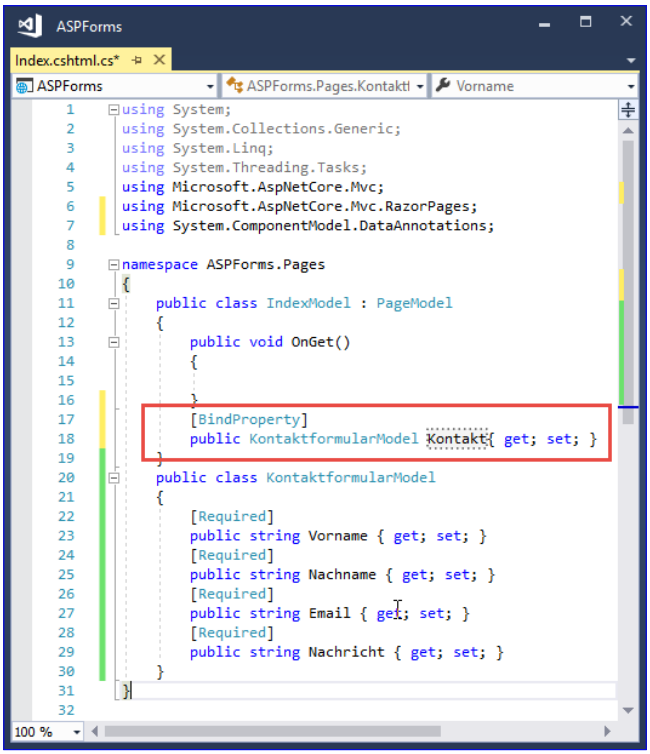

Bild 2: Aktueller Stand der Klassendatei **Index.cshtml.cs**

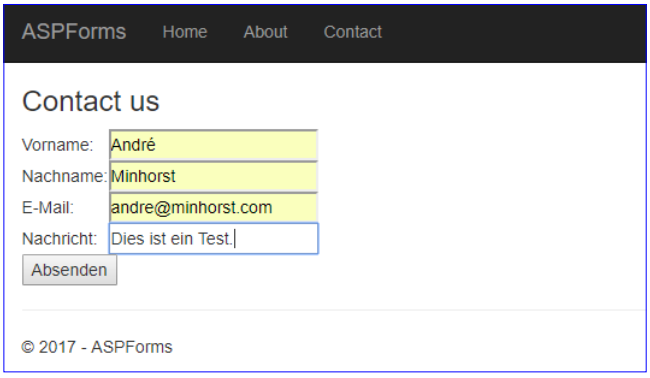

Bild 3: Start des ersten Beispiels

```
    </table>
</form>
```

Wenn wir nun auf Absenden klicken, geschieht allerdings noch nichts. Das liegt daran, dass wir nicht – wie früher üblich – die Zielseite angegeben haben, die nach dem Anklicken der **Submit**-Schaltfläche aufgerufen werden soll. Diese wurde sonst im Attribut **action** des **form**-Elements angegeben. Wir geben hingegen mit dem Attribut **method** nur die Art der Übermittlung der Inhalte des Formulars an, und zwar als **post** (es gibt auch noch **get**). Wenn Sie hier **get** angeben, werden die in das Formular eingegebenen Daten an die durch das Formular aufgerufene Seite über die URL übermittelt, für eine PHP-Seite also etwa mit **/index.php?vorname=andre&nachname=minhorst&email=andre@minhorst.com**. Der Nachteil dieser Version ist, dass die Anzahl der übermittelbaren Zeichen je nach Browser begrenzt ist. Deshalb verwenden wir **post** als **action**.

Auswertung des Formulars

Wie aber wollen wir das Formular nun nach dem Senden auswerten? Unter PHP beispielsweise gibt man eine Zielseite an, die dann den notwendigen Code enthält, um die entweder in der Variablen **$_GET** oder **$_POST** übermittelten Informationen zu verarbeiten. Unter ASP.NET Core läuft das ganz anders. Hier enthält die Code behind-Datei der Seite mit dem Formular eine Ereignismethode, welche die Daten auswertet. Sie brauchen also keine Zielseite für das Formular anzugeben, die Verarbeitung erfolgt in der Seite selbst beziehungsweise in ihrem serverseitigen Code. Die einfachste Version dieser Methode, die lediglich zu der Seite mit dem Namen **About.cshtml** weiterleitet, sieht wie folgt aus:

```
public async Task<IActionResult> OnPostAsync() {
    return RedirectToPage("About");
}
```

Wenn Sie die Anwendung nun starten, die Daten in das Formular eingeben und dann auf Absenden klicken, wird anschließend die Seite **About.cshtml** geladen.

Validieren

Und nun kommt ein wenig ASP.NET Core-Magie ins Spiel. Wir fügen nun eine **if**-Bedingung ein, welche die Eigenschaft **IsValid** des Objekts **ModelState** prüft und erneut die aktuelle Seite aufruft, wenn **IsValid** nicht den Wert **true** liefert:

```
public async Task<IActionResult> OnPostAsync() {
    if (!ModelState.IsValid) {
        return Page();
    }
    return RedirectToPage("About");
}
```

Wie funktioniert das? Nun: Wir haben ja mit dem **[Required]**-Element festgelegt, dass die Daten für die vier Felder der Klasse **KontaktformularModel** angegeben werden müssen. Diese werden bei der Prüfung durch **ModelState.IsValid** untersucht. Ist eines der Felder leer, hat **IsValid** den Wert **False** und man landet wieder auf der auslösenden Seite. Erst wenn alle Felder gefüllt sind, wird die Seite **About** mit der **RedirectToPage**-Methode aufgerufen.

Mail versenden

Um die Mail mit den Kontaktinformationen schließlich zu versenden, fügen wir der Methode **OnPostAsync** noch den Aufruf einer weiteren Methode namens **SendMail** hinzu:

```
public async Task<IActionResult> OnPostAsync() {
    if (!ModelState.IsValid) {
        return Page();
    }
    SendMail();
    return RedirectToPage("About");
}
```

Diese verwendet in diesem Fall nicht den Mail-Anbieter **Sendgrid** wie im Artikel **E-Mails per Sendgrid verschicken** beschrieben, sondern die Klasse **MailMessage** der Bibliothek **System.Net.Mail**, um die E-Mail zusammenzustellen und die Methode **Send** der Klasse **SmtpClient**, um die Mail zu versenden. Dabei nutzen wir auch schon die String-Interpolation, die wir im Artikel **Interpolierte Zeichenketten** beschrieben haben, um den Inhalt der E-Mail zusammenzustellen. Dabei pflegen wir die Inhalte des Objekts **Kontakt** auf Basis der Klasse **KontaktformularModel** direkt in geschweiften Klammern in den Mailtext ein:

```
private void SendMail() {
    var inhalt = $@"Hallo André,

hier kommt eine Kontaktanfrage:

Vorname: {Kontakt.Vorname}
Nachname: {Kontakt.Nachname}
E-Mail: {Kontakt.Email}
Nachricht: {Kontakt.Nachricht}";

    using (var mailMessage = new MailMessage(Kontakt.Email, "andre@minhorst.com")) {
        mailMessage.To.Add(new MailAddress("andre@minhorst.com"));
        mailMessage.From = new MailAddress(Kontakt.Email);
        mailMessage.Subject = "Kontaktanfrage";
        mailMessage.Body = inhalt;
        using (var smtpClient = new SmtpClient("minhorst.com")) {
            smtpClient.Send(mailMessage);
        }
    }
}
```

Im hinteren Teil der Methode erstellen wir ein neues Objekt namens **mailMessage** auf Basis der Klasse **MailMessage** und weisen dieser die Empfängeradresse, die Absenderadresse sowie den Betreff und für die Eigenschaft **Subject** den Inhalt der

Variablen **inhalt** zu. Dann verschickt sie die E-Mail über die Methode **Send** der Klasse **SmtpClient**, der wir beim Erstellen die Adresse des SMTP-Servers übergeben. In diesem Fall können wir den SMTP-Server problemlos nutzen, da wir eine Mail an eine der E-Mail-Adressen schicken, die von diesem Server verwaltet wird. Wollten Sie E-Mails an andere Adressen schicken, etwa um der Person, welche die Kontaktanfrage gesendet hat, eine Kopie davon zu schicken, sind Sie mit Sendgrid wohl besser bedient.

Optimierung: Validierung und Layout

Nun wollen wir noch etwas Komfort hinzufügen, beispielsweise indem wir die Validierung, die ja bereits durch die **[Required]**-Elemente vorgegeben ist, auch noch in die Benutzeroberfläche einbeziehen, sprich: ins Formular.

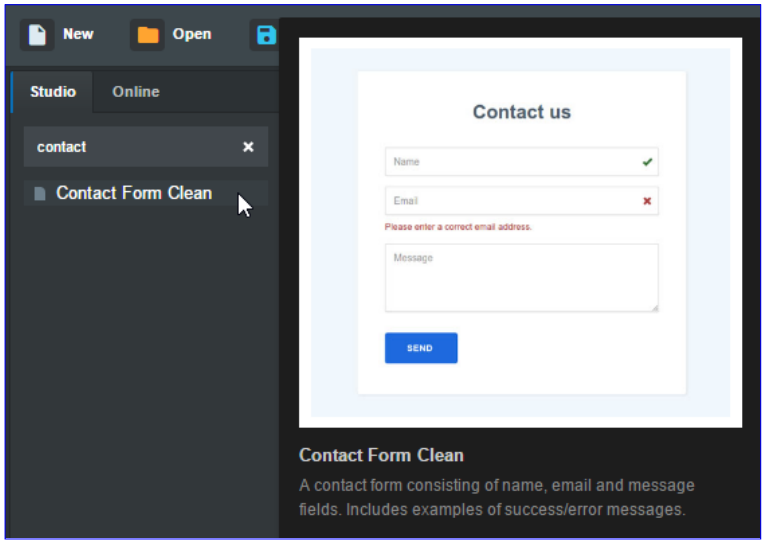

Bild 4: Vorschau eines Formulars

Gleichzeitig wollen wir auch noch die Optik anpassen und statt der Tabelle mit **div**-Elementen arbeiten, die ihre Format-Informationen aus entsprechenden CSS-Elementen beziehen.

Zum Erstellen des Designs wollen wir wieder Bootstrap Studio nutzen, das Sie bereits im Artikel **Webdesign mit Bootstrap Studio** kennengelernt haben.

Da wir ein Kontaktformular benötigen, können wir in Bootstrap Studio links oben im Feld **Search components** beispielsweise den Suchbegriff **contact** eingeben. Dies liefert nur einen Eintrag, der uns ein wenig schmal erscheint (siehe Bild 4).

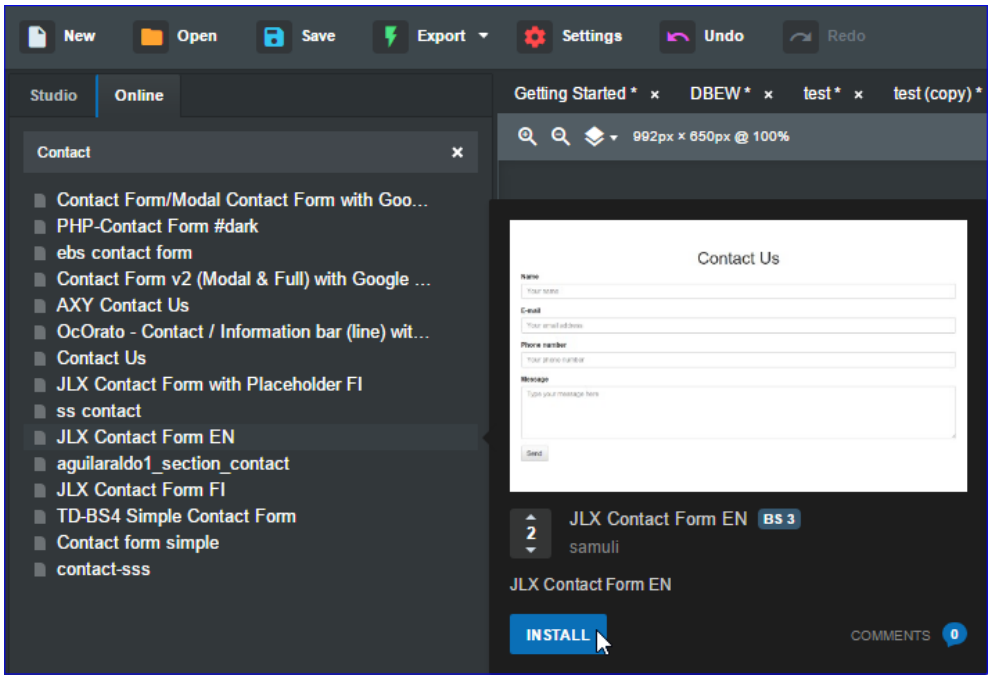

Bild 5: Alle Vorlagen mit dem Schlüsselwort **Contact**

Wir möchten lieber einen anderen Entwurf haben. Dazu gibt es den Bereich **Online**. Dieser zeigt standardmäßig die im Trend liegenden und die neuesten Vorlagen an. Wenn Sie hier den Suchbegriff **Contact** eingeben,

Razor Pages mit Datenbankanbindung

Nachdem Sie in weiteren Artikeln einige Grundlagen zur Programmierung von Razor Pages mit ASP.NET Core kennengelernt haben, geht es nun einen Schritt weiter: Wir wollen eine erste kleine Anwendung für den Zugriff auf die Daten einer Datenbank programmieren. Dazu beginnen wir mit einer kleinen Tabelle, für die wir eine Übersicht und eine Detailseite zum Ansehen und Bearbeiten der einzelnen Felder erstellen. Die Übersicht soll natürlich auch eine Möglichkeit zum Hinzufügen und Löschen der Datensätze bieten.

Voraussetzungen

Sie benötigen Visual Studio in der Version von 2017, die Sie mit den Komponenten wie im Artikel Visual Studio 2017 Community Edition beschrieben installieren.

Projekt erstellen

Wir erstellen ein Projekt auf Basis der Vorlage **Visual C#|Web|ASP.NET Core Webanwendung** (siehe Bild 1).

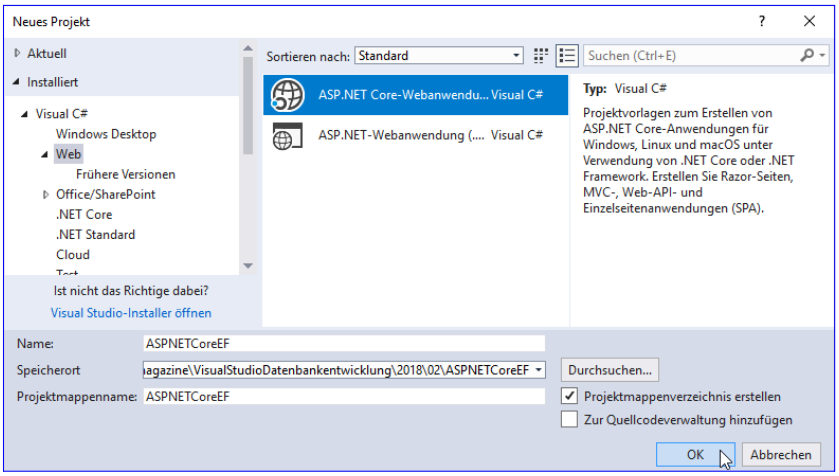

Bild 1: Projekt anlegen

Im folgenden Dialog wählen Sie den Eintrag **Webanwendung** aus, da wir hier schon einige Elemente bekommen, die wir sonst manuell nachreichen müssten (siehe Bild 2).

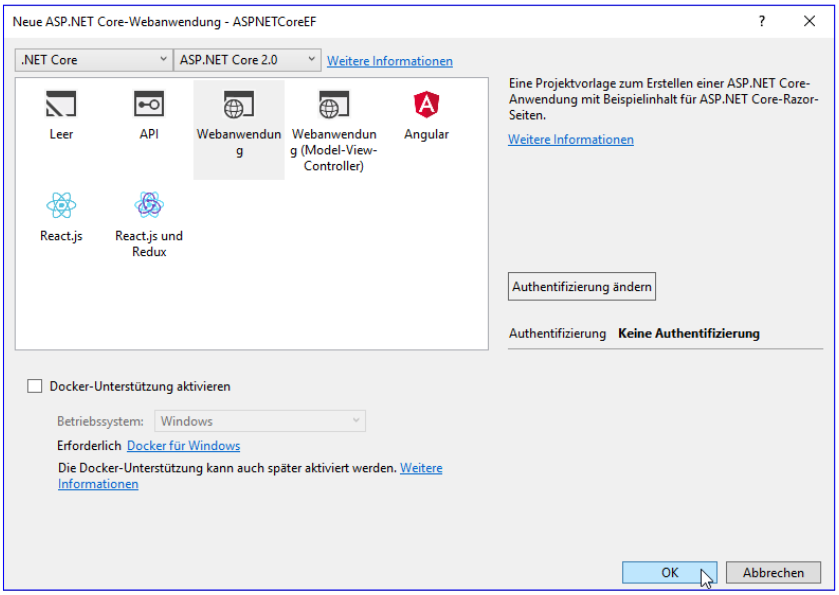

Bild 2: Projektvorlage definieren

Anpassen des Rahmens

Mit Rahmen meinen wir die Seite, welche die Navigationsleiste und den Fußbereich der Seite definiert. In unserem automatisch erstellten Demo-Projekt ist dies die Seite **_Layout.cshtml**. Wie die enthaltenen Elemente einer Webanwendung anpassen können, erfahren Sie beispielsweise im Artikel **ASP.NET Core-Anwendung anpassen**. In unserem Fall wollen wir die Bezeichnung unseres Projekts, das an verschiedenen Stellen in der Datei **_Layout.cshtml** gelandet ist, durch die Bezeichnung unserer Anwendung ersetzen – also beispielsweise **Kundenverwaltung**. Außerdem wollen wir das Navigationsmenü so ändern, dass es hinter den Einträgen **Home** und **About** den Eintrag **Kunden** anzeigt.

Damit dies geschieht, ändern wir in Bild 3 markierten Zeilen in der Datei **_Layout.cshtml**. Die eingerahmten Stellen markieren die zu ändernden Stellen, der Pfeil zeigt den Befehl **@RenderBody()**, an dem später die eigentlichen Inhalte eingefügt werden.

Den Inhalt der Index-Seite, also der Seite, die direkt beim Starten der Anwendung im Browser angezeigt wird, können Sie auch anpassen. Den ersten **div**-Teil mit **id="myCarousel"** entfernen wir komplett. Im folgenden **div**-Element entfernen wir die hinteren Auflistungen. Den vorderen Teil können wir wie folgt anpassen. Dabei wollen wir eine kleine Übersicht über die enthaltenen Funktionen bieten. Der Code sieht anschließend wie folgt aus:

Bild 3: Datei **_Layout.cshtml** anpassen

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}
<div class="row">
    <div class="col-md-3">
        <h2>Die Beispielanwendung Kundenverwaltung</h2>
        <p>Diese Beispielanwendung zeigt, wie Sie eine Demo-Anwendung um Datenbankfunktionen erweitern. Dazu gehören:</p>
        <ul>
            <li>Übersichten für Kunden und andere Entitäten mit Schaltflächen zum Anzeigen oder Löschen von Kunden</li>
            <li>Anzeige der Kundendetails zum Bearbeiten der Daten</li>
```

```
            <li>Anzeige der Kundendetails zum Anlegen neuer Kundendatensätze</li>
        </ul>
    </div>
</div>
```

Wenn Sie die Anwendung danach starten, zeigt der Browser die Seite aus Bild 4 an. Für die wenigen Änderungen, die wir vorgenommen haben, sieht diese schon recht ordentlich aus. Ein Klick auf den Link Kunden in der Navigationsleiste zeigt nun leider noch keine Kundenseite an, was wir als Nächstes in Angriff nehmen werden.

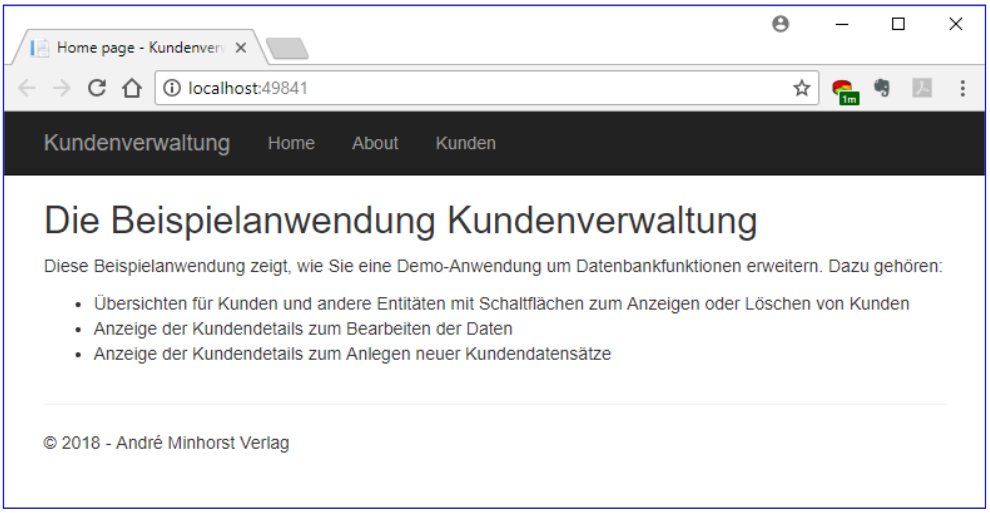

Bild 4: Die angepasste Startseite

Kunden-Klasse hinzufügen

Nun wollen wir uns um die anzuzeigenden Daten kümmern. Dazu benötigen wir eine Datenbank und ein entsprechendes Entity Data Model mit den Klassen für die einzelnen Entitäten. Wir verwenden den Code First-Ansatz, bei dem wir erst die Entitätsklassen definieren und daraus dann automatisch die Datenbank erstellen lassen. Für die Entitätsklassen legen wir direkt unterhalb des Projektordners ein neues Verzeichnis namens **Models** an.

In diesem Verzeichnis erstellen wir mit dem Kontextmenü-Eintrag **Hinzufügen|Klasse...** eine neue Klassendatei, der wir im folgenden Dialog **Neues Element hinzufügen** den Namen **Customer** geben. Wir könnten auch die deutsche Bezeichnung verwenden, aber beim Entity Framework ist es einfacher, wenn man die Bezeichnungen in englischer Sprache angibt. So kann man leichter Singular durch einfaches Anhängen des Buchstaben **s** in Plural verwandeln. Außerdem muss man dort nicht aufpassen, ob eine Bezeichnung gleichzeitig Singular und Plural ist (im Deutschen beispielsweise **Artikel**!).

Die Klasse sieht wie folgt aus und wird in der Klassendatei **Customer.cs** angelegt:

```
namespace ASPNETCoreEF.Models {
    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; }
    }
}
```

Das Feld **ID** soll der Primärschüssel der noch zu erstellenden Tabelle sein, in der die Daten zu den Kunden gespeichert werden. Die übrigen Felder benötigen keine weitere Erläuterung. Fremdschlüsselfelder und weitere Tabellen lassen wir aus Gründen der Übersicht zunächst außen vor.

Beachten Sie, dass durch das Anlegen der Klassendatei im Verzeichnis **Models** auch der Namespace auf **ASPNETCoreEF.Models** eingestellt wurde (**ASPNETCoreEF** ist der Name des Projekts und somit auch als Namespace des Projekts voreingestellt).

Datenbankkontext erstellen

Nun benötigen wir noch die Klasse, welche die Informationen der einzelnen Entitätsklassen – hier nur eine – zusammenfasst und Informationen zum Erstellen des Datenmodells speichert. Dazu fügen wir dem Projekt zunächst einen weiteren neuen Ordner namens **Data** hinzu. Hier legen wir eine neue Klasse namens **CustomerManagementContext** an. Dieser fügen wir im Kopf der Klassendatei zunächst einen Verweis auf den Namespace **Microsoft.EntityFrameworkCore** hinzu:

```
using Microsoft.EntityFrameworkCore;
```

Da die **Customer**-Klasse sich im Verzeichnis **Models** und somit auch im Namespace **ASPNETCoreEF.Models** befindet, fügen wir auch diesen Namespace noch hinzu:

```
using ASPNETCoreEF.Models;
```

Die Klasse soll die Schnittstelle **DbContext** implementieren. Außerdem erhält Sie ein **DbSet**-Objekt namens **Customers**, welches Elemente des Typs **Customer** aufnehmen soll:

```
public class CustomerManagementContext : DbContext {
    public CustomerManagementContext(DbContextOptions<CustomerManagementContext> options) : base(options) {

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

Wir überschreiben noch die Methode **OnModelCreating** und geben dort an, dass die Tabelle für die Kunden nicht **Customer** heißen soll, sondern im Plural **Customers** angelegt werden soll:

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Customer>().ToTable("Customers");
}
```

DbContext beim Anwendungsstart registrieren

Damit die Datenbank über das **DbContext**-Objekt gleich nach dem Start zur Verfügung steht, fügen wir der Methode **ConfigureServices** der Datei noch eine entsprechende Anweisung hinzu. Die Datei **Startup.cs** erhält jedoch zuvor zwei zusätzliche Verweise auf die benötigten Namespaces:

```
using Microsoft.EntityFrameworkCore;
using ASPNETCoreEF.Data;
```

Dann ergänzen wir **ConfigureServices** wie folgt:

```
public void ConfigureServices(IServiceCollection services) {
    services.AddMvc();
    services.AddDbContext<CustomerManagementContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
}
```

Dadurch greift das Projekt auf die Verbindungszeichenfolge zu, die wir noch in der Datei **appsettings.json** speichern müssen. Diese erweitern wir wie folgt:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=CustomerManagement;ConnectRetryCount=0;
        Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

Hier legen wir in der Einstellung **ConnectionStrings** fest, dass die Standardverbindungszeichenfolge den Server **localdb** verwenden soll und der Datenbankname **CustomerManagement** lautet. Außerdem soll Windows-Authentifizierung für den Zugriff auf die Datenbank verwendet werden (**Trusted_Connection=True**).

Ein paar Daten vorbereiten

Nun wollen wir noch dafür sorgen, dass die Tabelle **Customers** beim Start der Anwendung nicht völlig leer ist. Daher fügen wir der Anwendung im Verzeichnis **Data** eine neue Klassendatei namens **DbInitializer.cs** hinzu. Diese füllen wir wie folgt:

```
public class DbInitializer {
```

```
    public static void Initialize(CustomerManagementContext dbContext) {
        dbContext.Database.EnsureCreated();
        if (dbContext.Customers.Any()) {
            return;
        }
        var customers = new Customer[] {
            new Customer{FirstName="André", LastName="Minhorst", Street="Borkhofer Str. 17", Zip="47137", City="Duisburg"},
            new Customer{ FirstName="Klaus", LastName="Müller", Street="Teststr. 1", Zip="12345", City="Berlin" },
            new Customer{ FirstName="Herbert", LastName="Meier", Street="Beispielweg 2", Zip="23456", City="Bremen"}
        };
        foreach (Customer c in customers) {
            dbContext.Customers.Add(c);
        }
        dbContext.SaveChanges();
    }
}
```

Die Klasse enthält eine Methode namens **Initialize**, welche mit dem Parameter **dbContext** ein Objekt des Typs **CustomerManagementContext** entgegennimmt. Sie prüft zunächst ob die Datenbank bereits erstellt wurde und erledigt dies gegebenenfalls im gleichen Schritt mit der Methode **EnsureCreated**. Dann prüft sie, ob die **Customers**-Tabelle bereits Daten enthält. Falls ja, bricht sie an dieser Stelle ab. Anderenfalls erstellt sie ein Array mit drei Elementen des Typs **Customer**. Die folgende **foreach**-Schleife durchläuft dieses Array und fügt die einzelnen Elemente des Typs **Customer** zum **DbSet** namens **Customers** des **dbContext** hinzu. Die Änderungen werden schließlich mit der **SaveChanges**-Methode gespeichert.

Nun müssen wir noch dafür sorgen, dass diese Methode zum Erstellen beziehungsweise Füllen der Datenbank auch ausgeführt wird. Dazu ändern wir in der Klassendatei **Program.cs** die Methode **Main** wie folgt:

```
public static void Main(string[] args) {
    //            BuildWebHost(args).Run();
    var host = BuildWebHost(args);
    using (var scope = host.Services.CreateScope()) {
        var services = scope.ServiceProvider;
        var dbContext = services.GetRequiredService<CustomerManagementContext>();
        DbInitializer.Initialize(dbContext);
    }
    host.Run();
}
```

Dabei kommentieren wir die dort bereits vorhandene Anweisung aus. Diese ein neues Webhost-Objekt und führte direkt seine **Run**-Methode aus. Hier wollen wir das Erstellen der Datenbank zwischenschalten. Dazu teilen wir die Anweisung auf, sodass diese zunächst das neue Objekt mit **BuildWebHost** erstellt und in der neuen Variable namens **host** speichert. Die **Run**-Methode

für die Variable **host** wird dann als letzte Anweisung ausgeführt. Dazwischen fügen wir ein paar neue Anweisungen ein. Die erste erstellt ein neues **Scope**-Element und speichert es in der Variablen **scope**. Über die Eigenschaft **ServiceProvider** holt die Prozedur dann ein **Service**-Objekt und speichert es in der Variablen **services**. Diees stellt dann über die Methode **GetRequiredService** mit dem Typ **CustomerManagementContext** den Datenbankkontext zur Verfügung und speichert diesen in der Variablen **dbContext**. Dieses Objekt übergeben wir dann der Methode **Initialize** der zuvor programmierten Klasse **DbInitializer**.

Damit alle als fehlerhaft markierten Stellen verschwinden, müssen Sie wieder die folgenden Namespaces referenzieren:

```
using Microsoft.Extensions.DependencyInjection;
using ASPNETCoreEF.Data;
```

Datenbank erstellen

Damit ist es soweit: Sie können die Anwendung nun starten und damit die Datenbank erstellen. Sobald der Browser aufgerufen wurde und die Startseite der Anwendung anzeigt, können Sie diese wieder beenden. Wir blenden dann den **SQL Server-Objekt-Explorer** ein (Menüeintrag **Ansicht|SQL Server-Objekt-Explorer**) und klappen die Einträge wie in Bild 5 auf. Wenn die Datenbank **CustomerManagement** dort angezeigt wird, navigieren Sie weiter zur Tabelle **dbo.Customers** und wählen den Kontextmenü-Eintrag **Daten anzeigen** aus. Damit sollten auch die soeben per Code hinzugefügten Datensätze erscheinen.

Die beiden Datenbankdateien **CustomerManagement.mdf** und **CustomerManagement.ldf** wurden auf dem lokalen Rechner im Verzeichnis **C:\Users\<Benutzername>** gespeichert. Dabei wurde automatisch der Name als Datenbankname verwendet, den wir in der Verbindungszeichenfolge angegeben haben. Der Tabellenname entspricht wie von uns gewünscht der Plural-Form der verwendeten Klasse **Customer**, also **Customers**. Die Eigenschaftsnamen der Klasse **Customer** wurden als Feldnamen verwendet und das Feld mit dem Namen **ID** wurde automatisch als Primärschlüsselfeld festgelegt.

Kundenübersicht hinzufügen

Da wir neben den Kunden später noch weitere Daten zur Anwendung hinzufügen werden, legen wir für jede Entität einen eigenen Ordner an. Als Erstes benötigen wir also einen Ordner für Kunden, der die Übersicht, die Detailseite et cetera aufnehmen soll. Diesen Ordner legen wir unterhalb des Ordners **Pages** an, und zwar über dessen Kontextmenü-Eintrag **Hinzufügen|Neuer Ordner**. Der neue Ordner soll analog zur englischen Bezeichnung der Tabelle **Customers** heißen.

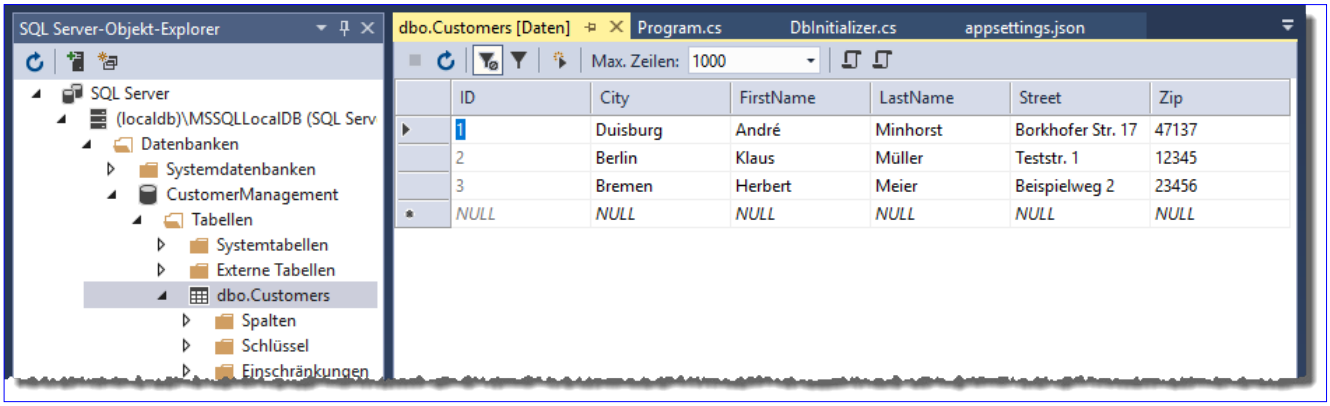

Bild 5: Die neue Datenbank mit den frischen angelegten Datensätzen

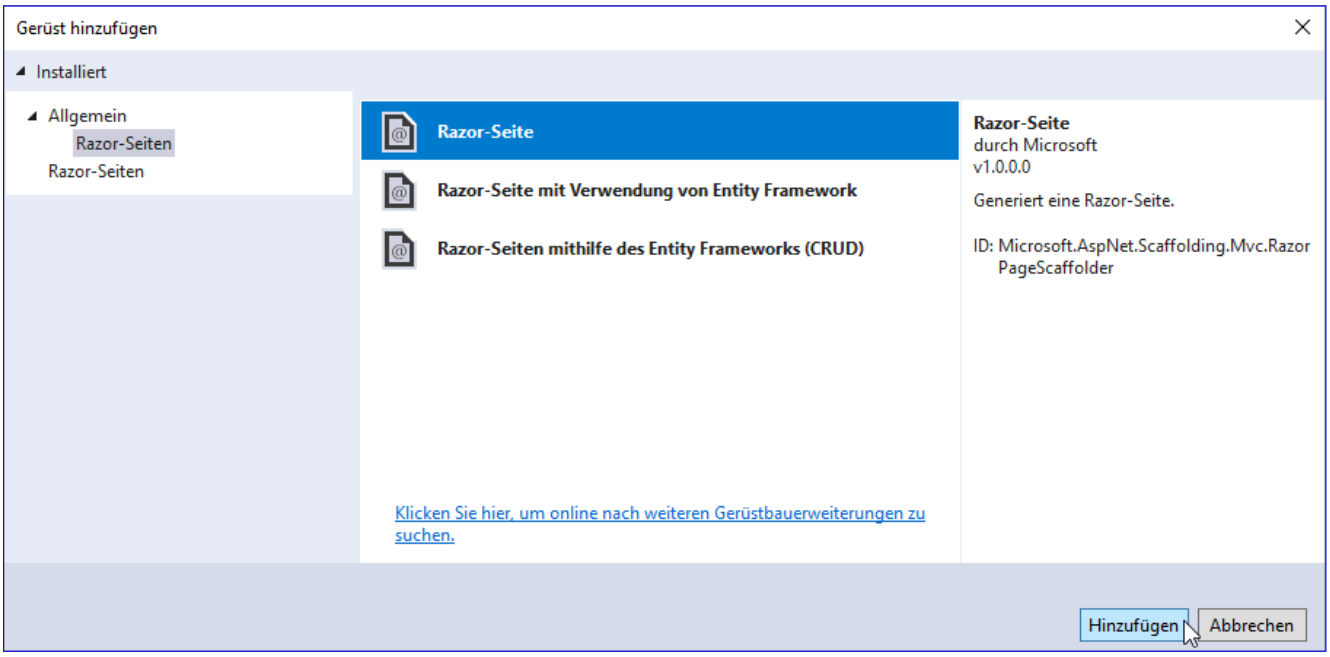

Bild 6: Anlegen einer neuen Razor-Seite

Diesem Ordner fügen wir gleich eine neue Razor-Seite hinzu, die wir **Index.cshtml** nennen. Eine neue Razor-Seite fügen Sie dem Ordner hinzu, indem Sie aus seinem Kontextmenü den Eintrag **Hinzufügen|Razor-Seite...** auswählen. Im folgenden Dialog finden Sie dann drei Möglichkeiten (siehe Bild 6):

- **Razor-Seite**
- **Razor-Seite mit Verwendung von Entity Framework**
- **Razor-Seiten mithilfe des Entity Frameworks (CRUD)**

Einfache Razor-Seite anlegen

Wählen Sie die erste Option wählen, geben Sie im folgenden Schritt einfach den Namen der gewünschten Seite an und erstellen diese. Damit erhalten Sie dann eine fast leere Razor-Seite mit zugehöriger **.cshtml.cs**-Klassendatei, die Sie weitgehend selbst füllen müssen. Diese Option ist interessant, wenn Sie komplett neue Seiten aufbauen möchten. Die anderen beiden Optionen hören sich aber wesentlich interessanter an!

Razor-Seite mit Verwendung von Entity Framework

Die zweite Option unterscheidet sich vom Titel her von der dritten Option, weil sie offenbar nur eine Seite anlegt und nicht mehrere. Schauen wir uns diese an. Nach der Auswahl erscheint der Dialog aus Bild 7. Hier geben wir zunächst den Namen der Razor-Seite an. Diese soll Index heißen, weil wir zunächst eine Übersicht der Kunden erstellen wollen. Warum nennen wir die Seite dann nicht **CustomerOverview** oder ähnlich? Weil die Übersicht die Standardseite sein soll, die auch beim Aufrufen des Ordners **/Customer** bereits erscheint. Wird ein Ordner in der URL angegeben und keine spezielle Datei, ruft die Anwendung die Datei **Index.cshtml** auf, sofern diese in dem Ordner enthalten ist.

Authentifizierung unter ASP.NET Core

In weiteren Artikeln dieser Ausgabe haben wir uns mit ASP.NET Core und den Razor Pages beschäftigt. Der vorliegende Artikel zeigt, wie Sie beim Erstellen eines neuen Projekts automatisch die Elemente für eine Benutzerverwaltung zum Projekt hinzufügen können. Damit werden automatisch etwa die notwendigen Datenbanktabellen angelegt und Elemente für die Anmeldung und die Registrierung hinzugefügt.

Wenn Sie eine Webanwendung bauen wollen, die dynamische Inhalte anzeigt, sind Sie mit ASP.NET Core und den Razor Pages aktuell auf der richtigen Seite. In vielen Fällen sollen solche Webanwendungen nicht nur dynamische Inhalte anzeigen, sondern diese unter Umständen teilweise nur angemeldeten Benutzern bereitstellen. Das ist ja beispielsweise auch bei den Artikeln und Downloads zu diesem Magazin der Fall – als Kunde können Sie sich im Shop unter **shop.minhorst.com** anmelden, nachdem Sie sich registriert haben, und erhalten dann Zugriff auf einen Bereich namens **Meine Sofortdownloads**. Oder Sie schauen die einzelnen Artikel im HTML-Format im Know-how-Bereich der Seite ein. Auch hier wird geprüft, ob überhaupt ein Benutzer angemeldet ist und falls ja, ob er das Produkt mit den gewünschten Inhalten abonniert hat.

Wenn Sie selbst eine Internetseite mit einem geschützten Bereich programmieren wollen, auf den nur angemeldete Benutzer zugreifen können, haben Sie eine Menge Arbeit vor sich. Sie müssen Datenbanktabellen erstellen, welche die Benutzer und ihre Berechtigungen verwalten und gegebenenfalls auch noch Benutzergruppen definieren. Sie benötigen eine Seite, auf der Benutzer sich registrieren können und eine weitere Seite, auf der sie sich später erneut einloggen können. Die Registrierungsseite muss Automatismen aufweisen, die prüfen, ob der Benutzername bereits vorhanden ist und ob das Kennwort den gewünschten Konventionen entspricht – zum Beispiel hinsichtlich der Mindestanzahl von Zeichen und Sonderwünschen wir das Vorkommen von Zahlen, großen Buchstaben oder Sonderzeichen. Dann muss die Seite dafür sorgen, dass der neue Benutzer angelegt wird.

Die Login-Seite soll die Daten des registrierten Benutzers, also Benutzername und Kennwort, entgegennehmen und für die aktuelle Session speichern, welcher Benutzer angemeldet ist. Außerdem soll die Anwendung jederzeit prüfen können, ob aktuell ein Benutzer angemeldet ist.

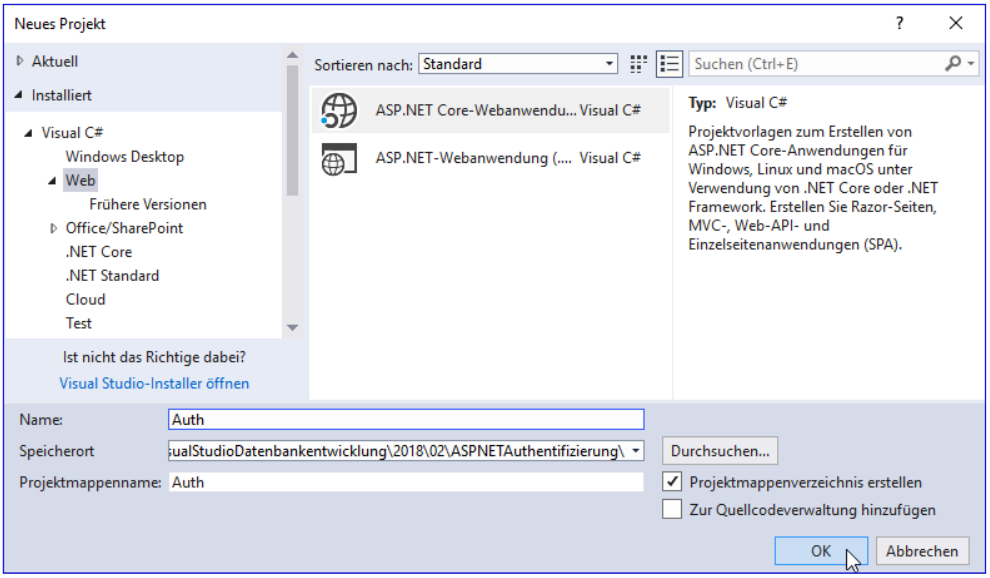

Bild 1: Anlegen eines neuen Projekts auf Basis der Vorlage **ASP.NET Core Webanwendung**

Wenn Sie das alles selbst programmieren wollen, haben Sie eine Weile zu tun. Doch warum sollte man, wenn es dazu fertige Pakete

gibt, die Sie direkt beim Anlegen des Projekts zu diesem hinzufügen können?

Projekt mit Authentifizierung anlegen

Um ein Projekt direkt mit Authentifizierung anzulegen, erstellen Sie dieses zunächst wie gewohnt über den Dialog **Neues Projekt** unter dem gewünschten Namen und Verzeichnis (siehe Bild 1).

Im nächsten Schritt wählen Sie als Projekttyp den Eintrag **Webanwendung** aus. Hier finden Sie außerdem die Schaltfläche **Authentifizierung ändern** (siehe Bild 2). Bisher zeigt der Text darunter noch den Wert **Keine Authentifizierung** als Authentifizierungsmethode an.

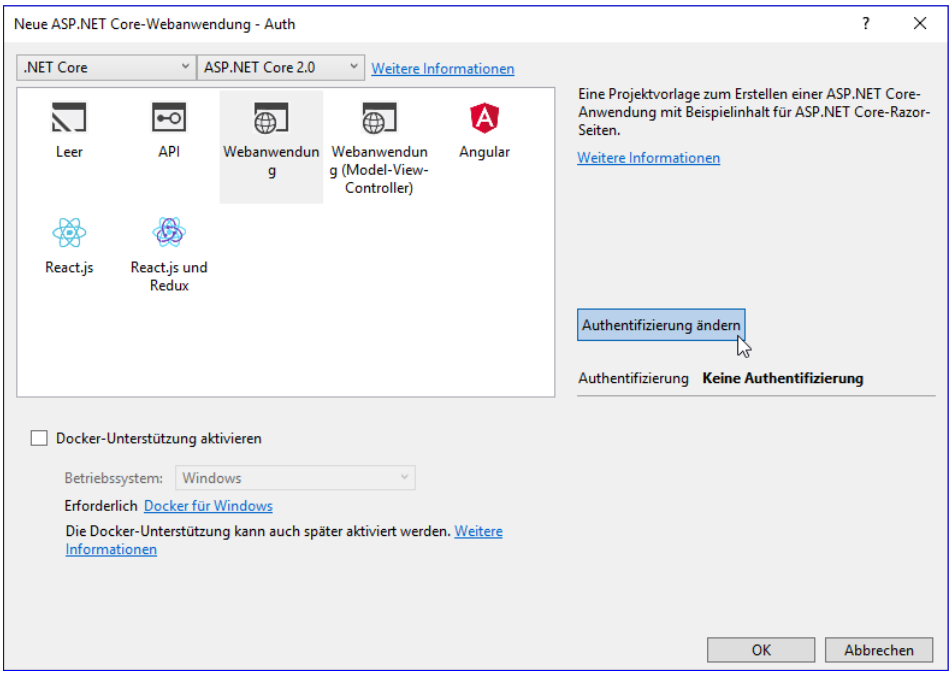

Bild 2: Aktivieren des Eintrags **Webanwendung** und Klick auf **Authentifizierung ändern**

Das ändern wir nach dem Anklicken der Schaltfläche im Dialog **Authentifizierung ändern** (siehe Bild 3). Hier haben Sie die folgenden Wahlmöglichkeiten:

Bild 3: Einstellen der Authentifizierungsmethode

- **Keine Authentifizierung**: Es werden keine Elemente zur Unterstützung einer Authentifizierung angelegt.
- **Einzelne Benutzerkonten**: Die Benutzerkonten werden lokal gespeichert, in diesem Fall in der Datenbank der Anwendung.
- **Geschäfts-, Schul oder Unikonten**: Die Authentifizierung erfolgt über Active Directory, Microsoft Azure Active Directory oder Office 365.
- **Windows-Authentifizierung**: Die Authentifizierung basiert auf den Windows-Konten. Diese Methode ist etwa für Intranet-Anwendungen geeignet.

Wir wollen eine Webanwendung für beliebig viele Benutzer programmieren und nutzen daher die Option **Einzelne Benutzerkonten**. Dabei behalten wir die Einstellung **In-App-Speicherung von Benutzerkonten** bei, da wir eine lokale Datenbank für die Benutzerdaten nutzen wollen.

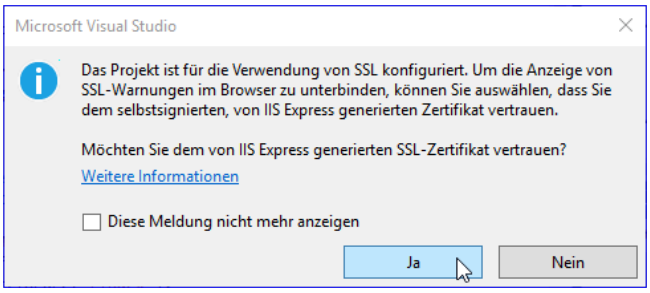

Bild 4: Hinweis auf die Verwendung von SSL

Was ist nun der Unterschied zu einer Anwendung ohne Authentifizierung? Dazu starten wir die Anwendung zunächst einmal. Dabei erhalten wir die Meldung aus Bild 4. Im Gegensatz zur Webanwendung ohne Authentifizierung verwendet diese hier eine SSL-Verschlüsselung. Das macht Sinn, denn niemand möchte Kennwörter im Klartext über das Internet verschicken.

Die nächste Meldung kündigt an, dass für den Rechner **localhost**, also den aktuellen Rechner, ein Zertifikat installiert wird (siehe Bild 5).

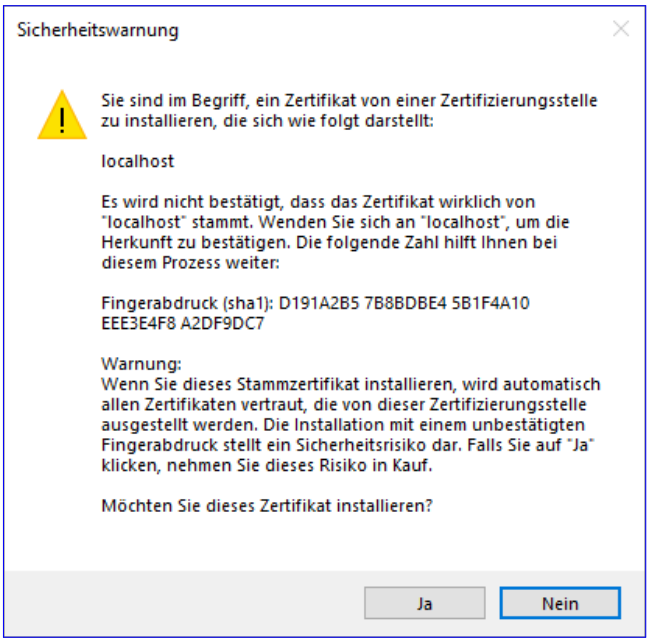

Bild 5: Installation eines Zertifikats auf dem Localhost

Nach dem Starten der Webanwendung zeigen sich zunächst keine offensichtlichen Änderungen. Erst ein Blick nach oben rechts liefert zwei Menüeintrage, die bei dem Projekt ohne Authentifizierung noch nicht vorhanden waren – nämlich **Register** und **Log in** (siehe Bild 6).

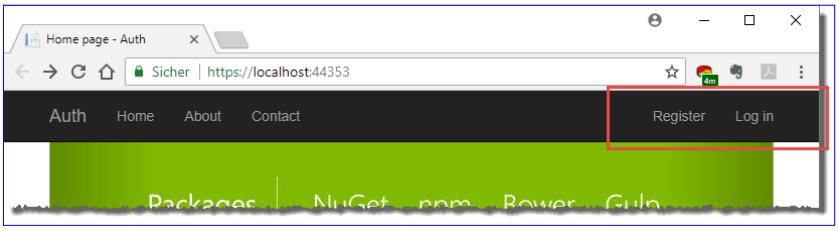

Bild 6: Zwei neue Einträge zum Registrieren und Einloggen

Ein Klick auf den Link **Register** zeigt dann das Formular aus Bild 7 an. Hier finden Sie die üblichen Textfelder zur Eingabe der E-Mail-Adresse sowie Textfelder zur zweifachen Eingabe des Kennworts. Hier finden wir bereits einige Mechanismen zur Prüfung der Validität des Kennworts.

Zum Beispiel meckert das Formular, wenn Sie ein Kennwort eingeben, das weniger als sechs Zeichen enthält oder wenn die beiden Kennwörter nicht übereinstimmen.

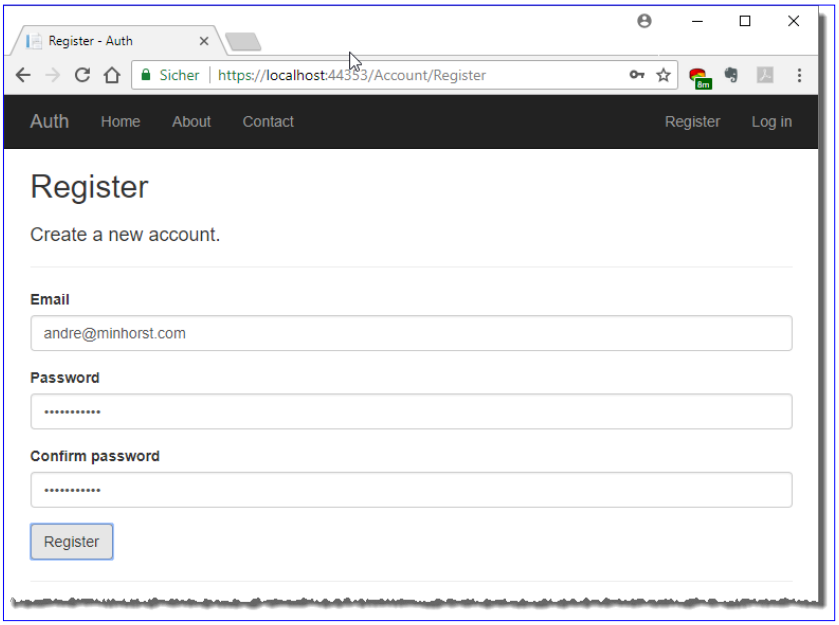

Bild 7: Registrieren eines neuen Benutzers

Weitere Validierungsmeldungen treten erst nach dem Bestätigen der Eingabe über die Schaltfläche **Register** auf – zum Beispiel, wenn das Kennwort nicht mindestens ein nicht alphanumerisches Zeichen enthält oder wenn das Kennwort keinen Großbuchstaben aufweist (siehe Bild 8).

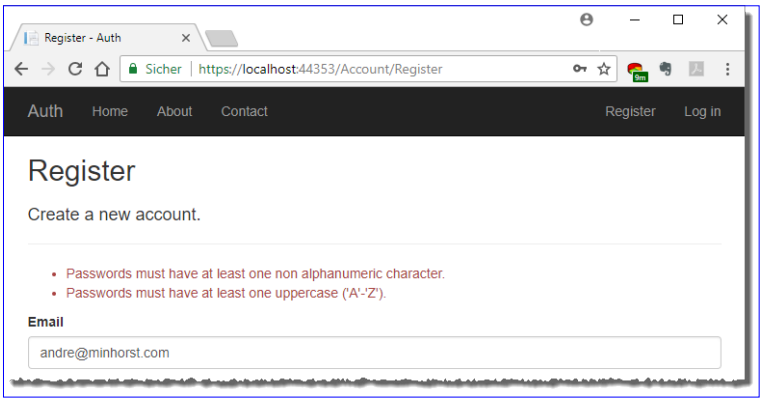

Bild 8: Fehlgeschlagene Registrierung

Fehler beim Registrieren ohne Datenbank

Wenn wir es schließlich geschafft haben, eine gültige E-Mail-Adresse und zwei den Anforderungen entsprechende und übereinstimmende Kennwörter einzugeben, folgt leider noch eine Fehlermeldung (siehe Bild 9). Diese weist uns darauf hin, dass eine Datenbank mit einem recht kryptischen Namen nicht geöffnet werden konnte. Zum Glück liefert die Meldung auch gleich noch einen Hinweis, wie wir die fehlende Datenbank bereitstellen können.

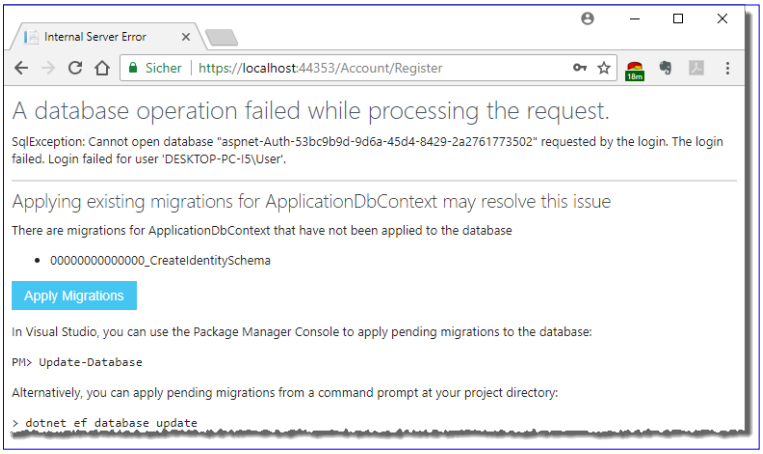

Bild 9: Fehler nach dem Absenden der Registrierung

Verbindungszeichenfolge anpassen und Datenbank erstellen

Dies holen wir nun nach, allerdings erst nach dem Anpassen der Verbindungszeichenfolge. Diese enthält nämlich auch den Namen der zu verwendenden Datenbank. Die Verbindungszeichenfolge finden wir in der Datei **appsettings.json**, die aktuell wie in Bild 10 aussieht. Hier können wir, bevor die Datenbank tatsächlich erstellen, noch den Namen der Datenbank auf eine eingängigere Bezeichnung einstellen, zum Beispiel **DbAuth**.

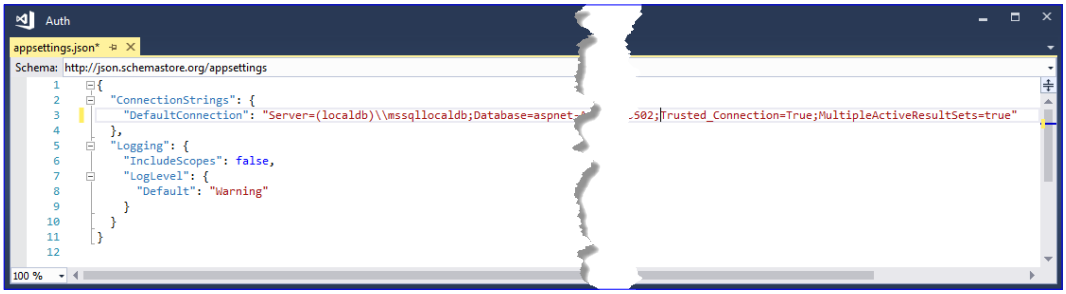

Bild 10: Ändern des Datenbanknamens in der Verbindungszeichenfolge

Danach erstellen wir die Datenbank. Dazu öffnen wir mit dem Menü-Eintrag **Extras|NuGet-Paket-Manager|Paket-Mana-**

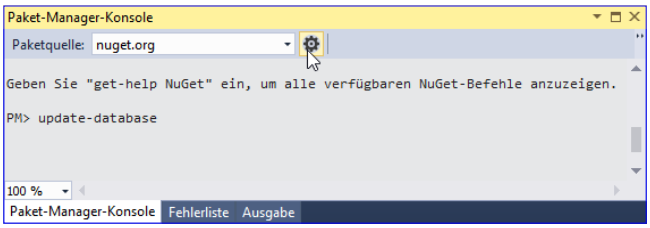

Bild 11: Starten der Erstellung der Datenbank

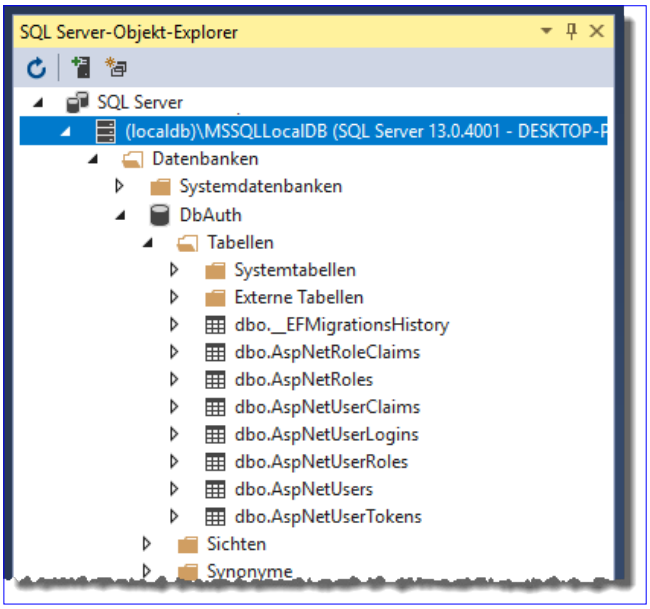

Bild 12: Die neue Datenbank **DbAuth**

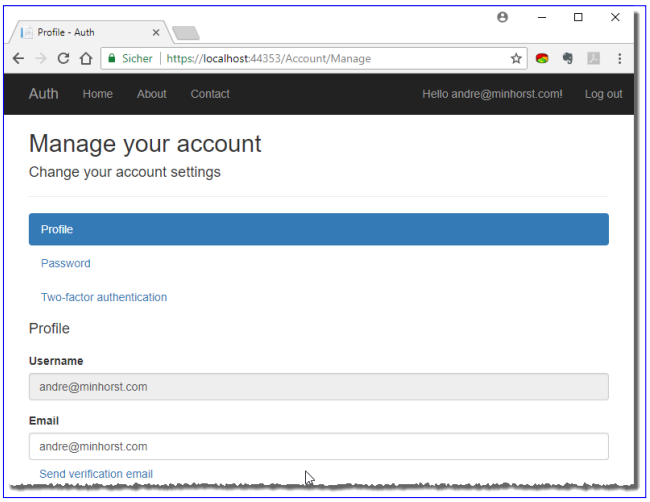

Bild 13: Die Registrierung hat funktioniert!

ger-Konsole den Bereich **Paket-Manager-Konsole**. Hier geben wir nun schlicht und einfach den Befehl **Update-Database** ein (siehe Bild 11). Dies startet die Erstellung der Datenbank mit dem in der Verbindungszeichenfolge angegebenen Namen **DbAuth**.

Nun wollen wir uns vergewissern, ob die Datenbank angelegt wurde und vor allem wie die Tabellen der Datenbank aussehen. Dazu starten wir mit **Ansicht|SQL Server-Objekt-Explorer** die gleichnamige Ansicht in Visual Studio 2017. Öffnen wir hier die Einträge **DbAuth** und darunter Tabellen, sehen wir, dass die Datenbank nicht nur angelegt wurde, sondern auch noch einige Tabellen enthält. Diese heißen etwa **AspNetUsers** oder **AspNetUserTokens** (siehe Bild 12).

Neuer Registrierungsversuch

Damit können wir nun erneut probieren, einen Benutzer zu registrieren. Dazu starten wir die zwischenzeitlich beendete Webanwendung erneut und klicken im nun erscheinenden Browser-Fenster auf den Link **Register**.

Diesmal gelingt die Registrierung! Nach dem Mausklick auf die Schaltfläche **Register** wird die Seite aktualisiert und zeigt im oberen Bereich den Eintrag **Hello andre@minhorst.com!** an. Ein Klick auf diesen Eintrag liefert die Profil-Informationen zum neu erstellten Benutzerkonto (siehe Bild 13).

Wenn wir uns nun mit einem Klick auf den Link **Log out** abmelden, können wir uns anschließend per Klick auf **Log in** erneut anmelden.

Ein Blick in die Tabelle **dbo.AspNetUsers**, die wir mit dem Kontextmenü-Eintrag **Daten anzeigen** des Elements

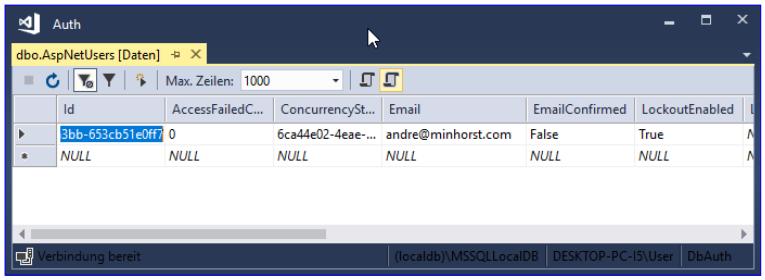

Bild 14: Gespeicherter Benutzer-Datensatz

Datenbanken\DbAuth\Tabellen\dbo.AspNetUsers öffnen, zeigt einen neuen Datensatz mit der von uns angegebenen E-Mail-Adresse (siehe Bild 14).

Änderungen durch die Benutzerverwaltung

Durch das Auswählen der Benutzerverwaltung beim Anlegen des Projekts wurden noch einige weitere Elemente zum Projekt hinzugefügt. Da wären zunächst einige Anweisungen, die der Datei **Startup.cs** hinzugefügt wurden und die in der Methode Konstruktor-Methode **Startup** dafür sorgen, dass alle für die Benutzerverwaltung notwendigen Klassen initialisiert werden. Dazu gehören auch die Elemente, die für den Zugriff auf die Datenbank notwendig sind und die Sie schon aus den Artikeln kennen, die sich mit der Programmierung von Desktop-Anwendungen auf Basis des Datenbankzugriffs mittels Entity Framework beschäftigen. Wichtig ist auch, dass die Methode **Configure** nun eine Anweisung enthält, welche die Authentifizierung aktiviert:

```
app.UseAuthentication();
```

Die Datei **appsettings.json**, welche die Anwendungseinstellungen aufnimmt, wurde um die Verbindungszeichenfolge erweitert, die wir ja schon geändert haben.

Bestätigen eines Kontos

Wenn der Benutzer ein neues Konto anlegt, erstellt die Anwendung wie oben gezeigt einen neuen Datensatz in der Tabelle **dbo.AspNetUsers**. Wenn Sie diesen Datensatz im Detail ansehen, finden Sie im Feld **EmailConfirmed** den Wert **False**. Das bedeutet, dass das Benutzerkonto noch nicht bestätigt wurde. Heutzutage ist dies allerdings ein wichtiger Bestandteil, um sicherzustellen, dass das Konto auch tatsächlich durch den Benutzer erstellt wurde, dessen E-Mail-Adresse bei der Anmeldung verwendet wurde. Erst wenn dieser Benutzer eine E-Mail an seine Adresse geschickt bekommt und einen darin enthaltenen Bestätigungslink anklickt, wird das Feld **EMailConfirmed** auf den Wert **True** eingestellt. Dieses Verfahren nennt sich Double-Opt-In und stellt sicher, dass niemand anderes ein Konto unter Ihrer E-Mail-Adresse anlegen kann.

Als Erstes ändern wir nun eine Einstellung, die wir in der Methode **ConfigureServices** der Klasse **Startup.cs** zuweisen. Dabei legen wir für den Aufruf der **AddIdentity**-Methode den Wert von **config.SignIn.RequireConfirmedEMail** auf **true** fest (siehe ab Markierung **//Änderung Start**):

```
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
    //Änderung Start:
    services.AddIdentity<ApplicationUser, IdentityRole>(config => {
        config.SignIn.RequireConfirmedEmail = true;
    })
    //Änderung Ende
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
    services.AddMvc()
        .AddRazorPagesOptions(options => {
```

ASP.NET Core: Validierung

Genau wie in Desktop-Anwendung ist auch in Webanwendungen die Validierung der Benutzereingaben ein wichtiges Feature. Unter ASP.NET Core gibt es für die Razor Pages erfreulicherweise eine Standardvorgehensweise, die wir in diesem Artikel vorstellen werden.

Vorbereitung

Für das Nachvollziehen der Beispiele in diesem Artikel legen Sie in Visual Studio 2017 ein neues Projekt namens Validierung an. Es wird mit der Projektvorlage **Visual C#\Web\ASP.NET Core Webanwendung** erstellt. Im zweiten Vorlagendialog wählen Sie dann den Typ **Webanwendung** aus. Wir wollen die Seite **Pages/Customers/Create.cshtml** nutzen, um unsere Beispiele auszuprobieren. Diese wollen wir über einen neuen Menüeintrag der Webanwendung aufrufen, den wir in der Datei **_Layout.cs** einfügen, und zwar an dieser Stelle:

```
...
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li><a asp-page="/Index">Home</a></li>
        <li><a asp-page="/About">About</a></li>
        <li><a asp-page="/Customers/
Create">Neuer Kunde</a></li>
    </ul>
</div>
...
```

Insgesamt werden wir neben Änderung der Datei **_Layout.cs** noch zwei Dateien samt Verzeichnissen hinzufügen, nämlich **Models/Customer.cs** und **Pages/Customers/Create.cshtml** (siehe Bild 1).

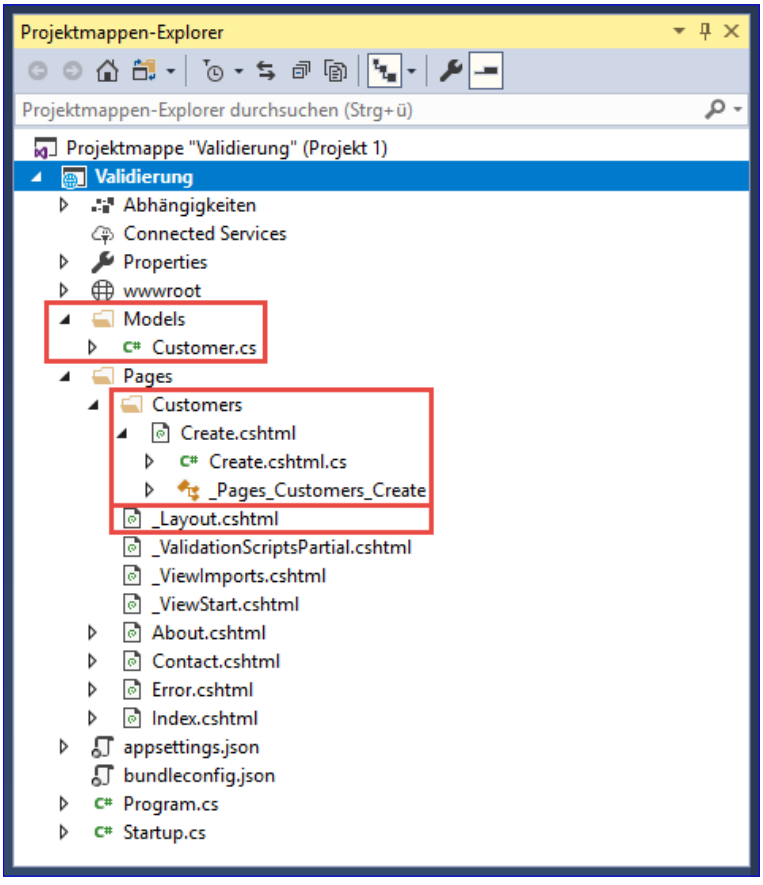

Bild 1: Anlage einiger Verzeichnisse und Dateien

Grundlagen

In Webanwendungen landen die Daten aus Tabellen zunächst in einem Objekt auf Basis einer Klasse, bevor diese auf einer Webseite dargestellt werden – diese Vorgehensweise kennen Sie ja auch schon von den Artikeln über den Einsatz des Entity Frameworks in Desktop-Anwendungen. Der Vorteil dieser Vorgehensweise ist, dass die Validierungsregeln nur an einer Stelle angelegt werden müssen und nicht etwa auf jeder Seite, welche die Eingabe oder Bearbeitung der Daten der betroffe-

nen Klasse erlauben soll. Die Validierungsregeln werden hier wie da direkt in den Klassen angelegt, welche die Tabellen der Datenbank repräsentieren. Dazu legen wir zu Beispielzwecken diesmal keine Klasse auf Basis einer Tabelle an, sondern eine einfache Klasse ohne Bindung an eine Datenbank.

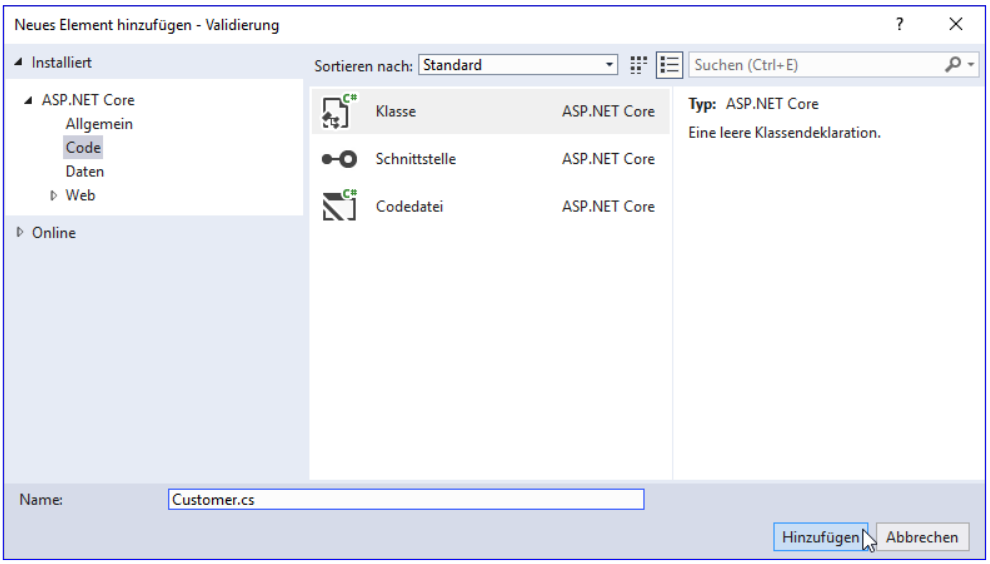

Bild 2: Hinzufügen einer Klasse zum Verwalten von Kunden

Zu Beispielzwecken legen wir nun eine Klasse namens **Customers.cs** an, und zwar in einem zuvor erstellten Verzeichnis namens Models direkt unterhalb des Projektordners. Dort wählen Sie dann den Kontextmenü-Eintrag **Hinzufügen|Klasse...** aus und geben im nun erscheinenden Dialog **Neues Element hinzufügen** den Namen für die Klasse ein – hier **Customer.cs** (siehe Bild 2).

Die Klassendatei mit der Klasse **Customer** ergänzen wir nun wie in folgendem Code um einige Beispielfelder. Wozu das Feld namens **Rating** dient? Es hat einen bestimmten Zweck – außer als Beispiel für ein **Decimal**-Feld ...

```
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 string Email { get; set; }
    public string Url{ get; set; }
    public string Phone { get; set; }
    public DateTime Birthday { get; set; }
    public decimal Rating { get; set; }
}
```

Wenn wir nun ein Formular auf Basis dieser Daten anlegen, sieht dies wie in gekürzter Form wie folgt aus:

```
@page
@model Validierung.Pages.Customers.CreateModel
```

```
@{ ViewData["Title"] = "Create"; }
<h2>Datensatz erstellen</h2>
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label class="control-label">Vorname:</label>
                <input asp-for="Customer.FirstName" class="form-control" />
                <span asp-validation-for="Customer.FirstName" class="text-danger"></span>
            </div>
            ...
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
```

Wir sehen hier im oberen Teil die Razor-Anweisungen, mit denen unter anderem die Klasse **Validierung.Pages.Customers.CreateModel** als Modell referenziert wird. Darunter folgen einige Elemente samt **form**-Element. Die für die Validierung wichtigen Elemente sind das Element mit dem Attribut **asp-validation-summary** über den übrigen Elementen sowie das Element mit dem Attribut **asp-validation-for** für jedes einzelne Steuerelement, das an eines der Felder der Modell-Klasse gebunden ist.

Bevor wir auf diese eingehen, schauen wir uns noch an, wie wir die Klasse mit ein paar Validierungen ausstatten und wie das **PageModel** und das Model für die Seite aussehen.

Model der HTML-Seite

Die Code behind-Datei enthält den folgenden Code, unter anderem mit dem **PageModel**:

```
... weitere Namespace-Referenzen
using Validierung.Models;

namespace Validierung.Pages.Customers {
    public class CreateModel : PageModel {
        public IActionResult OnGet() {
            return Page();
        }
        [BindProperty]
        public Customer Customer { get; set; }
        public IActionResult OnPost() {
```

```
            if (!ModelState.IsValid) {
                return Page();
            }
            return RedirectToPage("./Index");
        }
    }
}
```

Hier definieren wir einen Verweis auf das Verzeichnis Entitäten, also **Validierung.Models**. Dann fügen wir eine **OnGet**-Methode ein, welche beim Aufruf die auf Basis von **Create.cshtml** zusammengestellt Seite zurückliefert. Wir binden die Seite an die Klasse **Customer** als Model. Dazu versehen wir die öffentliche Variable **Customer** des Typs **Customer** mit dem Attribut **[BindProperty]**. Interessant ist die Methode **OnPost**, die letztlich die Validierung ausführt – und zwar durch die Methode **IsValid** des Objekts **ModelState**. Liefert dieses nicht das Ergebnis **True**, wird wieder die Seite mit dem zuvor ausgefüllten Formular zurückgeliefert. Erst wenn **IsValid** den Wert **True** liefert, zeigt die Webanwendung die dann vorgesehene Seite an – in diesem Fall die **Index**-Seite. Das Objekt **ModelState** ist übrigens ein Element des Namespaces **Microsoft.AspNetCore.MVC**. Wenn die Methode **IsValid** von **ModelState** aufgerufen wird, untersucht die Webanwendung, ob die im Model **Customer** festgelegten Validierungsregeln erfüllt sind. Diese müssen wir allerdings erst noch hinzufügen.

Validierungsregeln in das Model einfügen

Um dem Model, also der Klasse **Customer.cs**, die Validierungsregeln hinzuzufügen, müssen wir zunächst den Namespace **System.ComponentModel.DataAnnotations** hinzufügen:

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

Validierungsregel für Pflichtangaben

Die meistgenutzte Validierungsregel dürfte die sein, die prüft, ob erforderliche Werte angegeben wurden. Um ein Feld in der Klasse **Customer** mit dieser Validierungsregel zu belegen, fügen wir vor dem jeweiligen Feld die Data Annotation namens **Requery** ein – hier an einigen Beispielen zu sehen:

Bild 3: Validierungsmeldungen

```
public class Customer {
    public int ID { get; set; }
    [Required]
    public string FirstName { get; set; }
    [Required]
    public string LastName { get; set; }
    [Required]
    public string Street { get; set; }
    ...
}
```

Wenn wir die Anwendung nun starten, über den Menüeintrag **Neuer Kunde** zur Seite **Create.cshtml** wechseln und dort direkt auf die Schaltfläche mit der Beschriftung **Create** klicken, erhalten wir die Validierungsmeldungen aus Bild 3. Es klappt also alles wie gewünscht. Nun schauen wir uns weitere Möglichkeiten an.

Tag Helper für die Validierung

Bei **asp-validation-summary** und **asp-validation-for** handelt es sich um sogenannte Tag-Helper (mehr Tag-Helper stellen wir im Artikel **ASP.NET Core: Tag Helper** vor). Der Tag Helper **asp-validation-summary** wird in der Regel einmal oberhalb eines Formulars angegeben. Dort hat er die Aufgabe, eventuell auftretende Validierungshinweise auszugeben, die nicht nur ein spezielles Feld betreffen, sondern mehrere Felder oder allgemeine Hinweise. Für dieses Attribut gibt es drei mögliche Werte:

- **All**
- **ModelOnly**
- **None**

Für **asp-validation-summary** ist aktuell der Wert **ModelOnly** festgelegt. Wir könnten diesen Wert einmal ändern in **All** und schauen uns an, was geschieht, wenn wir das Formular erneut ohne Füllen eines der Felder absenden (siehe Bild 4):

```
<div asp-validation-summary="All" class="text-danger"></
div>
```

Es erscheint also schon vor dem Feld **Vorname** eine Übersicht aller fehlerhaft ausgefüllten Eingabefelder. Das ist ehrlich gesagt nicht notwendig, wenn schon jedes einzelne Feld mit einem entsprechenden Text ausgestattet wird. Leider sind

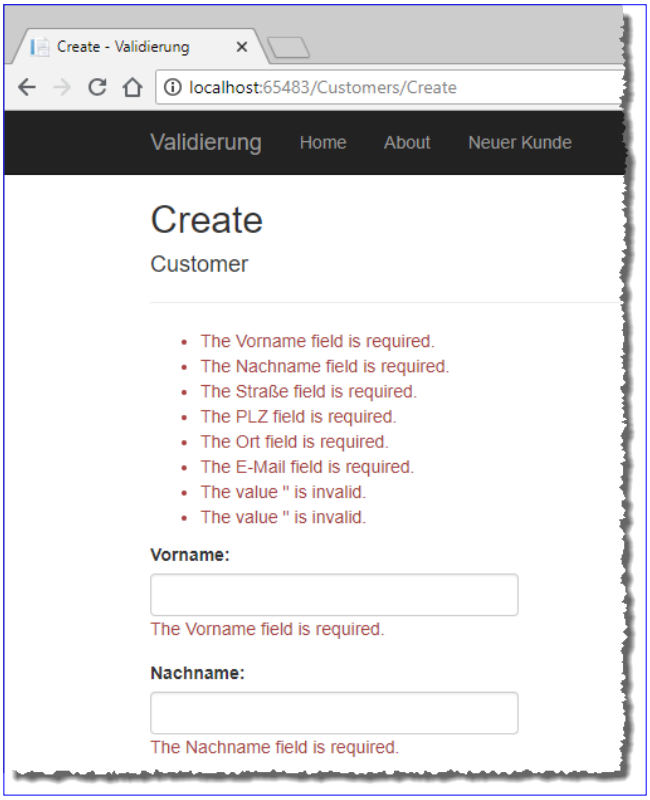

Bild 4: Validierungsmeldungen zusammengefasst im Kopfbereich

diese Texte auch noch in englischer Sprache – wir schauen uns also gleich an, wie Sie hier die deutschen Texte angeben können.

Wozu aber dient dann die Einstellung **ModelOnly**? Auch dieses hat seine Berechtigung, warum, sehen wir uns ebenfalls weiter unten an.

Validierungstexte in Deutsch

Die Validierungstexte können Sie direkt in den DataAnnotation-Elementen für die einzelnen Felder ergänzen, und zwar für das Element **Required**. Diesem fügen Sie in Klammern noch den Parameter **ErrorMessage** mit dem gewünschten Text hinzu.

Die Meldungen werden dann sowohl im Kontext mit den einzelnen Textfeldern als auch in der Übersicht der Validierungsmeldungen in der gewünschten Form ausgegeben (siehe Bild 5).

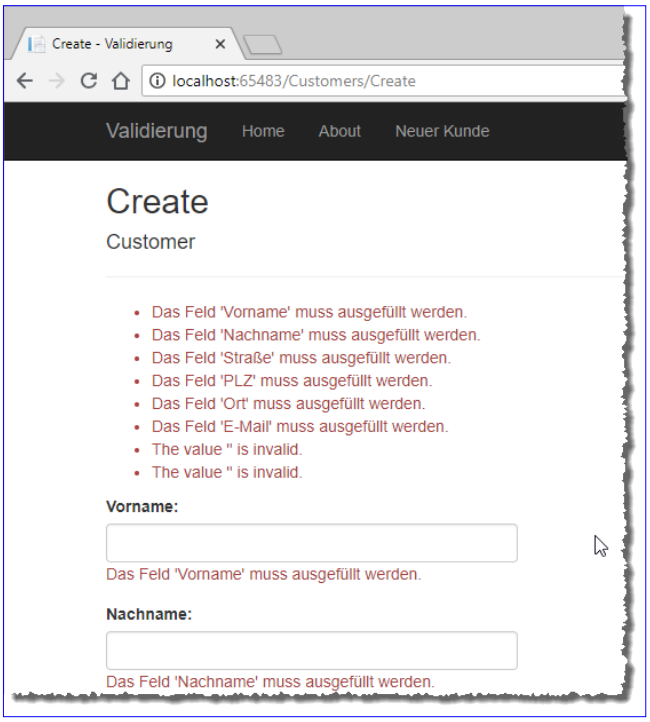

Bild 5: Deutsche Validierungsmeldungen

Das sieht dann etwa wie folgt aus:

```
public class Customer {
    public int ID { get; set; }
    [Required(ErrorMessage = "Das Feld 'Vorname' muss ausgefüllt werden.")]
    public string FirstName { get; set; }
    [Required(ErrorMessage = "Das Feld 'Nachname' muss ausgefüllt werden.")]
    public string LastName { get; set; }
    [Required(ErrorMessage = "Das Feld 'Straße' muss ausgefüllt werden.")]
    public string Street { get; set; }
    ...
}
```

Allerding sehen wir im Screenshot ganz unten in der Liste der Validierungsfehler noch zwei Mal den Inhalt **The value '' is invalid**. Was bedeutet dies? Es gibt hier anscheinend Validierungsregeln, die unsere angegebene Regel namens **Required** außer Kraft setzen. Das ist bei dem Datumsfeld **Geburtsdatum** und beim **Decimal**-Feld **Rating** der Fall.

Weitere Validierungsmöglichkeiten

Nachfolgend schauen wir uns die folgenden Validierungsmöglichkeiten an:

- **MaxLength(<Länge>)**: Die Zeichenkette darf maximal **<Länge>** Zeichen lang sein.

- **MinLength(<Länge>)**: Die Zeichenkette muss mindestens die in **<Länge>** angegebene Anzahl Zeichen aufweisen.
- **StringLength(...)**: Eingrenzen der maximalen und minimalen Länge einer Zeichenkette in einer DataAnnotation
- **EmailAddress**: Die Eingabe muss das Format einer E-Mail aufweisen.
- **Phone**: Die Eingabe muss das Format einer Telefonnummer aufweisen.
- **Url**: Die Eingabe muss wie eine URL formatiert sein.

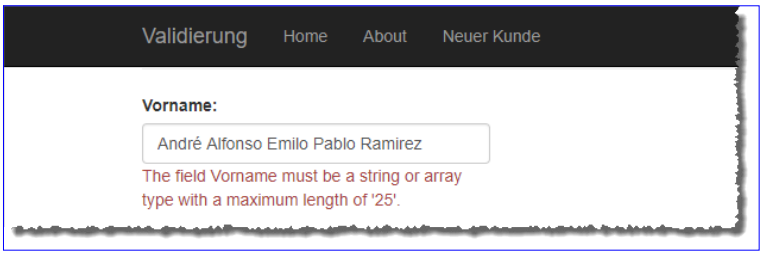

Bild 6: Validierung auf eine maximale Zeichenlänge

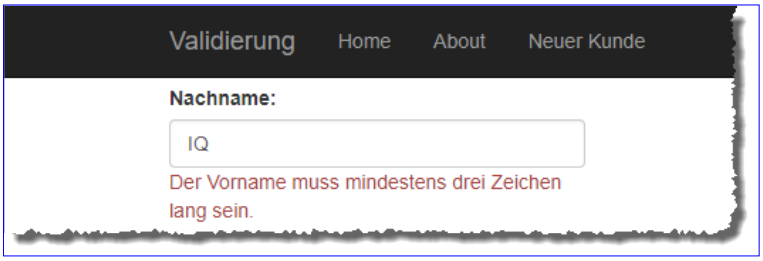

Bild 7: Validierung auf eine minimale Zeichenlänge

- **CreditCard**: Die Eingabe muss wie eine Kreditkartennummer formatiert sein.
- **DataType(<Datentyp>)**: Prüft, ob die Eingabe einem bestimmten Datentyp entspricht, zum Beispiel für **Datatype.Date**
- **Range(<Startwert>, <Endwert>)**: Bereich für Zahlenwerte
- **RegularExpression("<Regulärer Ausdruck>")**: Regulärer Ausdruck, der festlegt, wie die Eingabe aussehen soll
- **CustomValidation**: Benutzerdefinierte Validierung

Validierung einer Zeichenkette auf maximale Länge

Zu Testzwecken legen wir für das Feld **FirstName** die DataAnnotation **MaxLength** mit einem Wert von **25** fest:

```
public class Customer {
    ...
    [MaxLength(25)]
    [Required(ErrorMessage = "Das Feld 'Vorname' muss ausgefüllt werden.")]
    public string FirstName { get; set; }
    ...
}
```

Das Ergebnis sieht bei Eingabe von Texten größer als 25 Zeichen wie in Bild 6 aus.

Auch hier würden wir uns eine deutschsprachige Meldung wünschen. Kein Problem – diese fügen wir wie folgt hinzu:

ASP.NET Core: Anwendung veröffentlichen

Die ganzen Themen rund um die Entwicklung von Webanwendungen sind natürlich nutzlos, wenn Sie die Anwendung dann über das Internet nutzen können. Also schauen wir uns in diesem Beitrag an, wie wir unsere mit Datenbankanbindung versehene Webanwendung aus dem Artikel »Razor Pages mit Datenbankanbindung« ins Internet bringen.

Voraussetzungen

Um eine ASP.NET Core-Webseite online zu stellen, benötigen Sie zunächst ein Microsoft Azure-Konto. Dieses können Sie hier anlegen:

```
https://azure.microsoft.com/de-de/free/
```

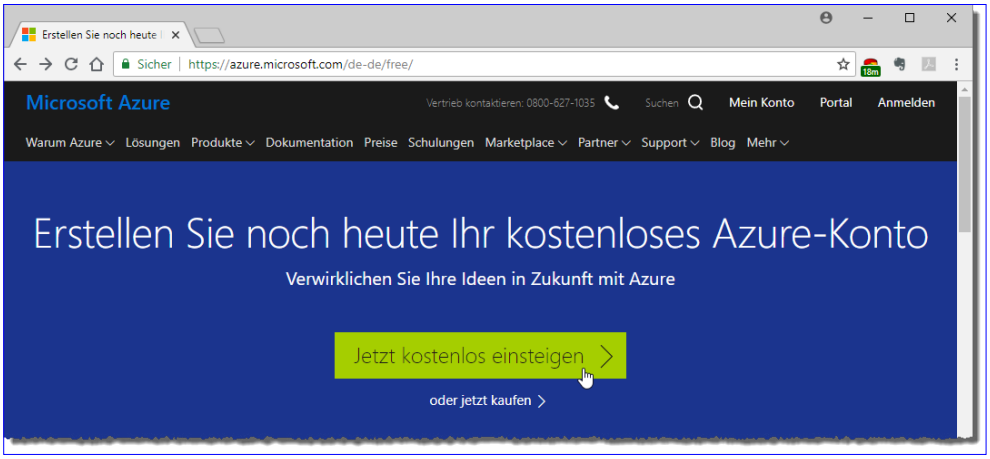

Bild 1: Erstellen eines Azure-Kontos

Dazu klicken Sie auf dieser Seite auf den Link **Jetzt kostenlos einsteigen** (siehe Bild 1). Danach melden Sie sich zunächst mit Ihrem Windows-Konto an, das Sie ja auch schon zur Installation von Visual Studio 2017 verwendet haben dürften.

Veröffentlichen des Projekts

Nachdem diese Schritte abgeschlossen sind, klicken Sie im Projektmappen-Explorer auf den Kontextmenü-Eintrag **Veröffentlichen...** des Projekts (siehe Bild 2).

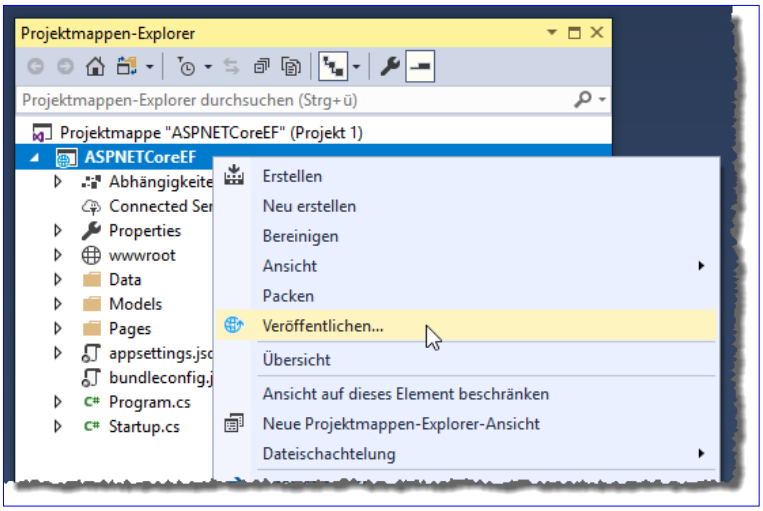

Bild 2: Veröffentlichen einer Webanwendung

Es erscheint der Dialog aus Bild 3, in dem Sie als Veröffentlichungsziel den Eintrag **App Service beibehalten** und rechts unter **Azure App Service** die Option **Neues Element erstellen** übernehmen. Dann klicken Sie auf die Schaltfläche **Veröffentlichen**.

Der folgende Dialog fragt, ob Sie ein kostenloses Azure Konto erstellen oder ein vorhandenes Konto nutzen wollen (siehe Bild 4).

Standardmäßig haben Sie, auch wenn sie bereits ein Windows-Konto besitzen, über welches Sie

auch Visual Studio 2017 heruntergeladen haben, noch kein Azure-Konto. Also legen wir eines über den Link **Kostenloses Azure-Konto erstellen** an.

Hier wählen Sie das bestehende Microsoft-Konto aus oder legen ein neues an und erstellen dann darauf basierend ein neues Azure-Konto. Hier erfolgen einige Sicherheitsabfragen, SMS, Eingaben von Sicherheitscodes et cetera bis Sie auf Ihr Konto zugreifen können.

Außerdem müssen Sie eine Kreditkartennummer angeben. Nach vollständiger Einrichtung werden Sie schließlich auf die Übersichtsseite Ihres neuen Azure-Kontos weitergeleitet (siehe Bild 5).

Damit sind die Arbeiten an dieser Stelle beendet und wir kehren wieder zu Visual Studio zurück.

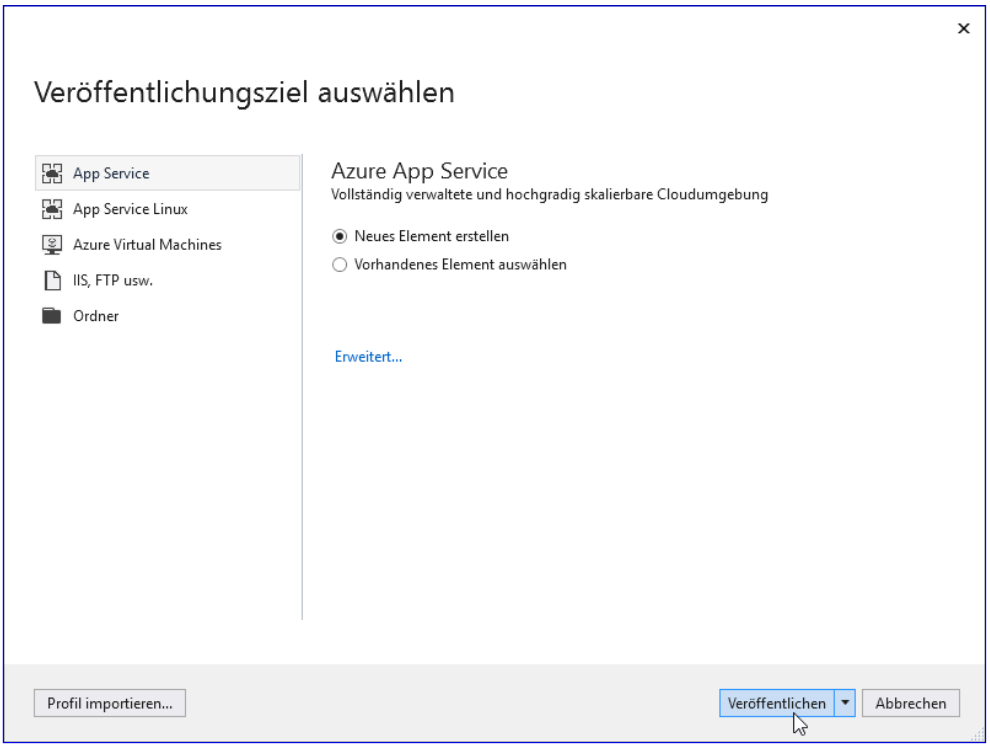

Bild 3: Veröffentlichungsziel auswählen

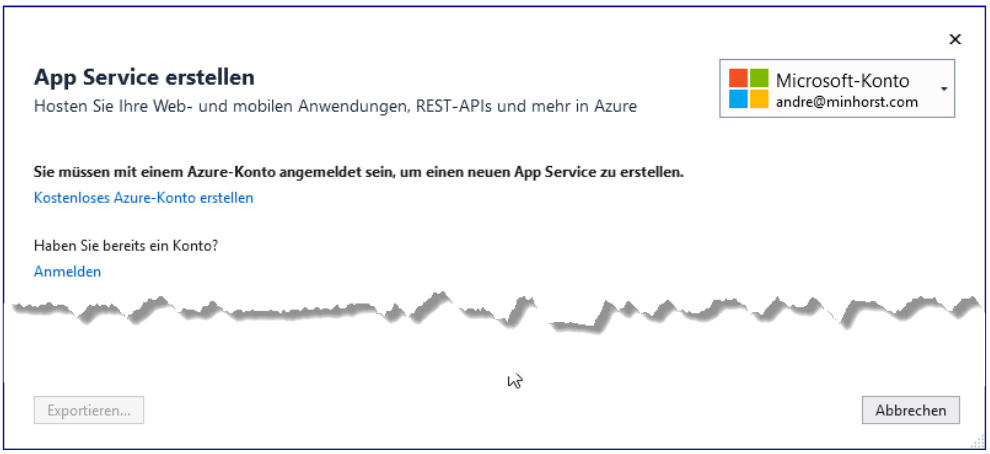

Bild 4: Azure-Konto erstellen oder an bestehendes Konto anmelden

App Service erstellen

Hier geht es nun mit dem Dialog aus Bild 6 weiter. Dieser zeigt automatisch Voreinstellungen für den neu zu erstellenden App-Service an, sobald Sie oben rechts ein Microsoft-Konto ausgewählt haben, das auch mit einem Azure-Konto verknüpft ist. Möglicherweise müssen Sie das Konto nochmals auswählen, damit dies geschieht. Hier sehen Sie nun einige Voreinstellungen, die wir zur Vereinfachung etwas anpassen:

- **App-Name**: Hier entfernen wir die Zahlen, damit wir einen griffigeren App-Namen erhalten.
- **Subscription**: Diesen Parameter stellen wir gleich über einen weiteren Parameter ein.

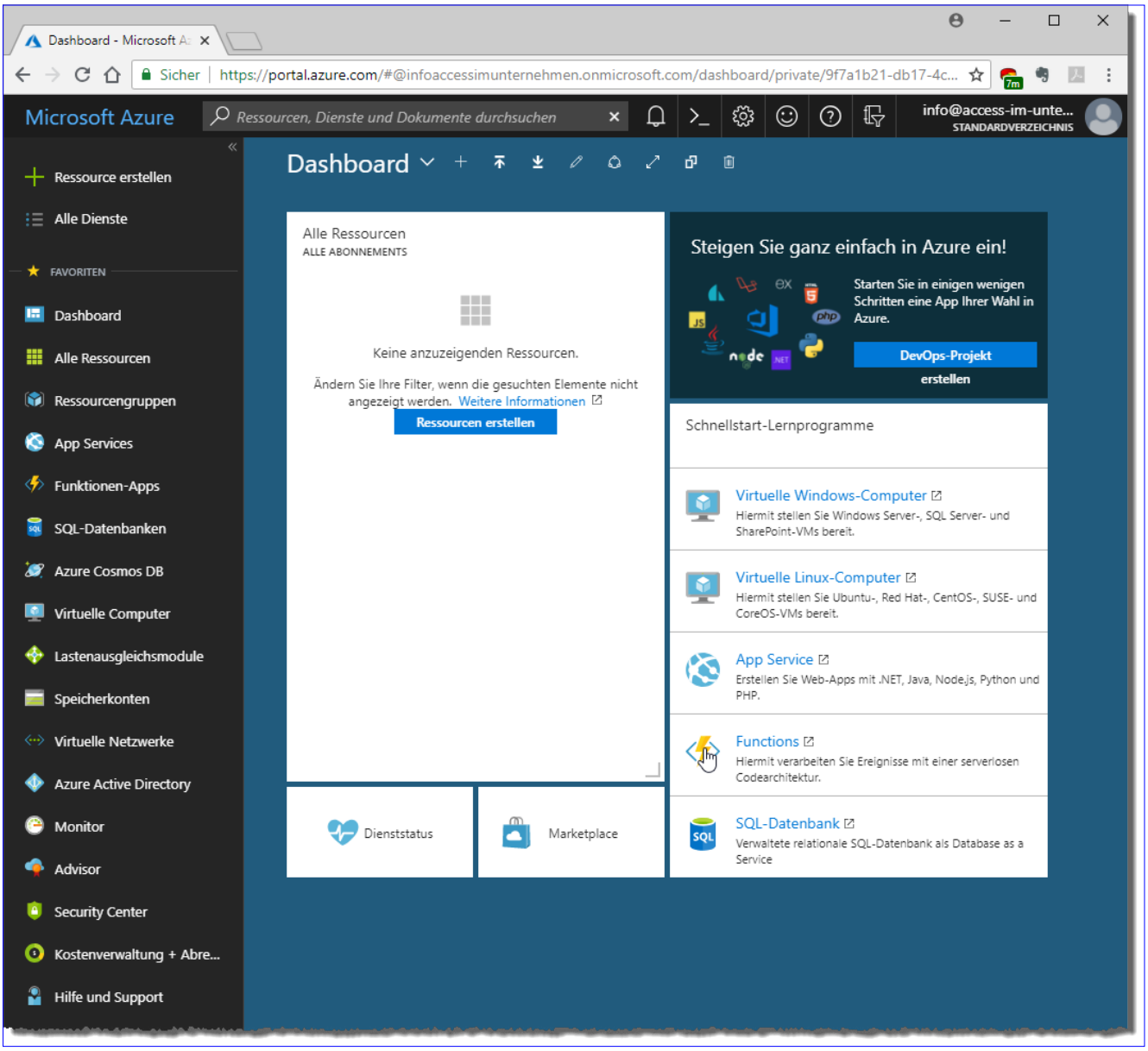

Bild 5: Ihr neues Azure-Konto

- **Ressourcengruppe**: Klicken Sie hier auf **Neu...**, um den Namen der Ressourcengruppe anzupassen.

- **Hostingplan**: Klicken Sie hier auf **Neu...**, erscheint der Dialog aus Bild 7. Hier geben Sie ebenfalls einen griffigeren Namen für das Feld **App Service Plan** ein, legen den Speicherort auf **West Europe** fest und wählen unter Größe vorsichtshalber den Eintrag **Kostenlos** aus. Dies passt dann auch den Wert des Parameters **Subscription** an.

Datenbank hinzufügen

Für diese Anwendung benötigen wir außerdem eine Datenbank auf dem Internetserver. Deshalb klicken Sie nun unter **Weitere Azure-Services durchsuchen** auf den Eintrag **SQL-Datenbank erstellen**. Wir haben aber noch keinen SQL Server angelegt.

Im nun erscheinenden Dialog **SQL-Datenbank** konfigurieren finden Sie daher neben der Einstellung **SQL Server** einen Link namens **Neu...**, den Sie nun betätigen.

Es erscheint der Dialog aus Bild 8. Hier geben Sie den Namen der Datenbank, einen Administratornamen und das Kennwort ein.

Nach dem Betätigen der **OK**-Schaltfläche erscheint nochmal die Seite **SQL-Datenbank konfigurieren**, in dem der neu angelegt SQL Server ausgewählt ist (siehe Bild 9). Wenn Sie später weitere Datenbanken für Webanwendungen anlegen, können Sie den nun erstellten SQL Server erneut auswählen.

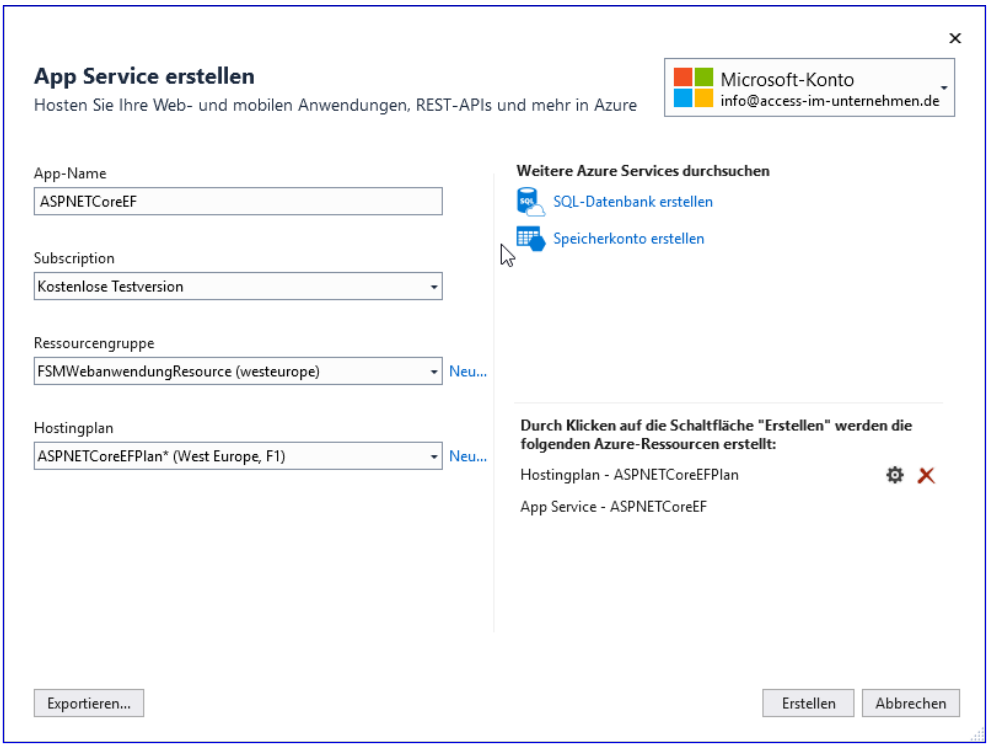

Bild 6: Erstellen des App-Services

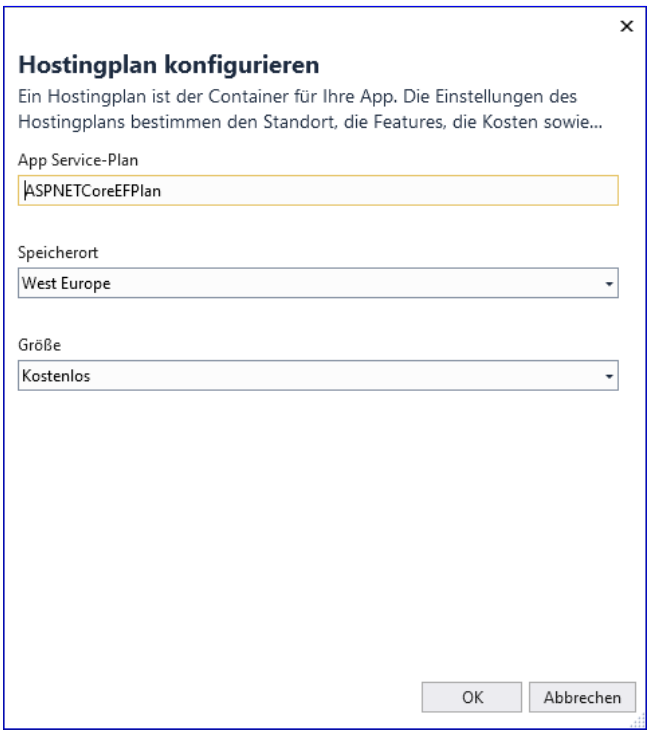

Bild 7: Einstellungen für den Hostingplan

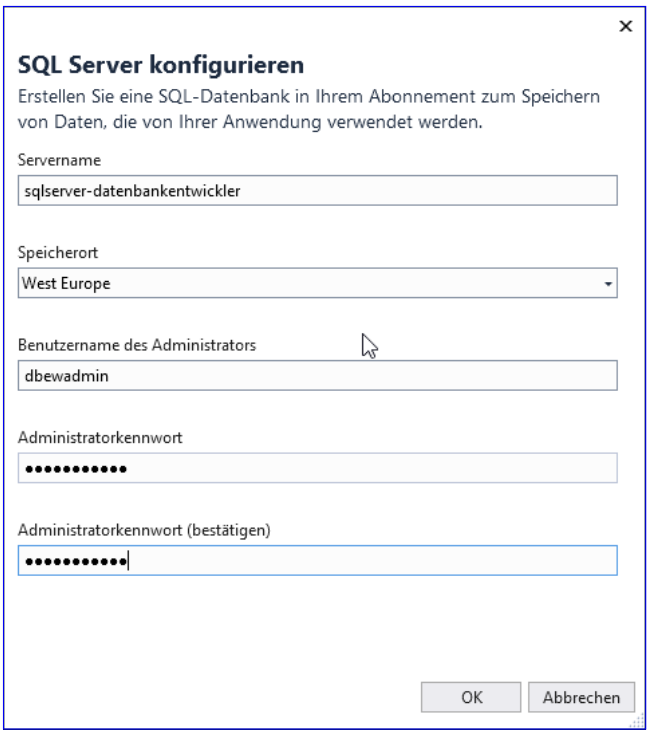

Bild 8: Angaben zum Erstellen der Datenbank