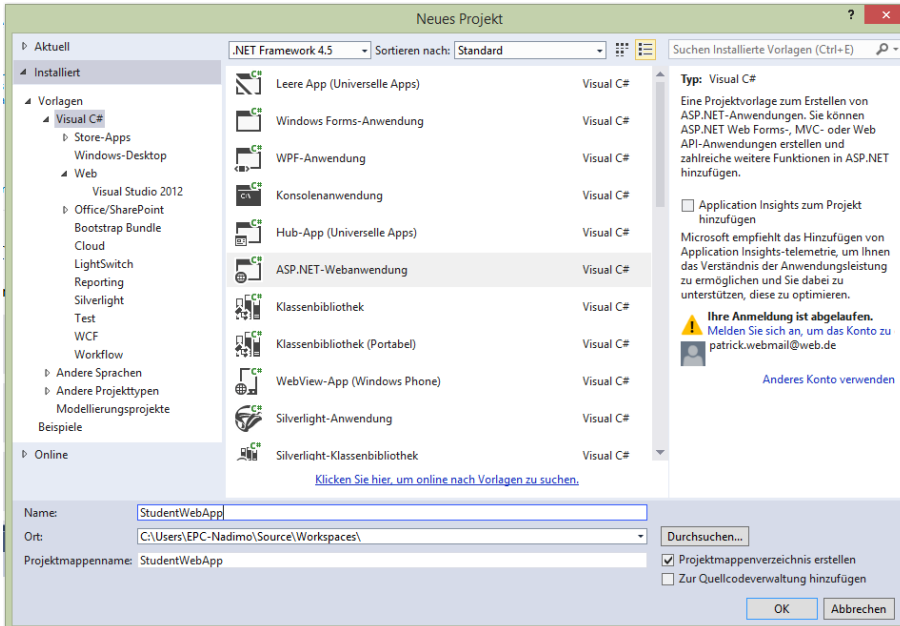


## Projektstart

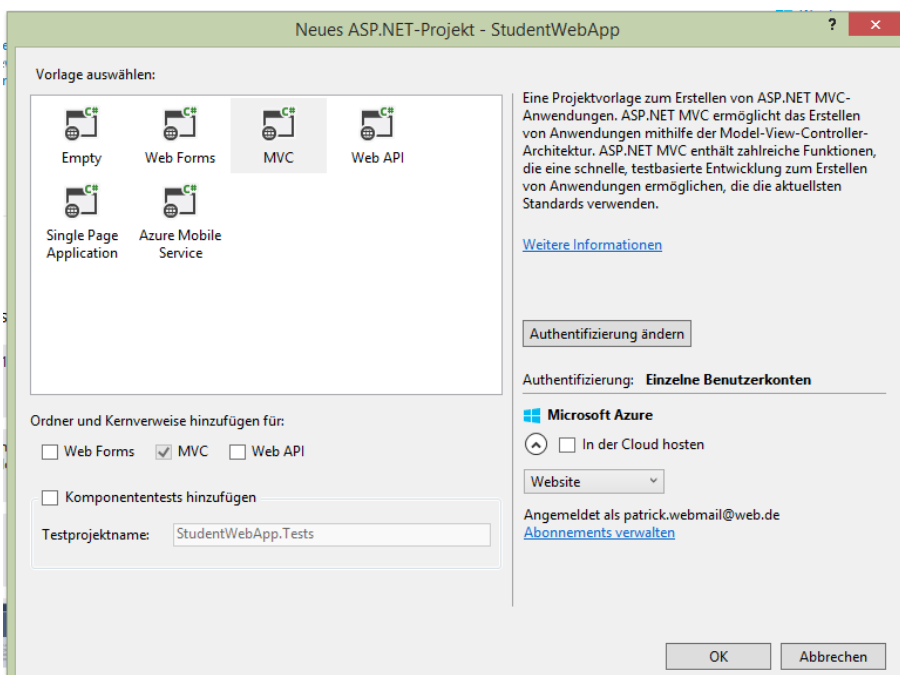
Wir beginnen ein ASP.NET MVC Projekt zu erstellen, dabei gehen wir wie folgt vor:



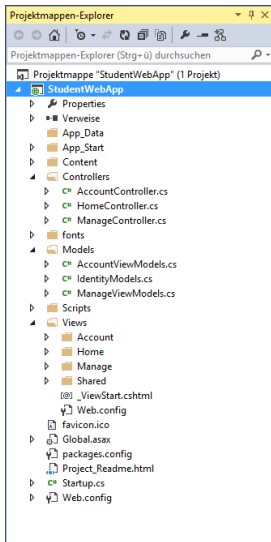
Zunächst wird ein neues Projekt als ASP.NET Webanwendung angelegt. Man sollte hierbei schon auf die Vergabe eines sprechenden Namens achten, da das Umstellen von Namespaces und im Nachhinein sehr mühsam sein kann und unnötig ist.

Hinweis:

Wählt eine nicht zu tiefe Ordertiefe, aufgrund von Pfadbeschränkung von 256 Zeichen in Gesamtlänge bei Windows kann es sonst zu Fehlern kommen.



Es sei darauf zu achten, dass die Authentifizierung auf „Einzelne Benutzerkonten“ (unter Authentifizierung ändern) gestellt ist – dies ist zwar eine Standardoption, hat man allerdings ein Projekt anders erstellt, so schleicht sich hier schon schnell mal ein irreparabler Fehler ein.



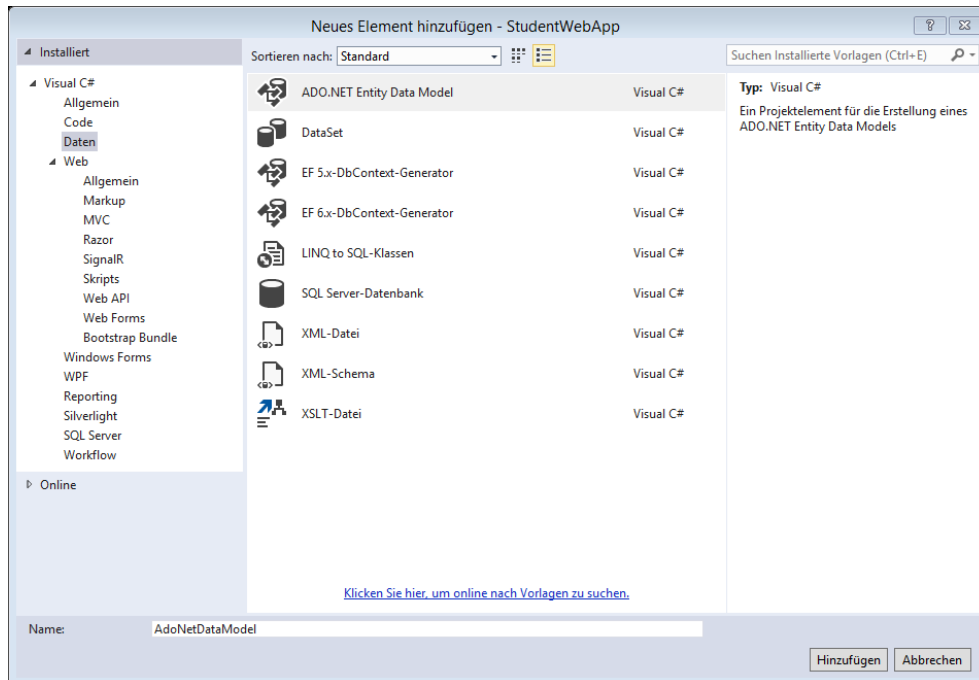
Nun, nachdem man das Projekt erstellen lassen hat, lässt sich schon gleich die klassische MVC Struktur im Projektmappe-Explorer erkennen.

Noch einmal zur Erinnerung:

- **Model:** Datenabstrahierungsschicht, sie hat die Attribute zu Objekten umgewandelten Datenstrukturen
- **View:** Die reine Darstellung, ihr Ergebnis ist immer in Form von HTML, CSS, JavaScript, etc. Output zu erkennen
- **Controller:** Hier findet die eigentliche Verarbeitungslogik statt, was aber auch heißt, dass alle View Anfragen von entsprechenden Controller hier verarbeitet werden.

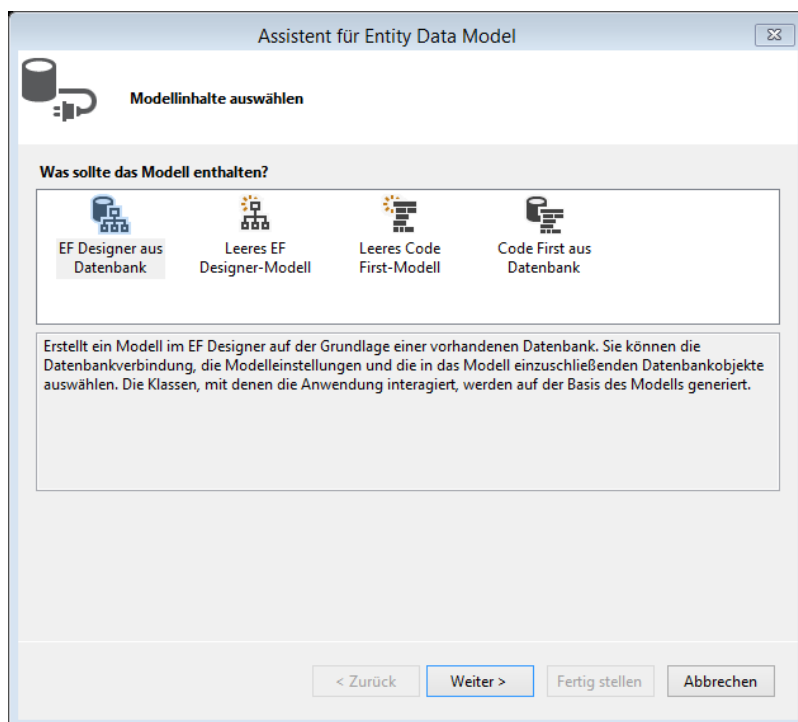
## Das Hinzufügen des Entity Data Models

Damit wir die Daten in Form von Objekten durch die Models im Code zugreifbar machen können, gehen wir wie folgt vor:



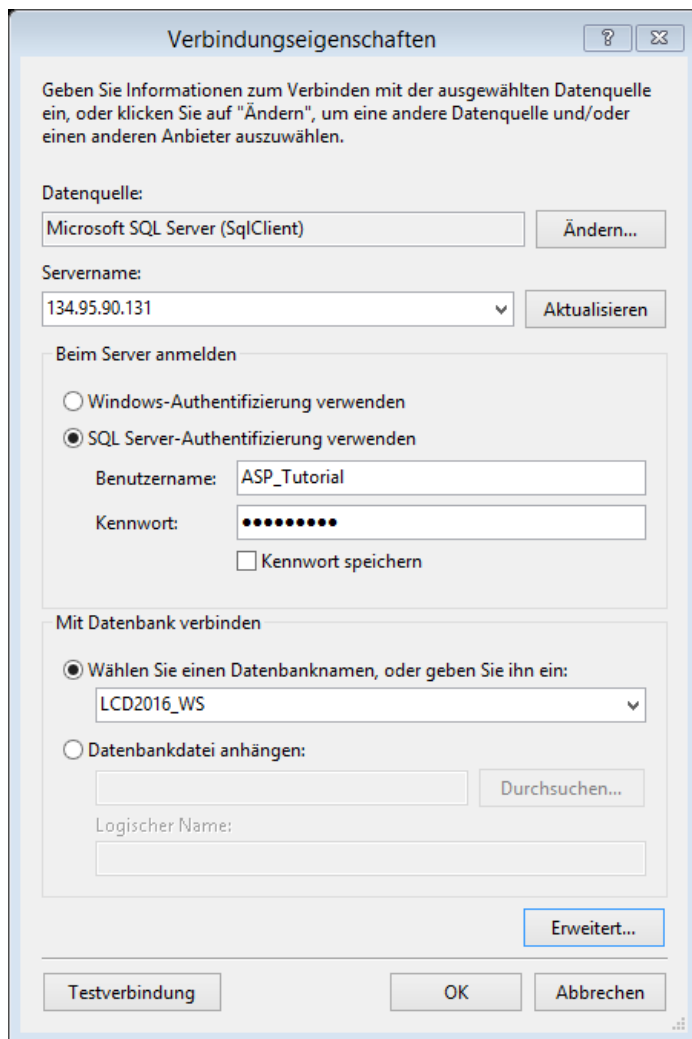
Mit einem Rechtsklick auf den Ordner der Models, können wir durch das Hinzufügen eines Elements und Navigieren zu „C# > Daten“ das „ADO.NET Entity Model“ auswählen, SPRECHEND benennen und somit hinzufügen.

Bsp.-Name: AdoNetDataModel



Aufgrund von Database-First Ansatz ist hier zu empfehlen, dass die Option „**EF Designer aus Datenbank**“ gewählt wird.

Stellt nun noch eine Verbindung zu euren SQL Servern her, die Daten können in Azure abgerufen werden. Hier heute in unserem Beispiel lauten sie wie folgt:



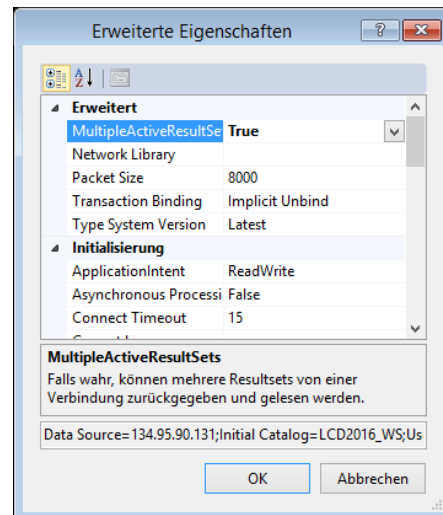
**Servername: 134.95.90.131**

**SQL Server Authentifizierung**

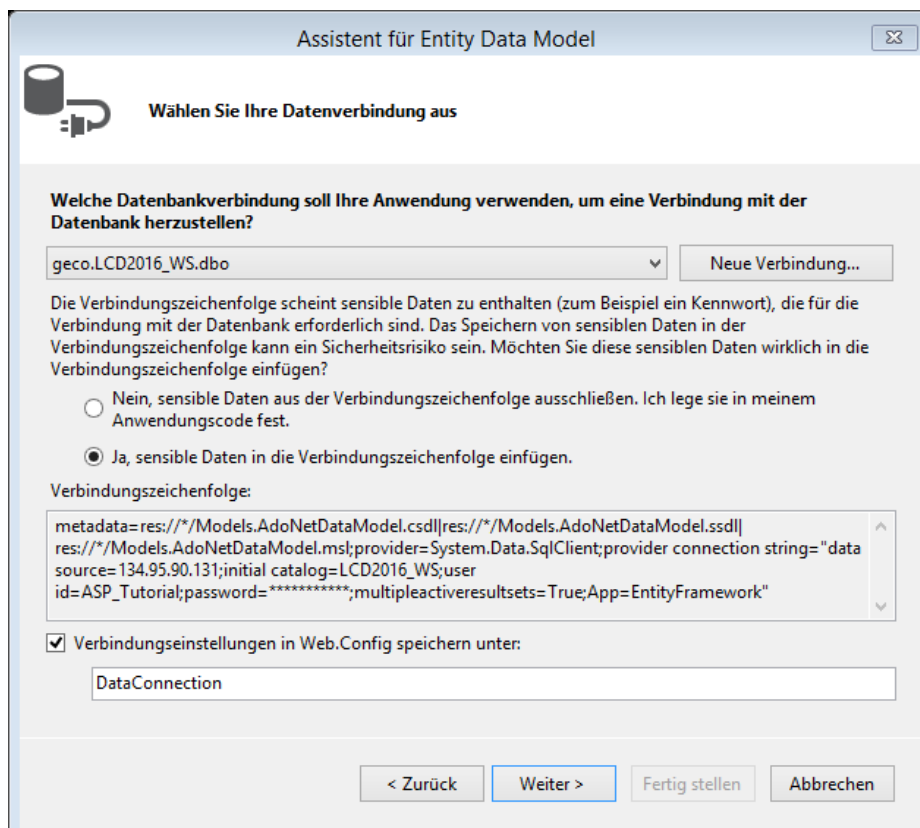
**Benutzer: ASP\_Tutorial**

**Passwort: LCD2017ws!**

**Datenbank: LCD2017\_WS**



Das **MultipleActiveResultSets** sollte auf **TRUE** gestellt werden.

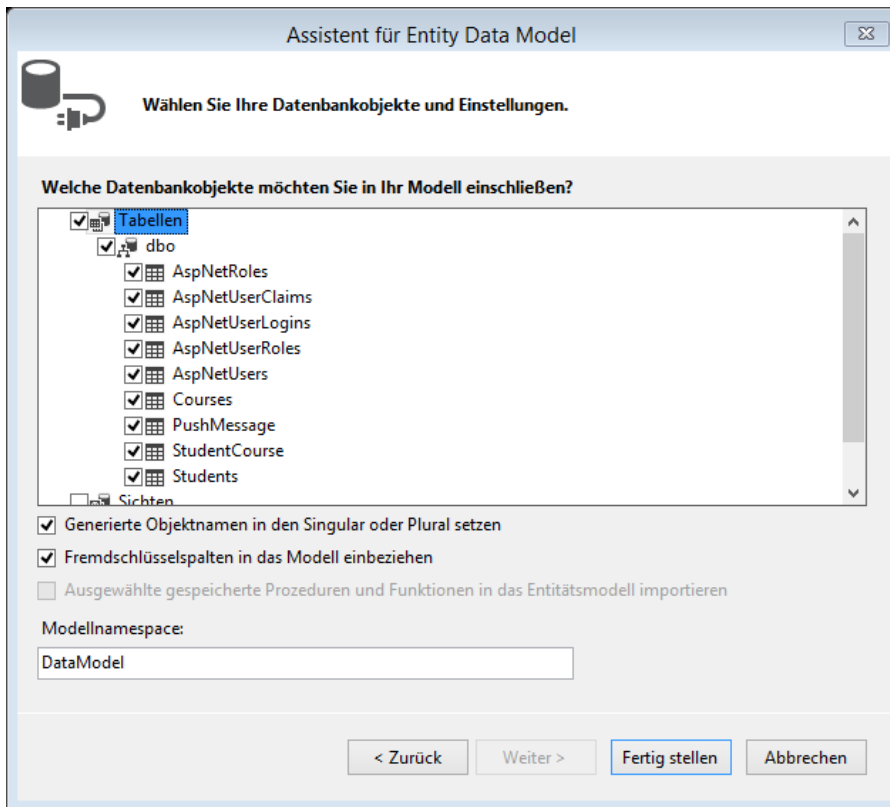


Vorerst kann die Option „Ja, sensible Daten in die Verbindungszeichenfolge einfügen“ ausgewählt werden. Mittel- bis Langfristig sollte man diese Daten aber über den Code füttern, denn niemand will unverschlüsselte Anmeldedaten herumfliegen haben!

Die Option Verbindungseinstellungen in Web.Config speichern unter auswählen. WICHTIG: Achtet auf den Namen.

Als Bsp. Hier:

**DataConnection**



Man wählt (denn nur so können alle Daten später abgefragt werden) alle Tabellen aus. Habt ihr momentan noch nicht die ASP.NET Tabellen in der DB? Nicht schlimm, sie können im Nachhinein hinzugefügt werden.

Folgende Option wählen: Generierte Objektnamen in den Singular oder Plural setzen.

Dieser Namespace ist wohl vom Namen der wichtigste, wählt in klug, denn das ist eure Klasse, den euch den Verweis auf die DB (auch via Linq) liefert.

Modellnamespace:  
**DataModel**

Im Anschluss: Das Projekt neu erstellen. Denn sonst sind die Klassen noch nicht von Visual Studio als bereits existierend erkannt.

Wo wir uns doch eh einmal mit der Connection befasst haben, können wir nun auch die standardmäßig für die Authentifizierung über OWIN ablaufende Connection Zeichenfolge in der Web.Config anpassen, die mit dem Namen „DefaultConnection“ vorliegt. Öffnet dazu das unterste Web.config File und ändert die Zeile mit dem Namen „DefaultConnection“ so ab, dass sie eine gültige Zeichenfolge für einen SQL Connection darstellt.

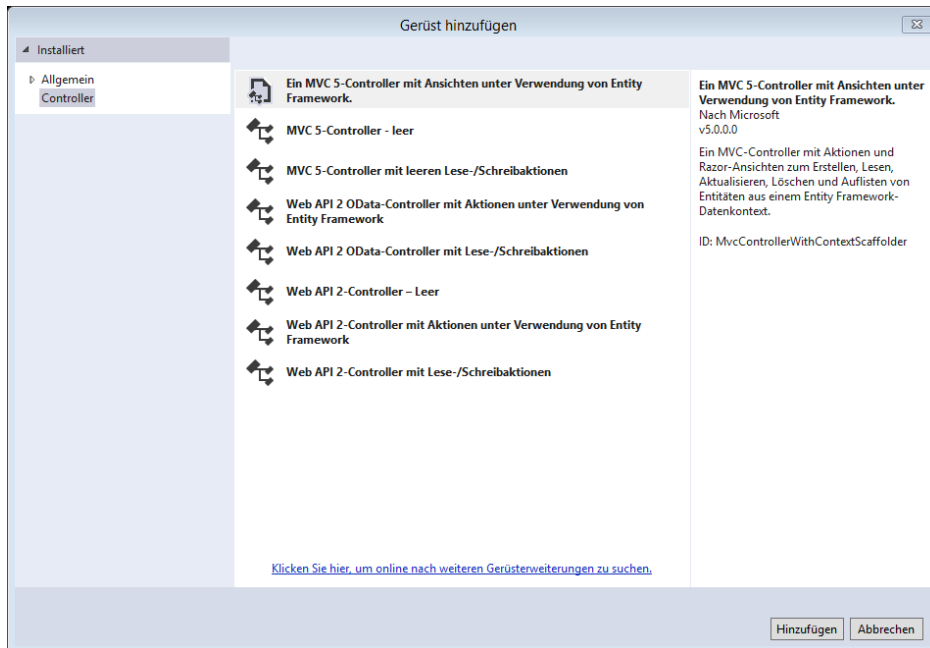
Hier sollte heute die Zeile wie folgt verändert werden:

```
<add name="DefaultConnection" connectionString="Data Source=134.95.90.131;Initial
Catalog=LCD2017_WS;Persist Security Info=True;User
ID=ASP_Tutorial;password=LCD2017ws!;MultipleActiveResultSets=True;Encrypt=True;TrustServerCertificat
e=True" providerName="System.Data.SqlClient" />
```

## Das Hinzufügen von Controllern

Damit nun auch ein wenig Logik und vor allem abrufbare Seiten in das Projekt kommen, gehen wir wie folgt vor, beim Hinzufügen von Controllern:

Im Rahmen des Vorkurses werden folgende Controller exemplarisch hinzugefügt: Students, Course, StudentCourse

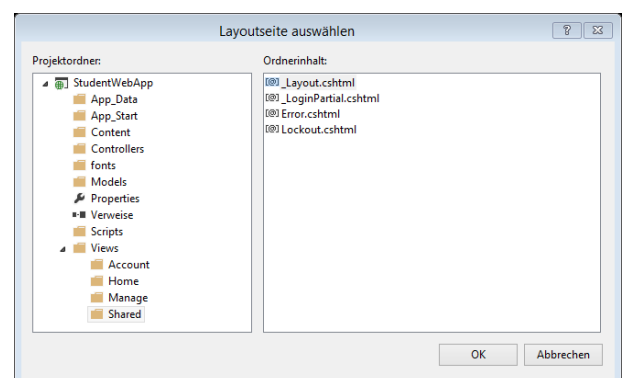
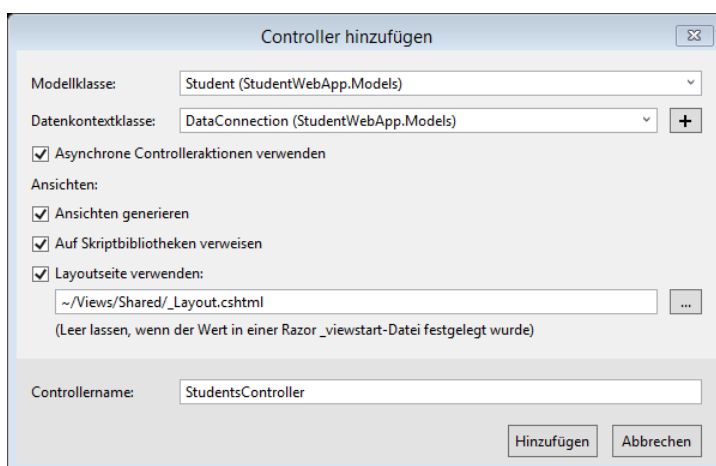


Mit einem Rechtsklick gehen wir auf den Oberordner der Controller, durch die Option Hinzufügen eines Controller wird uns die Option „MVC 5-Controllern mit Ansichten unter Verwendung von Entity Framework“ angeboten bzw. verwendet das englische äquivalente, dass auch Ansichten gleich mit hinzufügt.

Die folgenden Schritte sind intuitiv. Man wählt für die Modelklasse jeweils das Model, für das man Bearbeitungsmasken erstellen mag, man wählt die „DataConnection“ als Datenkontextklasse aus, und hakt die „asynchrone Verarbeitung“ an, ebenen wie „Ansichten generieren“, „Auf Skriptbibliotheken verweisen“ und „LayoutSeite verwenden“.

Durch die letzte Option wird die Möglichkeit geboten ein Layout (sprich eine HTML Schablone für den Inhalt der Edit, Delete, Details und Create Ansicht).

Die Namen können jeweils so belassen werden. Hier in unserem Beispiel machen wir dies nun also drei Mal:



Controller hinzufügen

Modellklasse: StudentCourse (StudentWebApp.Models)

Datenkontextklasse: DataConnection (StudentWebApp.Models) +

☒ Asynchrone Controlleraktionen verwenden

Ansichten:

☒ Ansichten generieren

☒ Auf Skriptbibliotheken verweisen

☒ Layoutseite verwenden: ~/Views/Shared/\_Layout.cshtml ...  
(Leer lassen, wenn der Wert in einer Razor \_viewstart-Datei festgelegt wurde)

Controllername: StudentCoursesController

Hinzufügen Abbrechen

Controller hinzufügen

Modellklasse: Cours (StudentWebApp.Models)

Datenkontextklasse: DataConnection (StudentWebApp.Models) +

☒ Asynchrone Controlleraktionen verwenden

Ansichten:

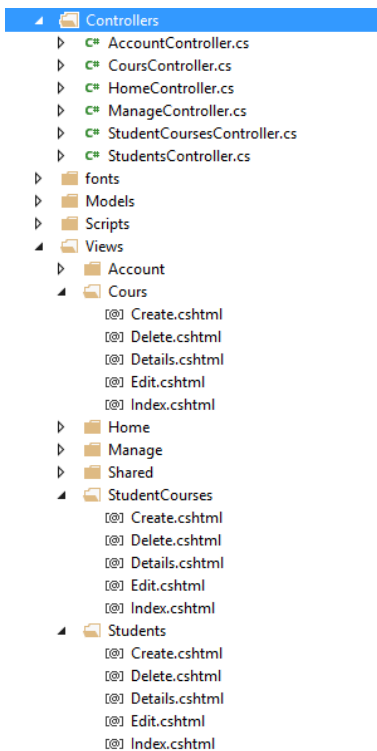
☒ Ansichten generieren

☒ Auf Skriptbibliotheken verweisen

☒ Layoutseite verwenden: ~/Views/Shared/\_Layout.cshtml ...  
(Leer lassen, wenn der Wert in einer Razor \_viewstart-Datei festgelegt wurde)

Controllername: CoursController

Hinzufügen Abbrechen



Somit wurden im Hintergrund sofort die Views automatisch mit angelegt. Im Wesentlichen funktioniert nun eigentlich alles.

Optional: Falls durch das Anlegen eines Benutzers automatisch auch noch andere Models berührt werden

Da wir beim Anlegen eines Benutzers gleichzeitig einen Studierenden in der Datenbank mit anlegen wollen, da dieser ganz eigene Attribute hat (und ein Abändern des OWIN Users nicht ratsam ist) verändern wir im Controller Account die Methode Register (bei dem ein das valide Model bereits übergeben wurde).

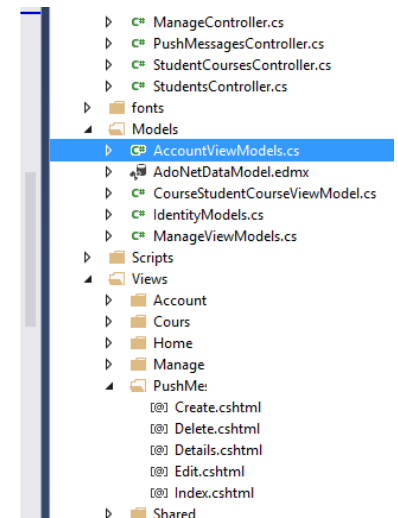
Zuerst ergänzen wir im RegisterViewModel weitere Attribute:

```
1-Verweis
public class RegisterViewModel
{
    [Required]
    [EmailAddress]
    [Display(Name = "E-Mail")]
    2-Verweise
    public string Email { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = @"\{0}\\" muss mindestens {2} Zeichen lang sein.", MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Kennwort")]
    1-Verweis
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Kennwort bestätigen")]
    [Compare("Password", ErrorMessage = "Das Kennwort entspricht nicht dem Bestätigungskennwort.")]
    0-Verweise
    public string ConfirmPassword { get; set; }

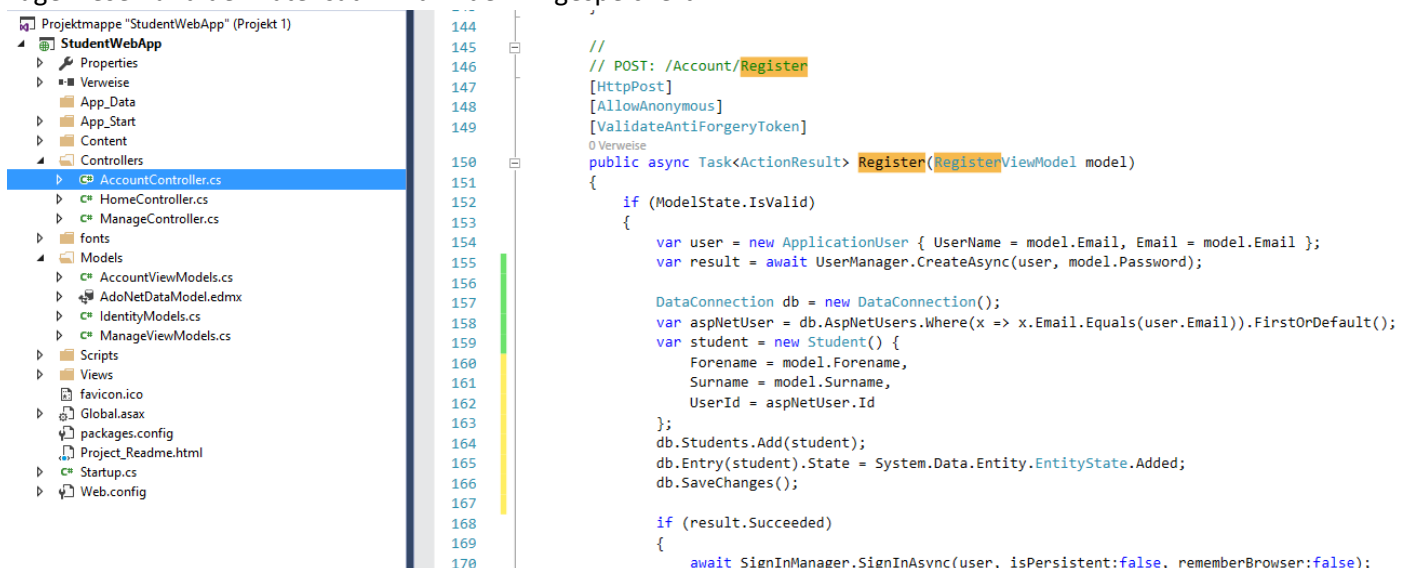
    1-Verweis
    public string Forename { get; set; }
    1-Verweis
    public string Surname { get; set; }
}
```



Folgende Attribute fügen wir dem RegisterViewModel hinzu:

```
public string Forename { get; set; }
public string Surname { get; set; }
```

Hier legen wir einen neuen Benutzer an, „userManager.CreateAsync(...)“ und greifen durch das Objekt der DataConnection den dadurch angelegten User ab (Object: aspNetUser). Nun wird ein neuer Studierender angelegt, die entsprechend zusätzlich angelegten Attribute des Models Account (s. unten) werden dem Studierenden zugewiesen und der Datensatz wird in der DB gespeichert.





Folglich fügen folgende Zeilen hinzu:

```
SqlConnection db = new SqlConnection();
    var aspNetUser = db.AspNetUsers.Where(x => x.Email.Equals(user.Email)).First();
    var student = new Student { Forename = model.Forename, Surname = model.Surname,
UserId = aspNetUser.Id};
    db.Students.Add(student);
    db.Entry(student).State = System.Data.Entity.EntityState.Added;
    db.SaveChanges();
```

## Anpassungen an der Register-View

Damit die im Model vorgenommenen Ergänzungen bezüglich der Angabe von Vor- und Nachnamen auch bei der Registrierung erfasst werden können, bedarf es der Ergänzung folgender Felder:

```
<div class="form-group">
    @Html.LabelFor(m => m.Forename, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.TextBoxFor(m => m.Forename, new { @class = "form-control" })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(m => m.Surname, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.TextBoxFor(m => m.Surname, new { @class = "form-control" })
    </div>
</div>
```

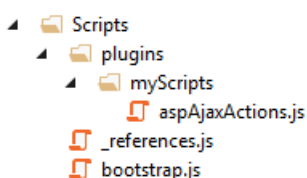
## Erweitern eines Menübereichs durch die neu gewonnen Seiten

Damit unsere Menüführung auch steht's aktuell ist, führen wir weitere Anpassungen in der Layout.cshtml für die Menüführung aus. So fügen wir weitere Action Links, verweisend auf die Index Seiten der hinzugefügten Controller in den Bereich „nav“ hinzu. Sobald man einen Bereich mit `@if(Request.IsAuthenticated){...}` umklammert, ist dieser Bereich nur für angemeldete Benutzer sichtbar.

```
<ul class="nav navbar-nav">
    @if (Request.IsAuthenticated)
    {
        //<li>@Html.ActionLink("Veranstaltungsbelegung", "Manage", "Students")</li>
        <li>@Html.ActionLink("Studierende", "Index", "Students")</li>
        <li>@Html.ActionLink("Veranstaltungen", "Index", "Cours")</li>
        <li>@Html.ActionLink("Belegung", "Index", "StudentCourses")</li>
    }
</ul>
```

## Nachladen von Paketen

Zum Hinzufügen von Paketen, bei Erweiterung des Systems um Frameworks oder Script Packages wird wie folgt vorgegangen:



Man erweitert das Projekt um die Daten der Pakete, die man hinzufügen will. Hier haben wir den Ordner Scripts um den Ordner plugins ergänzt und das Paket myScripts datentechnisch hinzugefügt.

Im BundleConfig müssen wir diese Pakete nun noch registrieren.

```
// myScripts
bundles.Add(new ScriptBundle("~/plugins/myScripts").Include(
    "~/Scripts/plugins/myScripts/aspAjaxActions.js"));
```

```
// jQueryUnobtrusive
bundles.Add(new ScriptBundle("~/bundles/jqueryunobtrusive").Include(
    "~/Scripts/jqueryunobtrusive-ajax.js",
    "~/Scripts/jqueryunobtrusive-ajax.min.js"));
```

Dies geschieht in unserem Fall durch das Ergänzen folgender Zeilen, Äquivalent also auch bei allen möglichen anderen Paketen.

```
// myScripts
bundles.Add(new ScriptBundle("~/plugins/myScripts").Include(
    "~/Scripts/plugins/myScripts/aspAjaxActions.js"));
```

```
// jQueryUnobtrusive
bundles.Add(new ScriptBundle("~/bundles/jqueryunobtrusive").Include(
    "~/Scripts/jqueryunobtrusive-ajax.js",
    "~/Scripts/jqueryunobtrusive-ajax.min.js"));
```

Damit wir diese Pakete und Frameworks auch wirklich verwenden können, müssen wir nun noch die eigentliche View anpassen. Dies geht wie folgt:

```
@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/bootstrap")
@Scripts.Render("~/plugins/myScripts")
@Scripts.Render("~/bundles/jqueryunobtrusive")
@RenderSection("styles", required: false)
@RenderSection("scripts", required: false)
</body>
```

Wir erweitern den Code um die Zeilen der benötigten Pakete. Falls wir die Pakete überall benötigen geschieht dies direkt auf der \_Layout.cshmtl, falls man das Packet nur auf manchen Seiten braucht, so fügt man es entsprechend nur dort ein wo man es verwendet (spart ja immerhin Ladezeit der Website).

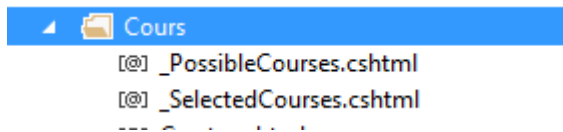
Innerhalb der Klammer von Render wird der in der BundleConfig vergebene BundleName übergeben, man wählt ihn gerne in Format eines Pfades:

```
@Scripts.Render("~/plugins/myScripts")
@Scripts.Render("~/bundles/jqueryunobtrusive")
@RenderSection("styles", required: false)
```

## Anlegen einer AJAX Partial View

Um eine PartialView anzulegen, gehen wir wie folgt vor:

Für unser Beispiel wollen wir dies mithilfe der unter Course, mit: `_PossibleCourses.cshtml` und `_SelectedCourses.cshtml` anzulegenden PartialViews zeigen.



Für die genannten beiden Partialen (per Konvention immer mit einem Unterstrich im Namen beginnend) kopieren wir jeweils die `Index.cshtml` von Course und benennen die Dokumente entsprechend in `_PossibleCourses.cshtml` und `_SelectedCourses.cshtml` um.

In diesen PartialViews sollte das Layout in immer auf null gesetzt werden:

Layout = null;

```
@{
    ViewBag.Title = "Ihre Belegungen";
    Layout = null;
}
```

```
@{
    ViewBag.Title = "Veranstaltungskatalog";
    Layout = null;
}
```

Der Inhalt der beiden Partialen ist im wesentlichen ähnlich und sieht wie folgt aus:

(`_SelectedCourses.cshtml`)

```
@model IEnumerable<StudentWebApp.Models.Course>

@{
    ViewBag.Title = "Ihre Belegungen";
    Layout = null;
}

<h2>Index</h2>

<h2>@ViewBag.Title</h2>
<h4>Status: Anfrage (0), Belegt (1), Abgelehnt (-1)</h4>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Description)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.MaxStudents)
        </th>
        <th>Status</th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Description)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.MaxStudents)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.StudentCourses.Where(x => x.CourseId ==
                    item.Id && x.Student.AspNetUser.UserName.Equals(User.Identity.Name)).First().StatusCode)
            </td>
        </tr>
    }
</table>
```

Bzw. so:

(\_PossibleCourses.cshtml)

```
@model IEnumerable<StudentWebApp.Models.Cours>

@{
    ViewBag.Title = "Veranstaltungskatalog";
    Layout = null;
}

<h2>Index</h2>

<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Description)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.MaxStudents)
        </th>
        <th></th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Description)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.MaxStudents)
            </td>
            <td>
                <input type="button" value="Belegen" onclick="ajaxJSONAction('/Cours/Request/', 'POST',
'courseId:@item.Id ', '', false); reloadPage()" />
            </td>
        </tr>
    }
</table>
```

Damit diese beiden Partial Views nun auf einer View zusammengefügt werden, erstellen wir die View Manage.cshtml unter Students. Wir gehen auf den Ordner Students und können durch Rechtsklick und Hinzufügen eine leere View hinzufügen, ohne Model und mit Standard-Layout.

```
Students
  Create.cshtml
  Delete.cshtml
  Details.cshtml
  Edit.cshtml
  Index.cshtml
  Manage.cshtml
  ViewStart.cshtml
```

Die Manage Datei besteht im Grunde nur aus zwei div Bereichen, die den Bereich für die PartialViews vorgeben. Im JavaScript Bereich, werden diese PartialViews mithilfe von AJAX und dem Tool myScripts in die View geladen. Dies geschieht auch schon direkt nach dem Öffnen der Webseite.

```
@{
    ViewBag.Title = "Manage";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Studierenden-Verwaltung / Veranstaltungsmanagement</h2>

<div id="_partialCourseList">
</div>

<br />

<div id="_partialMyCourses">
</div>

@section Scripts {
    <script type="text/javascript">
        $(document).ready(function () {
            reloadPage();
        });

        function reloadPage() {
            partialLoad('/Cours/_PossibleCourses/', 'GET', '_partialCourseList', 'html', '');
            partialLoad('/Cours/_SelectedCourses/', 'GET', '_partialMyCourses', 'html', '');
        }
    </script>
}
```

### Entfernen der „Create“ Aktionen in kopierten Views

Die „Create“ Aktionen der kopierten Index in in \_PossibleCourses.cshtml und \_SelectedCourses.cshtml sollten aus den Views gelöscht werden.

### Hinzufügen der Annehmen/Ablehnen Button in StudentCourse View

Die „Index“-View des „StudentCourses“ wird um die Button „Annehmen“ und „Ablehnen“, anstatt „Edit / Detail und Delete“ wie folgt angepasst:

```
<td>
    <input type="button" value="Annehmen"
        onclick="ajaxJSONAction('/StudentCourses/Answer/', 'POST',
'CourseId:@item.CourseId, StudentId:@item.StudentId, StatusCode:1', '', false); location.reload();"
/> |
    <input type="button" value="Ablehnen"
        onclick="ajaxJSONAction('/StudentCourses/Answer/', 'POST',
'CourseId:@item.CourseId, StudentId:@item.StudentId, StatusCode:-1', '', false); location.reload();"
/>
</td>
```

## Individuelle Anpassungen im Controller

Nur im Rahmen dieses Beispiels relevant sind zudem folgende Anpassungen in den Controllern.

### Anpassungen im CourseController:

```
// GET Partial: _PossibleCourses
public ActionResult _PossibleCourses()
{
    var student = db.Students.Where(x =>
x.AspNetUser.UserName.Equals(User.Identity.Name)).First();
    var studentCourses = student.StudentCourses.Select(x => x.CourseId);

    return PartialView(db.Courses.Where(x => !studentCourses.Contains(x.Id)).ToList());
}

// GET Partial: SelectedCourses
public ActionResult _SelectedCourses()
{
    var student = db.Students.Where(x =>
x.AspNetUser.UserName.Equals(User.Identity.Name)).First();
    var studentCourses = student.StudentCourses.Select(x => x.CourseId);

    return PartialView(db.Courses.Where(x => studentCourses.Contains(x.Id)).ToList());
}

[HttpPost]
public ActionResult Request(int courseId)
{
    var student = db.Students.Where(x =>
x.AspNetUser.UserName.Equals(User.Identity.Name)).First();
    var studentCourse = new StudentCourse ()
    {
        CourseId = courseId,
        StudentId = student.Id,
        StatusCode = 0
    };
    db.StudentCourses.Add(studentCourse);
    db.SaveChanges();
    return null;
}
```

### Anpassungen im StudentCourseController:

```
// GET: StudentCourses/Answer
// Belegungsstatus der Belegungsanfrage aendern
public ActionResult Answer(int StudentId, int CourseId, int StatusCode)
{
    var studentCourse = db.StudentCourses.Where(x => x.StudentId == StudentId &&
x.CourseId == CourseId).First();
    studentCourse.StatusCode = StatusCode;

    db.Entry(studentCourse).State = EntityState.Modified;
    db.SaveChanges();

    return null;
}
```

### Anpassungen im StudentController:

```
// GET: Students/Manage
public async Task<ActionResult> Manage()
{
    return PartialView();
}
```

### Anpassung d. Menüstruktur:

Damit unsere Menüführung auch nun auch die Veranstaltungsbelegung mit anzeigt, kommentieren wir den entsprechenden Eintrag wieder in unseren Code ein:

```
<li>@Html.ActionLink("Veranstaltungsbelegung", "Manage", "Students")</li>
```