Asp.NET MVC

Apanhado Geral

MVC

- Padrão arquitetural:
- Separação de responsabilidades
  - Acesso aos dados, lógica de visualização, etc
- Originalmente para aplicações desktop, muito aplicável em aplicações Web

MVC

- Três aspectos principais:
  - **Model**: Conjunto de classes que descrevem os dados e as regras de negócio para alteração e manipulação dos mesmos
  - **View**: Define como a interface de usuário será mostrada
  - **Controller**: Conjunto de classes que lidam com a comunicação do usuário, fluxo da aplicação e lógica específica da aplicação

MVC aplicado pelo Asp.NET

- **Models**: Representam o domínio, geralmente encapsulam dados armazenados em um banco e código para manipulá-los. É como um *data access layer*
- **View**: Template para gerar HTML dinâmico
- **Controller**: Classe especial que gerencia o relacionamento entre a view e o model. Responde à entrada de usuário, ”conversa” com o model e decide que view mostrar.

Estrutura da aplicação ASP.NET

Diretório	Objetivo
/Controllers	Local das classes de Controller que lidam com requisições de URLs
/Models	Local das classes de Model que representam e manipulam objetos de negócio
/Views	Local dos templates de UI que são responsáveis por mostrar a saída, como HTML
/Scripts	Local dos arquivos javascript (.js)
/fonts	O Bootstrap contém algumas fontes customizadas, que ficam nesse diretório
/Content	Local dos arquivos CSS, imagens e outros conteúdos que não são scripts
/App_Data	Onde ficam os arquivos de dados que serão lidos/escritos
/App_Start	Código de configuração para funcionalidades como roteamento, bundling e Web API

Convenções do ASP.NET MVC

- ASP.NET MVC se baseia em convenções
- Conceito originado do framework Ruby on Rails
- Três diretórios principais: Controllers, Views, Models, não são configurados em nenhum local
- Classes de controller terminam com *Controller*: `ProductController`, `HomeController`, etc. e ficam no diretório `Controllers`.
- Só há um diretório `Views` para todas as views da aplicação
- Views utilizadas por controllers ficam num subdiretório de `Views` que são nomeados de acordo com o nome do controller (-`Controller`). Por exemplo, as views de `ProductController` estão em `/Views/Product`
- Todos os elementos de UI reutilizáveis estão em um diretório `/Views/Shared`

Controllers

- Responsáveis pela entrada dos dados do usuário, geralmente alterando o model conforme a interação do usuário
- Lidam com o fluxo da aplicação: trabalha com os dados *entrantes* e provê dados que *saem* para a view.
- No ASP.NET MVC a URL indica ao mecanismo de rotas que classe de controller instanciar e qual método de *action* chamar, além de fornecer os argumentos necessários para esse método.
- O método do controller então decide que view utilizar e a view mostra o HTML

Um Controller exemplo

```
namespace MvcMusicStore.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult About()
        {
                    ViewBag.Message = "I like cake!";
            return View();
        }

        public ActionResult Contact()
        {
            ViewBag.Message = "Your contact page.";
            return View();
        }
    }
}
```

- Controller: HomeController
- 3 actions: Index, About, Contact
- Diferenciam-se apelas pelo conteúdo enviado para a view
- Acessíveis via: / , /About, /Contact
- Ou /Home/, /Home/About, /Home/Contact

Parâmetros em Actions

```
//
// GET: /Store/Browse?genre=?Disco
public string Browse(string genre)
{
    string message =
        HttpUtility.HtmlEncode("Store.Browse, Genre = " + genre);

    return message;
}
```

```
//
// GET: /Store/Details/5
public string Details(int id)
{
    string message = "Store.Details, ID = " + id;

    return message;
}
```

- Ao adicionar parâmetros aos métodos (actions), o framework automaticamente os reconhece e os parâmetros da URL são passados para o método
- Em uma URL como: `/Store/Details/5`, 5 é reconhecido automaticamente como o parâmetro ID, mostrado no código ao lado

Controllers em Resumo

- URLs são mapeadas para as classes de controller e suas actions
- Quase sempre utilizamos views como templates HTML que serão retornados aos navegadores
- Actions geralmente retornam um ActionResult, que lidam com códigos HTTP, redirecionamento de páginas, chamam os templates de views, etc

Views

- Responsável por prover a interface de usuário.
- Depois que o controller executa a lógica apropriada para uma URL, ele delega o *display* para a view
- Views não são acessíveis diretamente, são sempre renderizadas por um controller
- Em alguns casos a view não necessita de qualquer informação do controller, em outros necessita de dados
- Dados são transferidos através de um objeto de transferência chamado ”model”

Views Básicas

- Em `/Views/Home/Index.cshtml`, numa aplicação padrão, há uma view sem quaisquer dados do Controller. Basicamente um arquivo HTML.
- Primeiras 3 linhas setam o título da página

```
@{
    ViewBag.Title = "Home Page";
}

<div class="jumbotron">
    <h1>ASP.NET</h1>
    <p class="lead">ASP.NET is a free web framework for building
                    great Web sites and Web applications using HTML,
                    CSS and JavaScript.</p>
    <p><a href="http://asp.net" class="btn btn-primary btn-large">
       Learn more &raquo;</a></p>
</div>
```

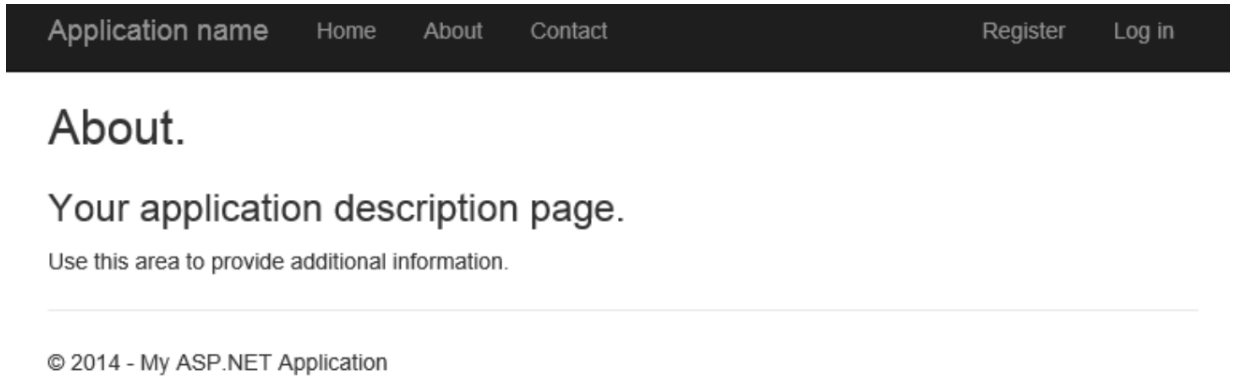
Passando informações básicas para a view

```
public ActionResult About()
{
        ViewBag.Message = "Your application description page.";

        return View();
}
```

```
 @{
     ViewBag.Title = "About";
 }
 <h2>@ViewBag.Title.</h2>
 <h3>@ViewBag.Message</h3>

 <p>Use this area to provide additional information.</p>
```



ViewBag.Title e ViewBag.Message são mostrados nessa view ”About”.
ViewBag.Message é setado no controller.

Convenções das views

- Dentro do diretório Views, é criado outro diretório para cada controller
  - Dentro desses diretórios dos controllers, há uma view para cada action
  - Que possuem o mesmo nome da action
- A lógica de seleção de views procura por /Views/NomeController/Action.cshtml
  - Para o método Index de Home, buscaria por: /Views/Home/Index.cshtml

Indicando views de maneira ”não-padrão”

- Passamos o nome da view, no controller:

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

Busca por /Views/Home/NotIndex.cshtml

```
public ActionResult Index()
{
    return View("~/Views/Example/Index.cshtml");
}
```

Busca por /Views/Example/Index.cshtml

~ = neste projeto

Views fortemente tipadas

- Geralmente é necessário passar mais dados do que uma simples string.
- ViewBag é como um dicionário de dados.
- ViewBag.Message = ”oi!”
- ViewBag.Albums = albums
- Tipagem e montagem dinâmica

Problemas com ViewBag

```
public ActionResult List()
{
 var albums = new List<Album>();
 for(int I = 0; i < 10; i++) {
   albums.Add(new Album {Title = "Product " + i});
 }
 ViewBag.Albums = albums;
 return View();
}
```

- Criamos uma lista, no controller
- E passamos via ”ViewBag” para a view
- Devemos fazer o ”cast” na view, para poder acessar os atributos de cada album

```
<ul>
@foreach (Album a in (ViewBag.Albums as IEnumerable<Album>)) {
 <li>@a.Title</li>
}
</ul>
```

View fortemente tipada

```
public ActionResult List()
{
 var albums = new List<Album>();
 for (int I = 0; i < 10; i++)
 {
   albums.Add(new Album {Title = "Album " + i});
 }
 return View(albums);
}
```

```
@model IEnumerable<MvcMusicStore.Models.Album>
<ul>
@foreach (Album p in Model) {
 <li>@p.Title</li>
}
</ul>
```

- Passamos o objeto diretamente para a view
- E indicamos, na view, qual o tipo do model, utilizando a declaração `@model`
- Cada view possui apenas *um* model.

View fortemente tipada

- Podemos inserir no web.config do diretório views o namespace dos models, para não der que usar o ”fully quallified name”:

```
<system.web.webPages.razor>
 …
 <pages pageBaseType="System.Web.Mvc.WebViewPage">
   <namespaces>
     <add namespace="System.Web.Mvc" />
     <add namespace="System.Web.Mvc.Ajax" />
     <add namespace="System.Web.Mvc.Html" />
     <add namespace="System.Web.Routing" />

     <add namespace="MvcMusicStore.Models" />
   </namespaces>
 </pages>
</system.web.webPages.razor>
```

ViewBag, ViewData e ViewDataDictionary

- Todos os dados são passados para as views através de um ViewDataDictionary, chamado de ViewData:
  - ViewData["CurrentTime"] = DateTime.Now;
- Embora ainda seja possível utilizar, apartir do ASP.NET MVC 3, a forma ”padrão” de se fazer é utilizando o ViewBag, que é uma espécie de ”wrapper” para o ViewData:
  - ViewBag.CurrentTime = DateTime.Now;
  - ViewBag.CurrentTime é equivalente a ViewData["CurrentTime"]

ViewData ou ViewBag?

- Para ”chaves com espaços” é necessário usar ViewData:
  - ViewData["Chave com espacos"]
- Além disso, é necessário fazer o ”cast” dos dados que passamos para a view, para podermos trabalhar com eles:
  - @Html.TextBox("name", ViewBag.Name) = Erro!
  - @Html.TextBox("name", (string) ViewBag.Name) = OK

View Models

- Às vezes necessitamos passar para a view mais do que um model, ou do que uma string
- Para isso, criamos uma espécie de ”model” específico para essa view → ViewModel
  - É como um agregador de atributos – empacotamos o necessário e enviamos para a view

View Models

- Por exemplo, um carrinho de compras que recebe os produtos, o valor total e uma mensagem

```
public class ShoppingCartViewModel {
   public IEnumerable<Product> Products { get; set; }
   public decimal CartTotal { get; set; }
   public string Message { get; set; }
}
```

- Na view, simplesmente aplicamos o tipo do `@model`

```
@model ShoppingCartViewModel
```

Razor Views

- Sintaxe C# em meio ao código HTML
- O marcador de código razor é o `@`
- O compilador é capaz de reconhecer em muitos casos a transição entre código razor e HTML, porém, podemos marcar um identificador razor com parênteses, para não haver dúvidas:

```
<span>@(rootNamespace).Models</span>

<li>Item_@(item.Length)</li>
```

- Escapamos o `@` com `@@`

```
<p>
 You should follow
 @@aspnet
</p>
```

Exemplos Razor

- Expressão Implícita:
```
<span>@model.Message</span>
```
- Expressão explícita
```
<span>1 + 2 = @(1 + 2)</span>
```
- Código não interpretado:
```
<span>@Html.Raw(model.Message)</span>
```
- Bloco de código
```
@{
    int x = 123;
    string y = "because.";
}
```

Exemplos Razor

- Combinar texto e marcação:

```
@foreach (var item  in itens) {
    <span>Item @item.Name.</span>
}
```

- Condicional:

```
@if (showMessage) {
    <text>This is plain text.</text>
}
```

Models

- Classes ”comuns” que representam o domínio da aplicação
- ASP.NET MVC fornece uma série de ferramentas para facilitar a persistência desses dados
- Podemos referenciar outra classe com um ID externo e um atributo de navegação:

```
public class Album
{
    public virtual int     AlbumId { get; set; }
    public virtual int     GenreId { get; set; }
    public virtual int     ArtistId { get; set; }
    public virtual string  Title { get; set; }
    public virtual decimal Price { get; set; }
    public virtual string  AlbumArtUrl { get; set; }
    public virtual Genre   Genre { get; set; }
    public virtual Artist  Artist { get; set; }
}
```

```
public class Artist
{
    public virtual int     ArtistId { get; set; }
    public virtual string  Name     { get; set; }
}
```

A classe DbContext

- Para que o Entity Framework persista as entidades (models) que declaramos, é necessaria uma classe DbContext, por exemplo:

```
public class MusicStoreDB : DbContext
{
   public DbSet<Album> Albums { get; set; }
   public DbSet<Artist> Artists { get; set; }
   public DbSet<Genre> Genres { get; set; }

}
```

- No controller, podemos utilizá-la:

```
var db = new MusicStoreDB();
var allAlbums = from album in db.Albums
                orderby album.Title ascending
                select album;
```

Eager Loading

- Para acessar dados relacionados de um model, necessitamos, explicitamente, declarar o ”eager loading” (algo como carregamento impaciente)

```
public class StoreManagerController : Controller
{
  private MusicStoreDB db = new MusicStoreDB();

  // GET: /StoreManager/
  public ActionResult Index()
  {
    var albums = db.Albums.Include(a => a.Artist).Include(a => a.Genre);
    return View(albums.ToList());
  }
// more later ...
```