

# はじめての ASP.NET MVC 5

# About me

- Tomo Mizoe
  - CEO & Founder of July Inc.
  - <http://www.july.co.jp>
  - Twitter: @tmizoe
- 
- Microsoft Certified Trainer
  - Microsoft Certified Solution Developer
    - Windows Store Apps using HTML5 and JavaScript
    - Web Applications

- Getting Started with ASP.NET MVC 5
- [Scott Guthrie](#) (twitter [@scottgu](#) ),  
[Scott Hanselman](#) (twitter: [@shanselman](#) ),  
[Rick Anderson](#) (twitter: [@RickAndMSFT](#) )

<http://www.asp.net/mvc/tutorials/mvc-5/introduction/getting-started>

# ASP.NET MVC の特徴

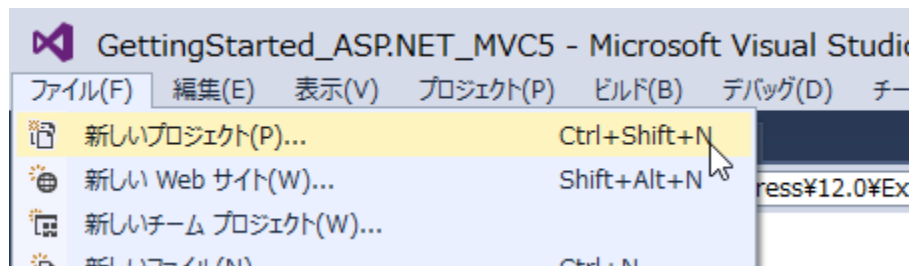
- MVC
- C#
- 統合環境 Visual Studio
  - コード補完
  - 文法チェック
- Facebook、Twitter、Office 365 など各種認証のモジュール導入も簡単
- Single Page Applicationにも対応（AngularJS 等）
- デプロイ
  - Azureへクリック1回
  - Azureだからスケールアップもクリック2-3回

# 用意するもの

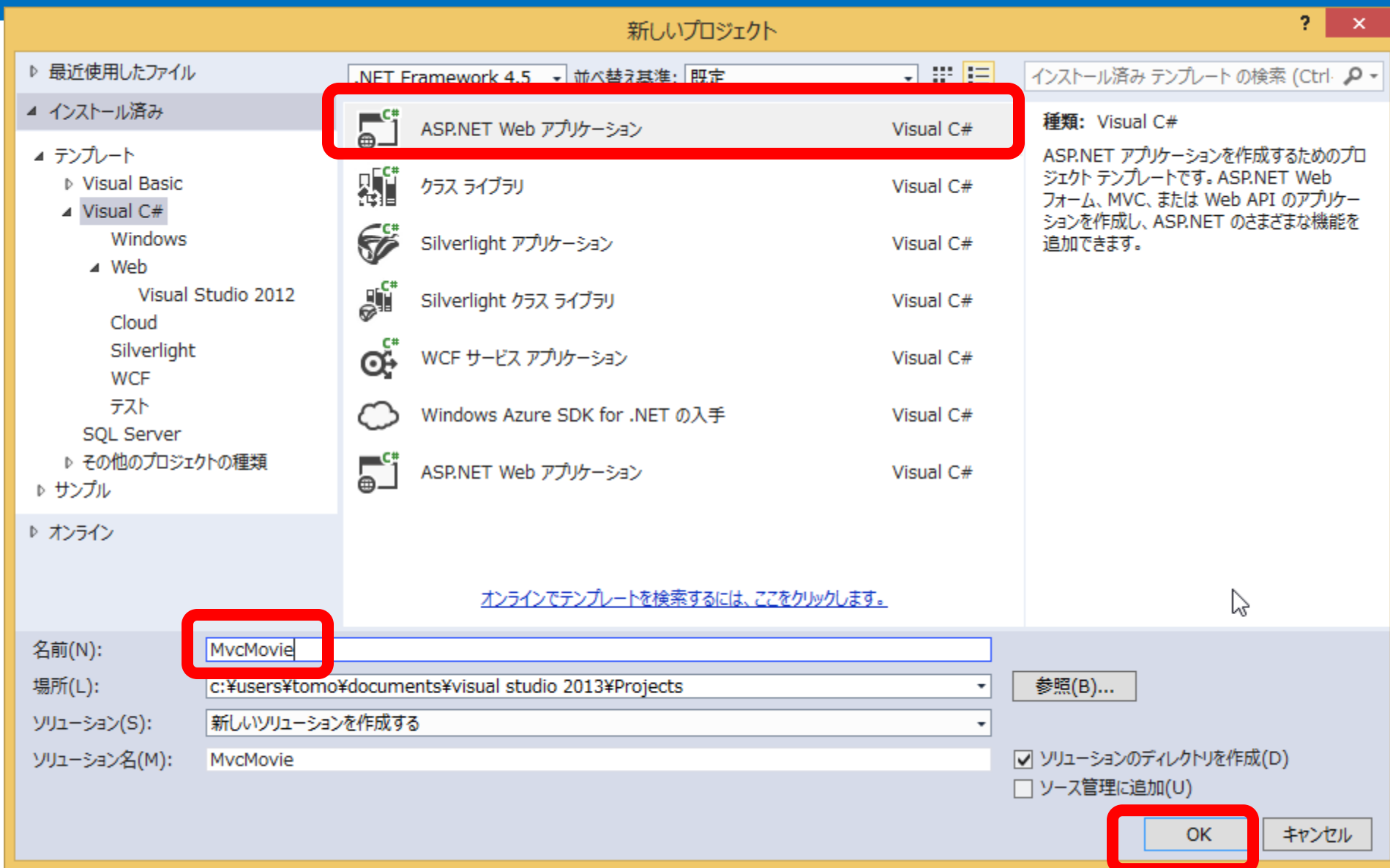
- Windows OS
- Visual Studio 2013 Express for Web (無償)  
有償版でも、もちろんOK

# 新プロジェクト作成







- ファイル ⇒ 新しいプロジェクト



# インストール済み : Visual C# : ASP.NET Webアプリケーション



テンプレートの選択(S):

 Empty	 Web Forms	 MVC	 Web API
 Single Page Application	 Facebook		

以下にフォルダーおよびコア参照を追加:


☐ Web Forms ☒ MVC ☐ Web API☐ 単体テストの追加テスト プロジェクト名: 

ASP.NET MVC アプリケーションを作成するためのプロジェクト テンプレート。ASP.NET MVC では、Model-View-Controller アーキテクチャを使用してアプリケーションを構築できます。ASP.NET MVC には、高速なテスト駆動開発をはじめとした最新の標準技術を使用するアプリケーションを作成するための多くの機能が備わっています。

[詳細](#)

認証の変更(A)

認証: 個人ユーザー アカウント

 **Windows Azure**☒ クラウド内のホスト(H)

Web サイト

[サブスクリプションの管理](#)

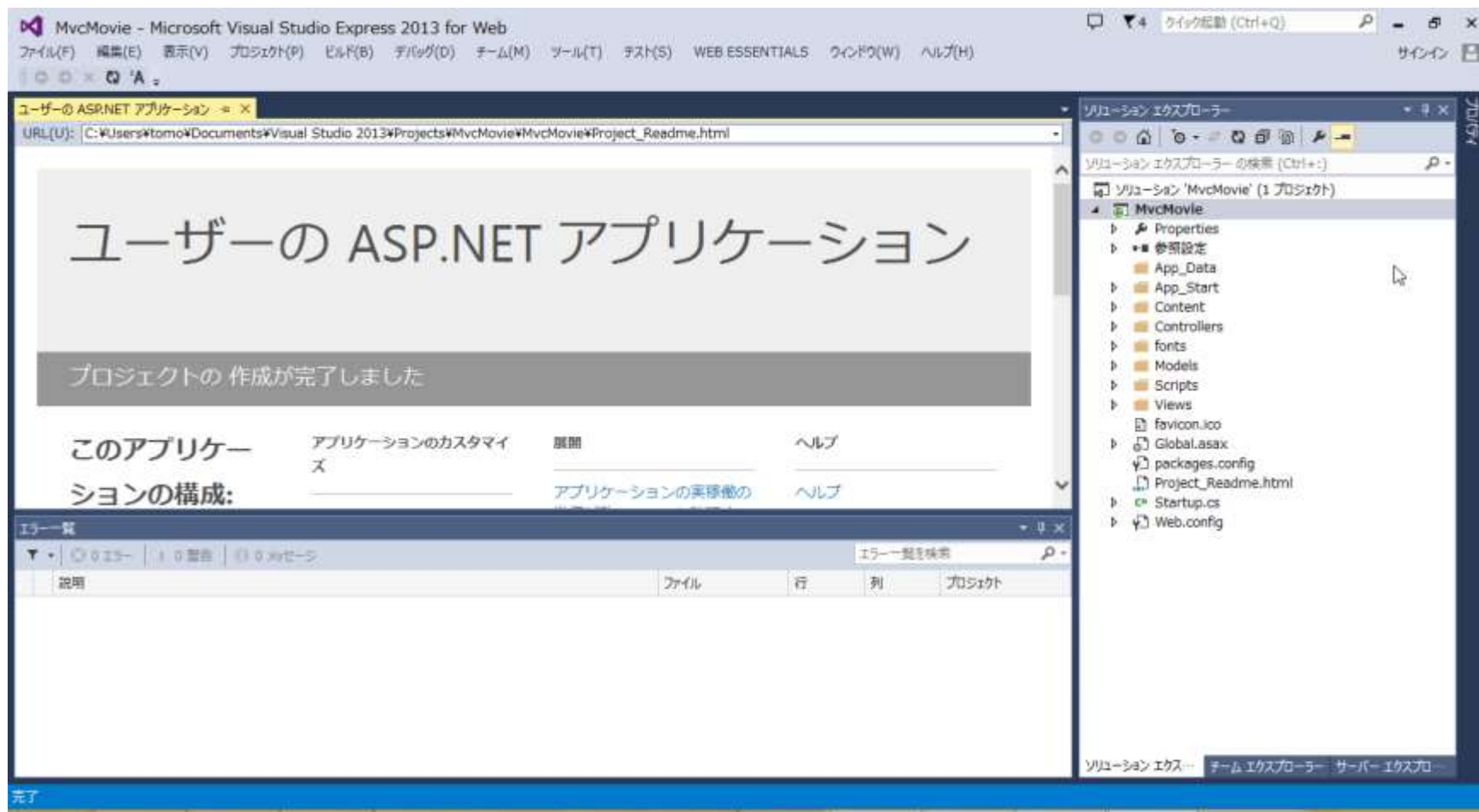
OK

キャンセル

テンプレート「MVC」  
Windows Azure「クラウド内のホスト : オフ」

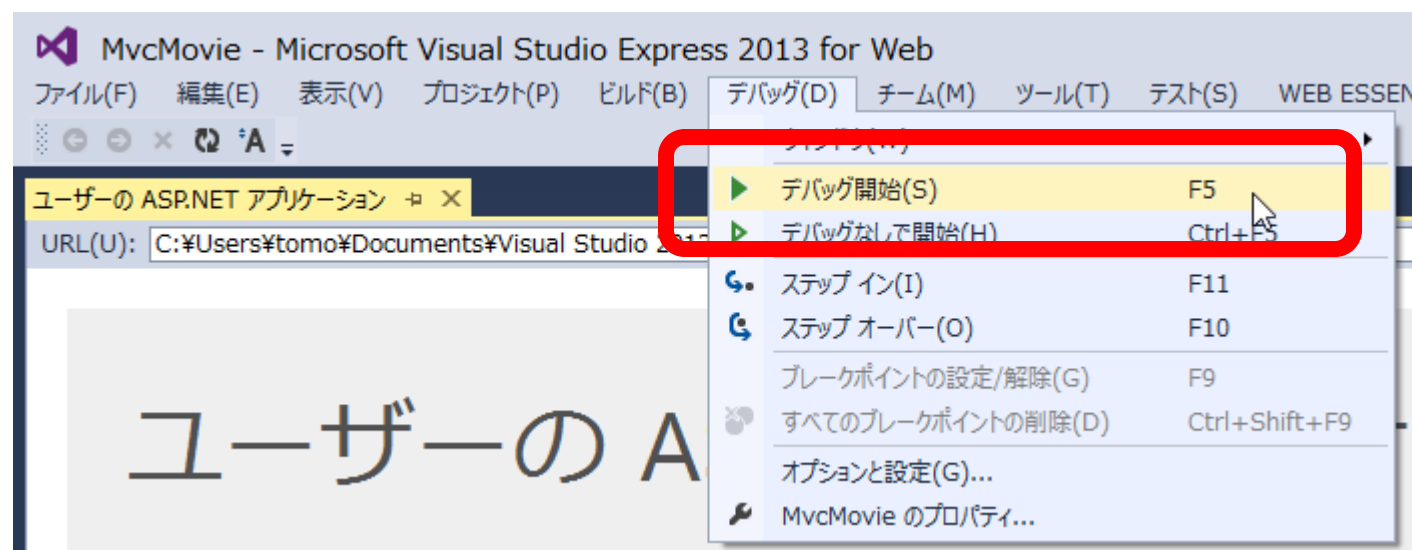


# 新しいプロジェクトの完成



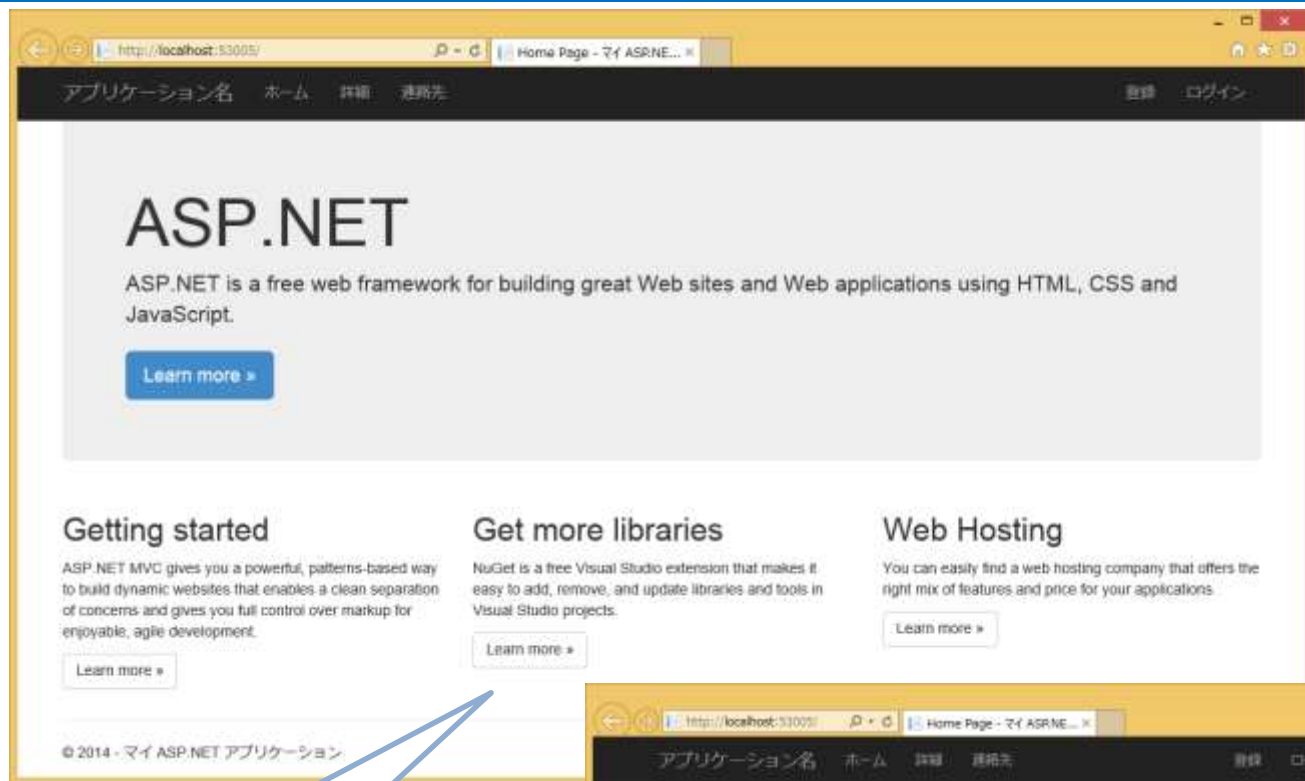
# まだ何もしてないけどコンパイルして実行してみる

- デバッグ⇒デバッグ開始
- または F5



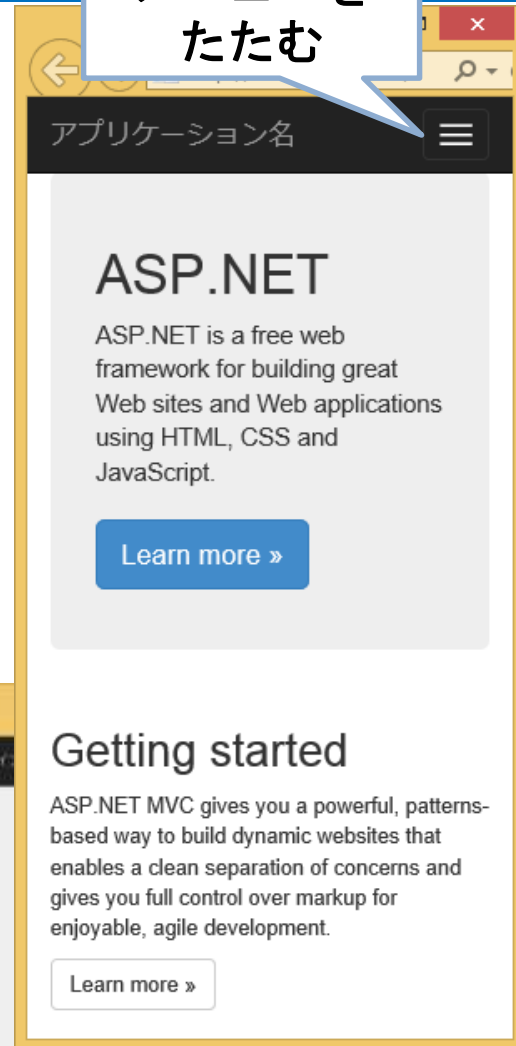
# 何もしてないけどレスポンス

スマホは  
メニューを  
たたむ



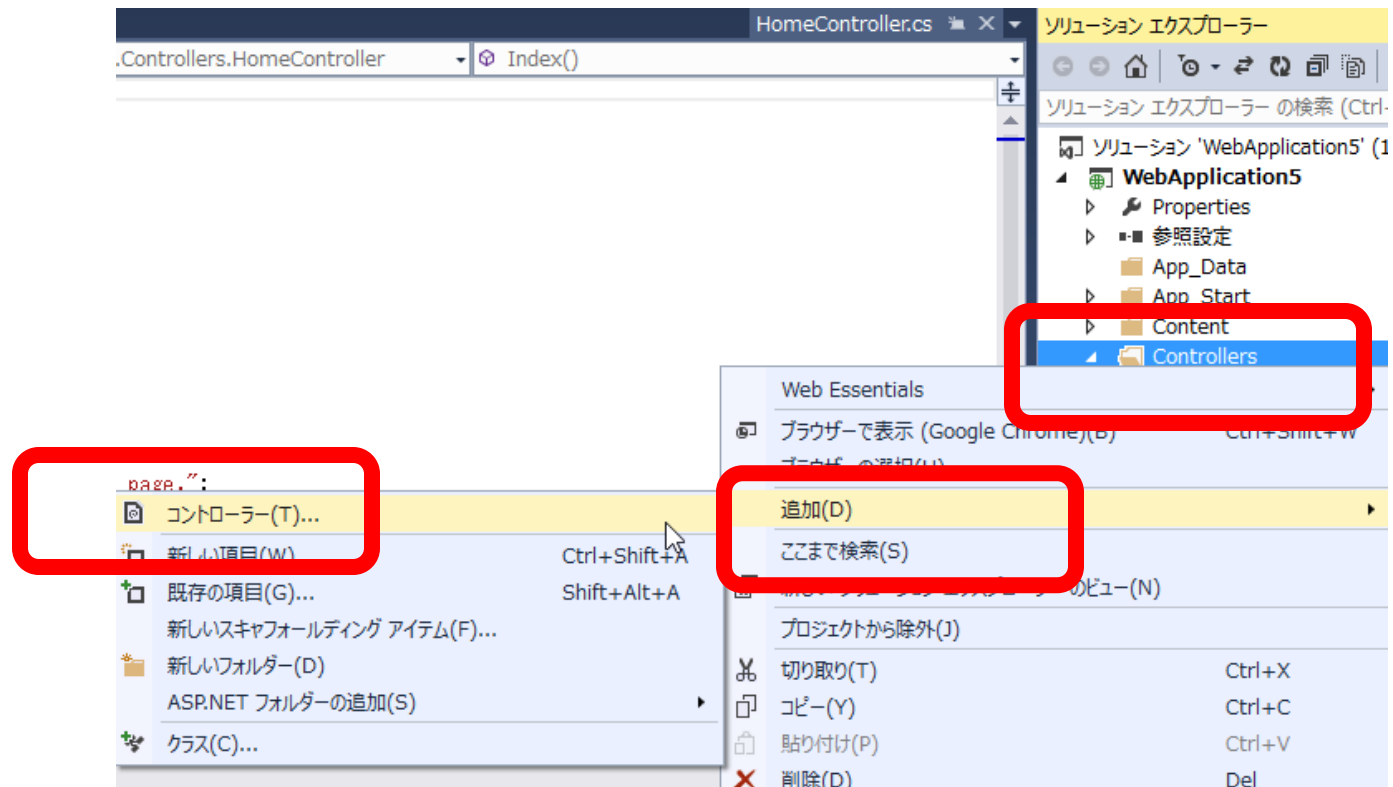
PCは3カラム

タブレットとスマホは  
1カラム



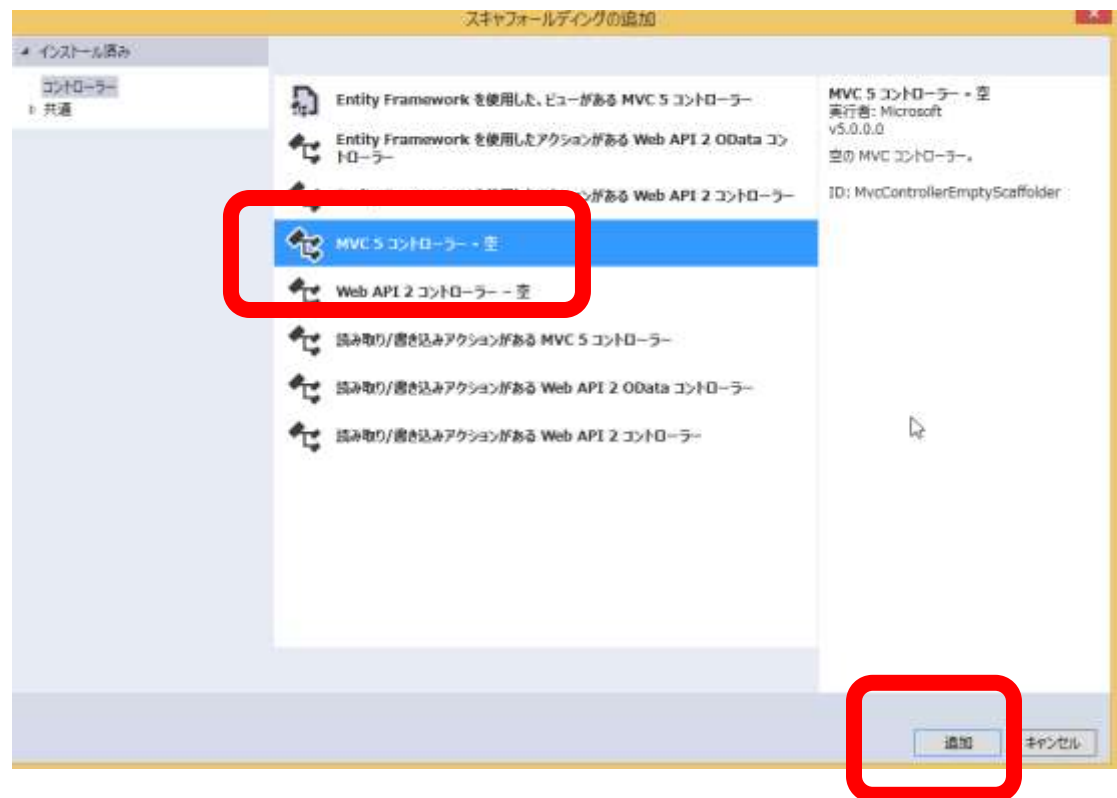
# コントローラ追加

- コントローラ：処理の振り分け担当
- ソリューションエクスプローラ⇒（プロジェクト名）  
⇒ Controllersフォルダ右クリック⇒追加  
⇒コントローラー



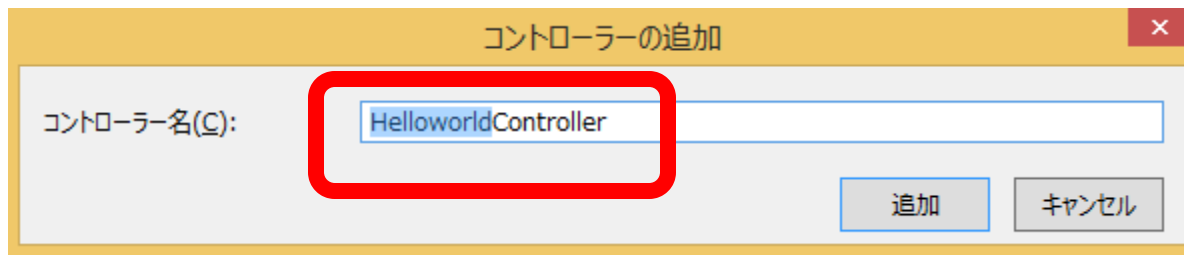
# コントローラ追加

- MVC 5 コントローラ⇒追加



# コントローラ追加

- コントローラ名 : HelloworldController

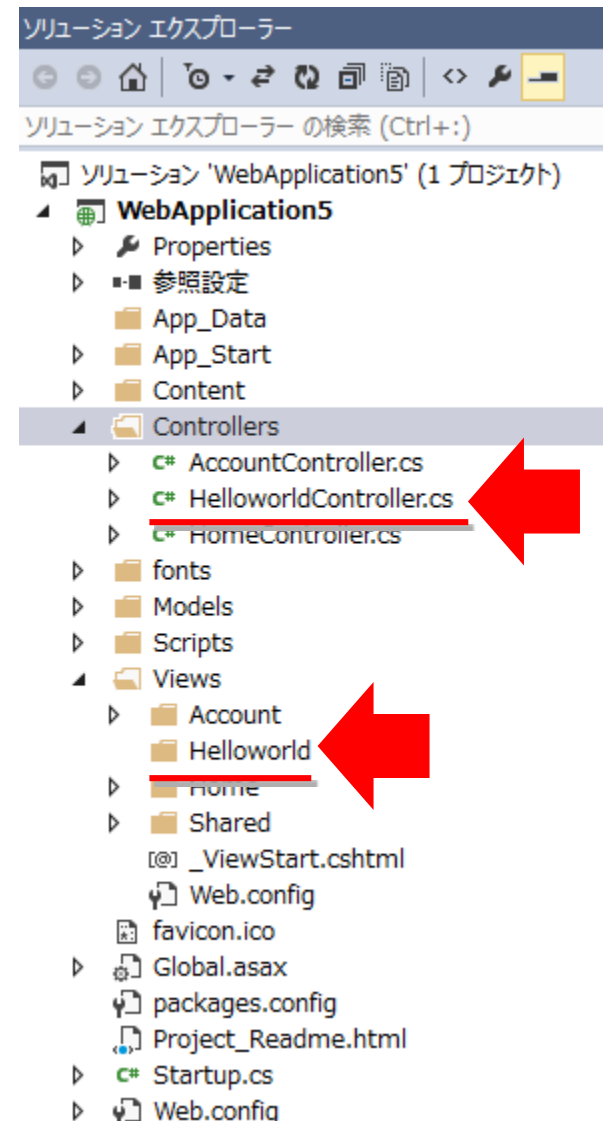


A screenshot of a Windows-style dialog box titled 'コントローラーの追加' (Add Controller). The dialog has a yellow title bar with a close button (X) in the top right corner. Inside the dialog, there is a label 'コントローラー名(C):' followed by a text input field. The text 'HelloworldController' is entered in the field and is highlighted with a blue selection box. A red rectangular box is drawn around the entire text input field. At the bottom right of the dialog, there are two buttons: '追加' (Add) and 'キャンセル' (Cancel).

# コントローラ追加

## 追加されたもの

- Controllers
  - HelloworldController.cs
- Views
  - Helloworldフォルダ



# HelloWorldController.cs編集

```
namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "Hello World...";
        }

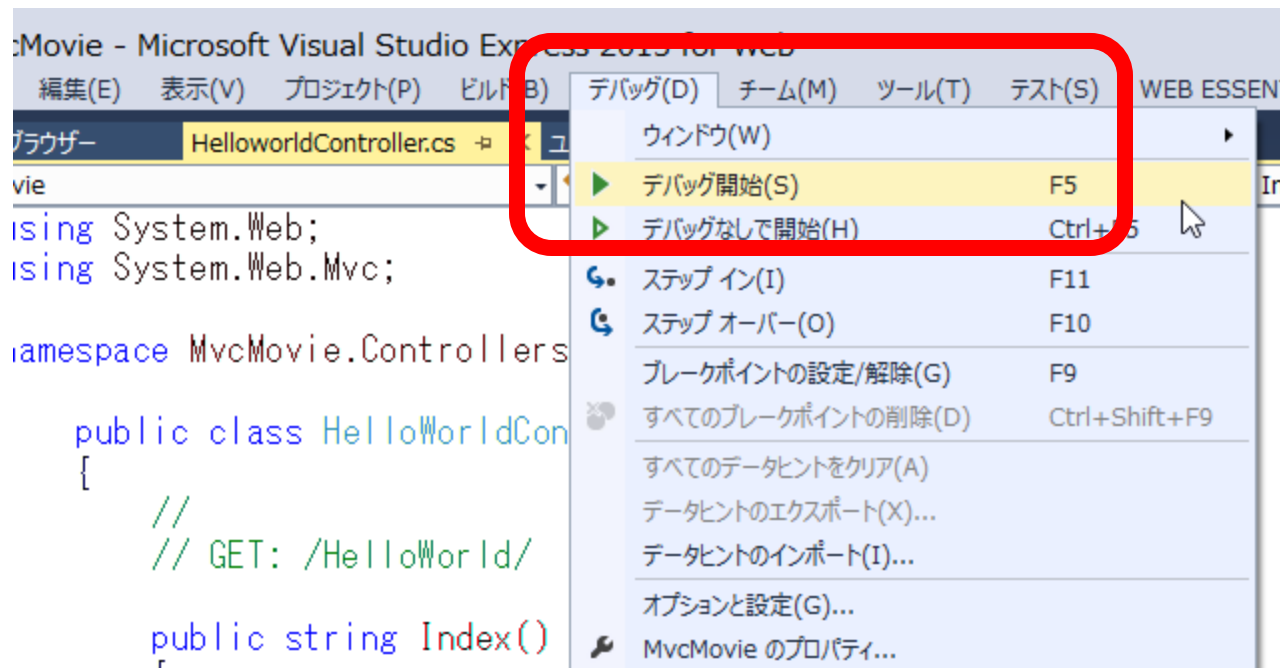
        //
        // GET: /HelloWorld/Welcome/

        public string Welcome()
        {
            return "ここは Weeeeeeelcome";
        }
    }
}
```



# デバッグ実行

- F5キーまたは  
デバッグ⇒  
デバッグ開始

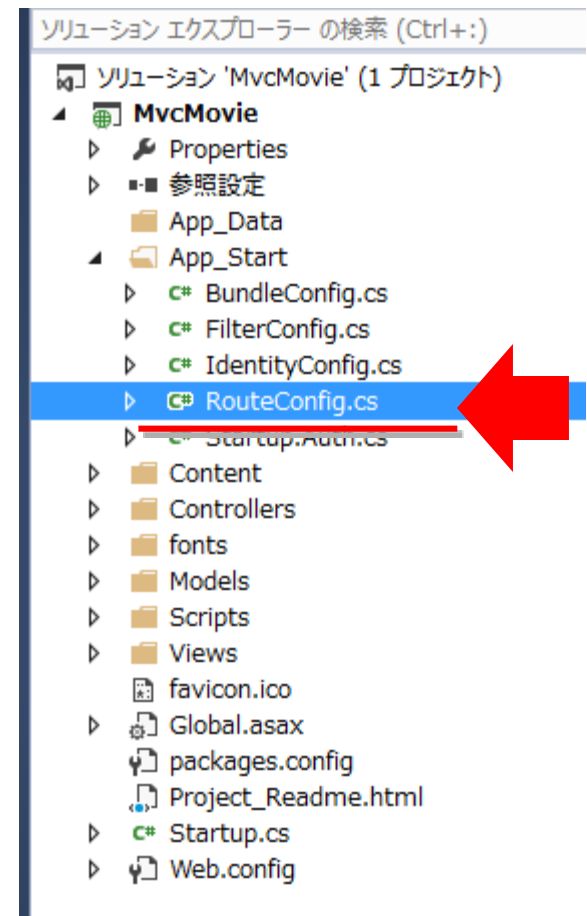


- ブラウザでURL指定
  - http://localhost:55880/Helloworld/
  - http://localhost:55880/Helloworld/Welcome

※ 55880の部分はポート番号。実行タイミングやIISの設定で異なります

# ルーティング

- どのコントローラーを呼び出すか指定
- どのメソッドを呼び出すか指定
- 引数の処理
- App\_Start / RouteConfig.cs



# ルーティング

## デフォルトのコントローラとアクションを変更

```
controller = "Helloworld",  
action = "Welcome"
```


```
defaults: new { controller = "Helloworld",  
action = "Welcome", id = UrlParameter.Optional }
```

# パラメータ

- HelloWorldController.cs の Welcomeアクションを編集

デフォルト値の  
指定もできるよ

```
public string Welcome(string name, int numTimes = 1)
{
    return HttpUtility.HtmlEncode("ここは Weeeeeelcome: へろー "
        + name + "さん. numTimes = " + numTimes);
}
```



ここは Weeeeeelcome: へろー Jacobさん. numTimes = 3

**http://サーバ名/?name=文字列&numTimes=整数**

**HttpUtility.HtmlEncode**  
特殊文字等を使った攻撃を無効化

# パラメータに変な値が入ったらどうなる？

- 試しにやってみよう

**http://サーバ名/?numTimes=2.3**

**http://サーバ名/?numTimes=5%**

**http://サーバ名/?numTimes="8"**

- コントローラ・アクションの大文字小文字は？

**/helloWORLD/welCOMe/?name=aaa**

# アクション追加

## HelloWorldController.csにLoginアクション追加

```
public string Login(string name, int ID = 1)
{
    return HttpUtility.HtmlEncode("Hello" + name + " ID: " + ID);
}
```

## アクセスしてみる


/helloworld/login/?id=3&name=Taro

/helloworld/login/4?name=Jiro

# なぜ「id=」を省略できたか

- App\_Start / RouteConfig.cs

```
routes.MapRoute(  
    name: "Default",  
    url: "{controller}/{action}/{id}",  
    // defaults: new { controller = "Home", action = "Index", id = "" }  
);
```



## ここまででやったこと

- コントローラでHTMLをreturn (View Controller)
- まあ、難しくない。
- でも、HTMLが複雑になると大変になるよね？
- そんなあなたに！ View です



# View作成

- 作業内容 : HelloWorldControllerにViewテンプレートを適用
- Razor view engine
  - 拡張子 .cshtml
  - C#でHTMLを生成
  - テンプレート作成時のキータイピングを最小化
  - 効率よくコーディング

# ControllerがViewオブジェクトをreturnするよう に変更

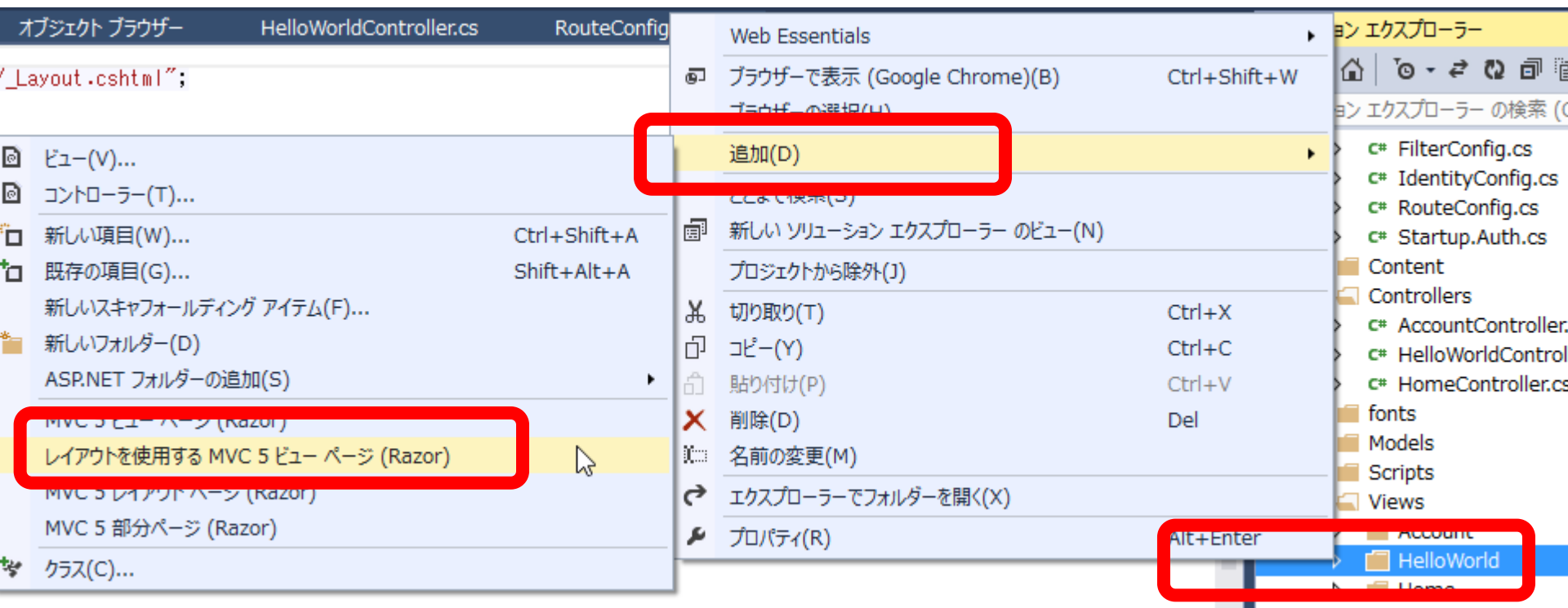
```
public ActionResult Index()  
{  
    return View();  
}
```

テンプレートを使うように、Viewに指令する。

コントローラのメソッド（＝アクションメソッド）は  
多くの場合、ActionResult（またはそれを継承したクラス）を  
returnする。

## View追加

- Views / HelloWorld 右クリック⇒追加  
⇒レイアウトを使用するMVC 5ビューページ (Razor)

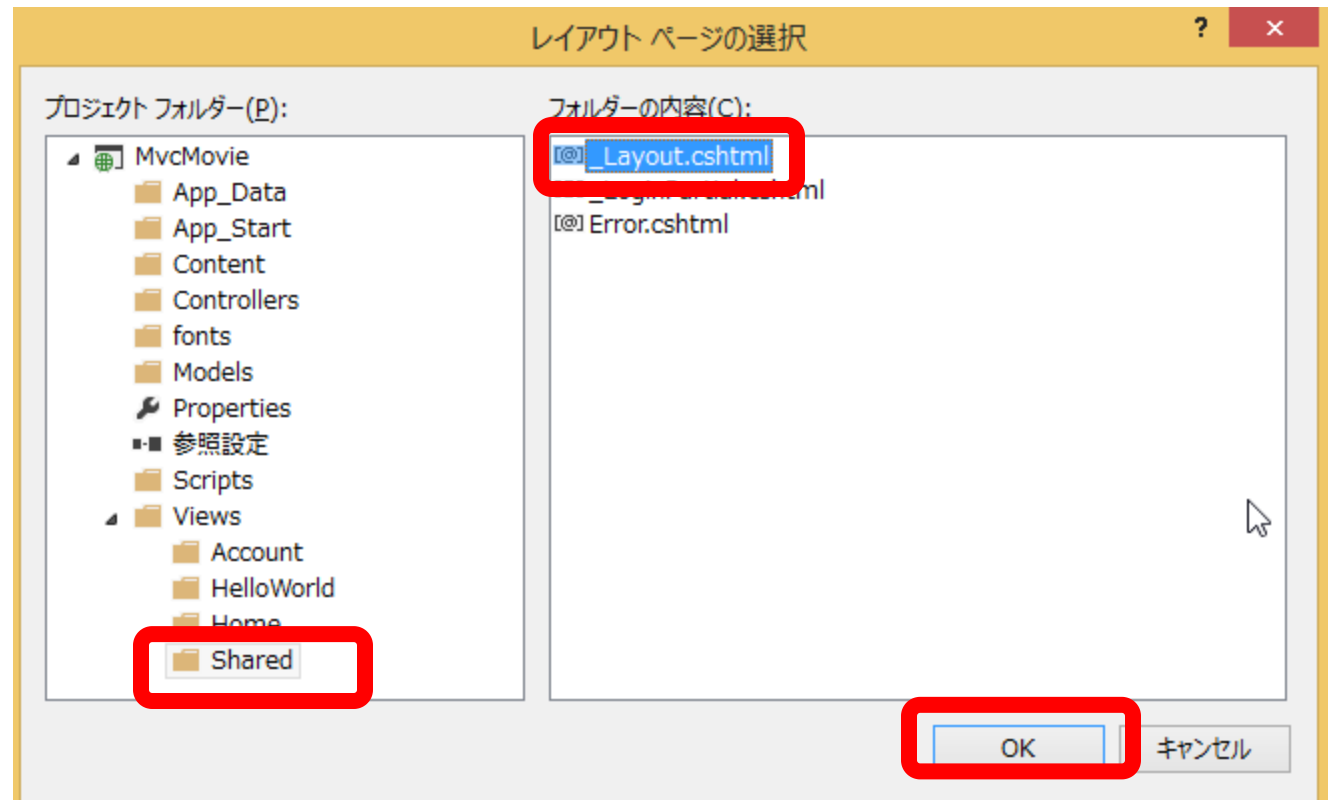


# 項目名とレイアウトページ

- 項目名 : Index

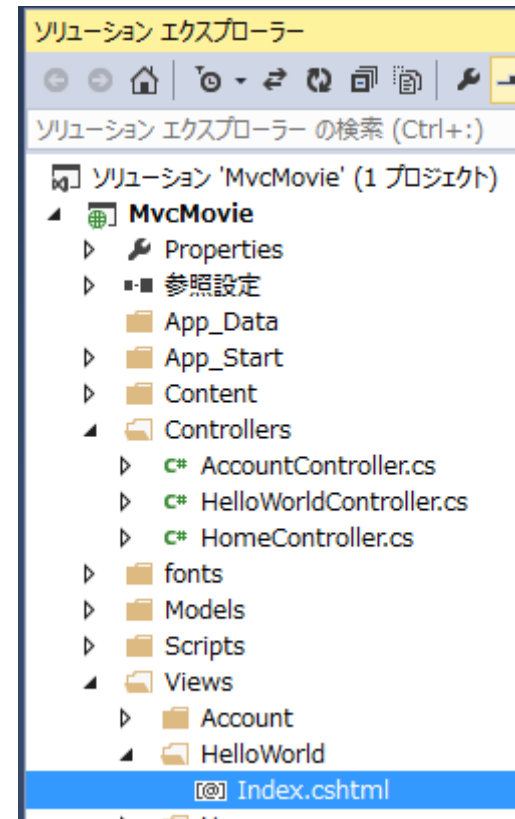


- レイアウトページの選択 : \_Layout.cshtml
- Shared選択
- OK



# 確認

- Views / HelloWorld / Index.cshtml  
が追加されている



# コード追加

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

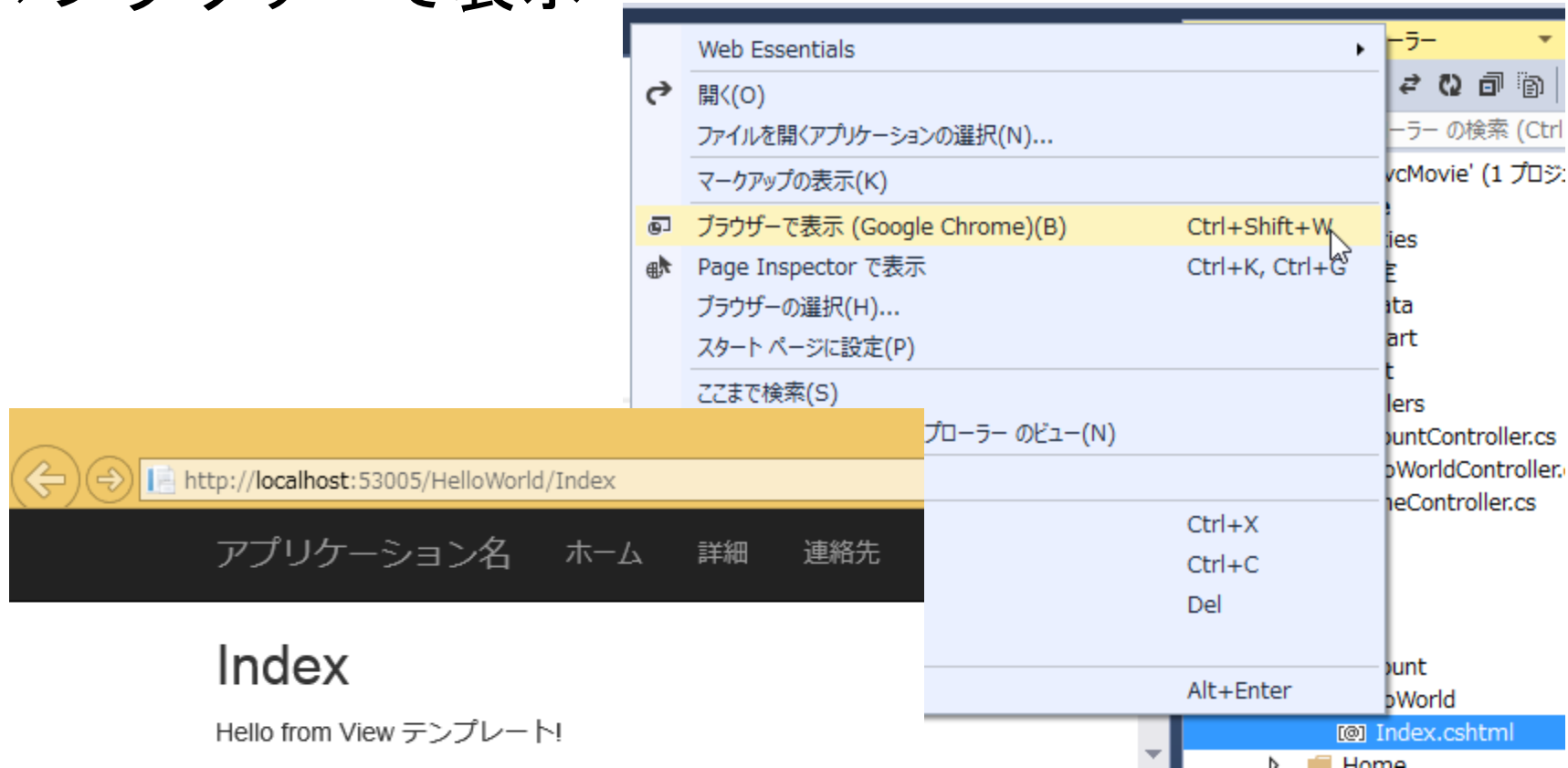
```
@{  
    ViewBag.Title = "Index";  
}
```

```
<h2>Index</h2>
```

```
<p>Hello from View テンプレート!</p>
```

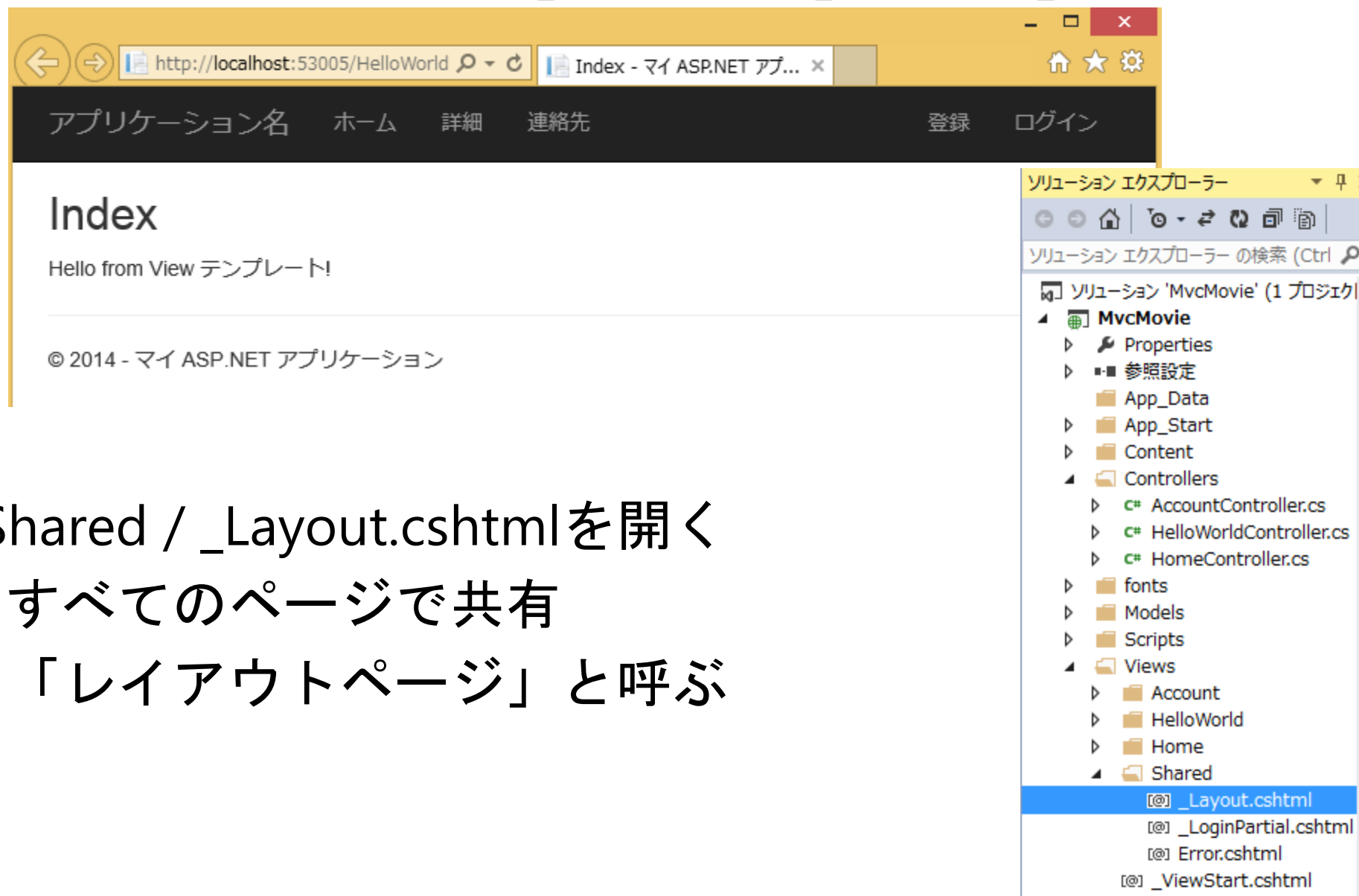
# 確認

- Index.cshtml右クリック  
⇒ ブラウザーで表示



# タイトル、トップのリンクを変更

「アプリケーション名」「ホーム」「詳細」のところ



The screenshot shows a web browser window at `http://localhost:53005/HelloWorld` displaying the 'Index' page. The page has a navigation bar with links: 'アプリケーション名', 'ホーム', '詳細', '連絡先', '登録', and 'ログイン'. The main content area says 'Hello from View テンプレート!' and '© 2014 - マイ ASP.NET アプリケーション'.

On the right, the Visual Studio Solution Explorer shows the project structure for 'MvcMovie'. The 'Views' folder is expanded, showing subfolders 'Account', 'HelloWorld', 'Home', and 'Shared'. The file '\_Layout.cshtml' is selected under the 'Shared' folder.

ソリューション エクスプローラー

ソリューション エクスプローラー の検索 (Ctrl F)

ソリューション 'MvcMovie' (1 プロジェクト)

- MvcMovie
  - Properties
  - 参照設定
  - App\_Data
  - App\_Start
  - Content
  - Controllers
    - AccountController.cs
    - HelloWorldController.cs
    - HomeController.cs
  - fonts
  - Models
  - Scripts
  - Views
    - Account
    - HelloWorld
    - Home
    - Shared
      - \_Layout.cshtml
      - \_LoginPartial.cshtml
      - Error.cshtml
      - \_ViewStart.cshtml

Shared / \_Layout.cshtmlを開く

- すべてのページで共有
- 「レイアウトページ」と呼ぶ

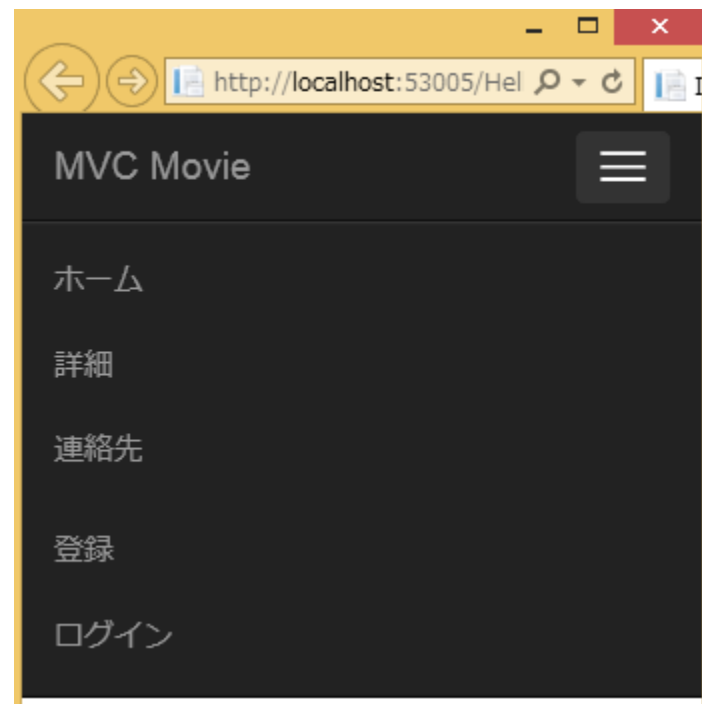


# レイアウトページ編集

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>@ViewBag.Title - Movie App</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>
<body>
  <div class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse">
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        @Html.ActionLink("MVC Movie", "Index", "Movies", new { area = "" }
      </div>
      <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
          <li>@Html.ActionLink("ホーム", "Index", "Home")</li>
          <li>@Html.ActionLink("詳細", "About", "Home")</li>
          <li>@Html.ActionLink("連絡先", "Contact", "Home")</li>
        </ul>
        @Html.Partial("_LoginPartial")
      </div>
    </div>
  </div>
```

# レイアウトページ確認

- Views/HelloWorld/Index.cshtml  
に次の記述があるのでレイアウトページが呼ばれる  
@{  
    Layout = "~/Views/Shared/\_Layout.cshtml";  
}



# Index.cshtmlを編集

```
@{
```

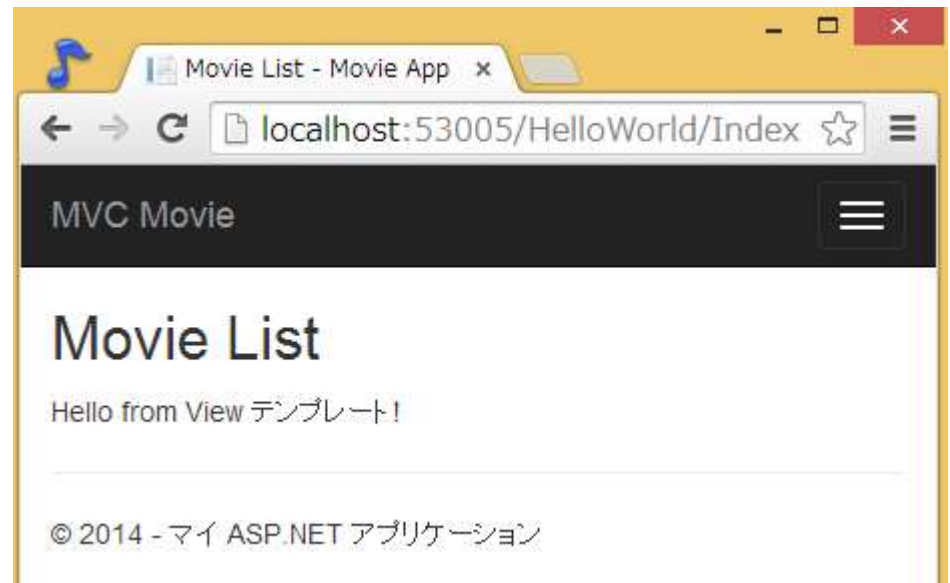
ページタイトル

```
    ViewBag.Title = "My Movie List";
```

```
}
```

```
<h2>Movie List</h2>
```

```
<p>Hello from View テンプレート!</p>
```

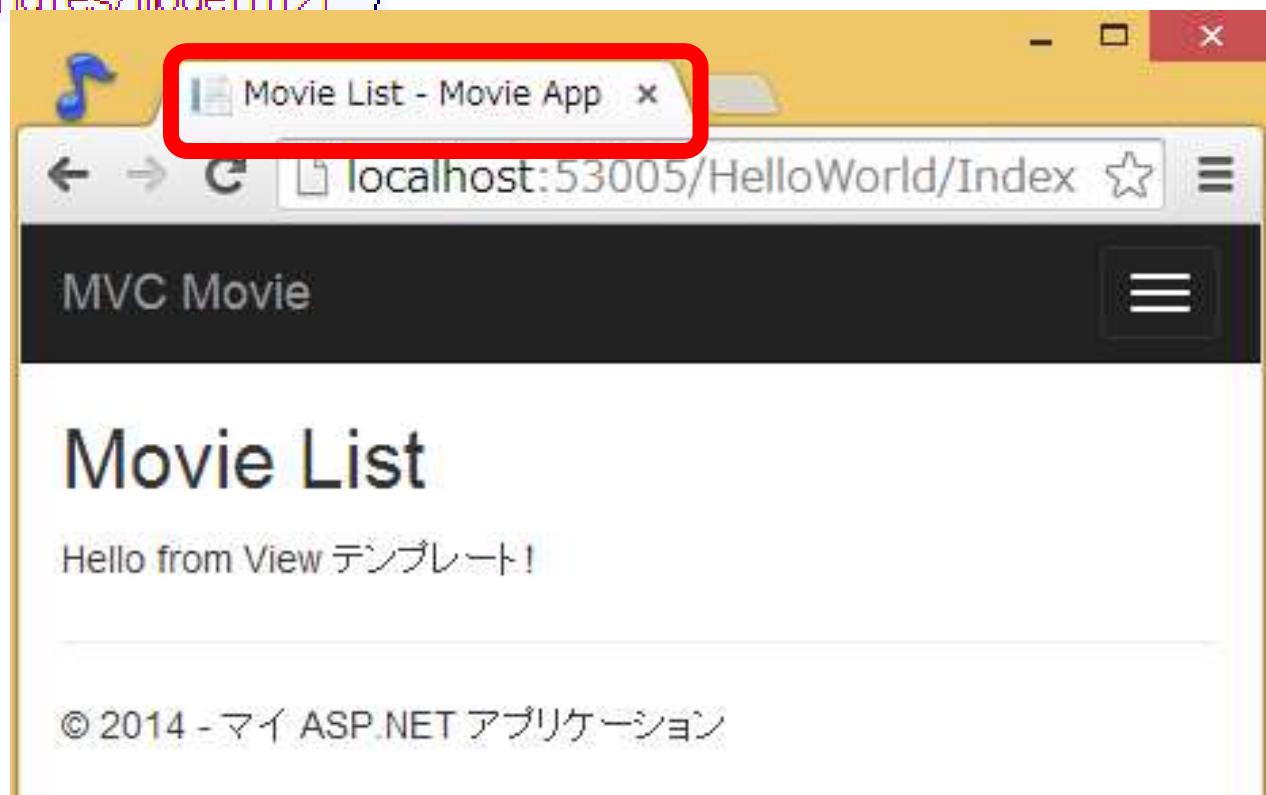


# <head> <title>

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>@ViewBag.Title - Movie App</title>
@Styles.Render("~/Content/css")
@Scripts.Render("~/bundles/modernizr")
```

レイアウトページの  
@ViewBag.Titleで読み  
込まれる

ViewBagにはパラメー  
タを自由に含めること  
ができる



## この段階でできたこと

- 固定のデータ「Hello from View テンプレート!」をViewに設定し、表示させる
- 次は：
  - データを可変にしたい。  
データベースに保存されたデータを表示させたい。
  - ⇒Modelをつくります

# データの流れ

- Model ⇒ Controller ⇒ View
- Viewでは処理を実行したり、データを作成しない。
- 処理ロジックはControllerへ
- データはModelからもらったものをそのまま使う

# HelloWorldController.cs の Welcomeアクション変更

```
public ActionResult Welcome(string name, int numTimes = 1)
{
    //return HttpUtility.HtmlEncode("ここは Weeeeeee!come: ^
    ViewBag.Message = "/\□ー" + name;
    ViewBag.NumTimes = numTimes;
    return View();
}
```

Viewに対してViewBagオブジェクトでデータを渡す。

このデータはURIで受け取った、ユーザからのパラメータとします。

ViewBagはダイナミックオブジェクトで、なんでも入れることが可能。

## この時点でのHelloWorldController.cs

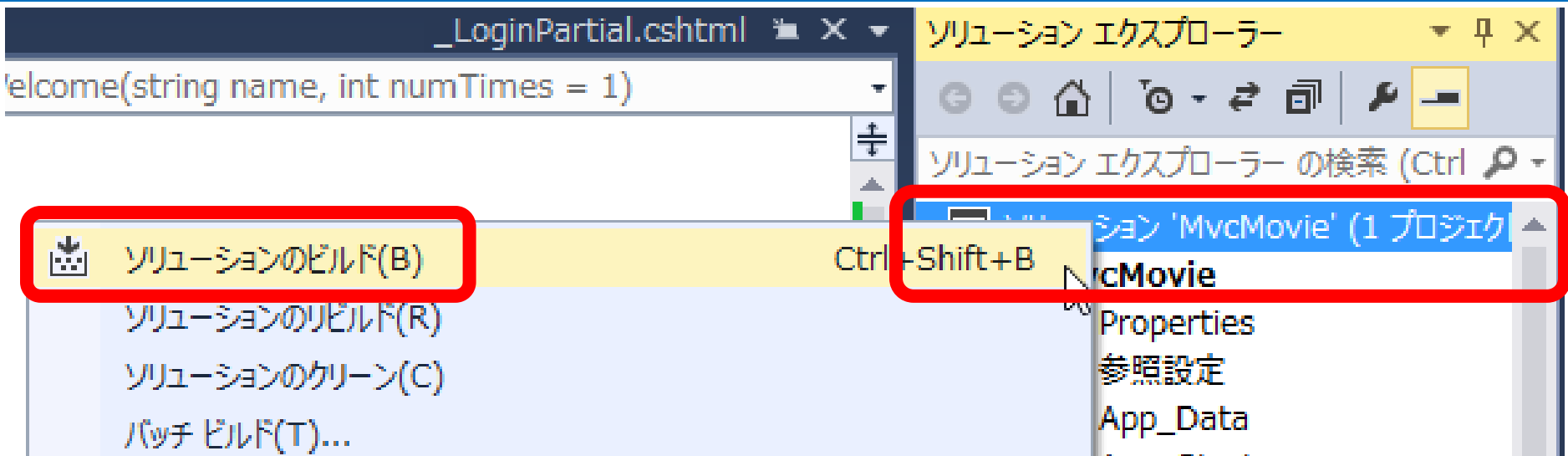
```
using System.Web;
using System.Web.Mvc;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Welcome(string name, int numTimes = 1)
        {
            ViewBag.Message = "ハロ〜" + name;
            ViewBag.NumTimes = numTimes;
            return View();
        }
    }
}
```

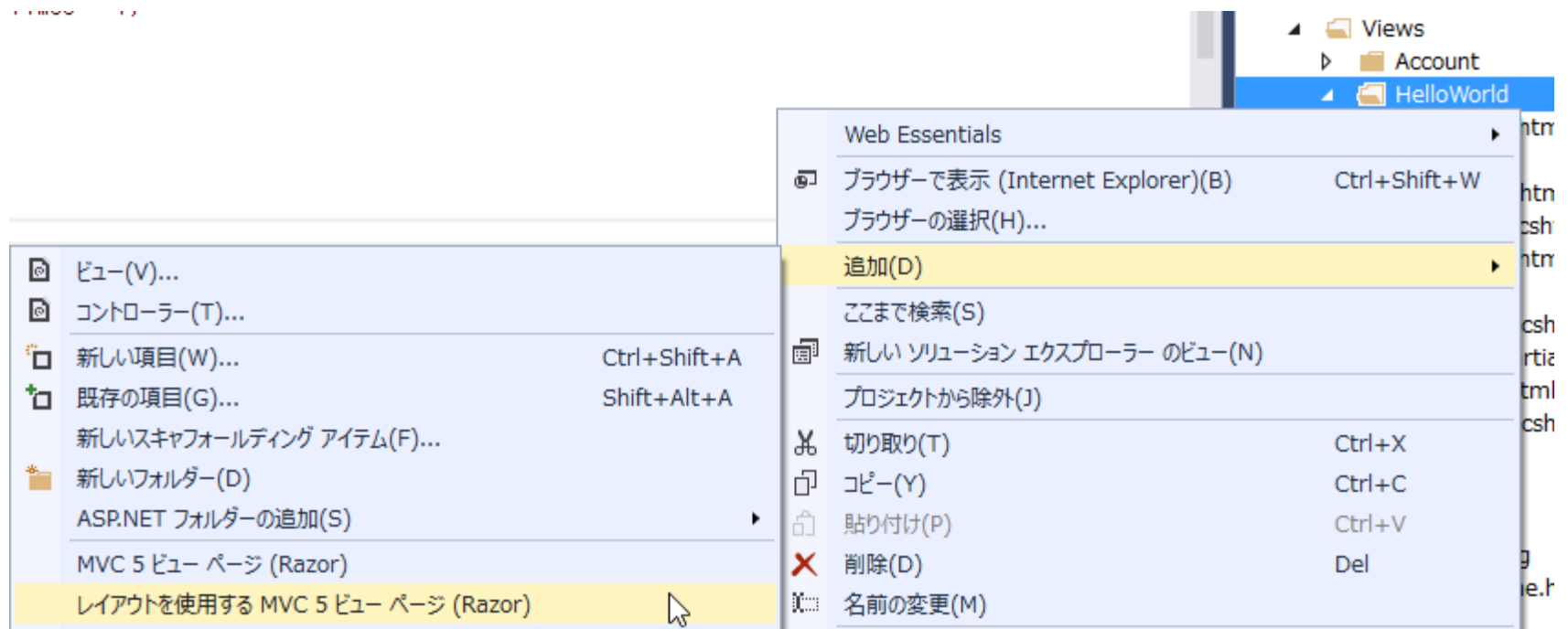


# ソリューションをビルドしておく



- ソリューションを右クリック  
⇒ソリューションのビルド

# View追加

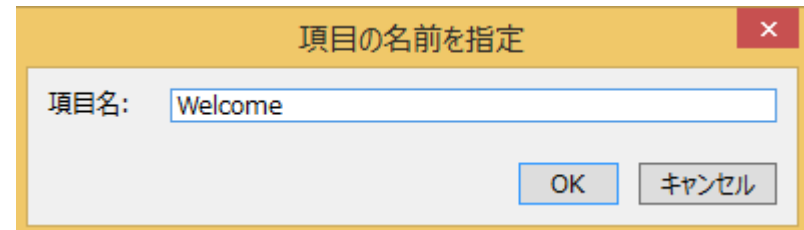


Views/HelloWorldを右クリック

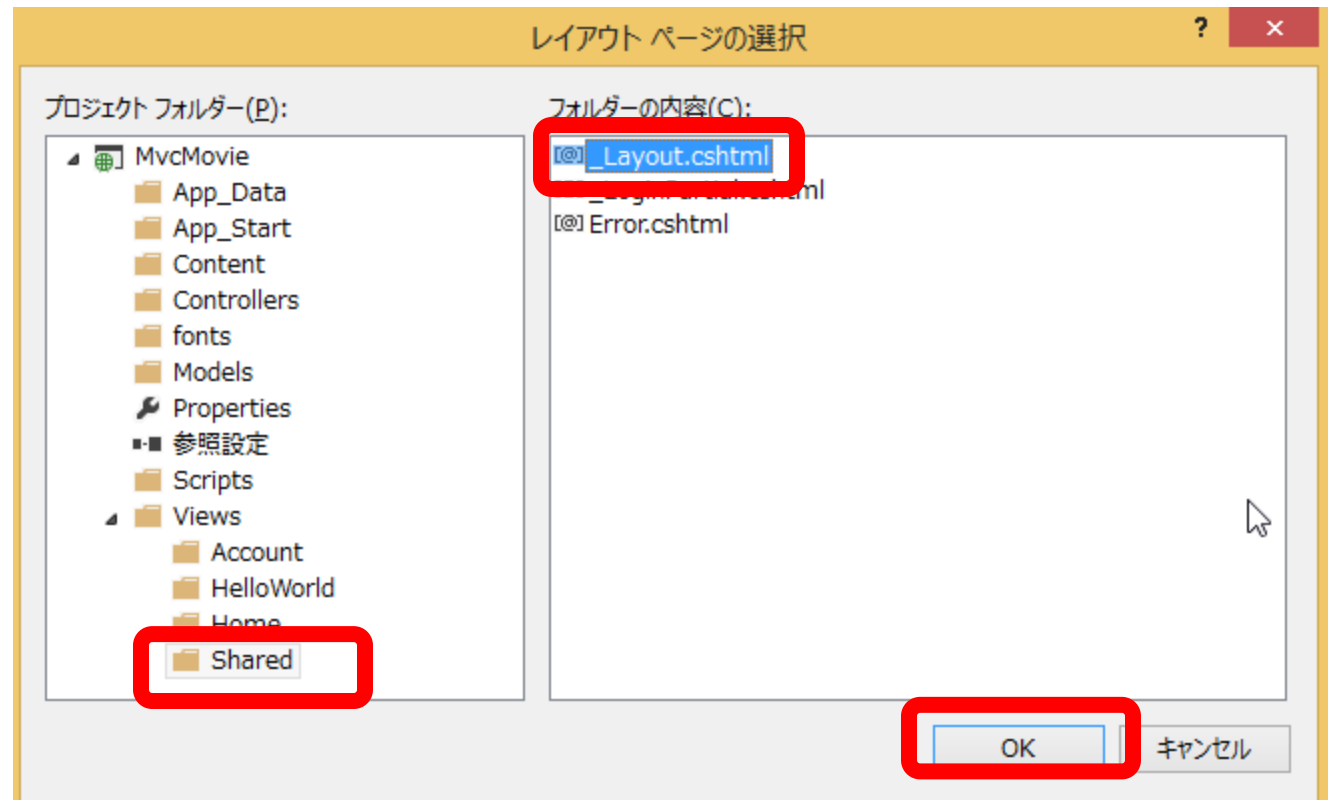
⇒追加⇒レイアウトを使用するMVC5ビューページ（Razor）

# View追加

- 項目名 : Welcome



- レイアウトページの選択 : \_Layout.cshtml
- Shared選択
- OK



# Welcome.cshtml

- ViewBagで受け取ったMessageをNumTimes個表示させる

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

```
<h2>Welcome</h2>  
<ul>  
    @for (int i = 0; i < ViewBag.NumTimes; i++)  
    {  
        <li>@ViewBag.Message</li>  
    }  
</ul>
```

# 動作確認

http://localhost:x/HelloWorld/Welcome?name=Taro&numtimes=4

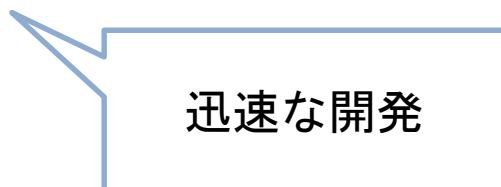
モデルバインダにより、  
URIから取得したデータを  
Controllerが受け取り、  
ViewBag経由でViewに渡す。



ViewBag の代わりに「View Model」を使う方法もある。後述。

# Model

- データベースとの接続部分
- Entity Framework (EF) による Code First 手法
- シンプルなクラスによる Model 定義  
(POCO クラス "plain-old CLR objects")
- クラスを定義 = データベースのテーブル作成



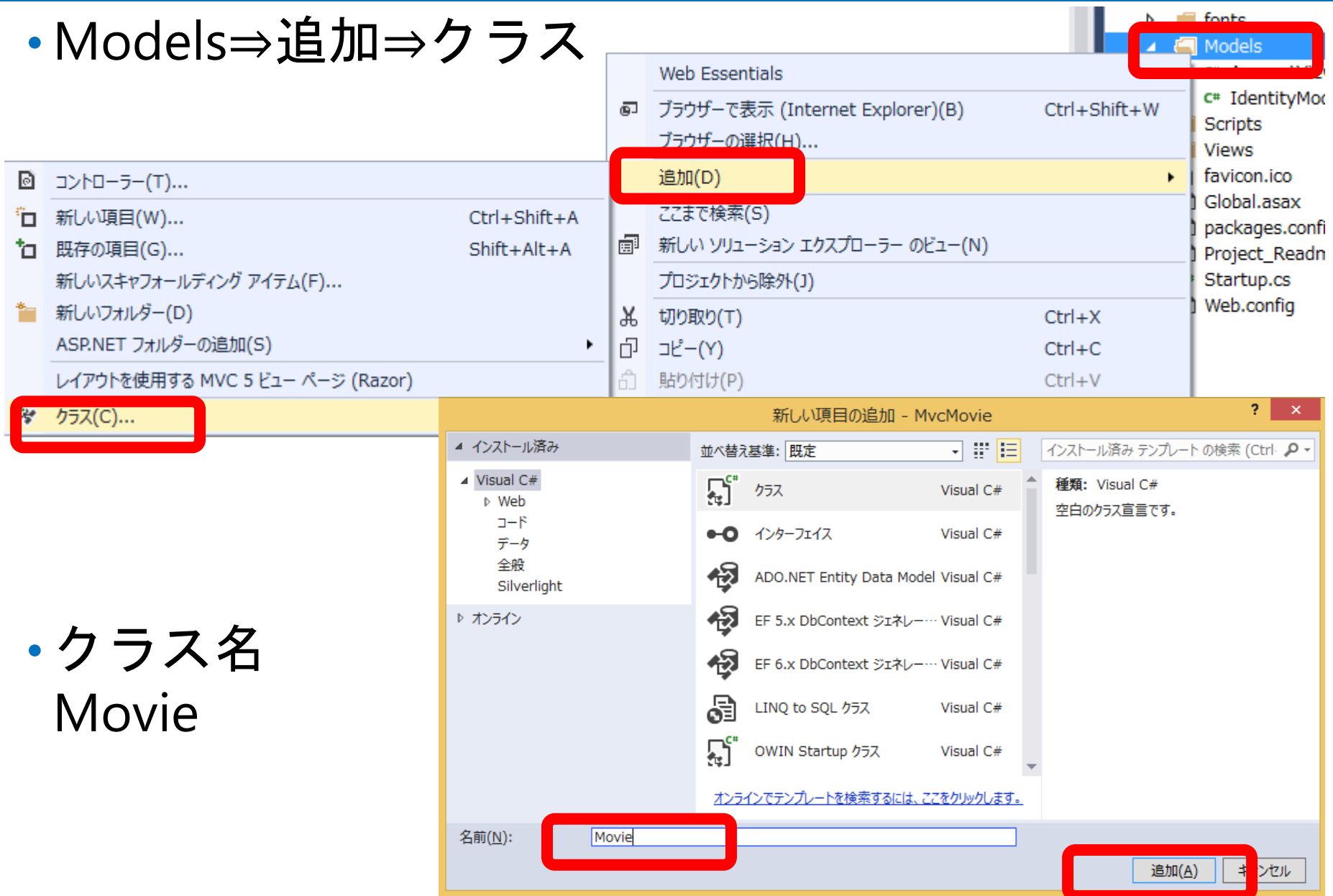
迅速な開発

- Database First 手法もあります

<http://www.asp.net/visual-studio/overview/2013/aspnet-scaffolding-overview>  
(by Tom Fizmakens)

# Model追加してみよう

- Models⇒追加⇒クラス



- クラス名  
Movie

# ModelにクラスMovie追加

```
namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

- 映画のタイトル等を管理するデータベース
- Movieオブジェクトのインスタンス  
= データベースのレコード
- Movieオブジェクトのプロパティ  
= データベースのカラム



# Modelにクラス MovieDbContext 追加

```
- using System;  
- using System.Data.Entity;
```

Entity Framework

```
- namespace MvcMovie.Models
```

```
-     public class Movie
```

```
    {  
        public int ID { get; set; }  
        public string Title { get; set; }  
        public DateTime ReleaseDate { get; set; }  
        public string Genre { get; set; }  
        public decimal Price { get; set; }  
    }
```

```
-     public class MovieDbContext : DbContext
```

```
    {  
        public DbSet<Movie> Movies { get; set; }  
    }
```

```
    }
```

# MovieDbContextクラスの役割

- Entity Frameworkの基底クラスDbContextを継承
- データベースに接続して：
  - 検索
  - 保存
  - 更新
  - 削除
- DbContextとDbSetの参照のためにはファイル上部で「using System.Data.Entity;」が必要。

# Note: 未使用のusingの削除



- ファイル右クリック
  - ⇒usingの整理
  - ⇒未使用のusingの削除

## ここまでのおさらい

- ついにModelも追加
- MVCのすべてが登場
  - Model
  - View
  - Controller
- 次は、データベースと接続していきます

# SQL Server LocalDBとの接続

- MovieDbContextクラスは、Movieオブジェクトをデータベースのレコードにマッピングする
  - では、どのデータベースに接続するか？
  - まだデータベースの設定はしていないはず...?
- 
- Entity Frameworkのデフォルト設定ではLocalDBを使います
  - 設定はアプリケーションのWeb.configファイルで

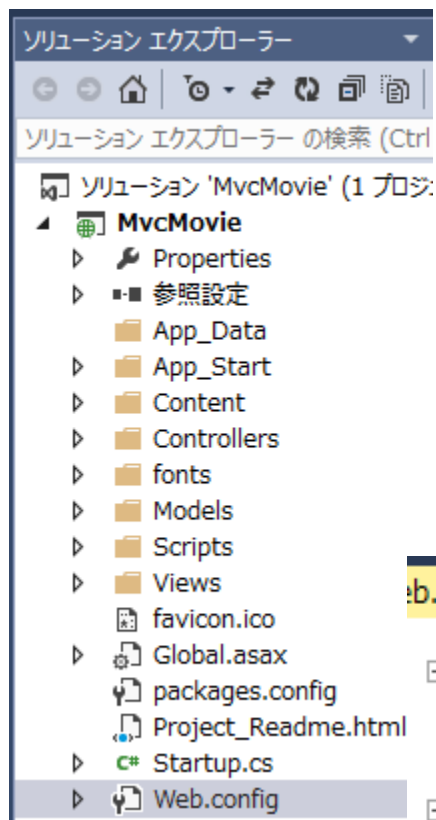
# SQL Server Express LocalDB

- LocalDB

- SQL Server Express の軽量版
- .mdfファイルをデータベースとして取り扱う
- 通常、プロジェクトのApp\_Dataフォルダに保存
- リリース時には使わないほうが良い  
⇒Webサーバと一緒に使うことを想定していないから
- ただしLocalDBからの変換は簡単
  - SQL Server
  - SQL Azure
- Visual Studio 2013と一緒にLocalDBがインストールされます（2012でも）
- Entity Frameworkは最初に、オブジェクトのコンテキストクラスと同じ名前の接続文字列を探します。  
(今の場合は MovieDBContext)

# 接続文字列の確認

- Web.configを開く
- <connectionStrings>を探す



```
Web.config AccountViewModels.cs Movie.cs IdentityModels.cs Welcome.cs
<?xml version="1.0" encoding="utf-8"?>
<!--
ASP.NET アプリケーションの構成方法の詳細については、
http://go.microsoft.com/fwlink/?LinkId=301880 を参照してください
-->
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkId=301880 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkConfiguration" />
  </configSections>
  <connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=MvcMovie;Integrated Security=True" providerName="System.Data.SqlClient" />
  </connectionStrings>
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
  </appSettings>
</configuration>
```

# MovieDBContextの接続文字列を追加

```
<add name="MovieDBContext"
      connectionString="Data Source=(LocalDB)\v11.0;
      AttachDbFilename=|DataDirectory|\Movies.mdf;
      Integrated Security=True"
      providerName="System.Data.SqlClient"/>
```

新規プロジェクトに入っている「<add name="DefaultConnection"」の部分とほとんど同じです。コピペしてから編集すると便利

DefaultConnectionはユーザのアクセス権を担当

会員ページの作成はここではやりません。下記を参照

<http://www.windowsazure.com/en-us/develop/net/tutorials/web-site-with-sql-database/>



# Modelと接続文字列

- 接続文字列のNameとDbContextクラスの名前は一致させる
- .mdfファイルの名前は自由に

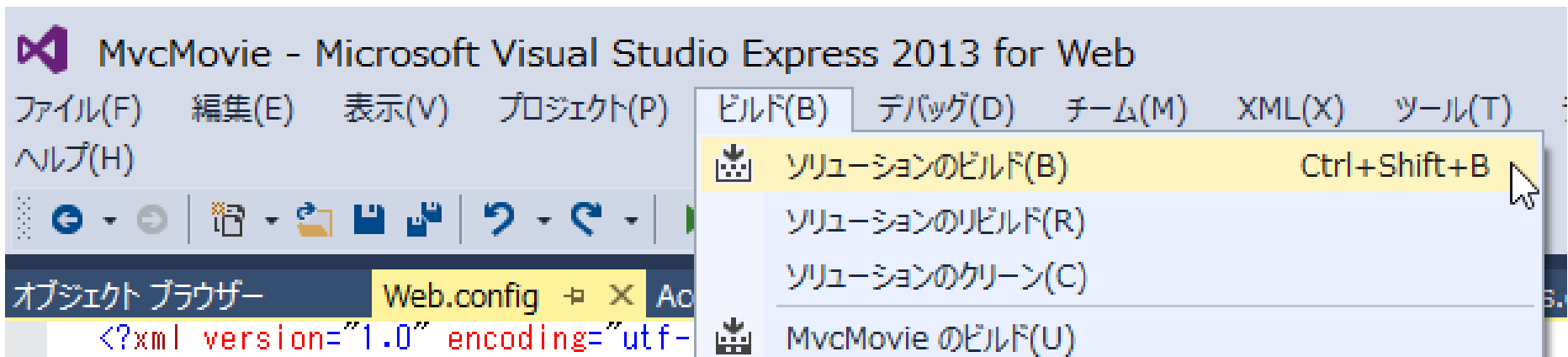
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }

    public class MovieDbContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

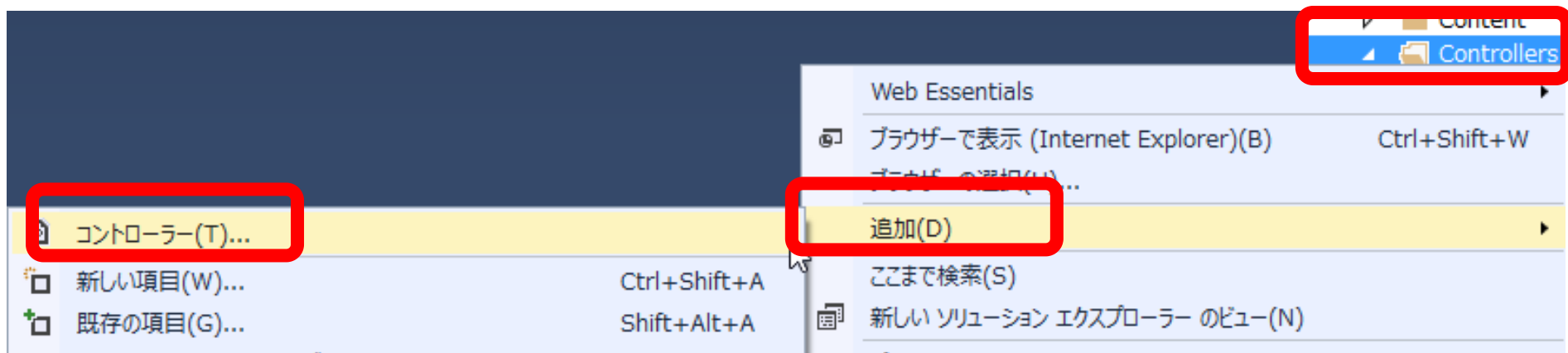
# ControllerからModelのデータにアクセス

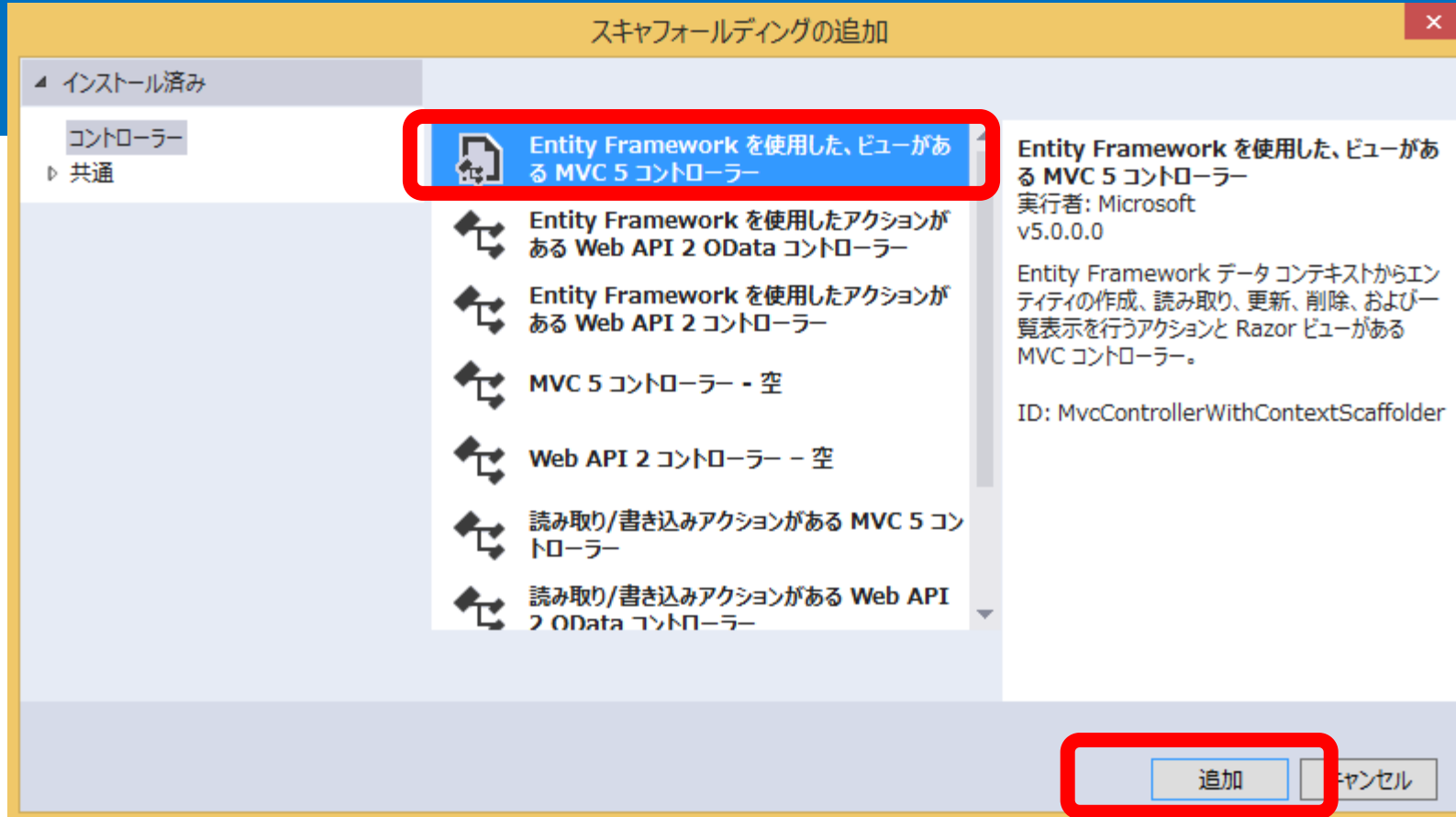
- MoviesControllerクラスを新規作成
  - 映画のデータを取り出す
  - 映画の一覧をViewテンプレートでブラウザに表示させる
- 作業の前にビルドしておいてください  
ビルドしないとエラーが出るかも



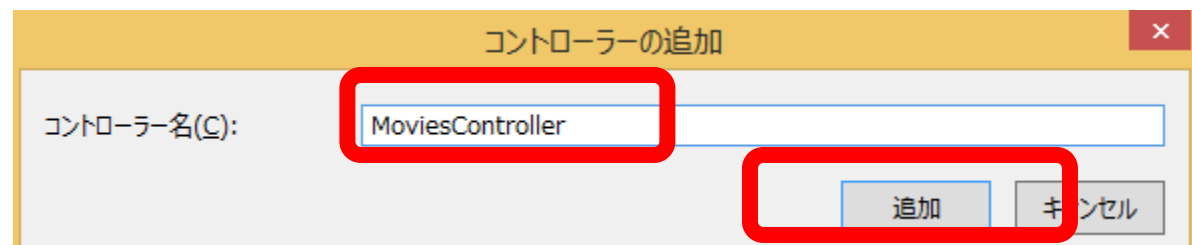
# Controller追加

- Controller右クリック⇒追加⇒コントローラー





- 「Entity Frameworkを使用した、ビューがある MVC5コントローラー」⇒追加
- MoviesController ⇒追加



# Controller追加

## プルダウンから選ぶ

- モデルクラス : Movie(MvcMovie.Models)
- データコンテキストクラス :  
MovieDbContext(MvcMovie.Models)

コントローラーの追加

モデル クラス(M): Movie (MvcMovie.Models)

データ コンテキスト クラス(D): MovieDbContext (MvcMovie.Models) +

☐ 非同期コントローラー アクションの使用(A)

ビュー:

☒ ビューの生成(V)

☒ スクリプト ライブラリの参照(R)

☒ レイアウト ページの使用(U):

(Razor \_viewstart ファイルで設定する場合は空のままにしてください)

コントローラー名(C): MoviesController

追加 キャンセル

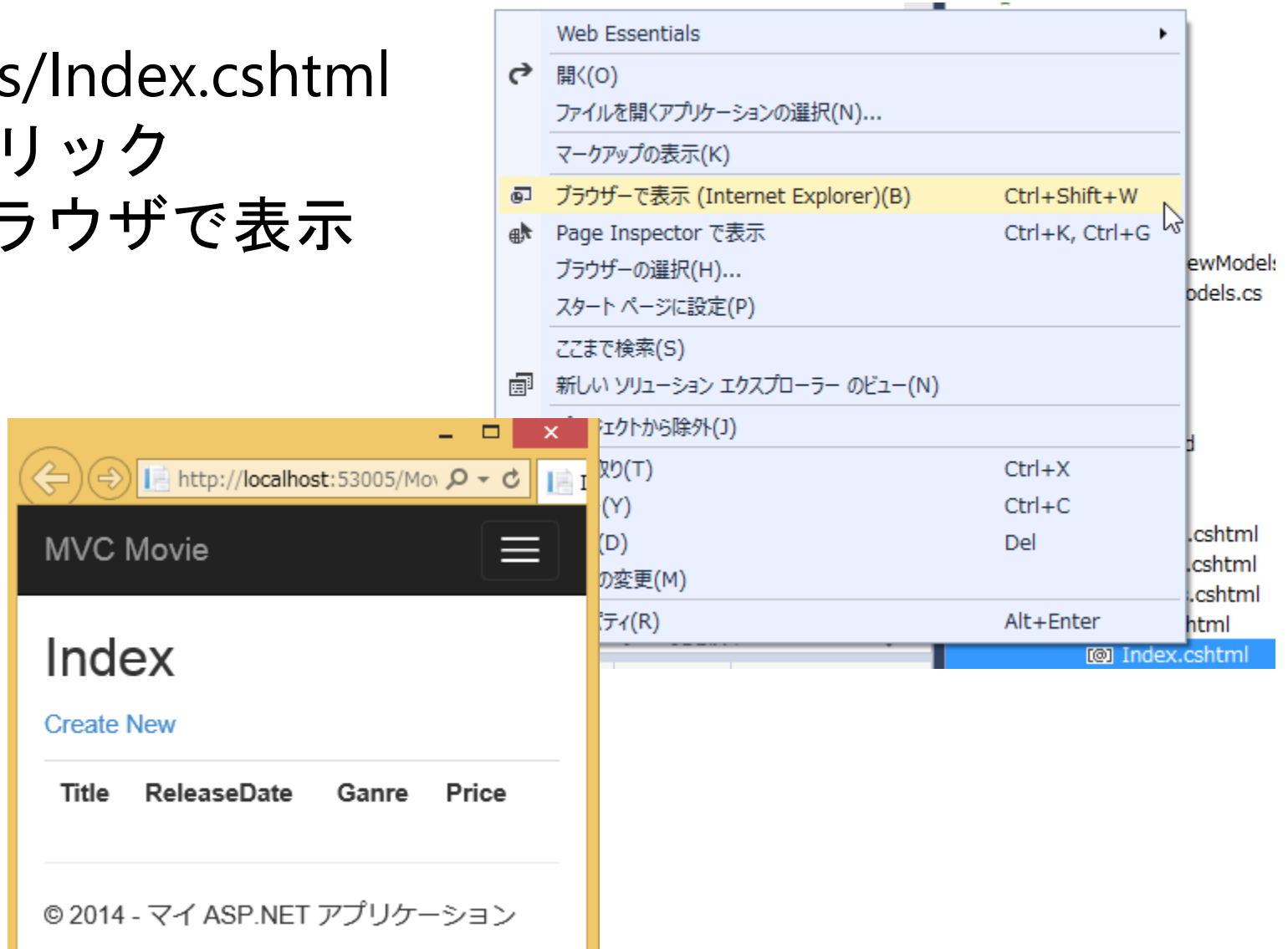
※追加ボタンでエラーになる場合は、ビルドしてください

# この操作でできたファイル

- Controllers/MoviesController.cs
- Views/Movies フォルダ
  - Create.cshtml
  - Delete.cshtml
  - Details.cshtml
  - Edit.cshtml
  - Index.cshtml
- 自動的にCRUD(Create, Read, Update, Delete)アクションメソッドと、Viewが作られる
- 映画データの作成、一覧表示、編集、削除ができるようになった！ ありがとう！

# 動作確認

- Views/Index.cshtml  
右クリック  
⇒ ブラウザで表示



# ルーティング設定

- App\_Start/RouteConfig.cs

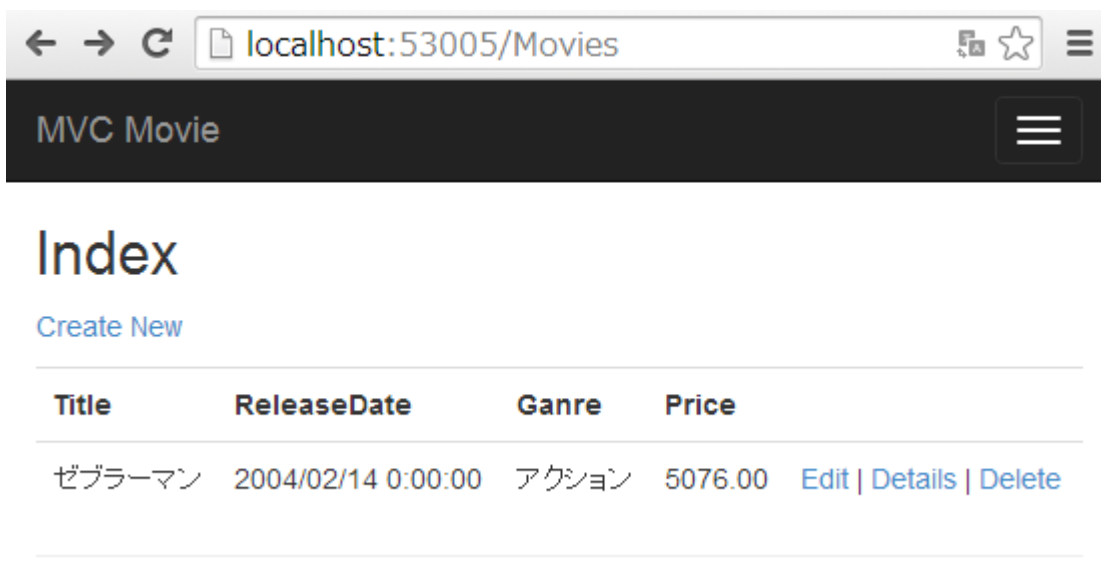
```
routes.MapRoute(  
    name: "Default",  
    url: "{controller}/{action}/{id}",  
    defaults: new { controller = "Movies",  
        action = "Index",  
        id = UrlParameter.Optional }  
);
```

http://localhost:xx/でアクセス可能に



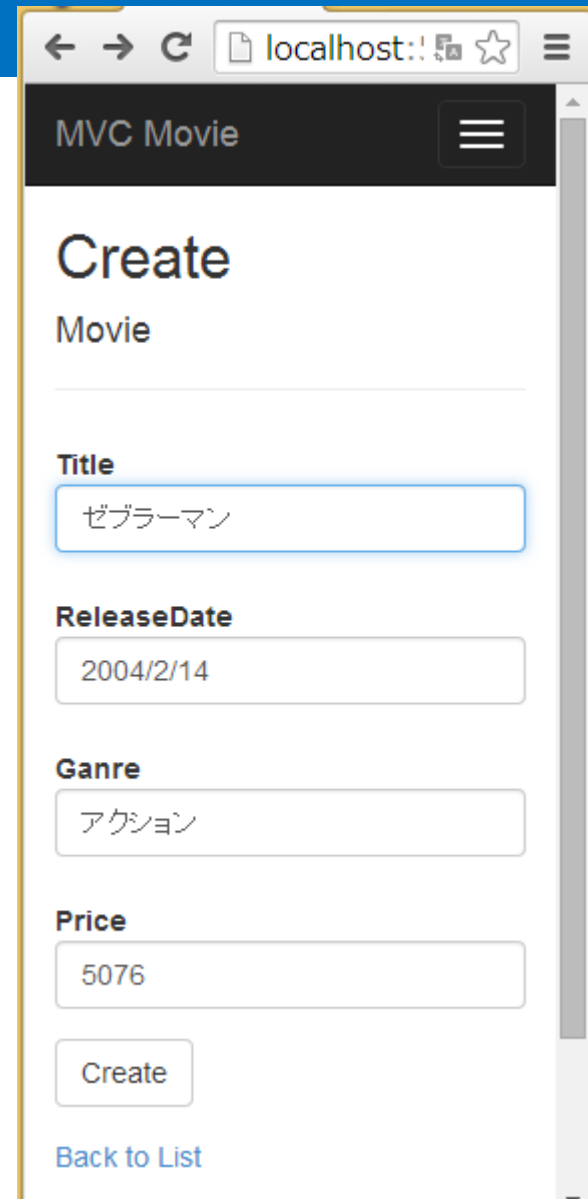
# 新しいデータを作成

- IndexページでCreate Newを押す
- 適当なデータを入力してみる
- わざと変な値を入れてみる  
Release Date: today  
Price: -9,999



A screenshot of a web browser showing the MVC Movie application's Index page. The browser's address bar displays 'localhost:53005/Movies'. The page has a dark header with 'MVC Movie' and a hamburger menu icon. Below the header, the word 'Index' is displayed in a large font, followed by a 'Create New' link. A table lists the existing movie data:

Title	ReleaseDate	Ganre	Price	
ゼブラーマン	2004/02/14 0:00:00	アクション	5076.00	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>



A screenshot of the MVC Movie application's Create page. The browser's address bar shows 'localhost:53005'. The page has a dark header with 'MVC Movie' and a hamburger menu icon. The main content area is titled 'Create Movie'. It contains four input fields: 'Title' with the value 'ゼブラーマン', 'ReleaseDate' with the value '2004/2/14', 'Ganre' with the value 'アクション', and 'Price' with the value '5076'. Below these fields is a 'Create' button and a 'Back to List' link.

- 入力したらEdit, Details, Deleteも動作確認

# MoviesController.csに自動生成されたコードを確認

- 先頭部分

データベースからの  
Movieコンテキスト

```
public class MoviesController : Controller
{
    private MovieDbContext db = new MovieDbContext();

    // GET: Movies
    public ActionResult Index()
    {
        return View(db.Movies.ToList());
    }
}
```


DBのMoviesテーブルはそのまま  
すべてViewに転送

# 厳密な型指定のModelとキーワード@model

- このチュートリアル最初のほうでViewBagを紹介しました。データやオブジェクトを転送する。これは遅延バインディングによるダイナミックオブジェクト。
- これに対し、厳密に型指定してViewテンプレートにオブジェクトを渡すこともできます。  
⇒ コンパイル時のチェックと、Visual StudioのIntelliSenseによるコードチェックを便利に利用できる
- 今回のスキャフォールディング機能は厳密な型指定によるMoviesControllerクラスとViewテンプレートを作成
- 次に、自動生成のDetailsメソッドを確認してみよう

# 自動生成のDetailsメソッド

```
// GET: Movies/Details/5
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}
```



Movieが見つかったらViewに渡す

- パラメータ「id」は基本としてルートデータから獲得
- 例) `http://localhost:xx/movies/details/1`
  - コントローラ : Movies
  - アクション : Details
  - ID: 1
- `http://localhost:xx/movies/details?1`でも同じ

# Views/Movies/Details.cshtml

```
@model MvcMovie.Models.Movie
```

@model によって、この View が参照すべき型を指定

```
@{  
    ViewBag.Title = "Details";  
}
```

ViewBag も一緒に使える

```
<h2>Details</h2>
```

```
<div>  
    <h4>Movie</h4>  
    <hr />  
    <dl class="dl-horizontal">  
        <dt>  
            @Html.DisplayNameFor(model => model.Title)  
        </dt>
```

Model のタイトルのカラム名

```
        @*(略)*@  
    </dl>  
</div>
```

後述のため略

```
<p>  
    @Html.ActionLink("Edit", "Edit", new { id = Model.ID }) |  
    @Html.ActionLink("Back to List", "Index")  
</p>
```

# Views/Movie/Index.cshtml

型指定のおかげで  
foreachできる

item: 厳密な型指定  
IEnumerable<Movie>  
オブジェクト

```
@foreach (var item in Model) {  
    <tr>  
        <td>  
            @Html.DisplayFor(modelItem => item.Title)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.ReleaseDate)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.  
        </td>  
        <td>  
            @Html.ActionLink("Edit", "Edit", ne  
            @Html.ActionLink("Details", "Detail  
            @Html.ActionLink("Delete", "Delete"  
        </td>  
    </tr>  
}  
</table>
```

型指定のおかげで  
IntelliSenseが効く

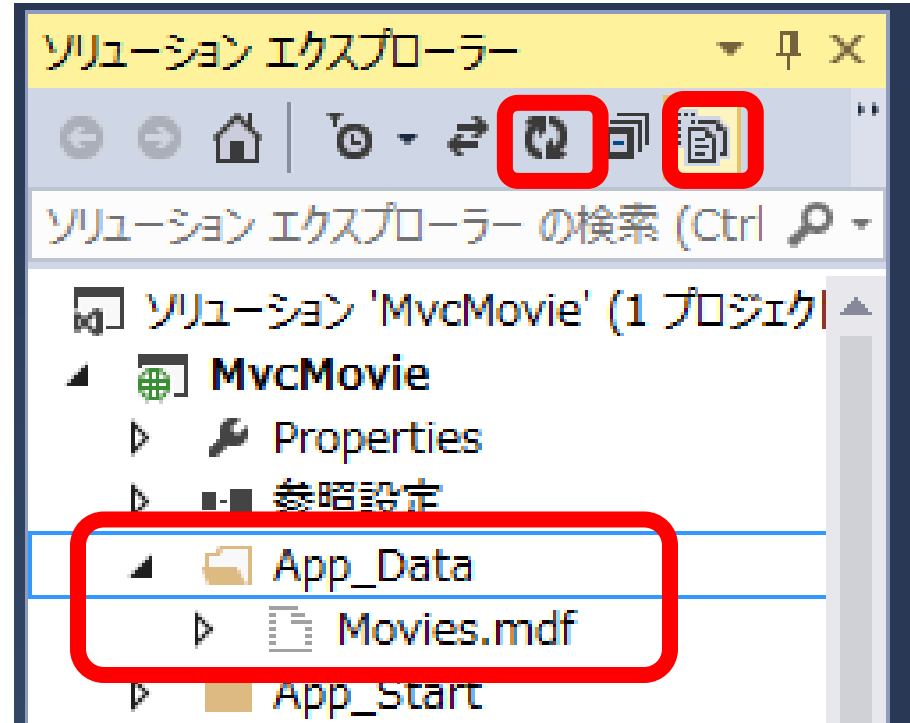
- Equals
- Genre
- GetHashCode
- GetType
- ID
- Price
- ReleaseDate
- Title
- ToString

bool ob  
指定した

D }) |  
})

# SQL Server LocalDB

- Entity FrameworkのCode Firstは、指定されたMoviesのデータベースがなければ自動的に生成
- App\_DataフォルダにMovies.mdfが保存される
- ソリューションエクスプローラで見えないとき：  
「すべてのファイルを表示」をオン  
「最新の情報に更新」を押す

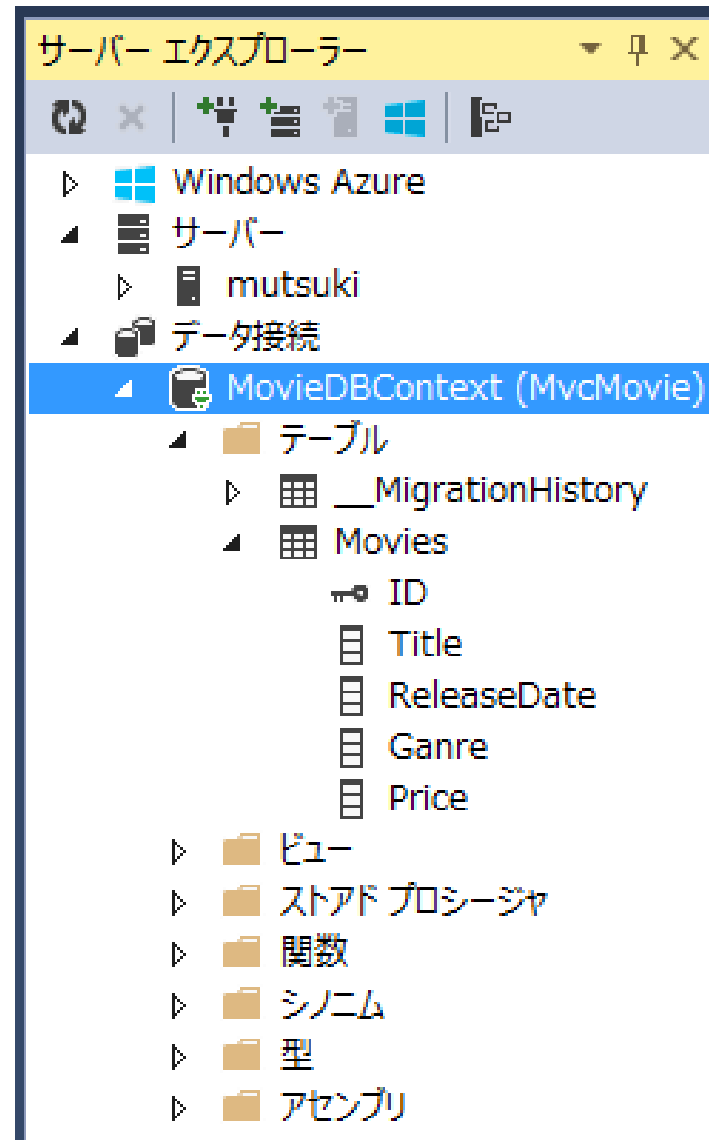


# サーバーエクスプローラー

- Movies.mdfをダブルクリック
- テーブルフォルダを開いて確認
- IDは鍵アイコン⇒プライマリキー

- 参考 : 「 Getting Started with Entity Framework 6 Code First using MVC 5」

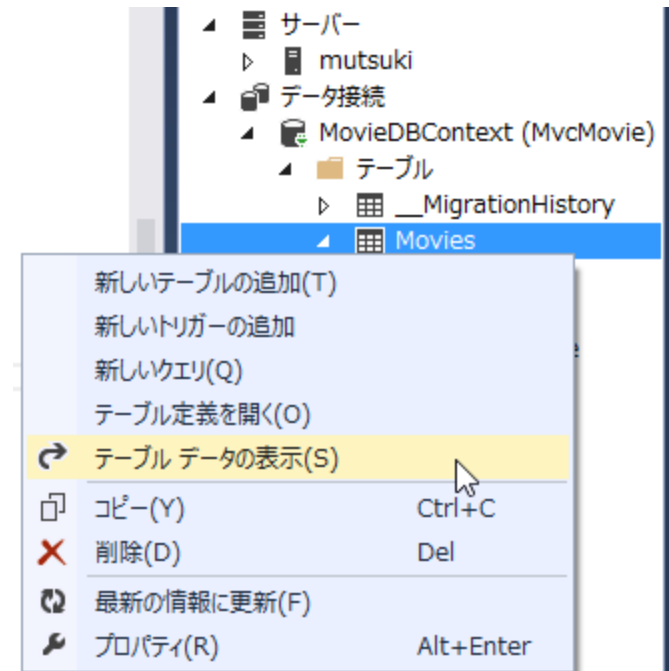
<http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application>





# テーブルデータの表示

- Moviesテーブルを右クリック  
⇒ テーブルデータの表示



- 入力したレコードが確認できる

dbo.Movies [データ]					
最大行数(O): 1000					
	ID	Title	ReleaseDate	Genre	Price
▶	1	ゼブラーマン	2004/02/14 0...	アクション	5076.00
*	NULL	NULL	NULL	NULL	NULL

# テーブル定義を開く

- Moviesテーブルを右クリック  
⇒ テーブル定義を開く
- テーブルの構造を確認

The screenshot displays the SQL Server Enterprise Manager interface. On the left, the 'Server Enterprise Explorer' tree shows the 'MovieDbContext (MvcMovie)' database with a 'Movies' table. The 'Movies' table is selected, and a context menu is open, showing options like '新しいテーブルの追加(T)', '新しいトリガーの追加', '新しいクエリ(Q)', 'テーブル定義を開く(O)', 'テーブルデータの表示(S)', 'コピー(Y)', '削除(D)', '最新の情報に更新(F)', and 'プロパティ(R)'. The 'テーブル定義を開く(O)' option is highlighted. In the background, the 'dbo.Movies [デザイン]' window is visible, showing the table's structure with columns: ID (int, primary key), Title (nvarchar(MAX)), ReleaseDate (datetime), Genre (nvarchar(MAX)), and Price (decimal(18,2)). The 'T-SQL' tab is also visible, showing the CREATE TABLE script for the 'Movies' table.

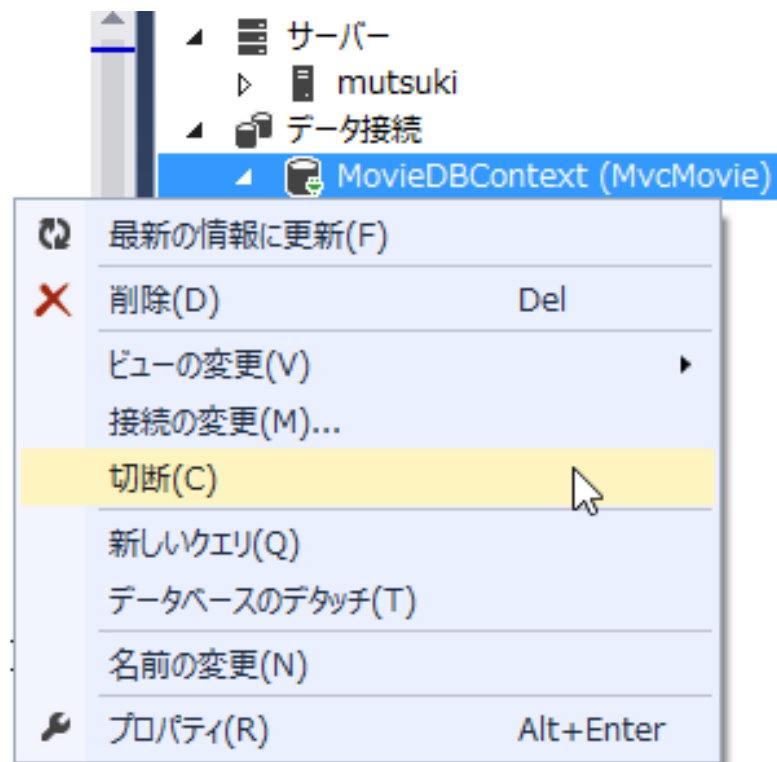
名前	データ型	Null を許容	既定
ID	int	<input type="checkbox"/>	
Title	nvarchar(MAX)	<input checked="" type="checkbox"/>	
ReleaseDate	datetime	<input type="checkbox"/>	
Genre	nvarchar(MAX)	<input checked="" type="checkbox"/>	
Price	decimal(18,2)	<input type="checkbox"/>	

```
CREATE TABLE [dbo].[Movies] (  
    [ID] INT IDENTITY (1, 1) NOT NULL,  
    [Title] NVARCHAR (MAX) NULL,  
    [ReleaseDate] DATETIME NOT NULL,  
    [Genre] NVARCHAR (MAX) NULL,  
    [Price] DECIMAL (18, 2) NULL,  
    CONSTRAINT PK_Movies PRIMARY KEY CLUSTERED ([ID])  
)
```

- MoviesテーブルとMoviesクラスが結合されている
- Entity Frameworkによる自動生成のスキーマによる

# 切断

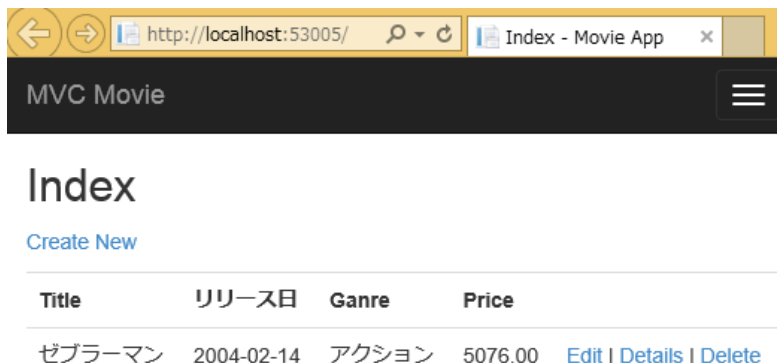
- 確認が終わったらMovieDBContextを右クリック⇒切断
- 切断しておかないと、次のプロジェクト実行時にエラーになる



- 次のトピック
  - スキャフォールディングの残りのコード
  - 検索インデックス

# EditメソッドとEdit View

- Editアクションメソッドの詳細を確認
- その前に : Release Dateを調整します
- Models/Movie.csを開く
- ハイライト部分を追加



Display(Name=“リリース日”)

```
using System;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;

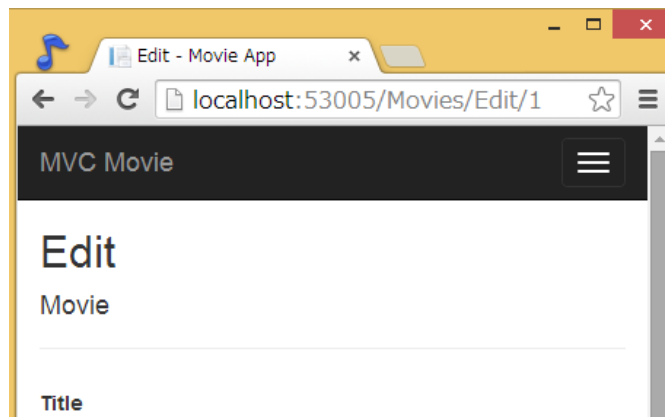
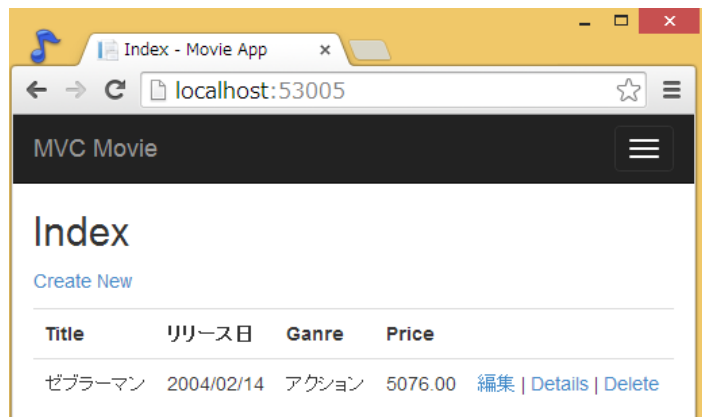
namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name=“リリース日”)]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }

    public class MovieDbContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

# 動作確認

- ビルドして一覧からEditを選択  
⇒URIを確認： /Movies/Edit/1
- Views/Movies/Index.cshtmlのActionLinkメソッドでURIを生成している  
`@Html.ActionLink("Edit", "Edit", new { id=item.ID })`
- リンク文字を日本語にするには：  
`@Html.ActionLink("編集", "Edit", new { id=item.ID })`
- 価格を `DataType.Currency` にしてみよう



## ActionLinkメソッド

- Htmlオブジェクト : `System.Web.Mvc.WebViewPage`を継承するヘルパー
- ヘルパーのActionLinkメソッドはControllerアクションメソッドへのHTMLリンクを動的に生成
- 引数1:表示される文字 (例 : `<a>編集</a>`)
- 引数2:アクションメソッド (例 : `Edit`アクション)
- 引数3:ルーティングのための匿名オブジェクト (例:ID)
- 結果としてURIは `http://localhost:xx/Movies/Edit/4`のようになる。参考 : `App_Start/RouteConfig.cs`
- 課題 : Create New, Details, Deleteも日本語にしてみよう

# パラメータの指定方法

- `http://localhost:xx/movies/edit?id=2`  
でも指定できる

Editアクションメソッドが2つある。  
こちらはGETの場合

```
// GET: Movies/Edit/5
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}
```

ValidateAntiForgeryToken  
リクエスト偽装のためのトークン発行  
Views/Movies/Edit.cshtml も参照

```
// POST: Movies/Edit/5
// 過多ポスト
// 詳細については、http://support.microsoft.com/fwlink/?LinkId=317598 を参照してください。
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include = "ID,Title,ReleaseDate,Ganre,Price")] Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Entry(movie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

POSTの場合

- ・ Bind属性による攻撃の無効化
- ・ 受け取るデータの選択（今回はすべて）

# Html.ValidateAntiForgeryToken()

- クロスサイトスクリプティング(XSRF, CSRF)攻撃対策
  - type=hiddenのinput要素を生成し、トークンを埋め込む
  - Movies ControllerのEditメソッドと一致していなければ先に進めない

```
@model MvcMovie.Models.Movie
```

```
@{  
    ViewBag.Title = "Edit";  
}
```

```
<h2>Edit</h2>
```

```
@using (Html.BeginForm())  
{
```

```
    @Html.AntiForgeryToken()
```

```
    <div class="form-horizontal">  
        <h4>Movie</h4>  
        <hr />
```

```
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })  
        @Html.HiddenFor(model => model.ID)
```



# HttpGetメソッド

- MovieのIDを取得
- Entity FrameworkのFindメソッドでDBのレコード検索
- 見つからないときは HttpNotFoundを返す
- スキャフォールディングのとき：
  - Viewに<label>と<input>要素も一緒に作る

```
<div class="form-group">
    @Html.LabelFor(model => model.Title, htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.Title, new { htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model => model.Title, "", new { @class = "text-danger" })
    </div>
</div>
```

# HTMLヘルパー

- `Html.LabelFor`  
フィールド名(Title, ReleaseDate, Genre, Price)を出力
- `Html.EditFor`  
<input> を出力
- `Html.ValidationMessageFor`  
バリデーションメッセージ
- `/Movies/Edit/id` にアクセスしてHTMLソースを確認

```
<h2>Edit</h2>
```

```
<form action="/Movies/Edit/1" method="post"><input name="__RequestVerificationToken" type="hidden"
value="xG4pPV6N8j-
I0QBAY604f41p00aJUBt7Iyp2Xd_Vo5eRLkw0j_1yD5DBcI7adqYyA9_JrGYmd3rJqG6w0UhaMt11KtJxCLEt8s8dd5-OKHI1" /
<div class="form-horizontal">
  <h4>Movie</h4>
  <hr />
```

# /Movies/EditにPOSTデータが渡ったとき

- Saveボタンが押されると<form>から/Movies/Editを呼ぶ

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include = "ID,Title,ReleaseDate,Ganre,Price")] Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Entry(movie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

モデルバインダがPOSTデータを取得し、Movieオブジェクトを作成

取得したデータがMovieオブジェクトの編集・更新に利用できるかどうかチェック

DBに保存

保存後Indexに戻り、Movieの一覧を表示。  
いま保存したばかりのデータも一緒に表示する。

# /Views/Movies/Edit.cshtml のバリデーション

- Html.ValidationMessageForヘルパーによる  
エラーメッセージ



The screenshot shows a web browser window with the address bar displaying `http://localhost:53005/movies/edit/`. The page title is "MVC Movie". The main heading is "Edit Movie". Below this, there are four input fields with labels: "Title", "リリース日" (Release Date), "Genre", and "Price". The "Title" field contains the text "半グレvsやくざ3". The "リリース日" field contains the text "aa", and below it is a red error message: "フィールド リリース日 は日付である必要があります。" (The field Release Date must be a date). The "Genre" field contains the text "やくざ" and has a clear button (X). The "Price" field contains the text "bbbb", and below it is a red error message: "フィールド Price には数字を指定してください。" (Please specify a number for the field Price).

MVC Movie

## Edit

Movie

---

**Title**

**リリース日**

フィールド リリース日 は日付である必要があります。

**Genre**

**Price**

フィールド Price には数字を指定してください。

# POSTとGET

- HttpGetの場合
  - Movieオブジェクト（Indexの場合はオブジェクトのリスト）をModelから取得する

Modelに変更を加える場合（作成、編集、削除）はPOSTを使う（セキュリティのためと、RESTパターンのため）

# 検索機能

- ジャンルや名前で検索できるようにしよう
- MoviesController の Indexアクションを編集

```
public ActionResult Index(string searchString)
{
    var movies = from m in db.Movies select m;
    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }
    return View(movies);
}
```

- movies: DBから検索するためのLINQクエリ
- searchString が存在するときはクエリを変更
- 「s=>s.Title」というのはラムダ表現
- Contains は SQL の Like にマップされる
- /movies/index?searchString=検索語

# 検索機能のためのルーティング調整

- /App\_Start/RouteConfig.cs の設定  
    {Controller}/{action}/{id}
- {id}でstringも受け付けるように調整

```
public ActionResult Index(string id)
{
    string searchString = id;
    var movies = from m in db.Movies select m;
    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }
    return View(movies);
}
```

- /movies/index/検索語

# 検索ボックス

- URLで指定するのも大変なので元に戻します

```
public ActionResult Index(string searchString)
{
    var movies = from m in db.Movies select m;
    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }
    return View(movies);
}
```

- /Views/Movies/Index.cshtml を編集  
@Html.ActionLink("新規作成", "Create") のすぐ下

<p>

```
@Html.ActionLink("新規作成", "Create")
```

```
@using (Html.BeginForm())
```

```
{
```

```
    <div>Title: @Html.TextBox("検索") <br />
```

```
    <input type="submit" value="Filter" /></div>
```

```
}
```

</p>



# 検索ボックス

- Html.BeginForm ヘルパー
  - <form>タグを作成
  - 検索ボタンを押したら、自分自身にPOST
- 検索するだけでデータの変更はない ⇒GETで良いので、HttpPostをオーバーロードする必要はない
- しかし、POSTを使うこともできる  
(MoviesController に追加)

```
// POST: Movies/SearchIndex
[HttpPost]
public string Index(FormCollection fc, string searchString)
{
    return "<h3>Form [HttpPost]Index: " + searchString + "</h3>";
}
```



# フレンドリーURL

- POSTだと次のような場合にページを再現できない
  - ブックマークしたとき
  - メールでURLを送るとき
- 解決策：BeginFormをオーバーロードして、POSTだった場合はGETバージョンのIndexメソッドに転送する

```
<h2>Index</h2>
```

```
]<p>
```

```
    @Html.ActionLink("新規作成", "Create")
```

```
    @using (Html.BeginForm("Index", "Movies", FormMethod.Get))
```

```
    {
```

```
        <div>@Html.TextBox("SearchString")
```

```
        <input type="submit" value="検索" /></div>
```

```
    }
```

```
</p>
```

```
    . . .
```

# ジャンル(Genre)で検索 : Controller

- MoviesController.csを再編集

```
public ActionResult Index(string movieGenre, string searchString)
{
    var GenreList = new List<string>();
    var GenreQry = from d in db.Movies orderby d.Genre select d.Genre;
    GenreList.AddRange(GenreQry.Distinct());
    ViewBag.movieGenre = new SelectList(GenreList);
    var movies = from m in db.Movies select m;
    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }
    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }
    return View(movies);
}
```

# ジャンル(Genre)で検索: Controller

パラメータ追加

ジャンルをデータベースから  
獲得し、リストGenreLstへ

```
public ActionResult Index(string movieGenre,
{
    var GenreList = new List<string>();
    var GenreQry = from d in db.Movies orderby d.Genre select d.Genre;
    GenreList.AddRange(GenreQry.Distinct());
    ViewBag.movieGenre = new SelectList(GenreList);
    var movies = from m in db.Movies select m;
    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }
    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }
    return View(movies);
}
```

Distinct()  
...重複を避ける

ViewBagにSelectList  
として保存

movieGenreを調べて、カラ  
でなかったら...

...今後のmovieはジャンルで  
フィルタリングする

# ジャンル(Genre)で検索: View

- Views/Movies/index.cshtml

```
<p>
    @Html.ActionLink("新規作成", "Create")

    @using (Html.BeginForm("Index", "Movies", FormMethod.Get))
    {
        <div>
            ジャンル: @Html.DropDownList("movieGenre", "All")
            タイトル: @Html.TextBox("SearchString")
            <input type="submit" value="検索" />
        </div>
    }
</p>
```

- movieGenre ... ControllerがViewBagに保存したやつ
- Html.DropDownList ... ViewBagから IEnumerable<SelectListItem> を探してドロップダウンリストを出力

# 動作確認と課題

- ジャンル、タイトル、およびその両方で検索して動作確認

MVC Movie 

## 映画一覧

新規作成

ジャンル: All ▼ タイトル:  検索

タイトル	リリース日	ジャンル	価格	
ゼブラーマン	2004/02/14	アクション	5076.00	<a href="#">編集</a>   <a href="#">詳細</a>   <a href="#">削除</a>
ゼブラーマン -ゼブラシティの逆襲-	2010/05/01	アクション	4104.00	<a href="#">編集</a>   <a href="#">詳細</a>   <a href="#">削除</a>
半グレvsやくざ3	2014/09/05	やくざ	5706.00	<a href="#">編集</a>   <a href="#">詳細</a>   <a href="#">削除</a>
仁義なき戦い	1973/01/13	やくざ	9999.00	<a href="#">編集</a>   <a href="#">詳細</a>   <a href="#">削除</a>

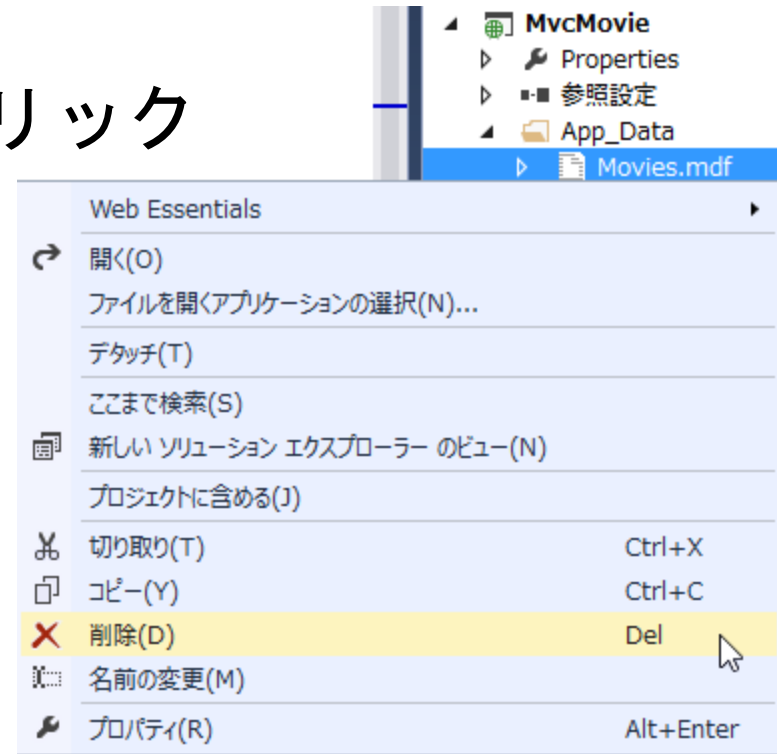
- やってみよう： リリース日、価格で検索できるようにしよう
- 映画の主演俳優を追加するにはどうしたら良いでしょうか？  
⇒データベースの変更が必要： 次の章へ。

# DBに新しいフィールドを追加

- Entity FrameworkのCode First では、Migrationにより Modelクラスへの変更をデータベースに反映させることができる
- このチュートリアルで最初のほうで試したように、Modelクラスに見合うデータベーススキーマに同期するように、Code Firstが自動的にテーブルを追加する。
- 同期していなければEntity Frameworkがエラーを返す
- エラーが出ることで、実行時の不具合の原因が明確になる

# Code First Migrations の設定

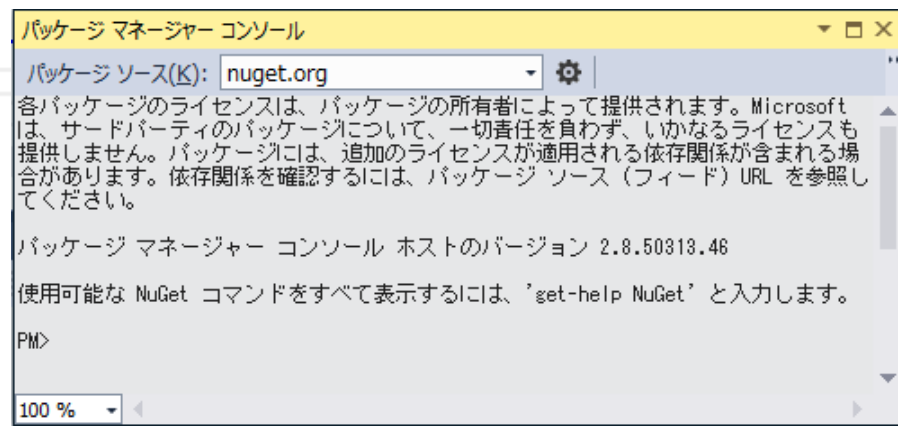
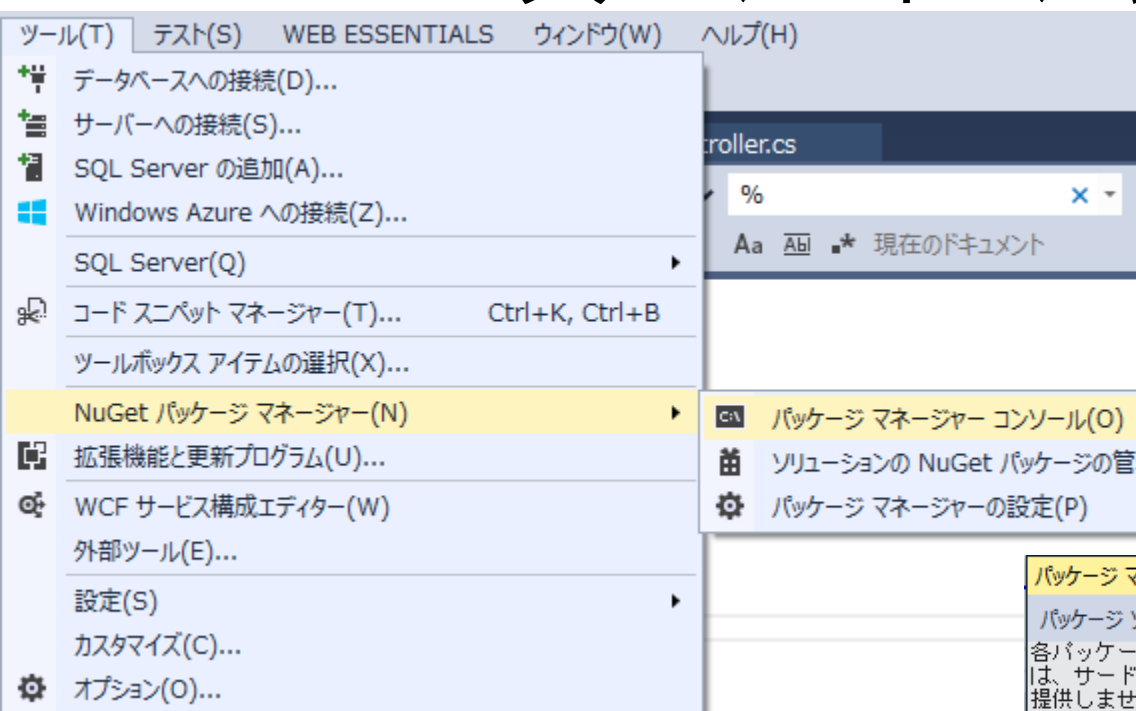
- 既存のデータベースを削除  
ソリューションエクスプローラ  
⇒ App\_Data/Movies.mdfを右クリック  
⇒ 削除





# Code First Migrations の設定

- ライブラリパッケージマネージャを開く  
ツール⇒NuGetパッケージマネージャー  
⇒パッケージマネージャーコンソール



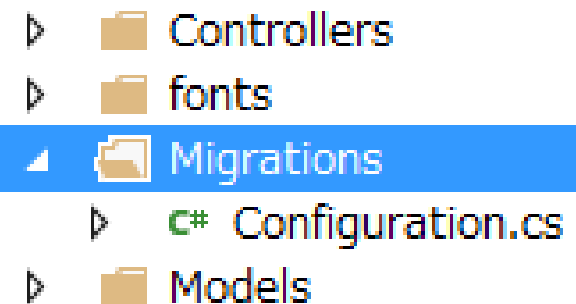
# パッケージマネージャーコンソール

- プロンプト 「PM>」 で次のコマンドレット

Enable-Migrations -ContextTypeName MvcMovie.Models.MovieDbContext

```
PM> Enable-Migrations -ContextTypeName MvcMovie.Models.MovieDbContext
コンテキストが既存のデータベースを対象にしているかをチェックしています...
Code First Migrations がプロジェクト MvcMovie で有効になりました。
PM> |
```

- プロジェクトにMigrationフォルダが作成される
- Configuration.csを開き、次ページのように編集



```

namespace MvcMovie.Migrations
{
    using MvcMovie.Models;
    using System;
    using System.Data.Entity.Migrations;

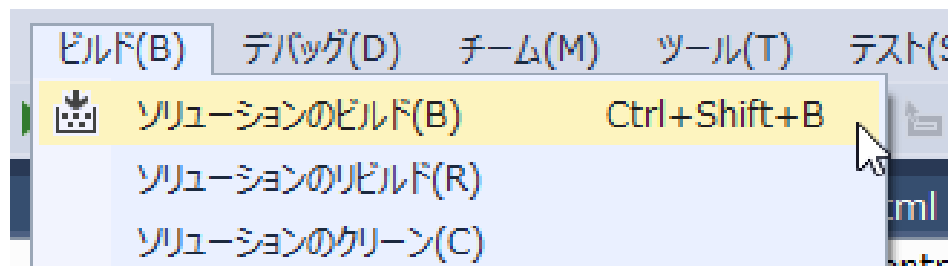
    internal sealed class Configuration : DbMigrationsConfiguration<MvcMovie.Models.MovieDbContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }
        protected override void Seed(MvcMovie.Models.MovieDbContext context)
        {
            context.Movies.AddOrUpdate(i => i.Title,
                new Movie
                {
                    Title = "仁義なき戦い",
                    ReleaseDate = DateTime.Parse("1973-1-13"),
                    Genre = "やくざ",
                    Price = 7990M
                },
                new Movie
                {
                    Title = "The Yakuza Papers 2: Deadly Fight in Hiroshima",
                    ReleaseDate = DateTime.Parse("1973-4-28"),
                    Genre = "Yakuza",
                    Price = 6990M
                }
            );
        }
    }
}

```

※ 映画のタイトルと数はご自由に

# Code First Migrations

- Code First Migrationsはマイグレーションの後に毎回Seedメソッドを呼び出す
- AddOrUpdate : レコードが存在しなければinsert、存在していればupdateを実行 (upsert)
- AddOrUpdateの最初の引数 : そのレコードが存在しているかを確認するためのカラム。今回はTitle (重複したTitleを登録しようとするとエラー)
- ソリューションのビルドを実行 (やっておかないとこれに続く操作でエラーになります)



# Code First Migrations

- パッケージマネージャーコンソールで  
add-migration Initial

「Initial」はmigrationファイルの名前で、任意に指定できる

```
PM>  
PM> add-migration Initial  
移行 'Initial' をスキャフォールディングしています。
```

この移行ファイルのデザイン コードには、現在の Code First モデルのスナップショットが含まれています。このスナップショットは次の移行をスキャフォールディングする際、モデルに対する変更の計算に使用されます。モデルに追加の変更を行い、この移行に含める場合は、'Add-Migration Initial' を再実行して再度スキャフォールディングできます。

```
PM>
```

- プロジェクトに Migrations/(タイムスタンプ)\_Initial.csが作成される。このファイルにはデータベーススキーマが含まれている。（開いて確認）
- 再びパッケージマネージャーコンソールで  
update-database

```
PM> update-database  
ターゲット データベースに適用されている SQL ステートメントを表示するには、'-Verbose' フラグを指定します。  
明示的な移行を適用しています: [201408021330398_Initial]。  
明示的な移行を適用しています: 201408021330398_Initial。  
Seed メソッドを実行しています。  
PM> |
```

# Code First Migrations

- update-databaseはレコードがすでに存在するとエラー「シーケンスに複数の要素が含まれています」
- この場合はデータベースの削除からやり直し
- 成功していれば、アプリケーションを実行して/MoviesにアクセスするとSeedのレコードが一覧に出る

## 映画一覧

[新規作成](#)

ジャンル:  ▼ タイトル:

タイトル	リリース日	ジャンル	価格	
仁義なき戦い	1973/01/13	やくざ	¥7,990	<a href="#">編集</a>   <a href="#">詳細</a>   <a href="#">削除</a>
The Yakuza Papers 2: Deadly Fight in Hiroshima	1973/04/28	Yakuza	¥6,990	<a href="#">編集</a>   <a href="#">詳細</a>   <a href="#">削除</a>

# カラム追加

- 「Director」カラムを追加したい
- 必要な作業は？
  - Model、View、Controller 修正
  - データベース更新
- まずModels/Movies.cs
- 変更したらビルド  
(画面左下に注目)



```
using System;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }

        [Display(Name = "タイトル")]
        public string Title { get; set; }

        [Display(Name = "監督")]
        public string Director { get; set; }

        [Display(Name = "リリース日")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }

        [Display(Name = "ジャンル")]
        public string Genre { get; set; }

        [Display(Name = "価格")]
        [DataType(DataType.Currency)]
        public decimal Price { get; set; }
    }

    public class MovieDbContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

## MovieController.csの編集

```
public ActionResult Create([Bind(Include =  
"ID,Title,Director,ReleaseDate,Genre,Price")] Movie  
movie)
```

```
public ActionResult Edit([Bind(Include =  
"ID,Title,Director,ReleaseDate,Genre,Price")] Movie  
movie)
```



# Views/Movies/Index.cshtml

- 一覧で監督が出るように

```
<tr>
  <th>
    @Html.DisplayNameFor(model => model.Title)
  </th>
  <th>
    @Html.DisplayNameFor(model => model.Director)
  </th>
  <th>
    @Html.DisplayNameFor(model => model.ReleaseDate)
  </th>
  <th>
    @Html.DisplayNameFor(model => model.Genre)
  </th>
  <th>
    @Html.DisplayNameFor(model => model.Price)
  </th>
</tr>

@foreach (var item in Model)
{
  <tr>
    <td>
      @Html.DisplayFor(modelItem => item.Title)
    </td>
    <td>
      @Html.DisplayFor(modelItem => item.Director)
    </td>
    <td>
      @Html.DisplayFor(modelItem => item.ReleaseDate)
    </td>
    <td>

```

# Views/Movies/Create.cshtml

- Createで監督が出るように

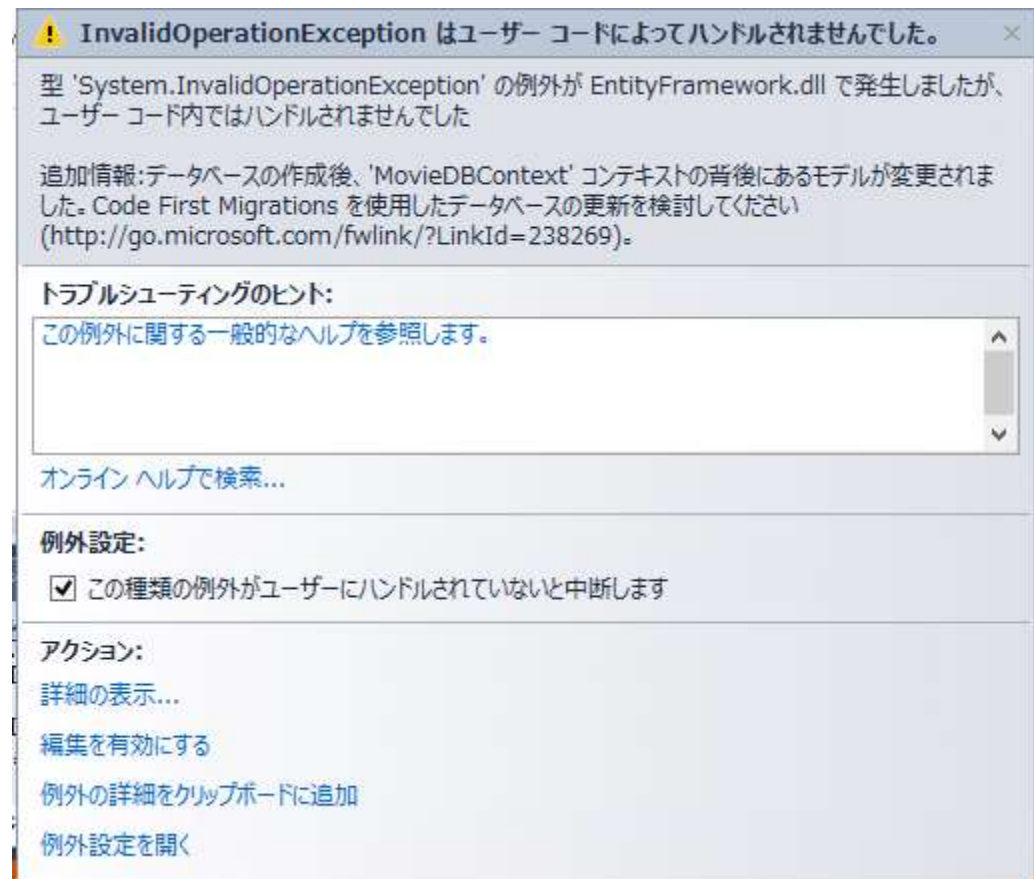
```
<div class="form-horizontal">
  <h4>Movie</h4>
  <hr />
  @Html.ValidationSummary(true, "", new { @class = "text-danger" })
  <div class="form-group">
    @Html.LabelFor(model => model.Title, htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
      @Html.EditorFor(model => model.Title, new { htmlAttributes = new { @class = "form-control" } })
      @Html.ValidationMessageFor(model => model.Title, "", new { @class = "text-danger" })
    </div>
  </div>
</div>
```

```
<div class="form-group">
  @Html.LabelFor(model => model.Director, htmlAttributes: new { @class = "control-label col-md-2" })
  <div class="col-md-10">
    @Html.EditorFor(model => model.Director, new { htmlAttributes = new { @class = "form-control" } })
    @Html.ValidationMessageFor(model => model.Director, "", new { @class = "text-danger" })
  </div>
</div>
```

- Details.cshtml、Edit.cshtml も同様に変更

# エラーの確認

- この時点でIndex.cshtmlにアクセスするとエラー  
⇒ カラムが変更されているため
- 対処法はいくつかあるが  
今回はCode Firstの  
「Migration」機能を使う



# Migrations/Configuration.cs の Seed を変更

- Moviesオブジェクトに「Director」を追加

```
context.Movies.AddOrUpdate(i => i.Title,  
    new Movie  
    {  
        Title = "仁義なき戦い",  
        Director = "深作欣二",  
        ReleaseDate = DateTime.Parse("1973-1-13"),  
        Genre = "やくざ",  
        Price = 7990M  
    },  
    ...)
```

- パッケージマネージャークンソールで次のコマンド  
add-migration Director

```
PM> add-migration Director
```

移行 'Director' をスキュールディングしています。

この移行ファイルのデザイン コードには、現在の Code First モデルのスナップショットが含まれています。このスナップショットは次の移行をスキュールディングする際、モデルに対する変更の計算に使用されます。モデルに追加の変更を行い、この移行に含める場合は、'Add-Migration Director' を再実行して再度スキュールディングできます。

# Migrationファイル

- 自動生成されたMigrationsファイルが開く

```
namespace MvcMovie.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class Director : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Movies", "Director", c => c.String());
        }

        public override void Down()
        {
            DropColumn("dbo.Movies", "Director");
        }
    }
}
```

# update-database

- パッケージマネージャークンソールで update-database

```
PM> update-database
ターゲット データベースに適用されている SQL ステートメントを表示する
には、'-Verbose' フラグを指定します。
明示的な移行を適用しています: [201408040030005_Director]。
明示的な移行を適用しています: 201408040030005_Director。
Seed メソッドを実行しています。
PM>
```

- 再実行して動作確認

## 映画一覧

新規作成

ジャンル:  ▼ タイトル:

タイトル	監督	リリース日	ジャンル	価格	
仁義なき戦い	深作欣二	1973/01/13	やくざ	¥7,990	<a href="#">編集</a>   <a href="#">詳細</a>   <a href="#">削除</a>
The Yakuza Papers 2: Deadly Fight in Hiroshima	深作欣二	1973/04/28	Yakuza	¥6,990	<a href="#">編集</a>   <a href="#">詳細</a>   <a href="#">削除</a>

# バリデーション

- セキュリティの観点から、ユーザから送られてくるデータは、すべてチェックしなければなりません。
- ASP.NET MVCのDRY(Don't Repeat Yourself)により、チェックは一度限りにする  
⇒ エラーの軽減、開発のスピードアップ
- バリデーションのルールを1箇所に書けば、アプリケーションの全体で共有する
- Movie Modelにバリデーションを追加  
⇒ Movie.csを編集

# バリデーション Movie.cs

```
[StringLength(60, MinimumLength=1), Required]  
[Display(Name = "タイトル")]  
public string Title { get; set; }
```

最大60文字  
最小1文字  
必須

```
[Display(Name = "監督")]  
public string Director { get; set; }
```

```
[Display(Name = "リリース日")]  
[DisplayFormat(DataFormatString="{0:yyyy-MM-dd}", ApplyFormatInEditMode=true)]  
public DateTime ReleaseDate { get; set; }
```

編集時も  
表示形式を適用

```
[Display(Name = "ジャンル")]  
public string Genre { get; set; }
```

```
[Display(Name = "価格")]  
[DataType(DataType.Currency)]  
[Range(1, 999999, ErrorMessage="{0}は{1}～{2}の間で入力してください。")]  
public decimal Price { get; set; }
```

{0} Display(Name)  
{1} Range(Min)  
{2} Range(Max)



# テーブル構造の確認、変更

- サーバーエクスプローラー  
⇒ Movies テーブル 右クリック ⇒ テーブル定義を開く
- すべての文字列カラムのデータ型は nvarchar(MAX)
- このスキーマを変更するため Migration を使う
- パッケージマネージャコンソールで  
add-migration DataAnnotations  
update-database

名前	データ型	Null を許
ID	int	<input type="checkbox"/>
Title	nvarchar(60)	<input checked="" type="checkbox"/>
ReleaseDate	datetime	<input type="checkbox"/>
Genre	nvarchar(MAX)	<input checked="" type="checkbox"/>
Price	decimal(18,2)	<input type="checkbox"/>
Director	nvarchar(MAX)	<input checked="" type="checkbox"/>

名前	データ型	Null を許容	既定
ID	int	<input type="checkbox"/>	
Title	nvarchar(MAX)	<input checked="" type="checkbox"/>	
ReleaseDate	datetime	<input type="checkbox"/>	
Genre	nvarchar(MAX)	<input checked="" type="checkbox"/>	
Price	decimal(18,2)	<input type="checkbox"/>	
Director	nvarchar(MAX)	<input checked="" type="checkbox"/>	

デザイン T-SQL

```
CREATE TABLE [dbo].[Movies] (  
    [ID] INT IDENTITY (1, 1) NOT NULL,  
    [Title] NVARCHAR (MAX) NULL,  
    [ReleaseDate] DATETIME NOT NULL,  
    [Genre] NVARCHAR (MAX) NULL,  
    [Price] DECIMAL (18, 2) NOT NULL,  
    [Director] NVARCHAR (MAX) NULL,  
    CONSTRAINT [PK_dbo.Movies] PRIMARY KEY CLUSTERED ([ID] ASC)  
);
```

# 自動生成されたDeleteメソッド

```
// GET: Movies/Delete/5
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}
```

削除ボタンを押すと最初にGETでリクエストする。この時点では確認ウィンドウを出すのみで、データベースを変更しない

POSTリクエストで、かつトークンのバリデーションがOKだったとき

```
// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    Movie movie = db.Movies.Find(id);
    db.Movies.Remove(movie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

DeleteConfirmedメソッドのアクション名を"Delete"に

データベースに変更を加える

# まとめ

- ASP.NET MVC5でWebアプリケーションを作成した
- データはLocalDBに保存できた
- 映画の情報のCRUD（Create, Read, Update, Delete）と検索の機能を作った

## 次のステップ

- 「Deploy a Secure ASP.NET MVC 5 app with Membership, OAuth, and SQL Database to an Azure Web Site」
- 「Getting Started with Entity Framework 6 Code First using MVC 5」
- マイクロソフト公式トレーニング#20486  
「ASP.NET MVC 4 Web アプリケーションの開発」（日本語）