Eintrag in AppSettings zur Laufzeit ändern

1. Zunächst benötigt man eine Referenz auf “System.Configuration”. Diese ist per Standard nicht in einem Projekt eingebunden
2. Man muss eine AppSettings.config erstellen, die in etwa so aussieht:
   [crayon-66496eae75189577860482/]
3. Anschließend noch eine Hilfsfunktion:
   [crayon-66496eae7518e737119347/]
4. Die man ausführen kann:
   [crayon-66496eae7518f274522778/]
5. Auslesen kann man die AppSettings ganz einfach mit
   [crayon-66496eae75191342053542/]

Snippet um Zip Dateien im Memory zu erstellen

Wenn Foldername gefüllt ist, wird ein Verzeichnis erstellt, wo die Dateien reinkommen.
[crayon-66496eae7569f502020450/]
[crayon-66496eae756a4395125929/]
In Verbindung mit MVC kann der Stream nun Chunk-weise ausgegeben werden:
[crayon-66496eae756a5942892550/]

mit Dapper ein SQL join mappen

Da ich von Entity Framework 6 aus Perfomancegründen überhaupt nicht zufrieden bin, frage ich mich ernsthaft, wie einige das produktiv einsetzen.

Daraufhin habe ich nach einer Alternative gesucht und bin bei Dapper, einem Projekt von Source stehen geblieben. Hier möchte ich einmal kurz zeigen, wie man mit Dapper eine Liste mit dem erstellten SQL Join erstellt.

Beispiel:

Wir haben in der SQL Datenbank eine Tabelle Automarken und eine andere mit Modelle.

Automarke: Id | Bezeichnung |

Modelle: Id | Bezeichnung | AutomarkeId

Es ist nicht schwer zu erkennen, dass hier eine 1:n Beziehung existiert.

Jetzt wollen wir das ganze in C# abbilden. Dazu bauen wir uns 2 Klassen
[crayon-66496eae758e9781187024/]
Wir wollen nun eine List<Automarken> gefüllt mit allen Modellen der Automarken haben. Wie machen wir das?

Vielleicht hat der eine oder andere bereits eine Idee, aber denkt mal an die Performance.

Hier kommt Dapper ins Spiel. Dapper mappt für mich mit wenigen Zeilen Code eine Collection ganz nach meinem Geschmack.

Wie das geht, zeige ich hier….

1. Am besten und schnellsten ist es, wenn man Dapper von Nuget herunterlädt: **Install-Package Dapper**
2. Wir brauchen ein Objekt der Klasse SqlConnection, welches natürlich erfolgreich zur Datenbank connected ist

Im Grunde war dies auch schon. Ich habe Bemerkungen im Programmcode angefügt, damit es direkt verständlicher ist.
[crayon-66496eae758f0541012597/]

Richtig casten

Ich will es mal kurz halten :

Casten ist nicht konvertieren, es ist eher eine Typzuweisung. Da in C# alles vom Typ Object erbt, können wir dem Object alles zuweisen was wir wollen. Um dann dem Compiler zu sagen, dieses Object ist aber vom Typ String und besitzt dementsprechend seine Eigenschaften, müssen wir es Casten. Dazu gibt es 2 Möglichkeiten : Einmal die (class) Object und die Object as class Möglichkeit.

Der Unterschied der beiden ist, dass im ersten sobald nicht gecastet werden kann eine Exception aufgerufen wird. Beim as casten wird das Object null.

Je nach Anwendungsfall ist beides Sinnvoll

Bulkupsert oder einfach Update, wenn bereits vorhanden ansonsten Insert in MSSQL

Lange habe ich nach einer vernünftigen Lösung gesucht, um riesige Datenmengen schnell in die Datenbank zu schreiben.

Als Quelle steht uns eine List<Class> mit der Klasse zur Verfügung, die der Datenbanktabelle gleicht und jede Menge Inhalte enthält. Versucht man diese Liste nun über das Entity Framework in die Datenbank zu jagen, merkt man schnell, dass das Entity Framework an seine Grenzen stößt. Alternativ haben wir das von der SQLCopy Klasse die BulkInsert() Methode, die aber nur Inserten kann. Findet das BulkInsert einen Index/Primärschlüssel, der bereits vorhanden ist, wird die ganze Show abgebrochen. Wie toll wäre es nun, wenn es nicht abbrechen würde, sondern diese Zeile einfach updated. Und das ganze Superschnell. Ich habe Testweise 10.000 Einträge in 400ms und 1.000.000 in 1 min in die Datenbank bekommen.

Methode Upsert:
[crayon-66496eae75b6d687669484/]
Methode BulkInsert:
[crayon-66496eae75b75539192470/]
und die beiden Extension Methods:
[crayon-66496eae75b79378376260/]

Angewendet wird das ganze ganz einfach so:

1. Parameter beinhaltet die Liste mit allen Inhalten
2. Parameter gibt an, welche Spalten geprüft werden sollen, ob es zu einem Update kommen soll oder Insert. In meinem Fall, wenn PersNr und CardId bereits in der Datenbank vorhanden sind, so mache ein Update, sonst Insert
3. Parameter gibt an, was inserted werden soll.
4. Parameter gibt an, was geupdated werden soll, wenn Parameter 2 zutrifft.

Bitte auf die tatsächlichen Primärschlüssel, Indizes und NOT NULL Schlüssel/Attribute achten. Ansonsten kommt man schnell zu einem Fehler
[crayon-66496eae75b7b983302766/]

String zusammensetzen Vergleich mit C#6

Wer kennt das nicht, da muss man z.B. einen SQL String basteln, der Werte aus Variablen zusammen setzen muss.

Hierfür haben wir die 3 Variablen, die in die Personen Tabelle hinzugefügt werden müssen:
[crayon-66496eae75df9450996820/]

1. Die meisten Entwickler arbeiten dann einfach ohne Parameter:
   [crayon-66496eae75dfd152635868/]
2. Der nächste sagt, ich arbeite mit dem StringFormater:
   [crayon-66496eae75dff964950194/]
3. Ein weiterer sagt, ich nutze den StringBuilder:

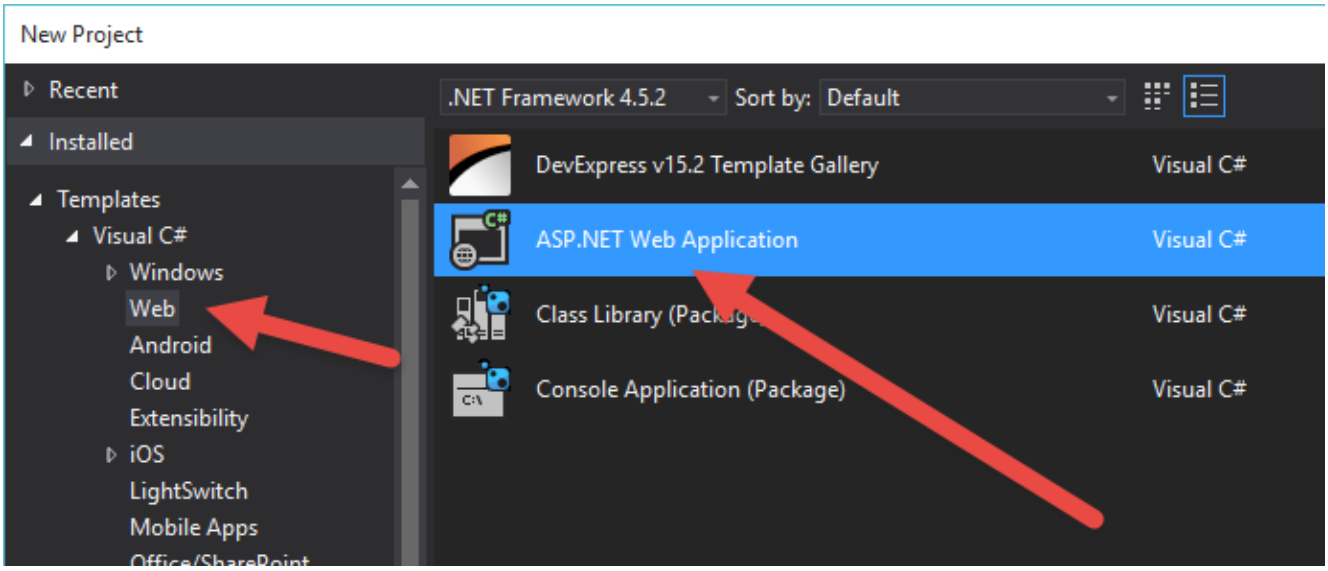
[crayon-66496eae75e00614244114/]

4. Neue Art einen String zusammen zu setzen mit C# Version 6:
   [crayon-66496eae75e02037324540/]
   Meiner Meinung nach ist die neue Möglichkeit deutlich übersichtlicher. Außerdem habe ich als Entwickler immer die Möglichkeit in den geschweiften Klammern Änderungen an der Variable vorzunehmen. Ich kann z.B. hingehen und Vorname.ToUpper() machen.

Unter ASP.Net 5 mit Entity Framework 7 arbeiten

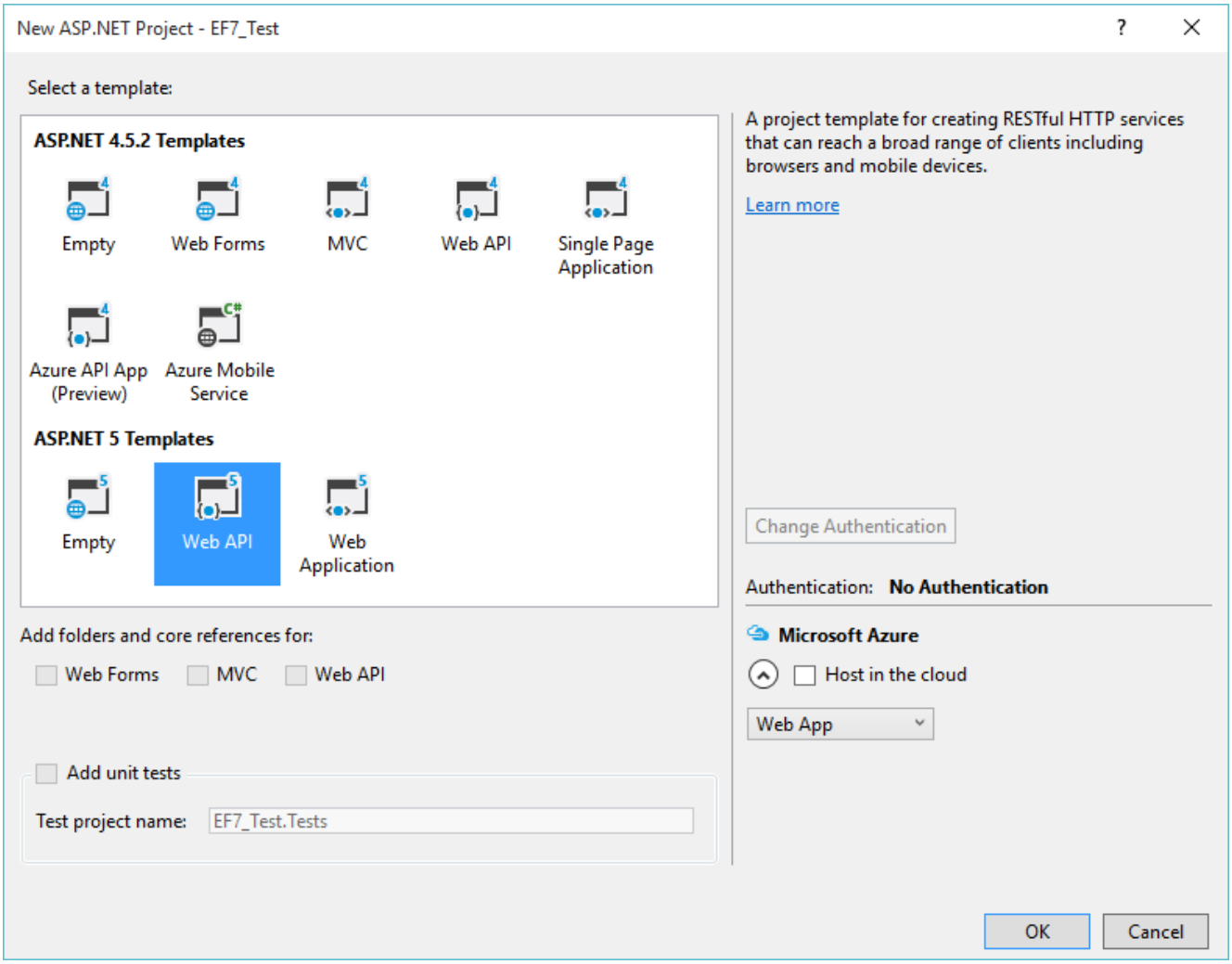
Mein Ziel in diesem Tutorial ist es, das neue ASP.Net 5 oder ASP MVC 6 mit dem Entity Framework 7 auszustatten. Dazu soll eine Tabelle Personal mit einer Adresstabelle in einer n:m Beziehung stehen. Na dann… lets go.

1. **Projekt erstellen**

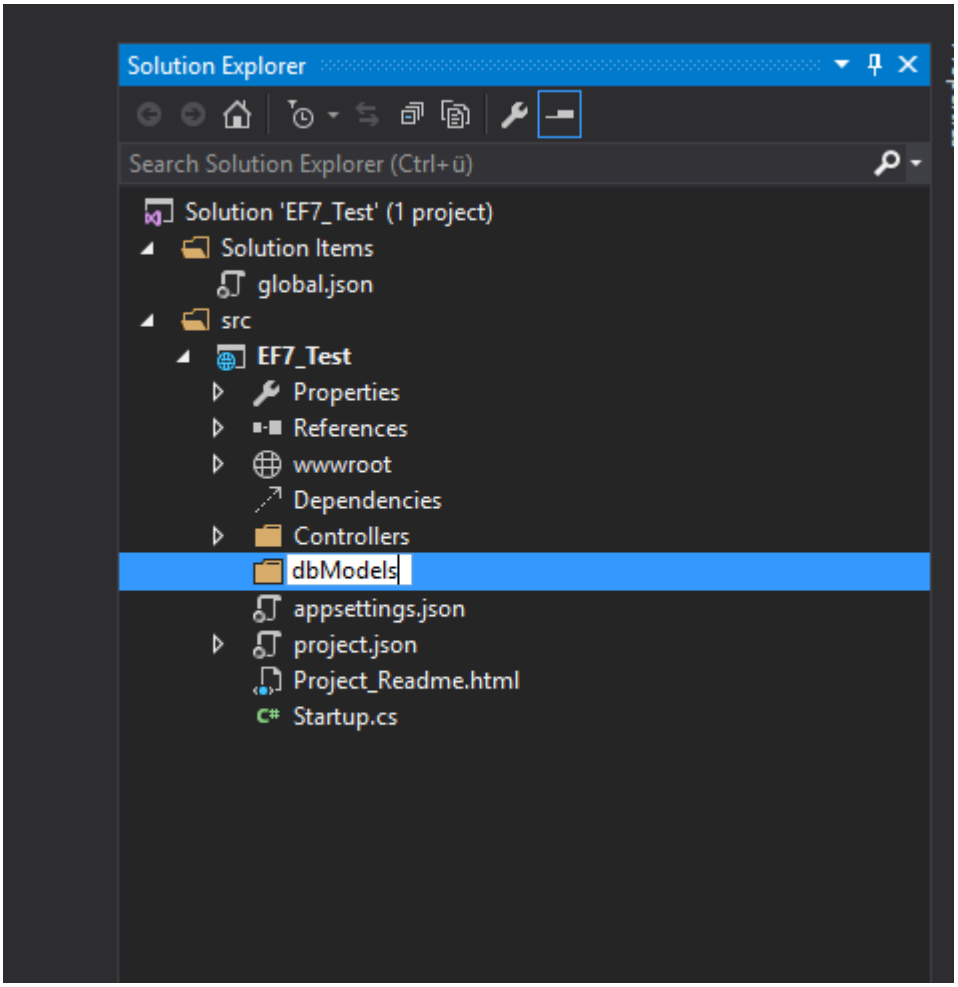


Unter Web finden wir in Visual Studio 2015 den Eintrag ASP.NET

Web Application



Es ist egal, ob wir ein leeres Projekt, die Web API oder Web Application wählen. Dies sind nur die Vorlagen. Wichtig ist nur, einen aus den unteren ASP.NET 5 Templates zu wählen



Ich habe meine Klassen gerne kategorisiert und daher lege ich für die Datenbank Modelle einen Ordner dbModels an.

2. Entity Framework 7 installieren und Klassen anlegen

a) *Entity Framework installieren*

mit Alt + T + N + O rufen wir die Nuget Console auf und geben dort folgendes ein um die neuste EF7 zu installieren:
[crayon-66496eae76015459479303/]
und dann noch die Commands:
[crayon-66496eae7601a964184173/]
Da das ganze noch in den Kinderschuhen ist, hat bei mir in der project.json Datei die Abhängigkeit zu den Commands gefehlt. Diese muss man evtl. manuell nachbessern. Dazu in der project.json Datei
[crayon-66496eae7601c620206135/]
mit

[crayon-66496eae7601e902467445/]
ersetzen. Und gleichzeitig einen Blick drauf werden, ob unter dependencies EF und Commands richtig instelliert sind:
[crayon-66496eae76020312671044/]
Falls man eine Änderung vornehmen musste, Project speichern, schließen und als Administrator neu aufmachen.

b) *Code First Modell erstellen*

Jetzt kann man das Abbild der Datenbank, das Modell der Personen Tabelle als Klasse in den neu angelegten Ordner erstellen:
[crayon-66496eae76022668672669/]
wobei die PersNr später der PrimaryKey sein wird. Heißt eine Property Id, dann erkennt Entity Framework diese automatisch als PrimaryKey an.

c) *dbContext Klasse erstellen*

Auch EF7 arbeitet mit einer Klasse, die von DbContext erbt. Hier legen wir Grundlegende Eigenschaften zur Verbindung und den Modellen an. Ich habe diese Klasse ebenfalls in den dbModels Ordner gelegt.
[crayon-66496eae76024952897102/]
Der 1.Teil mit dem Konstruktor ist da, der sagt, wenn die Datenbank nicht vorhanden ist, so soll doch bitte eine erstellt werden.

Der 2. Teil OnConfiguring: Hier gebe ich den Datenbankpfad an.

Der 3. Teil von OnModelCreating, hier kann ich unter anderem den Tabellennamen und den Namen des Schemas wählen. Ich habe das gerne, dass zusammenhängende Tabellen einen gemeinsamen Namen bekommen. Per Standard heißt das Schema immer dbo.

Der 4. Teil ist eine Property der dbContext Klasse vom Typ DbSet, damit wir über diese später Daten lesen,einfügen,ändern und löschen können.

d) *Migration erstellen*

Nun ist es an der Zeit eine Datenbankverbindung zu testen und gleichzeitig eine erste Migration zu erstellen.

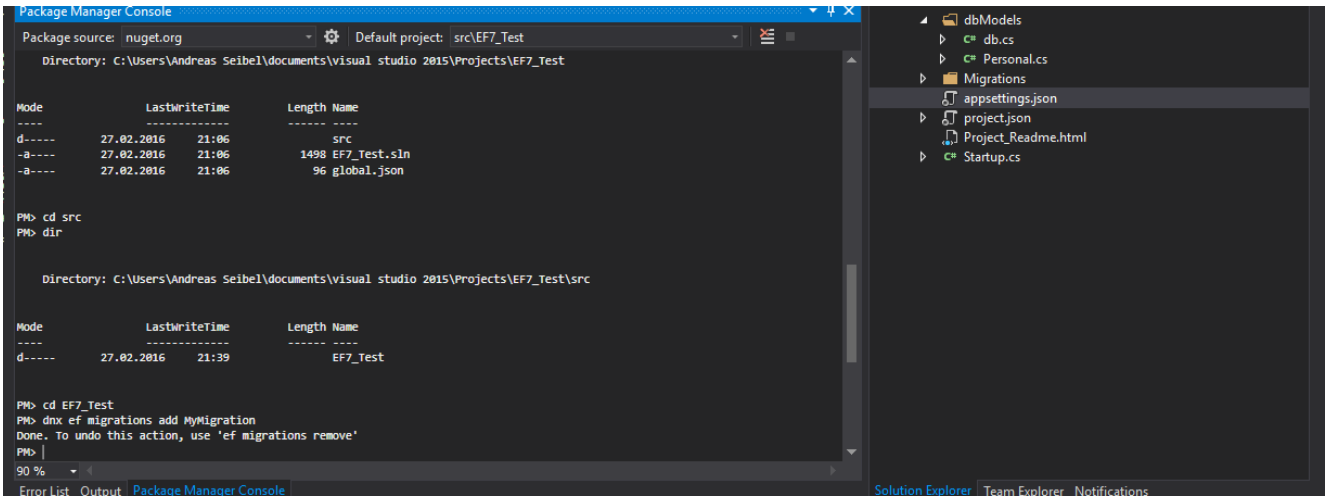
Dazu öffnen wir wieder die NuGet Console.

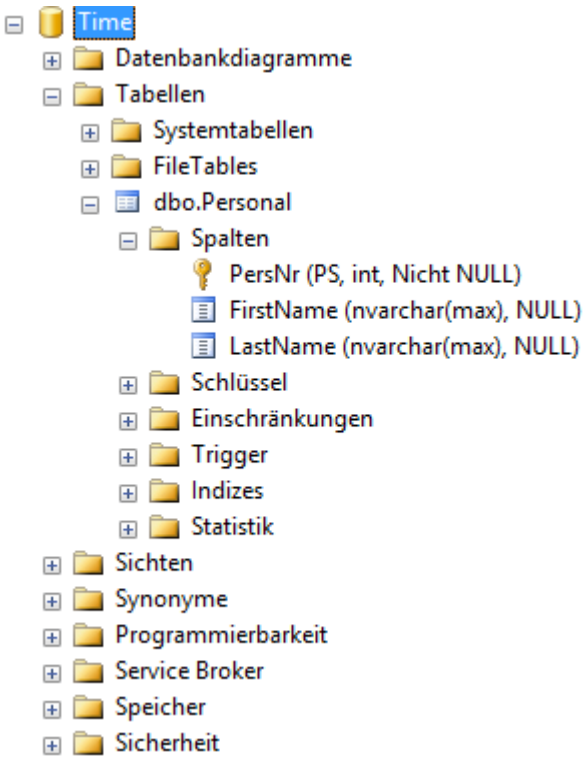
Da wir mit ASP.Net 5 arbeiten, gelten für uns nicht die alten Entity Framework Befehle, sondern die von DNX. Diese verlang von uns, dass wir die Befehle dort ausführen, wo sich die Datei befindet. Die EF Dateien befinden sich in der Ordnerstruktur src/[Projektname].

Navigieren kann man mit **cd Ordnername** nach vorne bzw. **cd ..** einen Satz zurück. Mit **dir** bekommt man eine Übersicht.

Konkret muss man in den Ordner **src\ProjektName** wechseln

Nun gibt man dort folgendes ein:
[crayon-66496eae76027908469251/]
Dann sollte man auch ein Done. Als Bestätigung bekommen. Weiterhin sieht man nun einen Ordner Migrations und in der Datenbank ist die Datenbank angelegt worden mit der Tabelle Personal





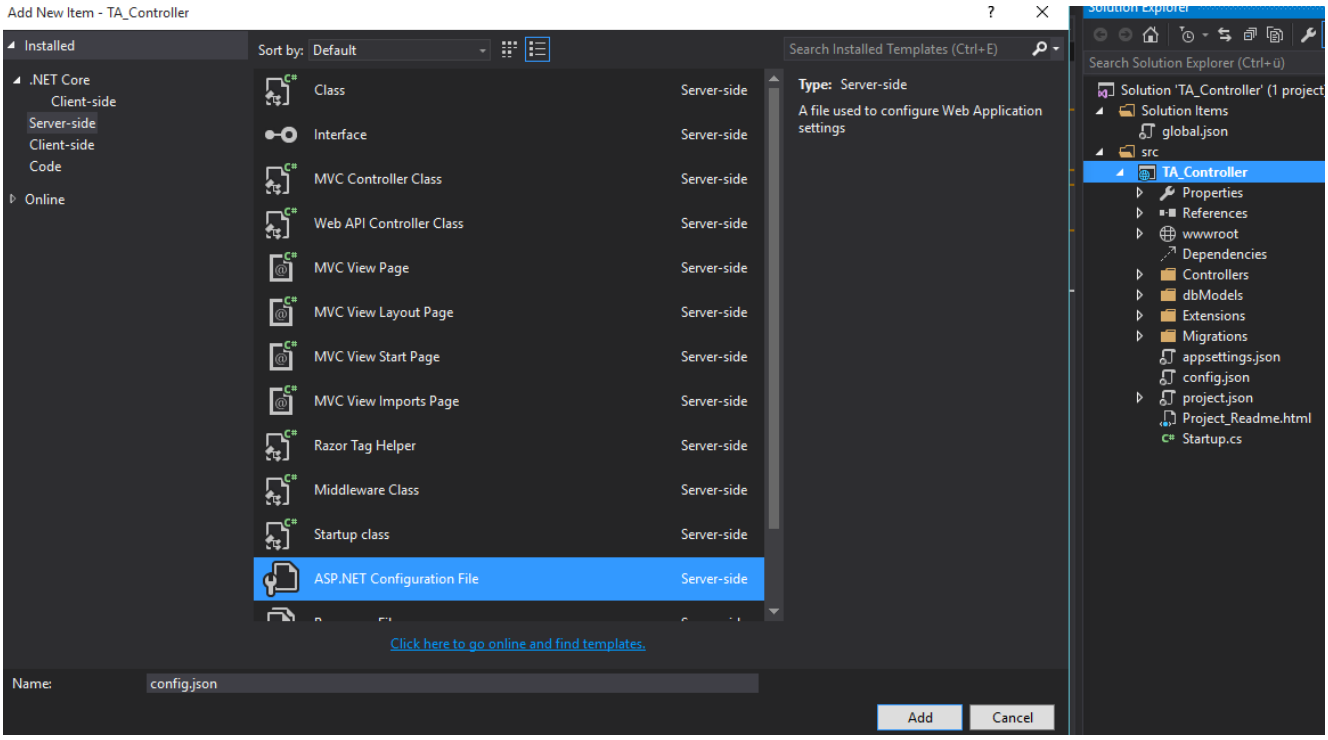
3. Klasse Adresse erstellen, welche eine Beziehung zu der Klasse Personal hat

Jetzt erstellen wir eine weitere Klasse Adress

Eine Adresse kann nur einer Person gehören. Daher bekommt jede Adresse nur eine Person zugewiesen. Gäbe es einen Fall, wo die Beziehungsklasse mehrere Personen enthalten kann, so würde man dies in eine ICollection<Person> packen.
[crayon-66496eae76029223213436/]

4. Default Connection String

Im Punkt 2c) haben wir einen Connection String direkt eingetragen. Will man das Projekt eines Tages publizieren, könnte man ihn so nicht ändern. Deshalb müssen wir diesen Connection String auslagern. Dazu erstellt man eine neue config.json Datei mit der Vorlage von **ASP.NET Configuration File**



und schreibt dort den Connection String rein.

Weiter muss nun auch die startup.cs angepasst werden:

1. Property erstellen:
   [crayon-66496eae7602b617381983/]
2. Den Konstruktor anpassen:
   [crayon-66496eae7602d579070994/]

3. die alte Context Klasse anpassen:
   [crayon-66496eae7602e909923212/]

4. Damit die Datenbank erstellt und aktuallisiert werden kann, fügen wir noch folgendes ein:
   [crayon-66496eae76030429600606/]

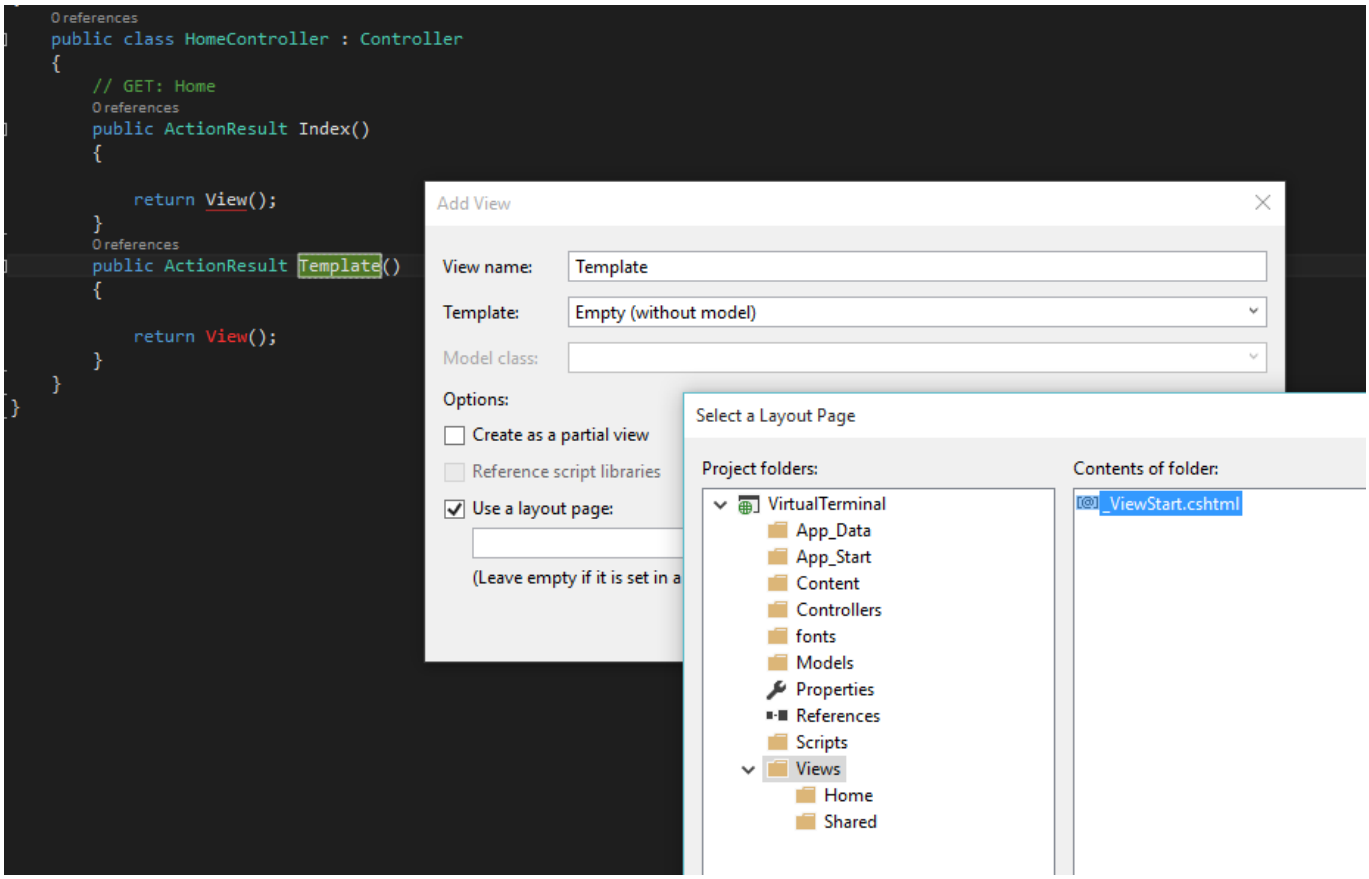
ASP MVC Html Template Erstellen und verwenden

Dazu erstellt man in den Ordner Views eine neue cshtml Datei, welche als Template dienen soll. Um diese vor anderen Views zu trennen empfiehlt es sich diese Template cshtml Seiten im shared Verzeichnis zu sichern.

Dieses Template kann z.B. ein Logo und den Footer beinhalten, die jede oder nur einige der Html Seiten beinhalten soll.
[crayon-66496eae76330946271074/]
Die Razor Anweisung @ViewBag.Title erlaubt es dann später jeder View einen anderen Titel vergeben zu können.

Die Anweisung @RenderBody() sagt aus, das hier der Inhalt der anderen Views stehen wird.

Im Controller kann man, wenn man nun eine neue View erstellt diese Template View auswählen:



Anschließend lässt sich im ViewBag.Title der Titel Der Seite anpassen und im Layout sehen wir wohin der Verweis führt.

Im unteren Block lässt sich nun einfach nur der gewünschte Inhalt anpassen
[crayon-66496eae76334268791016/]

Sections

Mann kann aber auch Innerhalb von Template Seiten an bestimmten Stellen Sektionen anfügen, die die Views mit eigenem Inhalt füllen soll.
[crayon-66496eae76335167625556/]
und in der View:
[crayon-66496eae76337015931468/]

ASP MVC View Razor Syntax

Eigentlich ist die Razor Syntax recht einfach gehalten, dennoch vergisst man sie schnell, wenn man nicht oft damit arbeitet.

- Einfachen Code Block erstellen: **@{ *C#* }**
- Variable ausgeben: **@VarName**
- String Zeichenkette ausgeben, ohne Variable(Ohne ” ”): **@:HalloWelt**
- Zeichenverkettung: **@var myVariable = “Baum”**
  @:ich habe einen schönen @(myVariable) der immer blüht.
- Textausgabe innerhalb des Code Blockes ohne HTML Tags: **@{ C# <Text> C# </Text>}**
- Textausgabe innerhalb des Code Blockes mit HTML Tags: **@{ *C#* <h1> *C#* </h1>}**
- Kommentar, der nicht in der HTML Seite sichtbar ist: **@*** ***mein Kommentar*** ***@**
- Normalerweise erkennt Razor automatisch, wenn es sich um eine E-Mail Adresse handelt und gibt diese auch so aus, falls dem nicht so ist, muss man das @ Zeichen mit **@@** Escapen, weil dieses Zeichen ja auch den Razor Code Block andeutet

---

Möchte man aus dem Controller etwas an die View übergeben, eignet sich der seit ASP 3 eingeführte dynamische Datentyp ViewBag. Dazu erstellt man im Controller einen neuen ViewBag: **ViewBag.VariablenName = “Wert”;**

und gibt diese in der View mit **@ViewBag.VariablenName** einfach

aus. Ein casten ist hiermit nicht notwendig.

Einfacher Logger mit C#

Wer kennt es nicht, in der Entwicklungsumgebung funktioniert die programmierte Software einwandfrei, auf einem anderen Testrechner aus unbekannten Gründen leider nicht mehr.

Um den Fehler schnell zu finden bieten sich Logfiles an. Gut dass .Net von Haus aus die Klasse Trace im namespace System.Diagnostics eine Lösung bietet.

Eine kleines Beispiel:

1. in der App.config den Tracer hinzufügen:
   [crayon-66496eae7653f633471868/]

2. Neue statische Classe erstellen:
   [crayon-66496eae76543350875047/]
   Nun kann man mit z.B: Logger.Info(“Programmstart”,”Main”) im Programmcode Informationen an die application.log Datei anhängen. Das tolle auch das Datum wird nun mit übertragen.