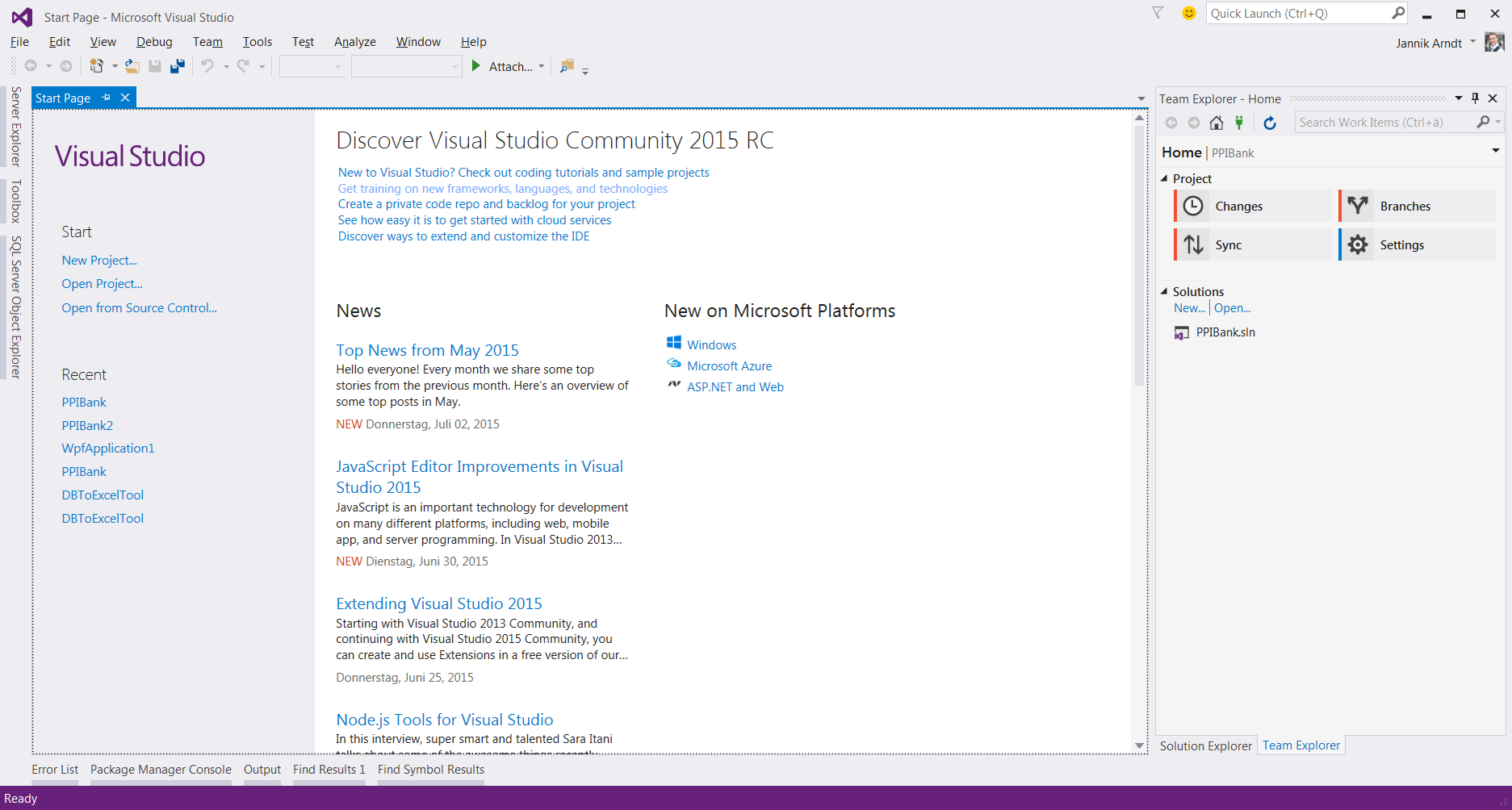
Beispielprojekt „MyBank“

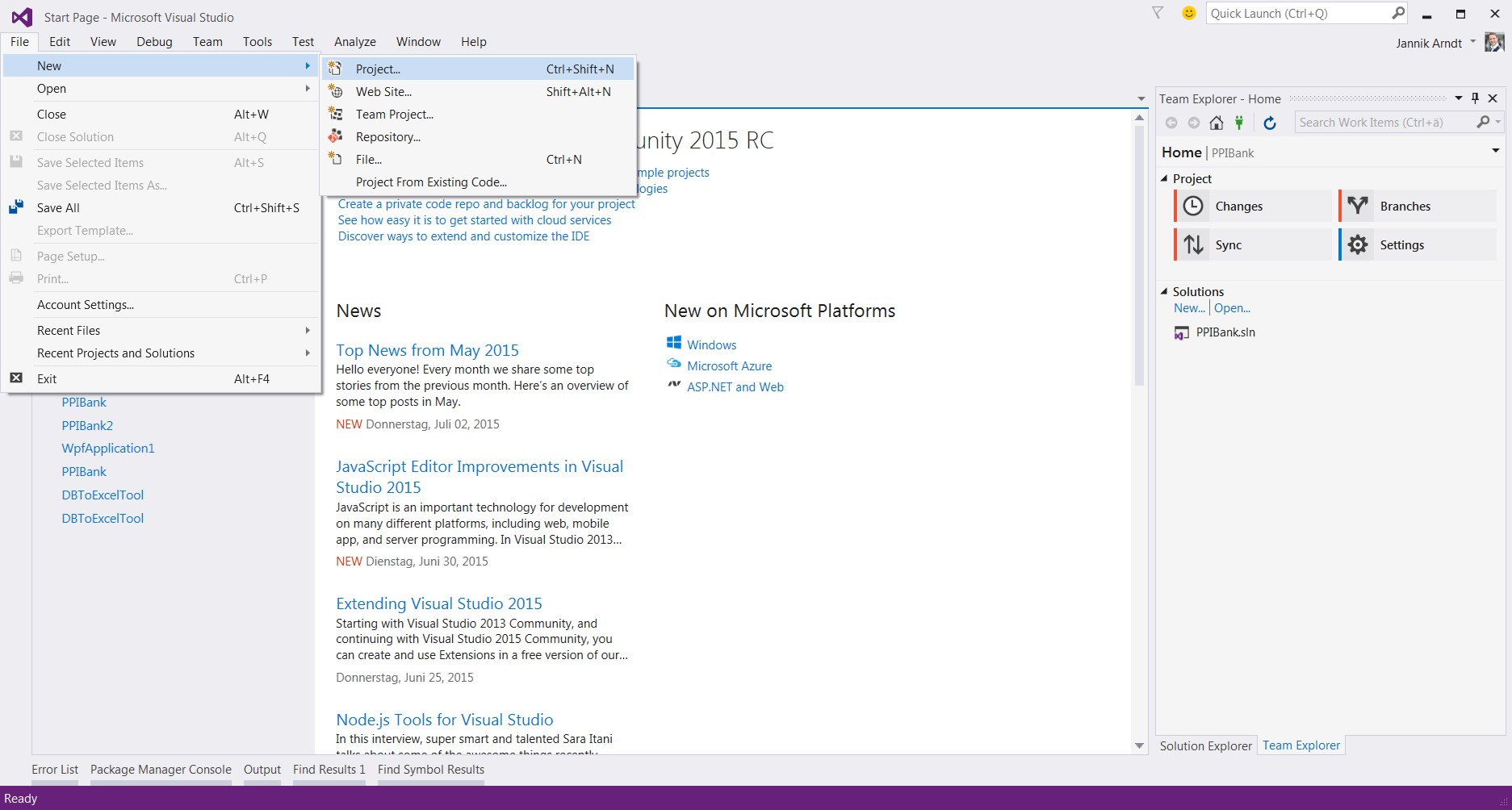
ASP.NET MVC 5 mit Entity Framework 6

# Projekt erstellen

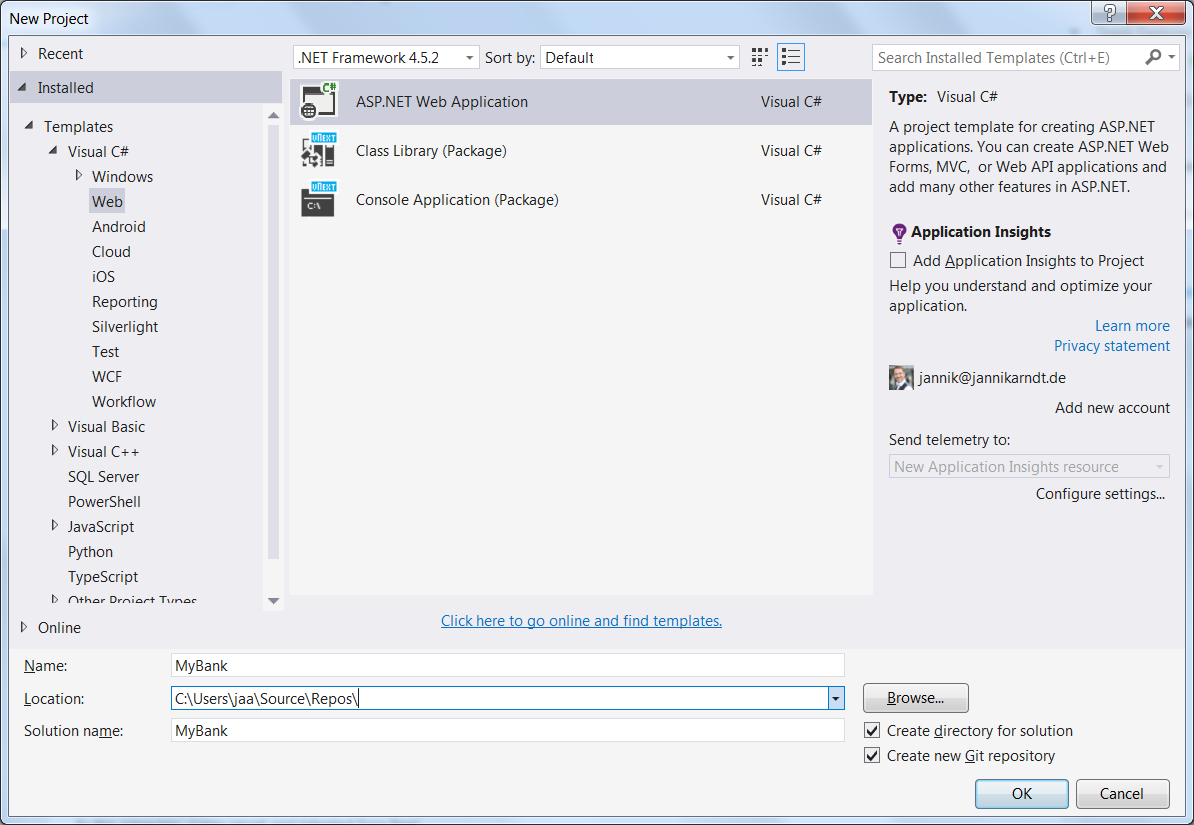
Visual Studio 2015 öffnen:



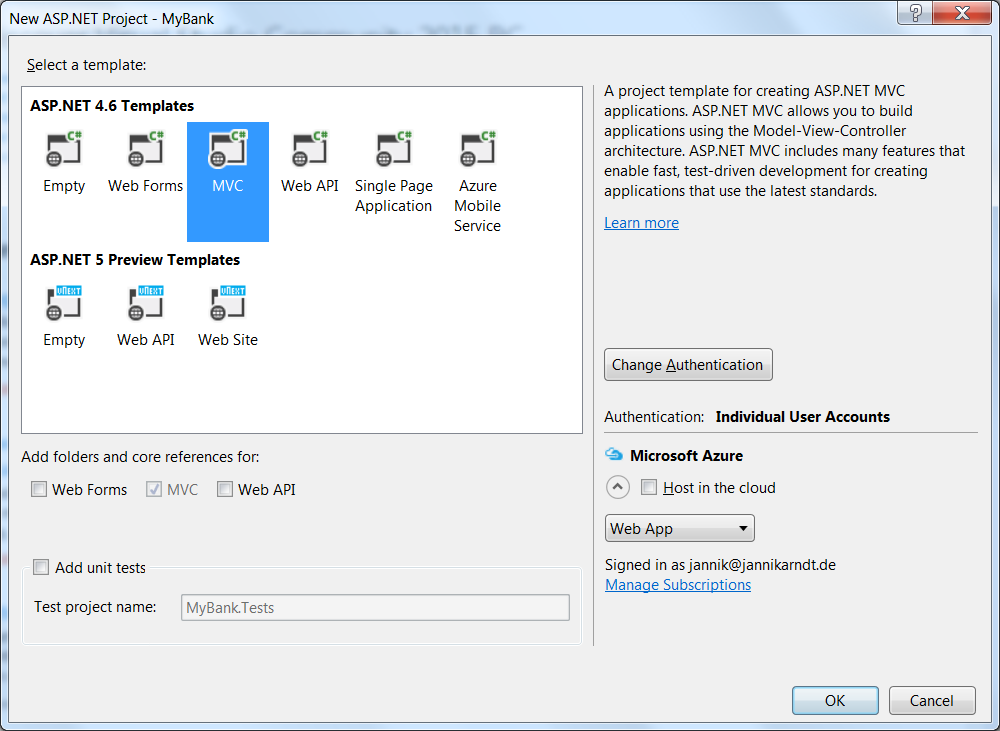
Neues Projekt erstellen:



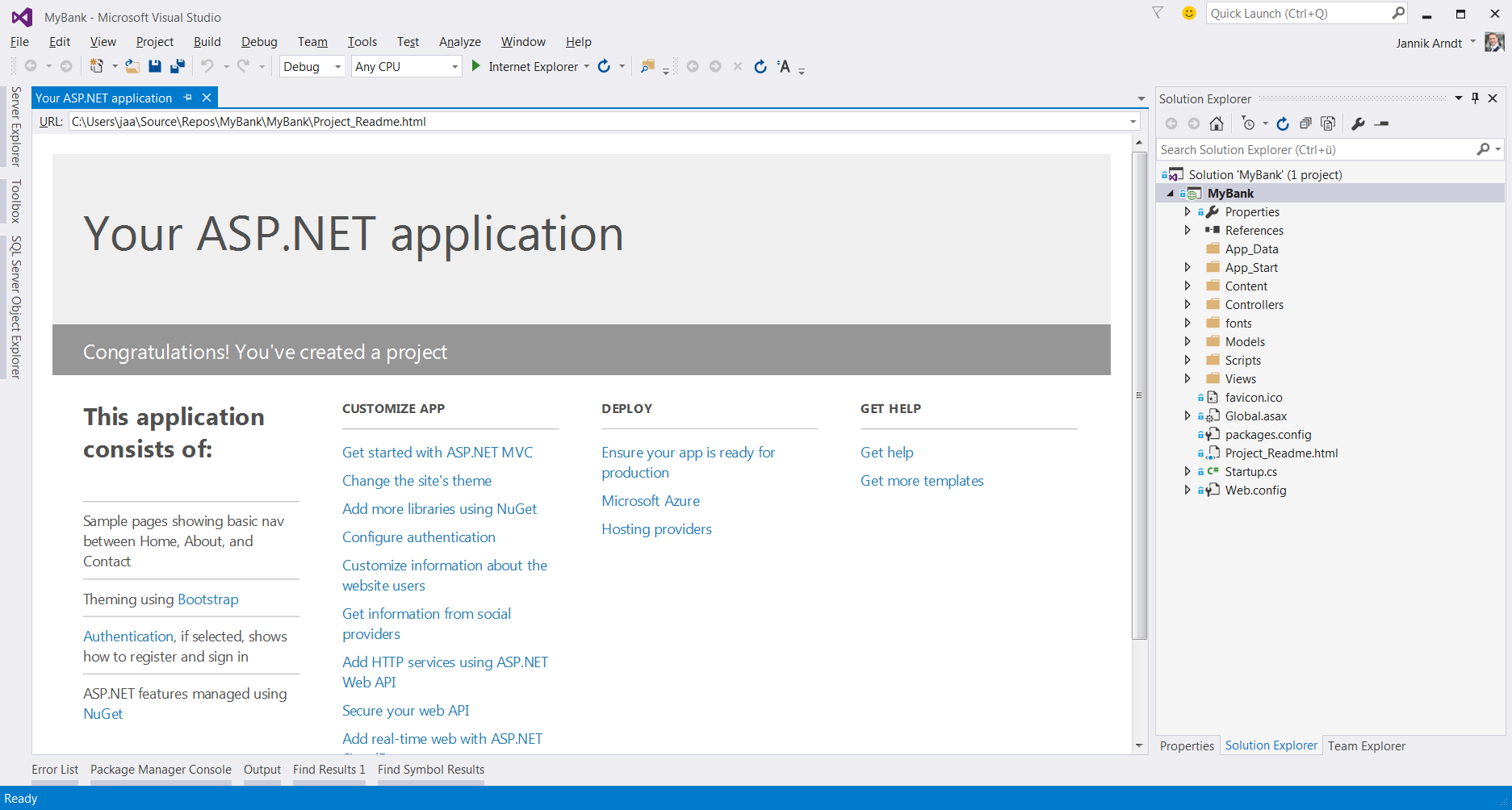
ASP.NET Web Application auswählen und Namen eingeben:



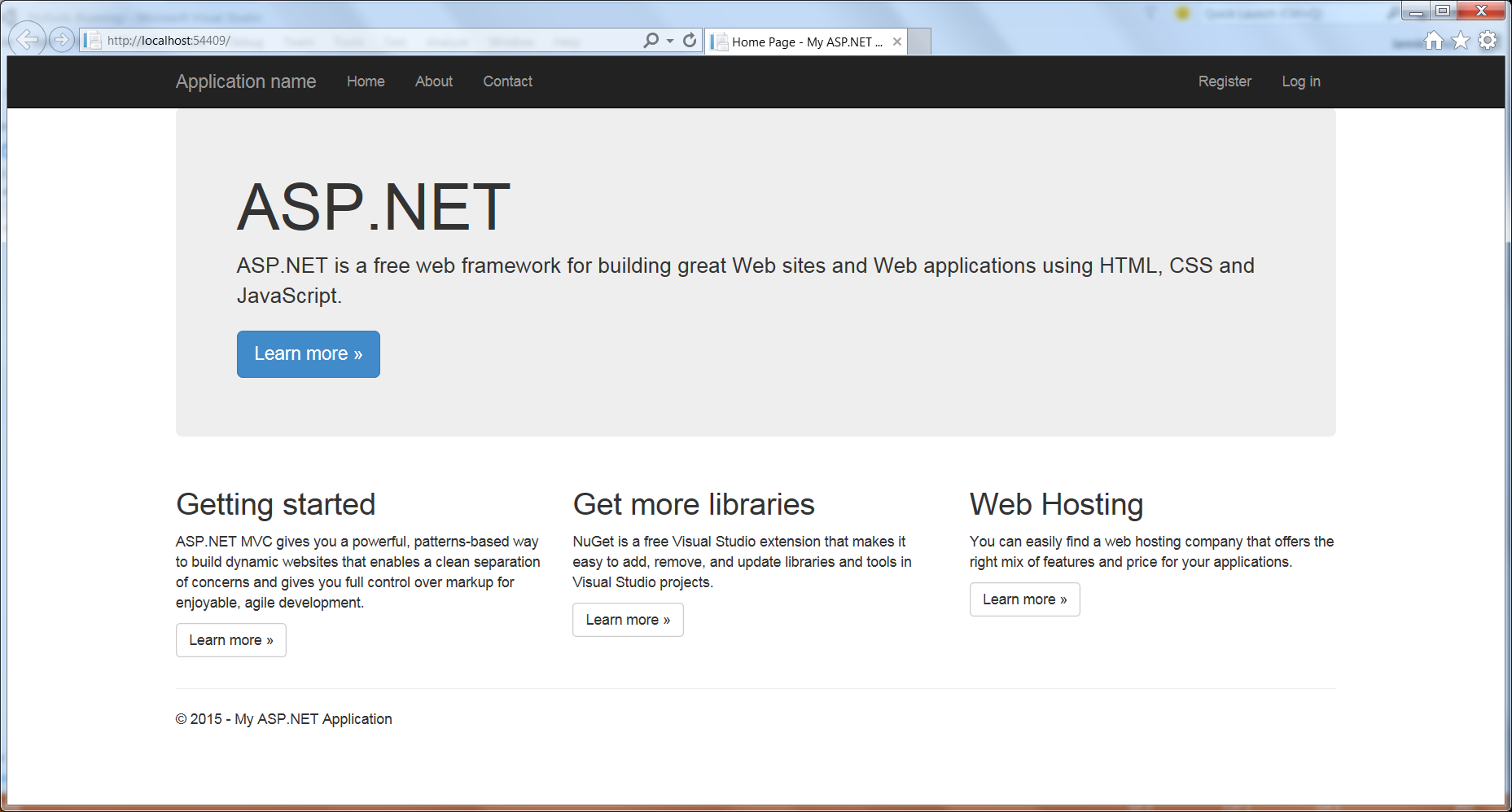
MVC Template auswählen (Authentication aus Individual User Accounts):



Es ist ein leeres Projekt entstanden:

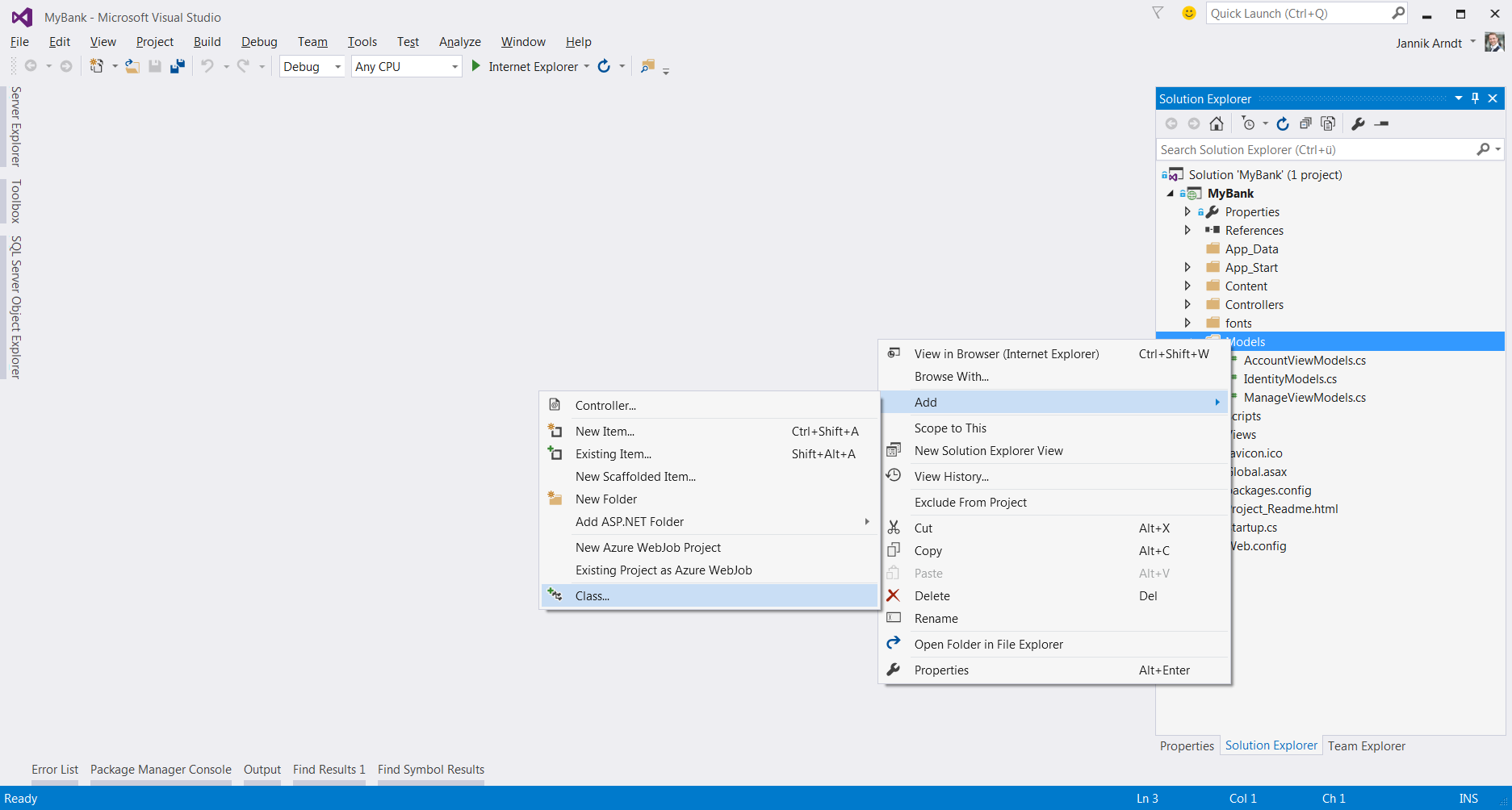


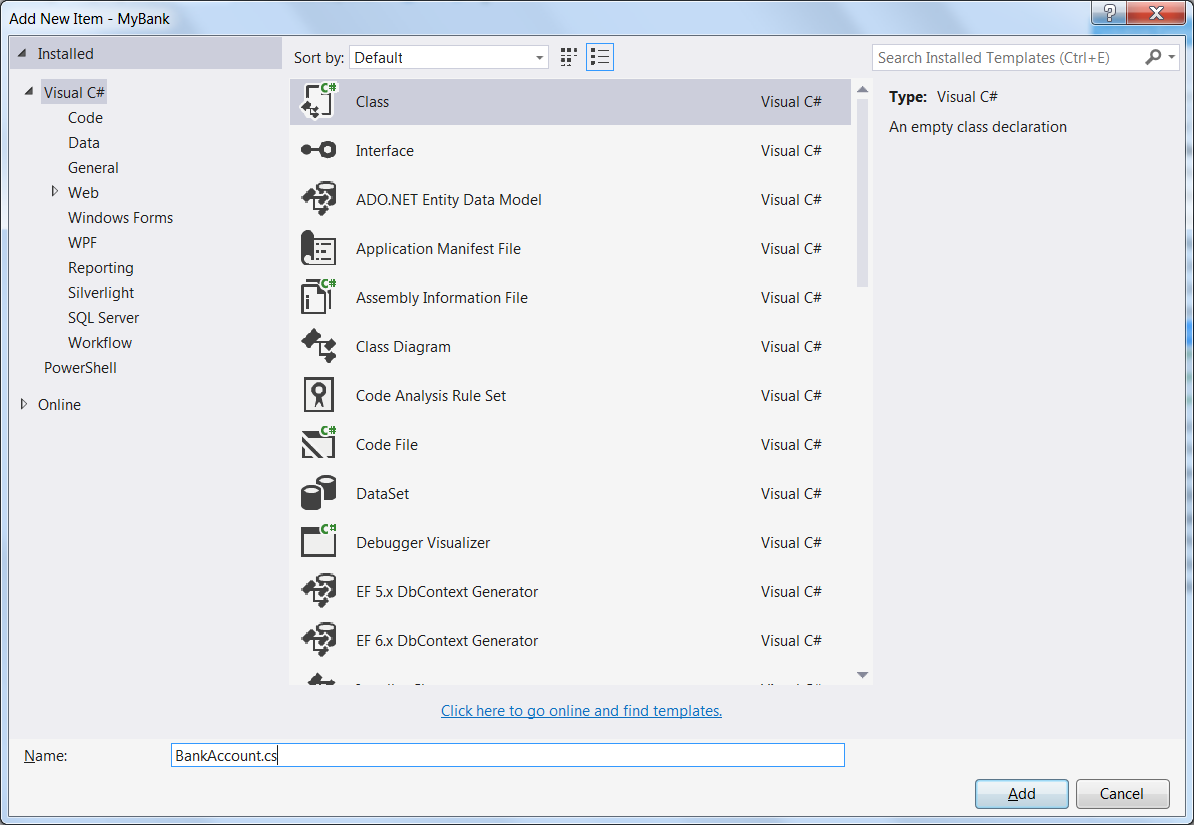
Mit einem Klick auf „Internet Explorer“ kann das schon ausgeführt werden:



# Model-Klassen erstellen

Debuggen beenden (Shift + F5 oder rotes Quadrat) und eine neue Datei in den Ordner Models einfügen:





Inhalt der Klasse:

namespace MyBank.Models

{

/// <summary>

/// Klasse für ein Konto

/// </summary>

public class BankAccount

{

/// <summary>

/// Die interne Datenbank-ID

/// </summary>

public int BankAccountId { get; set; }

/// <summary>

/// Die öffentliche Kontonummer

/// </summary>

public int Number { get; set; }

private double \_balance = 0;

/// <summary>

/// Das Guthaben auf dem Konto

/// </summary>

public double Balance { get { return \_balance; } internal set { \_balance = value; } }

/// <summary>

/// Die ID des Benutzers, dem das Konto gehört

/// </summary>

public string OwnerId { get; set; }

/// <summary>

/// Das Objekt des Besitzers, wird vom EF über die ID automatisch zugeordnet

/// </summary>

public virtual ApplicationUser Owner { get; set; }

/// <summary>

/// Ein leerer Konstruktor für die Erstellung aus der Datenbank / Deserialisierung

/// </summary>

public BankAccount()

{

}

/// <summary>

/// Standardkonstruktor

/// </summary>

/// <param name="ownerId"></param>

/// <param name="number"></param>

public BankAccount(string ownerId, int number)

{

OwnerId = ownerId;

Balance = 0;

Number = number;

}

/// <summary>

/// Methode um Geld abzuheben

/// </summary>

/// <param name="amount">Die Summe der Abhebung</param>

/// <returns>True bei Erfolg, False falls die Deckung nicht ausreicht</returns>

public bool Withdraw(double amount)

{

if (amount > Balance)

return false;

Balance -= amount;

return true;

}

/// <summary>

/// Methode zum Einzahlen

/// </summary>

/// <param name="amount"></param>

public void PayIn(double amount)

{

Balance += amount;

}

/// <summary>

/// Methode zur Überweisung von Geld auf ein anderes Konto

/// </summary>

/// <param name="amount">Die Summer der Überweisung</param>

/// <param name="otherAccount">Die (öffentliche) Kontonummer des anderen Accounts</param>

/// <returns>True bei Erfolg, False falls die Deckung nicht ausreicht</returns>

public bool Transfer(double amount, BankAccount otherAccount)

{

if (amount > Balance)

return false;

Balance -= amount;

otherAccount.PayIn(amount);

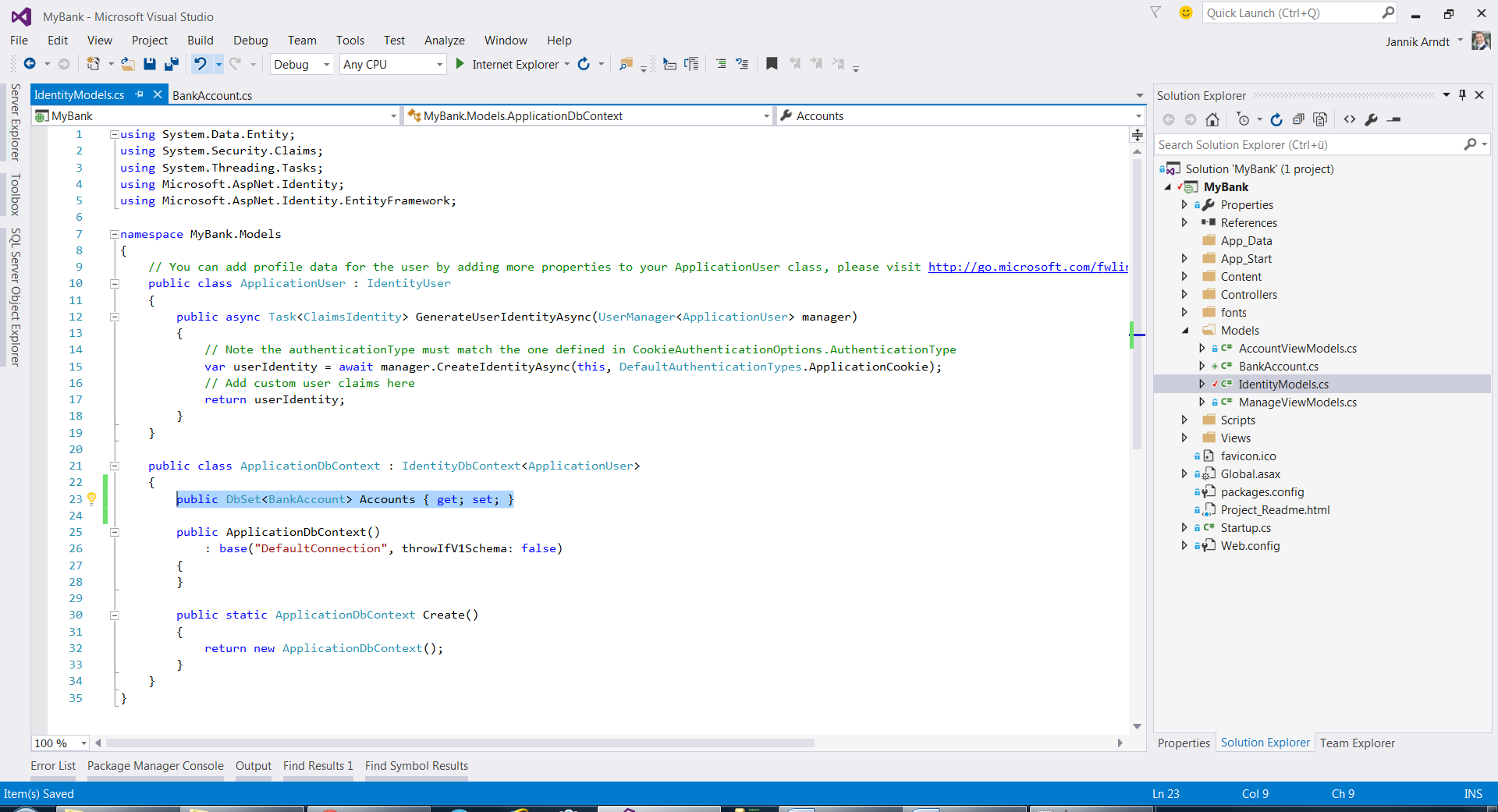
return true;

}

}

}

Einfügen der Tabelle in den Datenbank-Kontext (hier in IdentityModels.cs):



Anpassung des HomeControllers (Back-End für den View):

1. Klassenvariablen und Konstruktor:

private ApplicationDbContext \_bank { get; set; }

private ApplicationUserManager \_userManager { get { return HttpContext.GetOwinContext().GetUserManager<ApplicationUserManager>(); } }

public HomeController()

{

\_bank = new ApplicationDbContext();

}

1. Index-Methode:

[Authorize]

public ActionResult Index()

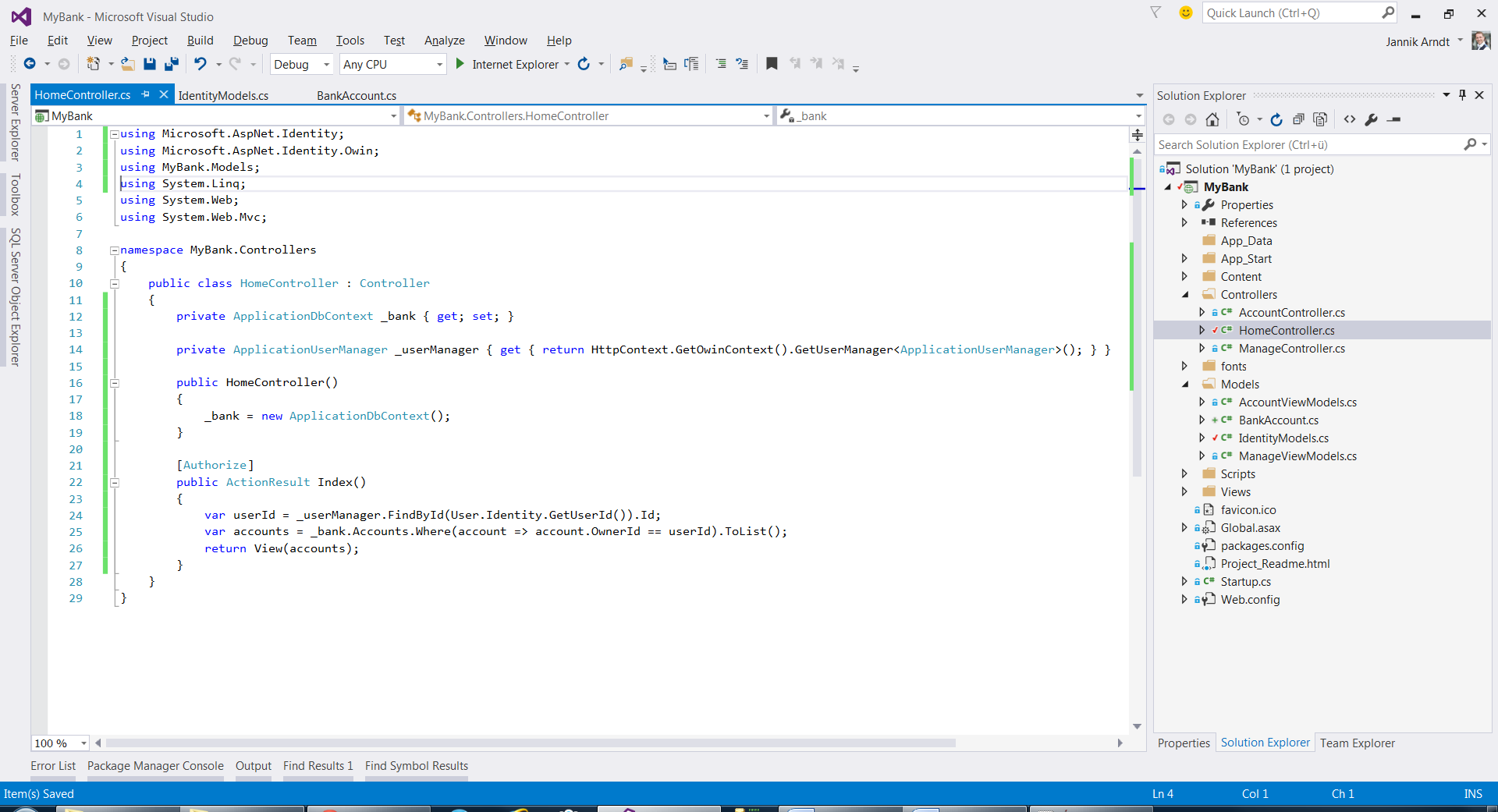
{

var userId = \_userManager.FindById(User.Identity.GetUserId()).Id;

var accounts = \_bank.Accounts.Where(account => account.OwnerId == userId).ToList();

return View(accounts);

}



Anpassung der View (View > Home > Index.cshtml):

@model List<MyBank.Models.BankAccount>

<h2>Meine Konten</h2>

<table class="table">

<tr>

<td>Kontonummer</td>

<td>Guthaben</td>

<td>Aktion</td>

</tr>

@foreach (var bankAccount in Model)

{

<tr>

<td>@bankAccount.Number</td>

<td>@bankAccount.Balance €</td>

<td>

@Html.ActionLink("Abheben", "Withdraw", new { account = bankAccount.Number }, new { @class = "btn btn-default" })

@Html.ActionLink("Einzahlen", "PayIn", new { account = bankAccount.Number }, new { @class = "btn btn-default" })

@Html.ActionLink("Überweisen", "Transfer", new { account = bankAccount.Number }, new { @class = "btn btn-default" })

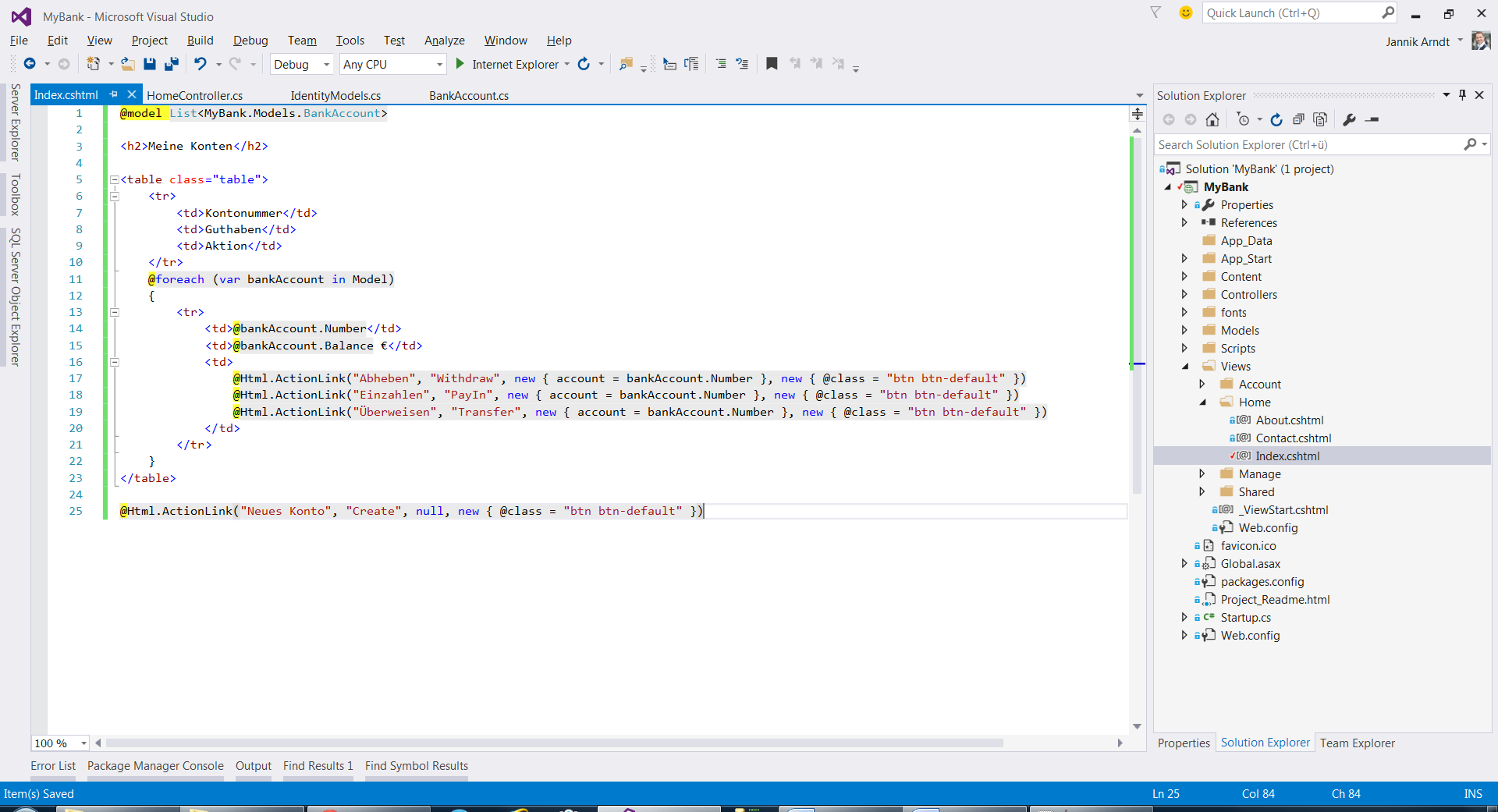
</td>

</tr>

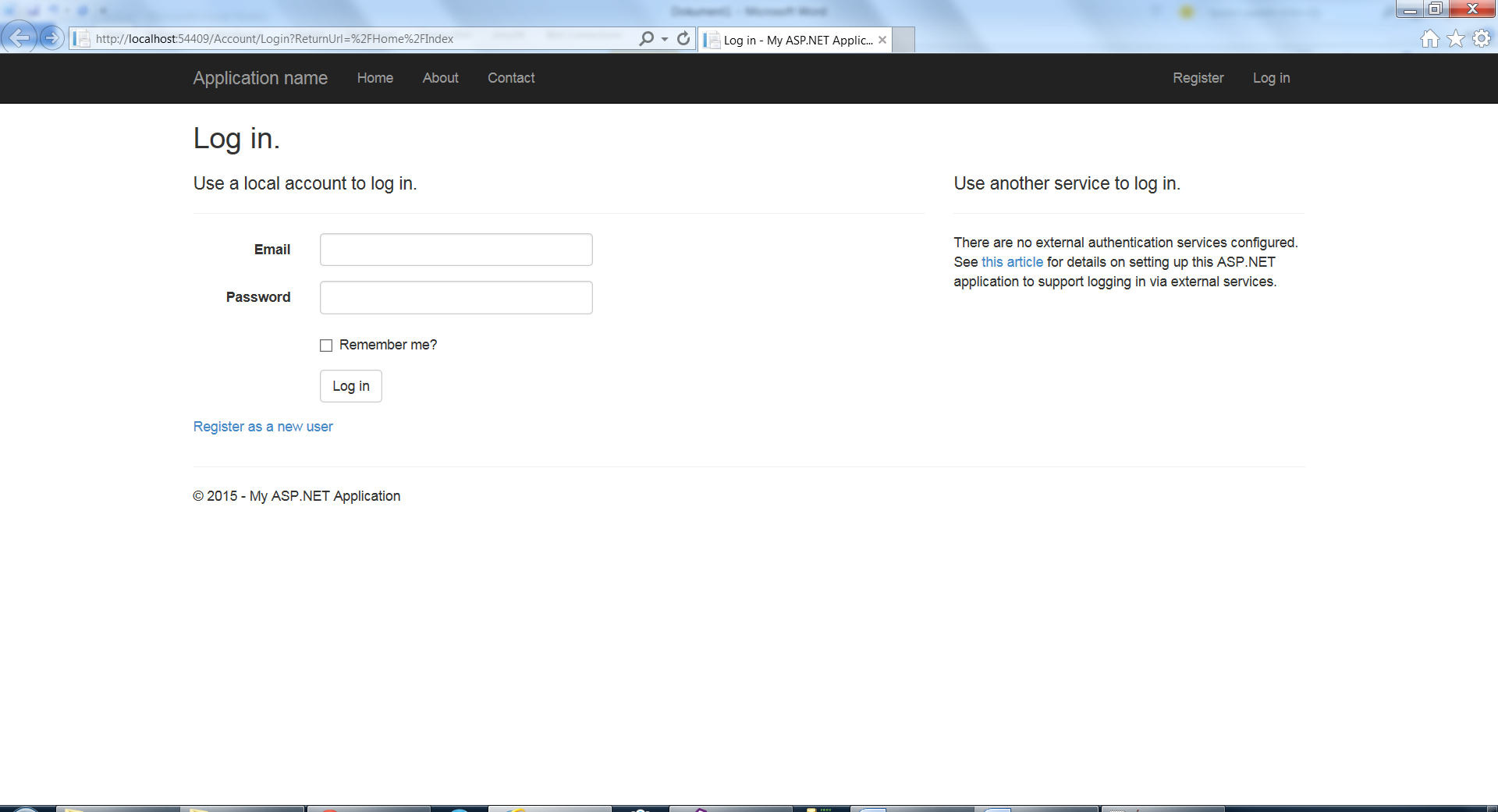
}

</table>

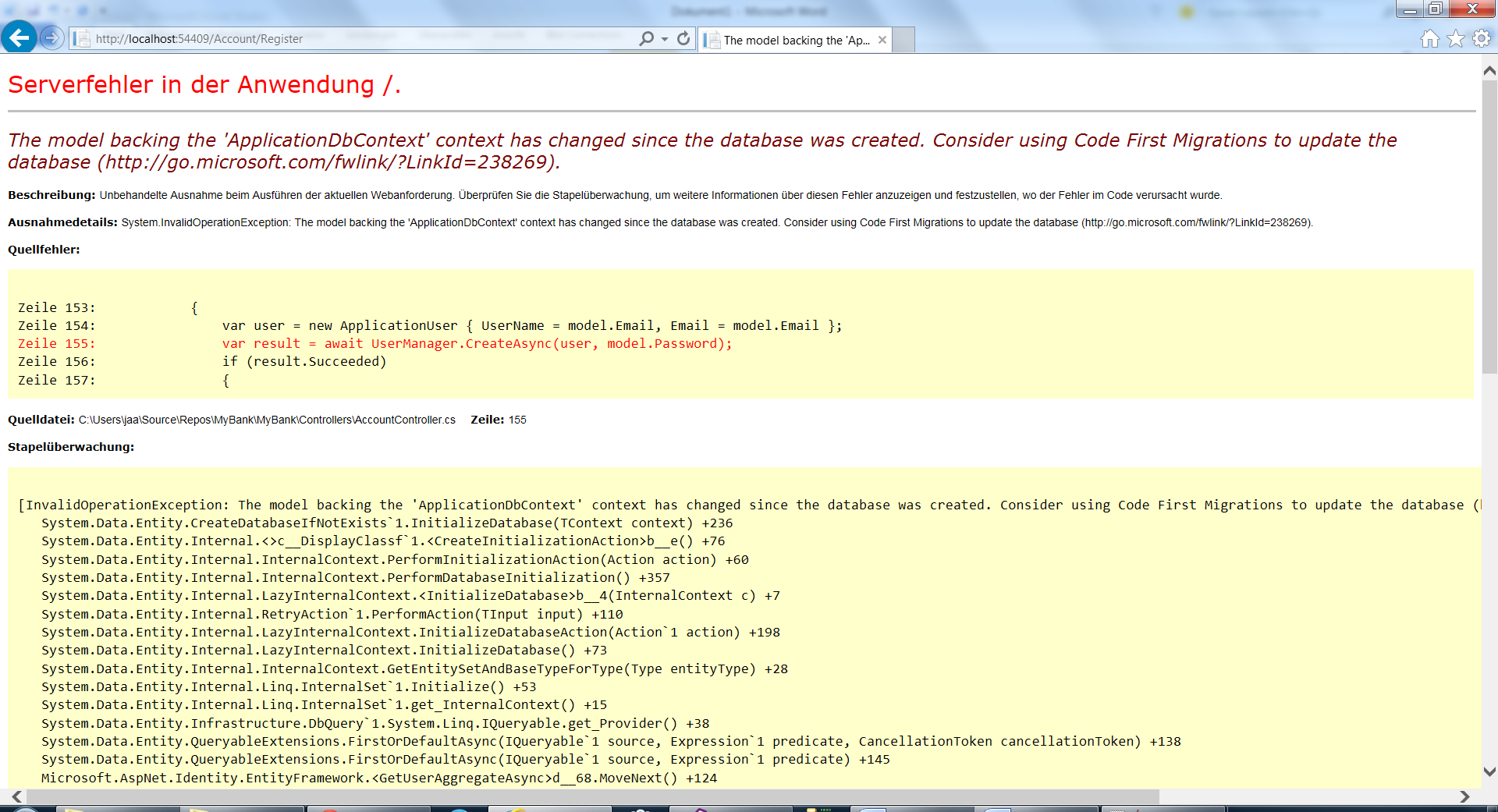
@Html.ActionLink("Neues Konto", "Create", null, new { @class = "btn btn-default" })



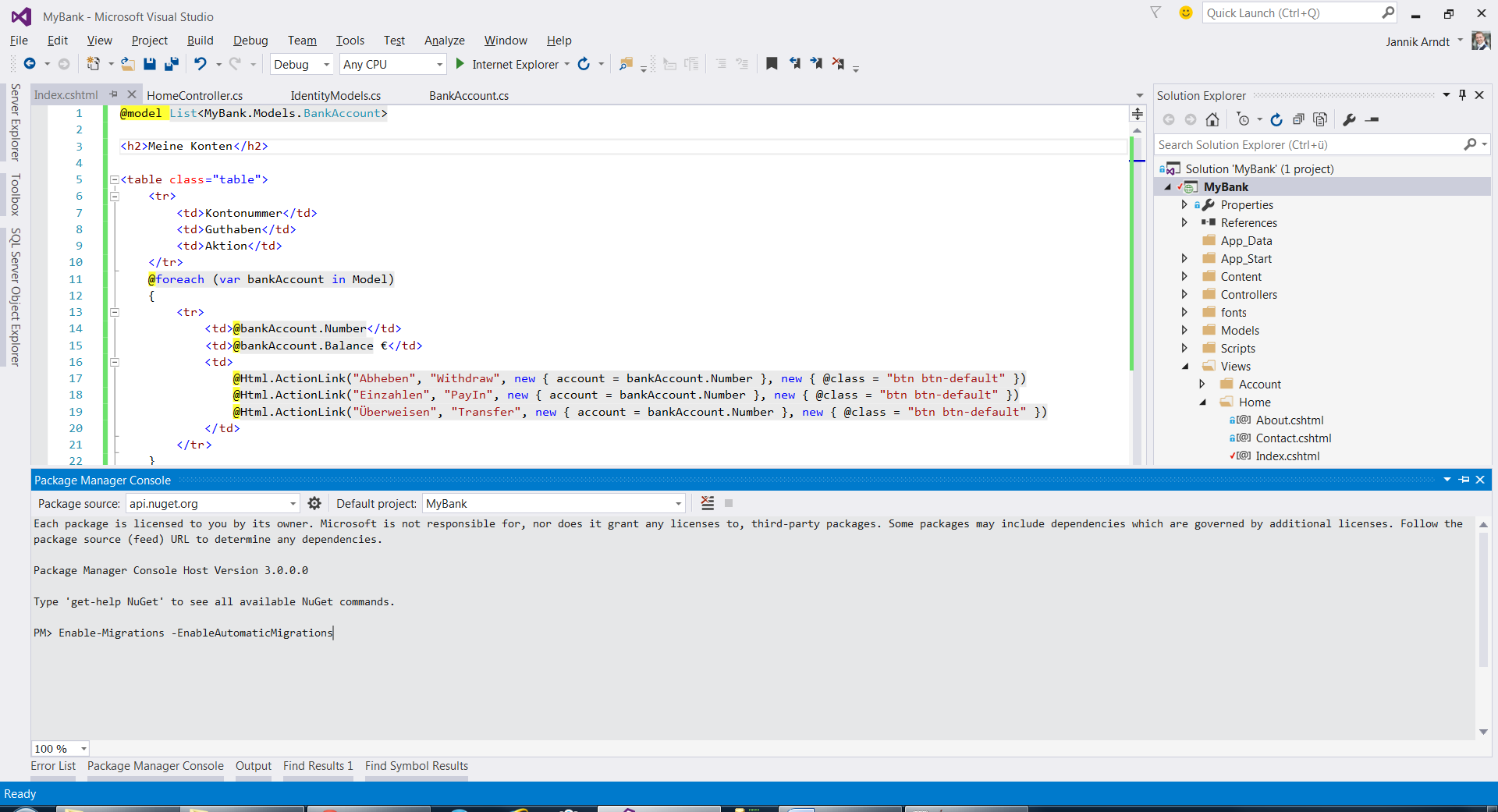
Mit einem Klick auf „Internet Explorer“ testen. Es kommt ein Log In-Dialog, da die Index-Methode das [Authorize]–Attribut hat:



Beim Registrieren eines Benutzers kommt eine Fehlermeldung:



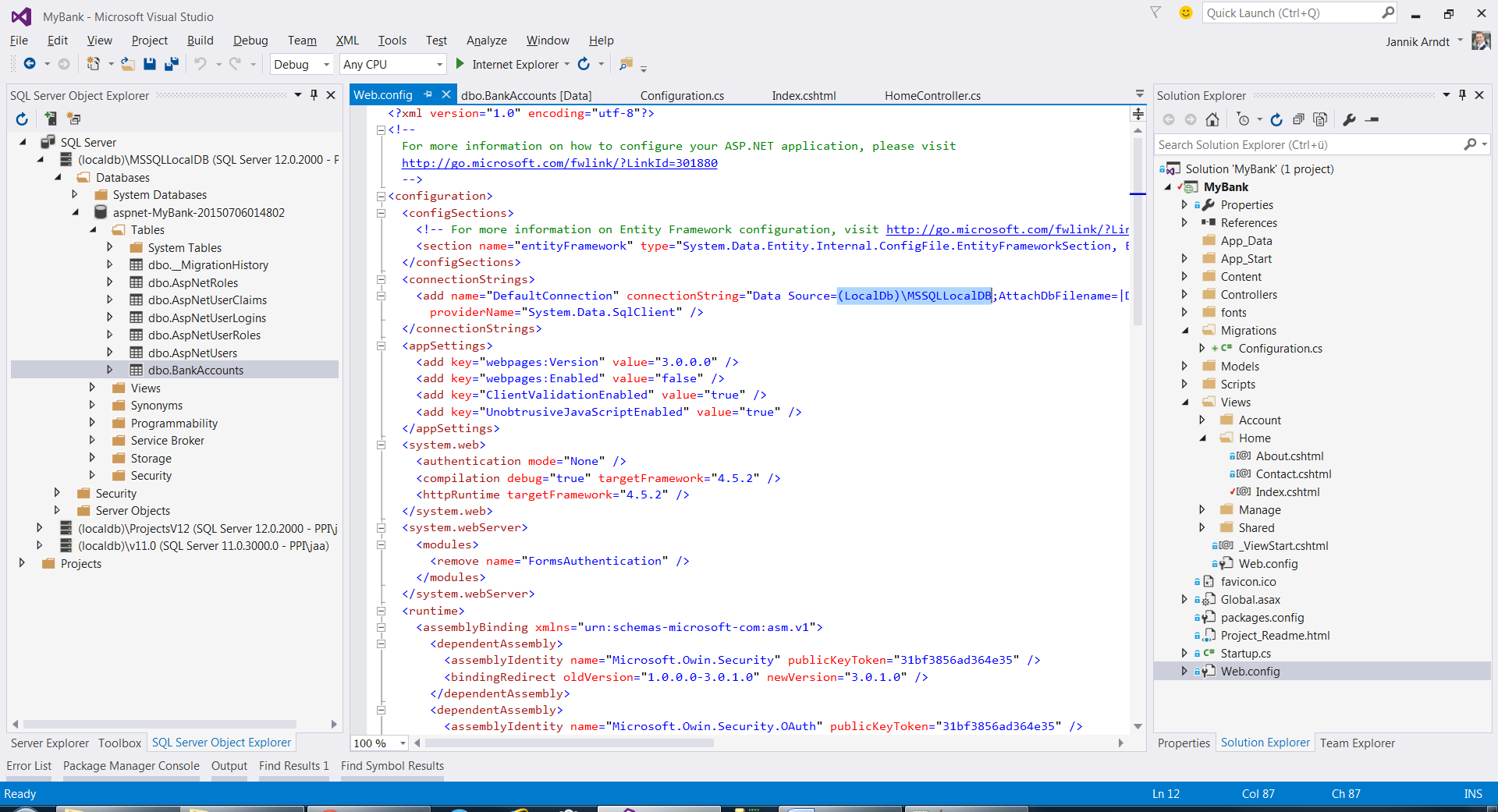
Das Model hat sich geändert, die Datenbank wurde aber noch nicht aktualisiert. Also Debugging beenden und Package Manager Console öffnen:



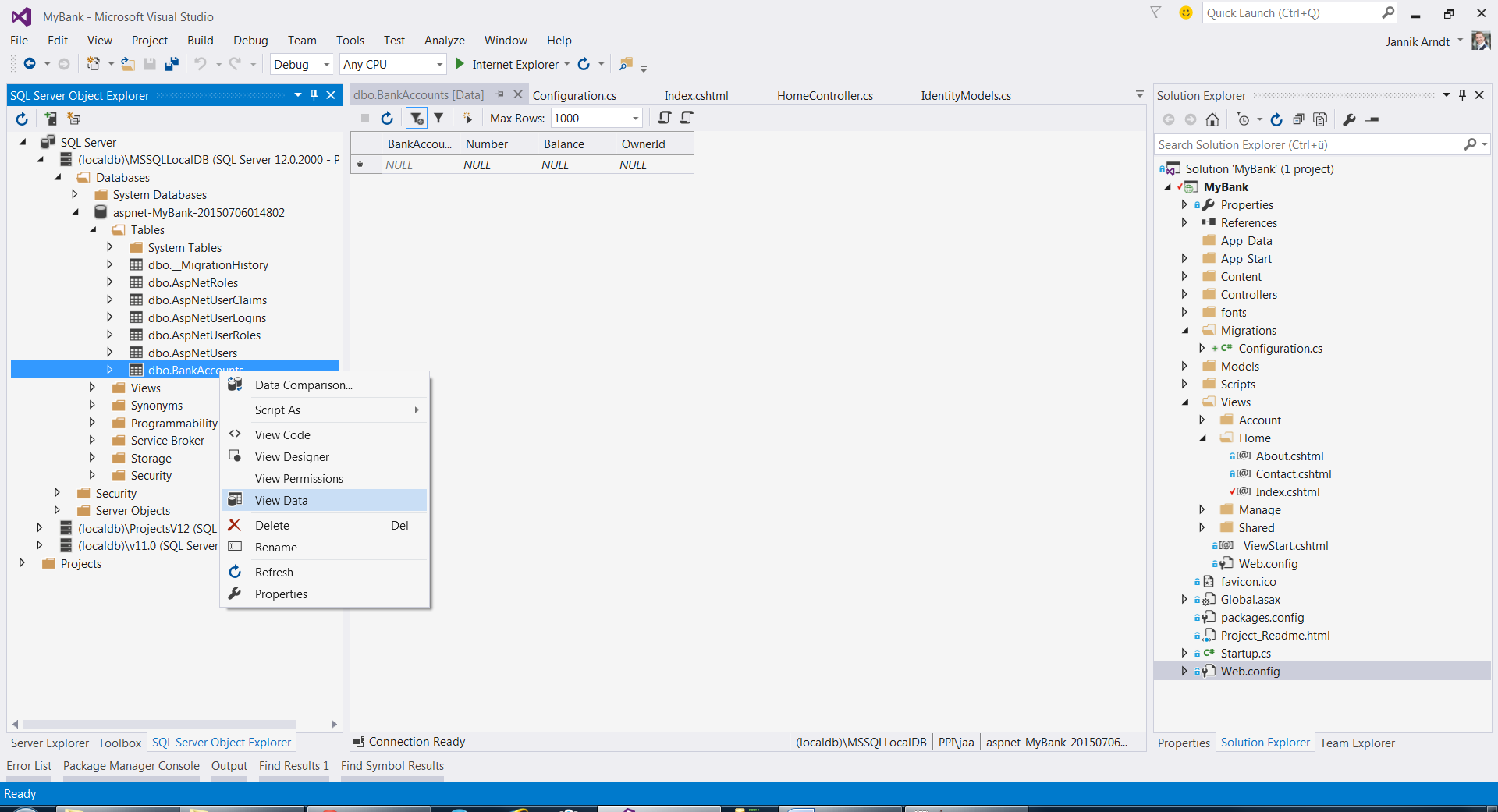
Mit dem Befehl

PM> Enable-Migrations -EnableAutomaticMigrations

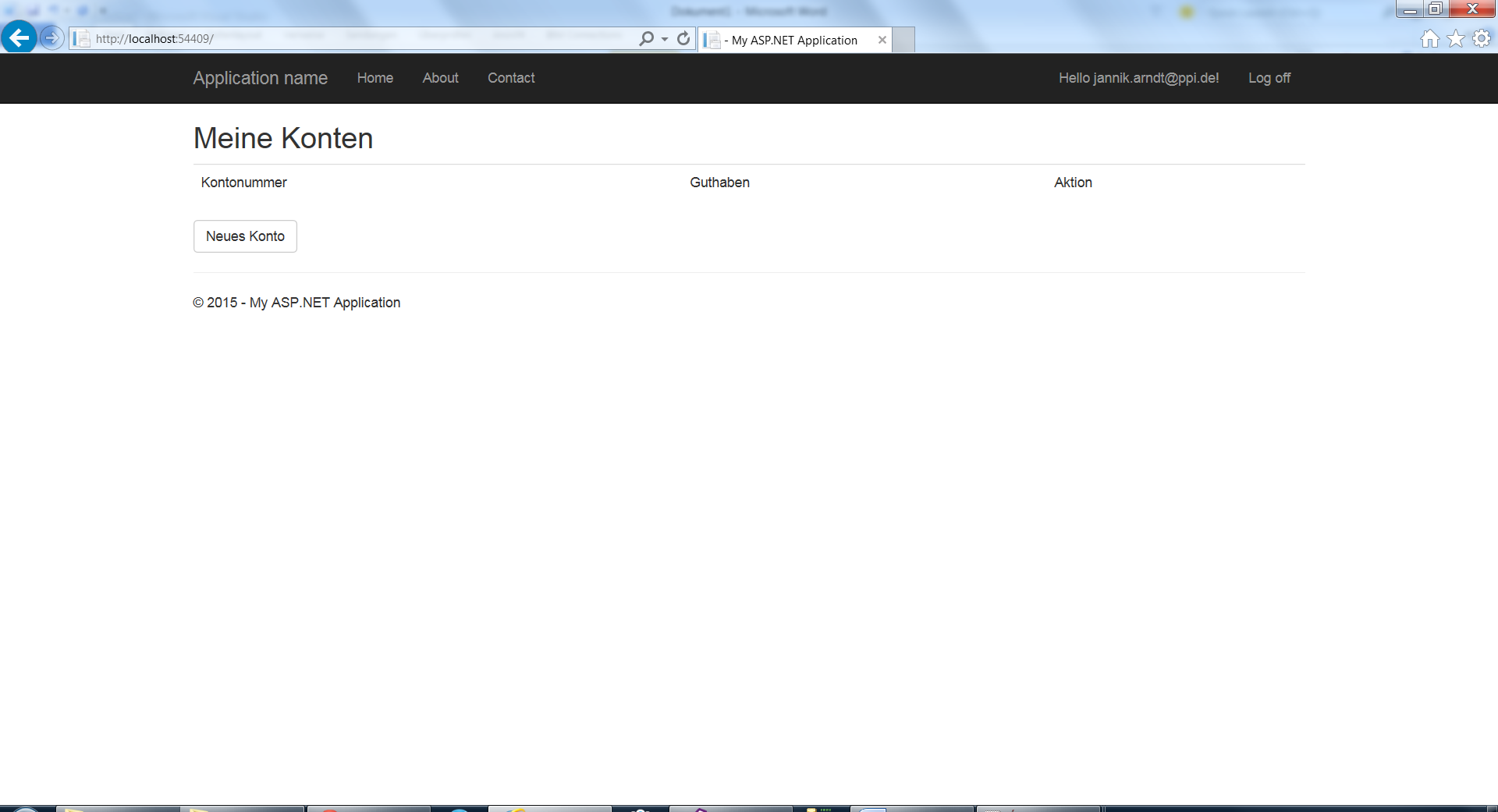
die automatische Migration aktivieren. Im *SQL Server Object Explorer* (links) kann man nun die frisch erstellte Datenbank anschauen. Die Verbindung hierzu ist i.d.R. (localdb)\MSSQLLocalDB, alternativ ist die genutzte Datenquelle auch in der Datei *Web.config* (mitte) im Hauptverzeichnis eingetragen:



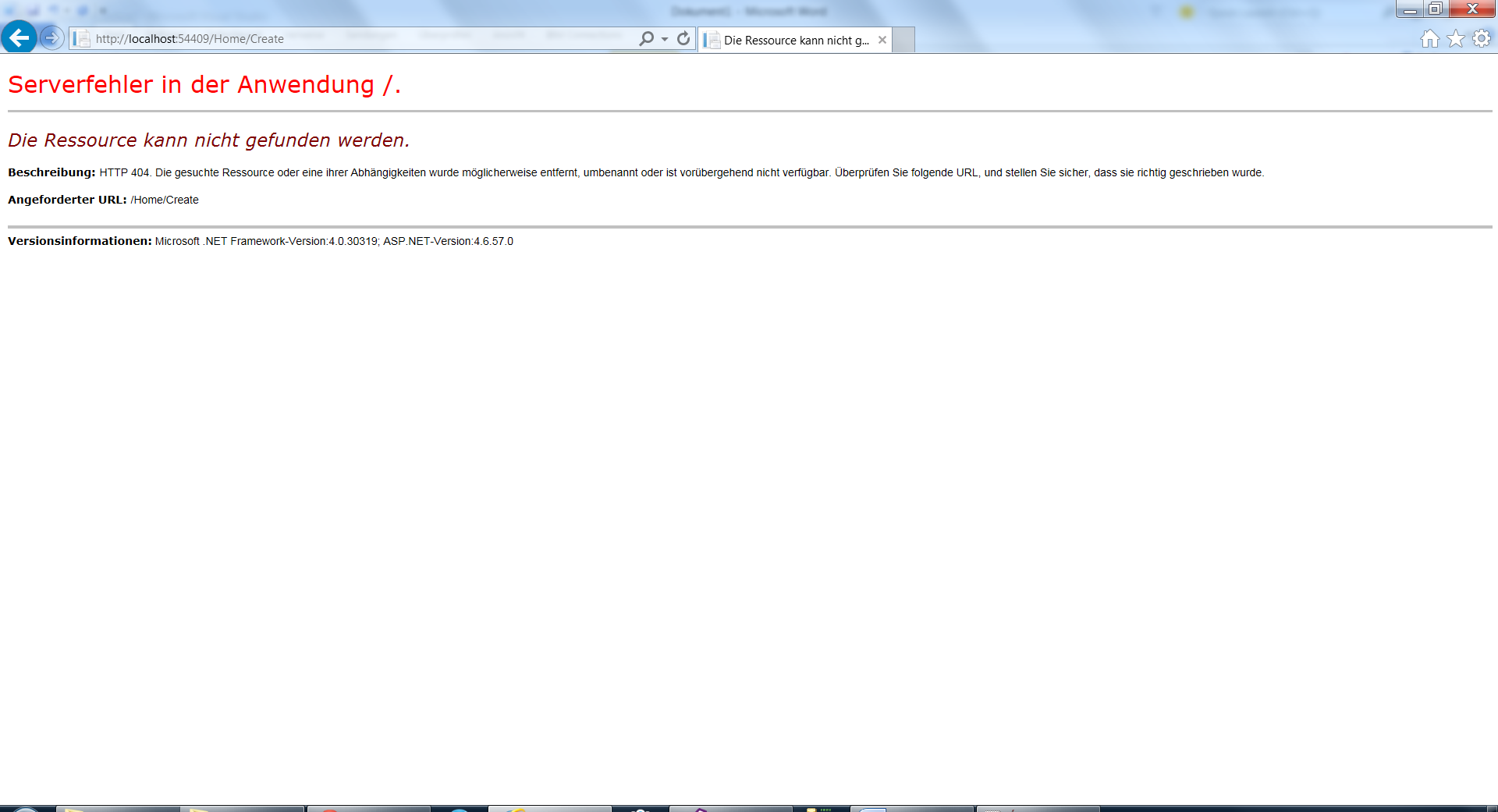
Der Aufruf von View Data zeigt aber, dass noch keine Daten vorhanden sind:



Nun also zurück zur Anwendung. Dafür das Debuggen erneut starten und einen Benutzer registrieren. Dieses mal klappt es. Es werden alle 0 Konten aufgelistet, die zum Benutzer gehören:



Der Klick auf *Neues Konto* führt noch zu einer Fehlermeldung:



Es wird die URL */Home/Create* aufgerufen. Erwartet wird also im *HomeController.cs* eine Methode mit dem Namen *Create()*.

# Einfügen der Create-Methode

Im HomeController.cs wird nun die Methode Create() eingefügt:

[Authorize]

public RedirectToRouteResult Create()

{

// Falls es schon Konten gibt, finde die höchste Kontonummer von allen und addiere 1, sonst nimm 1:

var accountNumber = \_bank.Accounts.Any() ? \_bank.Accounts.Max(account => account.Number) + 1 : 1;

// Erstelle ein neues Konto mit der aktuellen UserId als Besitzer und der eben herausgefundenen Kontonummer:

var newAccount = new BankAccount(User.Identity.GetUserId(), accountNumber);

// Füge das Konto der Liste der Konten der Bank hinzu:

\_bank.Accounts.Add(newAccount);

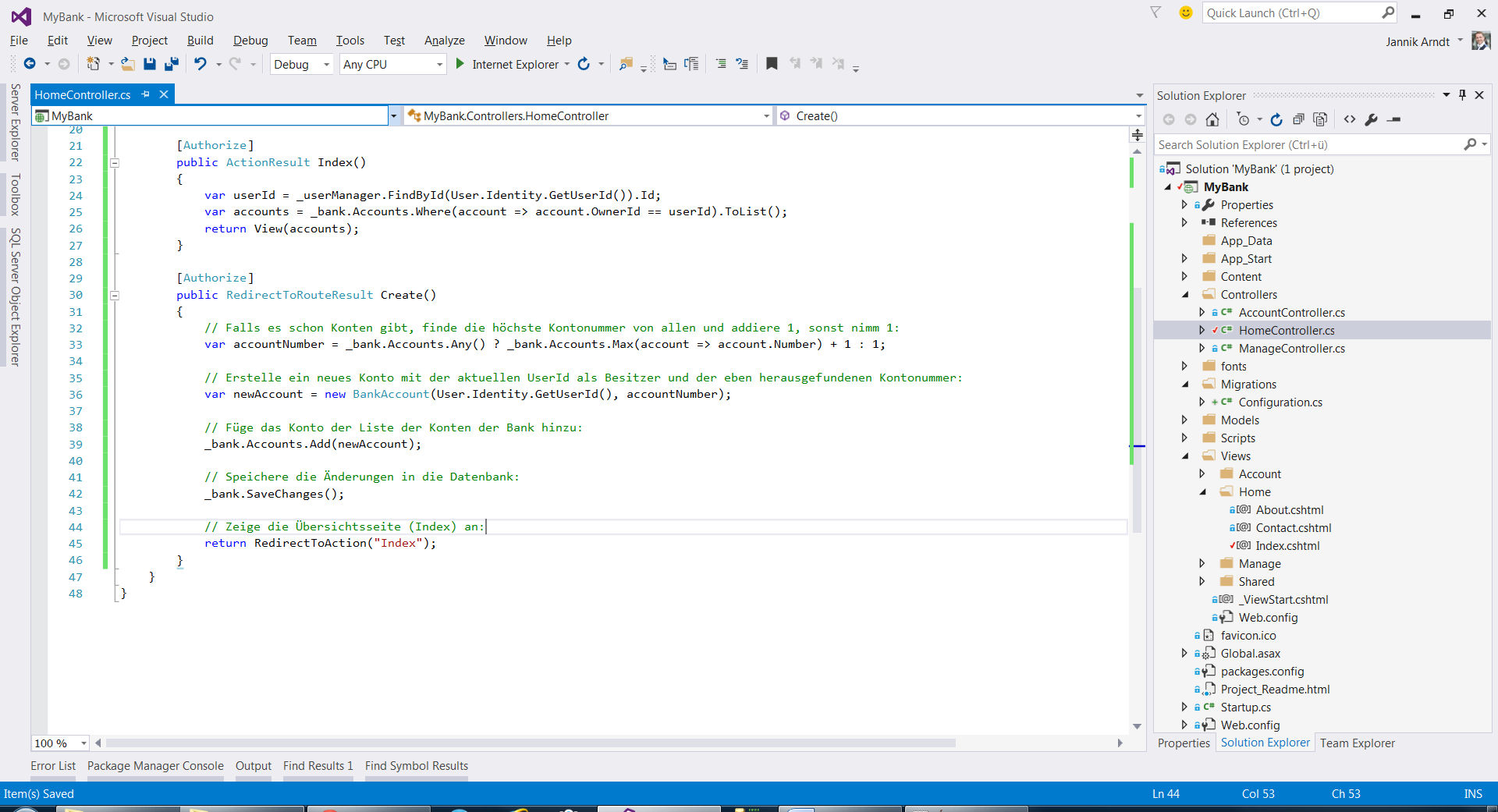
// Speichere die Änderungen in die Datenbank:

\_bank.SaveChanges();

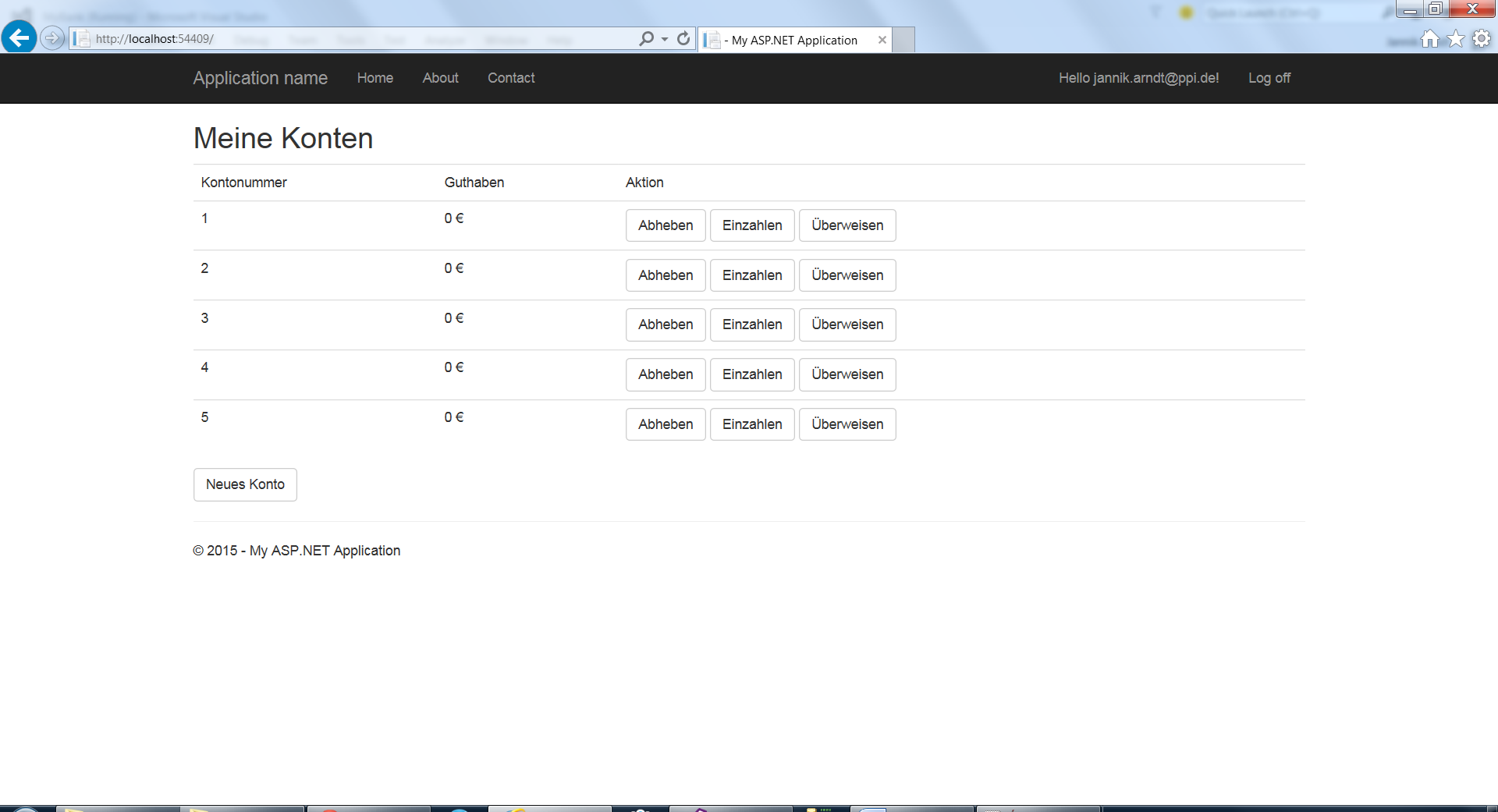
// Zeige die Übersichtsseite (Index) an:

return RedirectToAction("Index");

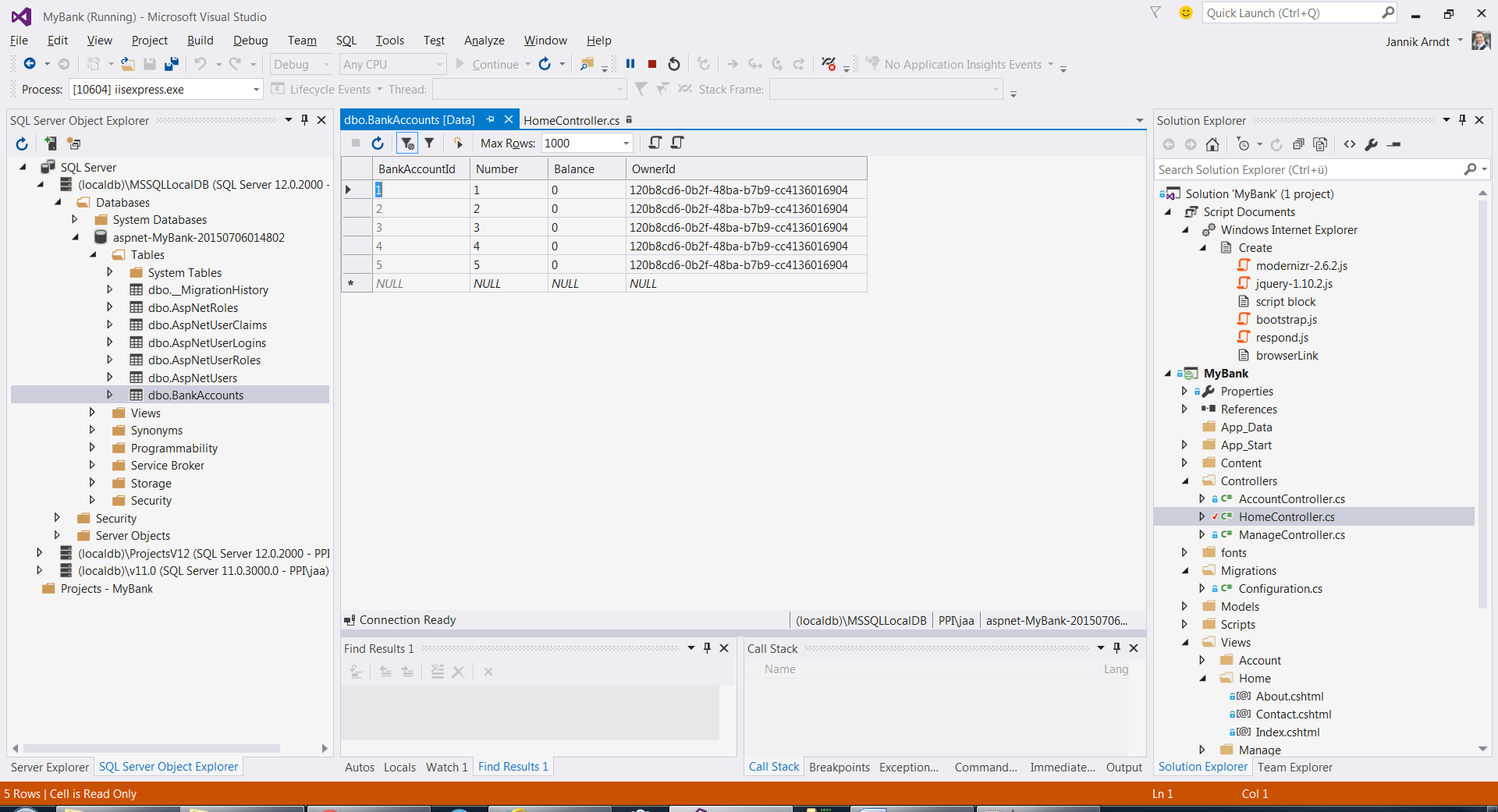
}



Mit einem Klick auf *Neues Konto* wird nun ein neues Konto angelegt und der Liste hinzugefügt:



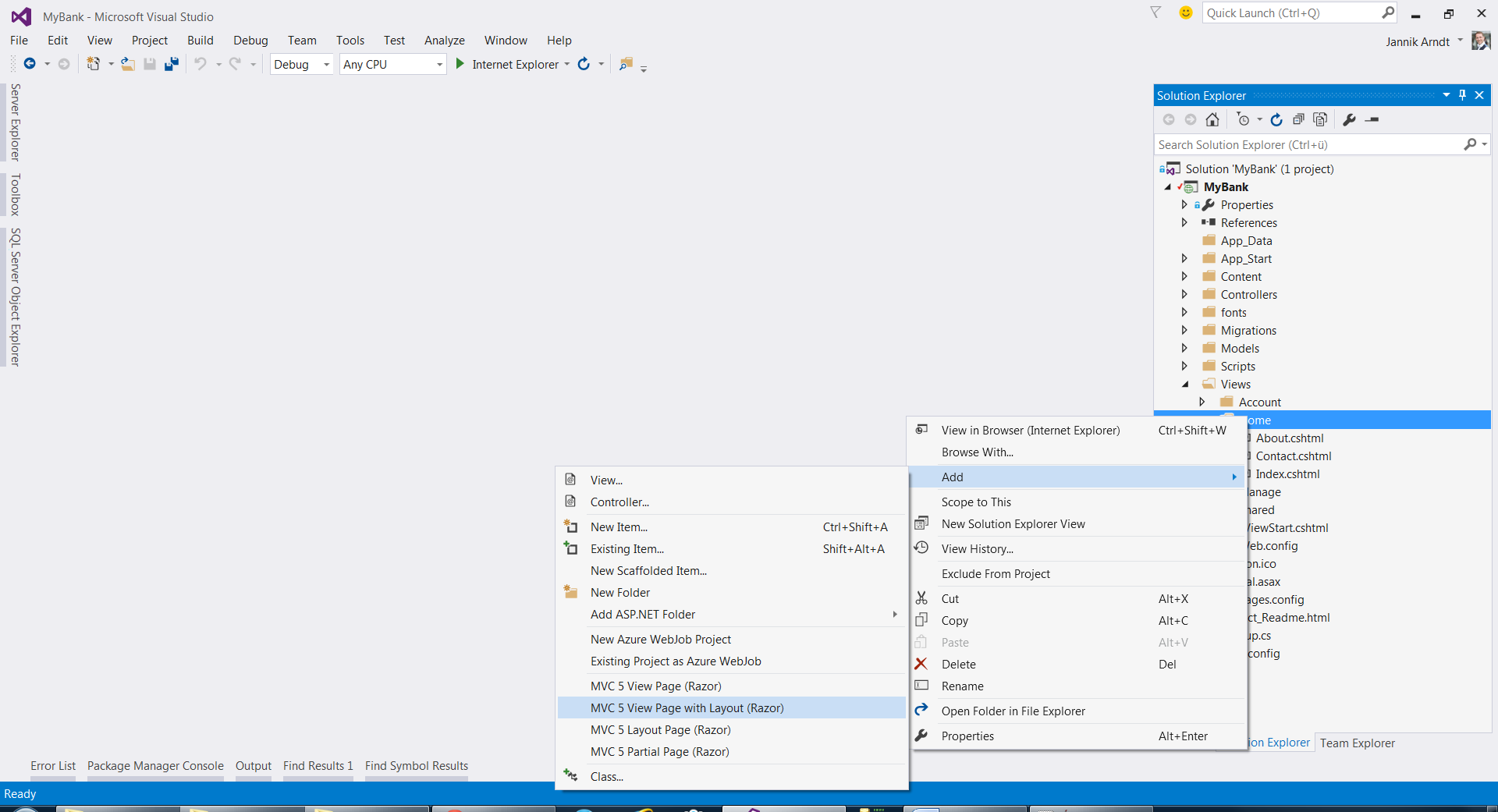
Diese Änderungen findet man auch gleich in der Datenbank wieder:

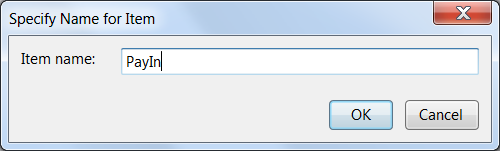


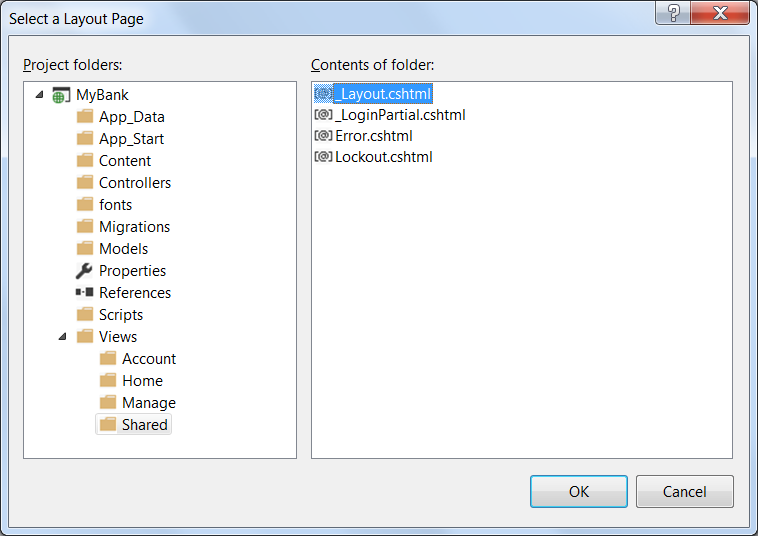
Leider führen die Methoden zum Einzahlen, Abheben und Überweisen noch nirgends hin.

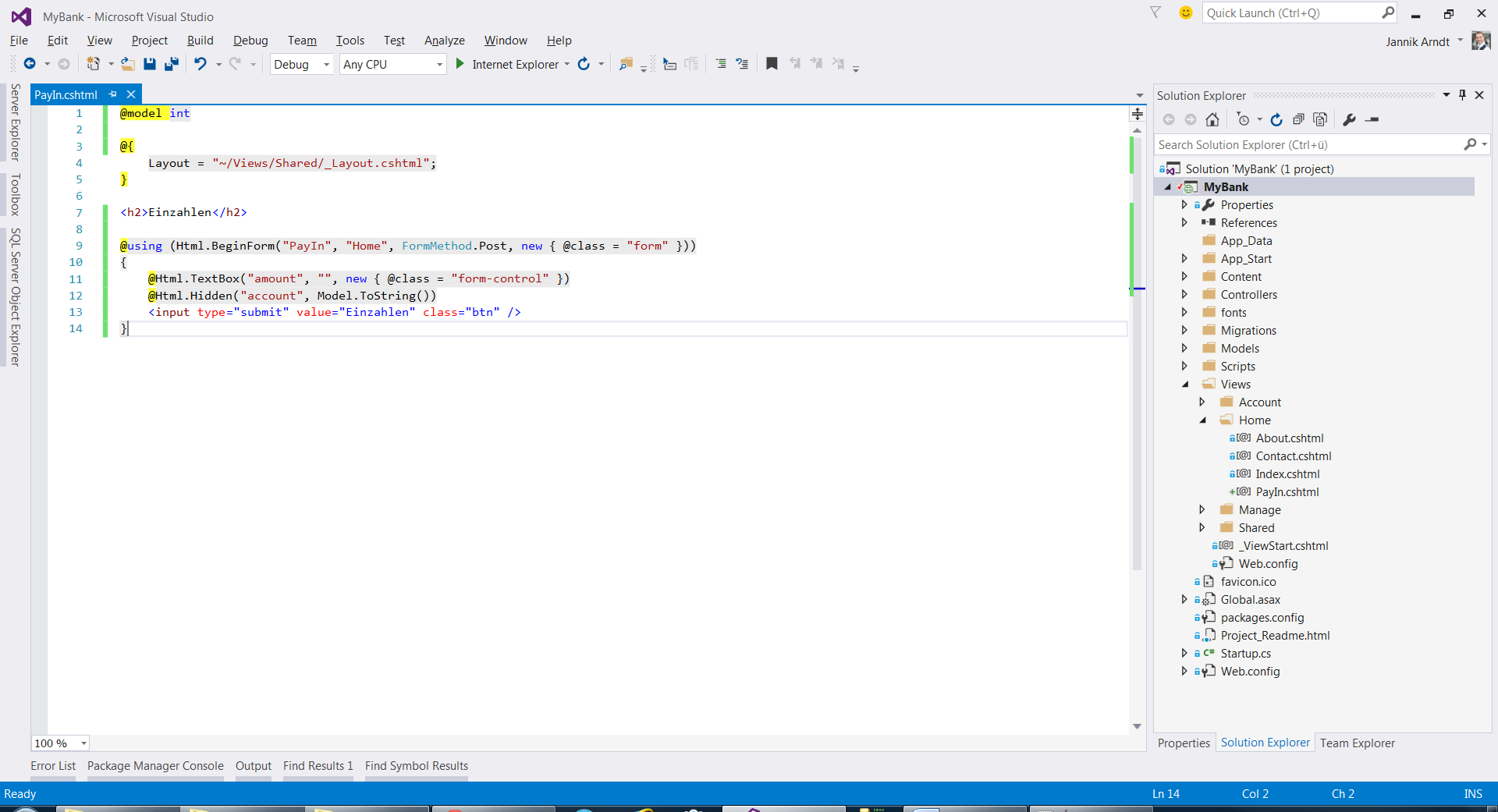
# Weitere Methoden und Views

Die Buttons in der Liste verweisen bereits auf die Methoden PayIn, Withdraw und Transfer, die wir im HomeController erstellen müssen. Allerdings sind für diese Aktionen weitere Benutzereingaben nötig, das heißt wir benötigen einen eigenen View für jede Methode:









View.cshtml:

@model int

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>Einzahlen</h2>

@using (Html.BeginForm("PayIn", "Home", FormMethod.Post, new { @class = "form" }))

{

@Html.TextBox("amount", "", new { @class = "form-control" })

@Html.Hidden("account", Model.ToString())

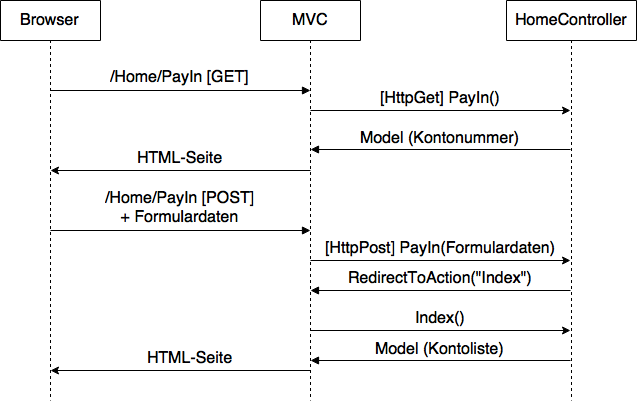
<br />

<input type="submit" value="Einzahlen" class="btn" />

}

Für jede der Aktionen benötigen wir zwei Aufrufe zum Server: Beim ersten wird der View generiert, beim zweiten werden die eigegebenen Daten (z.B. der Betrag) übermittelt und die Logik durchgeführt. Hierfür kann man zwei verschiedene Methoden benutzen (z.B. PayIn und PayInDo), oder dieselbe Methode überladen und von der Art des Aufrufs (GET oder POST) abhängig machen, welche gewählt wird.

Ruft der Benutzer die URL auf, wird die GET-Methode ausgeführt, schickt er dann das Formular ab, wird dieses per POST übermittelt:



[HttpGet]

[Authorize]

public ActionResult PayIn(int account)

{

return View(account);

}

[HttpPost]

[Authorize]

public RedirectToRouteResult PayIn(int account, double amount)

{

var myAccount = \_bank.Accounts.FirstOrDefault(acc => acc.Number == account);

if (myAccount == null)

return RedirectToAction("Index");

myAccount.PayIn(amount);

\_bank.SaveChanges();

return RedirectToAction("Index");

}

[HttpGet]

[Authorize]

public ActionResult Withdraw(int account)

{

return View(account);

}

[HttpPost]

[Authorize]

public RedirectToRouteResult Withdraw(int account, double amount)

{

var myAccount = \_bank.Accounts.FirstOrDefault(acc => acc.Number == account);

if (myAccount != null)

{

if (!myAccount.Withdraw(amount))

return RedirectToAction("Index");

}

else

return RedirectToAction("Index");

\_bank.SaveChanges();

return RedirectToAction("Index");

}

[HttpGet]

[Authorize]

public ActionResult Transfer(int account)

{

return View(account);

}

[HttpPost]

[Authorize]

public RedirectToRouteResult Transfer(int from, int to, double amount)

{

var sender = \_bank.Accounts.FirstOrDefault(acc => acc.Number == from);

var receiver = \_bank.Accounts.FirstOrDefault(acc => acc.Number == to);

if (sender != null && receiver != null)

{

if (!sender.Transfer(amount, receiver))

return RedirectToAction("Index");

}

else

return RedirectToAction("Index");

\_bank.SaveChanges();

return RedirectToAction("Index");

}

Bei der Methode Transfer gibt es einen Unterschied: Hier wird auf zwei Konten zugegriffen, deswegen hat die Methode im Controller die Parameter *from* und *to*, wobei *from* wie sonst auch aus einem Hidden-Input-Feld im View stammt und *to* vom Benutzer eingegeben wurde:

@model int

@{ Layout = "~/Views/Shared/\_Layout.cshtml";}

<h2>Überweisen</h2>

@using (Html.BeginForm("Transfer", "Home", FormMethod.Post, new { @class = "form-horizontal" }))

{

<label for="amount">Betrag</label>

@Html.TextBox("amount", "", new { @class = "form-control" })

<br />

<label for="to">Empfänger</label>

@Html.TextBox("to", "", new { @class = "form-control" })

@Html.Hidden("from", Model.ToString())

<br />

<input type="submit" value="Überweisen" class="btn btn-primary" />

}

# Erweiterung zum ViewModel

Die Methoden des Controllers übergeben ihrem zugehörigen View ein Model, zum Beispiel bei return View(accounts);. Das ist aber aktuell noch kein ViewModel, also keine eigens auf den View zugeschnittene Klasse. Dies werden wir nun ändern. Hintergrund ist, dass wir zwar im Controller wissen, ob die Aufrufe zum Einzahlen, Abheben und Überweisen funktioniert haben oder nicht, das aber dem Benutzer noch nicht anzeigen können. Wir fangen also damit an, beim Aufruf der Index-Methode eine Statusmeldung als optionalen Parameter zu übergeben:

public ActionResult Index(string message = "", bool error = false, bool success = false)

Und in den aufrufenden Methoden wird ein anonymes Objekt erstellt, das diese Parameter enthält:

return RedirectToAction("Index", new { message = "Das Konto Nummer " + accountNumber + "wurde erfolgreich angelegt.", success = true } );

Außerdem müssen wir für das ViewModel eine eigene Klasse erstellen:

using System.Collections.Generic;

namespace MyBank.Models

{

public class AccountOverviewViewModel

{

public List<BankAccount> Accounts { get; set; }

public string Message { get; set; }

public bool Error { get; set; }

public bool Success { get; set; }

}

}

Diese wird nun in der Index-Methode befüllt:

[Authorize]

public ActionResult Index(string message = "", bool error = false, bool success = false)

{

var userId = \_userManager.FindById(User.Identity.GetUserId()).Id;

var model = new AccountOverviewViewModel

{

Accounts = \_bank.Accounts.Where(account => account.OwnerId == userId).ToList(),

Message = message,

Error = error,

Success = success

};

return View(model);

}

Nun muss nur noch der View angepasst werden:

@model MyBank.Models.AccountOverviewViewModel // andere Model-Klasse

<h2>Meine Konten</h2>

@if (Model.Error)

{

<div class="alert alert-danger">@Model.Message</div>

}

@if(Model.Success)

{

<div class="alert alert-success">@Model.Message</div>

}

<table class="table">

<tr>

<td>Kontonummer</td>

<td>Guthaben</td>

<td>Aktion</td>

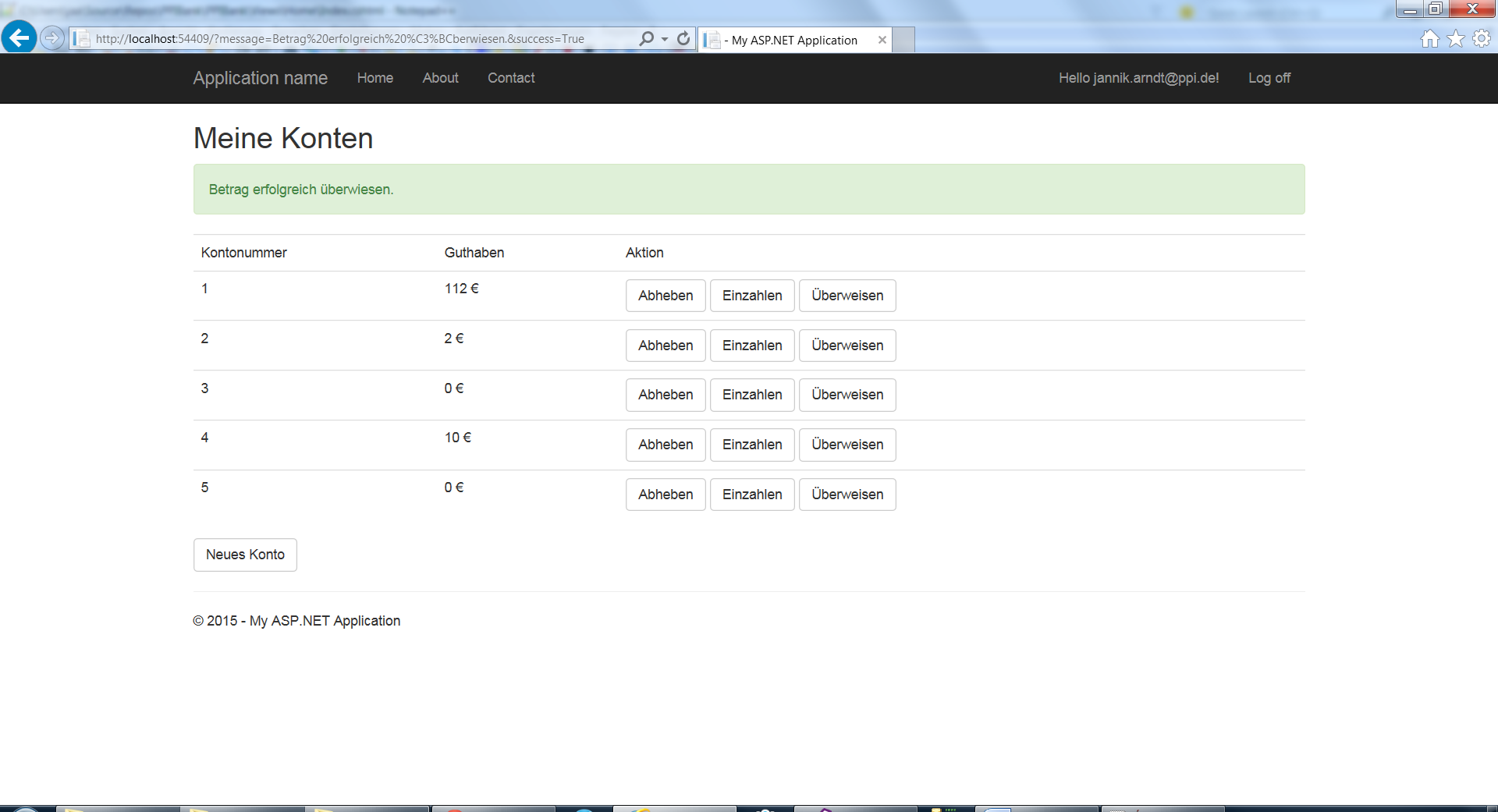
</tr>

@foreach (var bankAccount in Model.Accounts) // hier jetzt Model.Accounts!

{

…

Nun werden nach allen Aktionen Erfolgs- oder Fehlermeldungen ausgegeben:



# Didaktische Reserve: Asynchrone Aufrufe

Das MVC-Framework macht es sehr einfach, Aufrufe an den Controller asynchron auszuführen, d.h. während der Prozessor auf eine Antwort von z.B. einem Webservice wartet kann er andere Aufgaben z.B. von anderen Benutzern ausführen. Die FindById()-Methode des UserManagers bietet auch eine asynchrone Variante FindByIdAsync(). Über das Schlüsselwort await kann man dem Prozessor signalisieren, dass das Ergebnis der Methode im weiteren Verlauf benötigt wird, er also mit dieser Aufgabe erst fortfahren kann, wenn das Ergebnis geliefert wurde. Dadurch ändert sich auch die Signatur der Methode leicht, sie bekommt das async-Schlüsselwort hinzu und gibt statt eines ActionResults nun ein Task<ActionResult> zurück:

public async Task<ActionResult> Index(string message = "", bool error = false, bool success = false)

{

var user = await \_userManager.FindByIdAsync(User.Identity.GetUserId());

var userId = user.Id;

…

return View(model);

}