

Visual Studio 2015 - ASP.NET com C# Recursos Avançados



Visual Studio 2015 - ASP.NET com C# Recursos Avançados

Créditos

Copyright © Monte Everest Participações e Empreendimentos Ltda.

Todos os direitos autorais reservados. Este manual não pode ser copiado, fotocopiado, reproduzido, traduzido ou convertido em qualquer forma eletrônica, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da Monte Everest Participações e Empreendimentos Ltda., estando o contrafator sujeito a responder por crime de Violação de Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos os logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

"As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais."

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

Coordenação Geral
Marcia M. Rosa

Coordenação Editorial
Henrique Thomaz Bruscagin

Atualização
José Eduardo Machado Grasso

Revisão Ortográfica e Gramatical
Fernanda Monteiro Laneri

Diagramação
Carla Cristina de Souza

Edição nº 1 | 1803_0_WEB
julho/ 2016



Este material constitui uma nova obra e é uma derivação da seguinte obra original, produzida por TechnoEdition Editora Ltda., em Dez/2014: **ASP.NET 2013 com C# - Recursos Avançados**

Autoria: José Eduardo Machado Grasso

Sumário

Informações sobre o treinamento	08
Capítulo 1 – Componentes	09
1.1. Introdução	10
1.2. Componentes do modelo MVC	11
1.3. Fluxo de execução do modelo MVC.....	20
1.3.1. AreaRegistration	20
1.3.2. FilterConfig.....	23
1.3.3. RouteConfig.....	26
1.3.4. BundlesConfig.....	27
Pontos principais	31
 Teste seus conhecimentos.....	33
 Mãos à obra!.....	37
Capítulo 2 – ASP.NET Core	57
2.1. Introdução	58
2.2. .NET Core	59
2.3. Ferramentas de desenvolvimento	63
2.4. Aplicação Console.....	66
2.5. Aplicação App Web	69
2.5.1. project.json	71
2.5.2. Bower	74
2.5.3. Gulp.....	76
2.5.4. Node.js e npm	78
2.5.5. Outros componentes.....	80
2.5.6. Microsoft.AspNet.Mvc.TagHelpers	80
2.6. Fluxo de execução	81
Pontos principais	86
 Teste seus conhecimentos.....	87
 Mãos à obra!.....	91
Capítulo 3 – SignalR	111
3.1. Introdução	112
3.2. WebSockets e outros protocolos.....	113
3.3. Implementando uma comunicação SignalR.....	113
3.4. Nome da função em JavaScript	121
3.5. Escopo dos clientes.....	122
Pontos principais	128
 Teste seus conhecimentos.....	129
 Mãos à obra!.....	133

Visual Studio 2015 – ASP.NET com C# Recursos Avançados

Capítulo 4 – Segurança (ASP.NET Core)	161
4.1. Introdução	162
4.2. Autenticação e autorização: História	162
4.3. ASP.NET Identity no .NET Core	164
Pontos principais	179
 Teste seus conhecimentos.....	181
 Mãos à obra!.....	185
Capítulo 5 – Segurança (.NET 4.6)	201
5.1. Introdução	202
5.2. ASP.NET Identity	202
5.3. OWIN	209
5.4. Implementações do Visual Studio	218
5.4.1. Web Forms e contas individuais	218
5.4.2. Criando usuários.....	224
5.4.3. Web Forms e autenticação externa	228
5.4.3.1. OAuth.....	229
5.4.3.2. Exemplo de autenticação com o Google	231
5.4.4. Web Forms e Windows	249
Pontos principais	252
 Teste seus conhecimentos.....	253
 Mãos à obra!.....	257
Capítulo 6 – Resources, Localization e Globalization	281
6.1. Introdução	282
6.2. Culture	282
6.3. Resources	286
6.3.1. Global Resources	287
6.3.2. Local Resources	289
6.4. Resources via código	290
6.5. Alterando a cultura via código	291
Pontos principais	298
 Teste seus conhecimentos.....	299
 Mãos à obra!.....	303

Sumário

Capítulo 7 – MVC – Validação e filtros	317
7.1. Introdução	318
7.2. Validação	318
7.3. Data Annotation.....	319
7.3.1. Required	319
7.3.2. StringLength	322
7.3.3. Regular Expression	323
7.3.4. Range	324
7.3.5. Remote	324
7.3.6. Compare.....	325
7.4. ModelState.....	325
7.5. Display	326
7.6. DisplayFormat.....	331
7.7. EditorForModel	332
7.8. Geração automática de código	334
7.8.1. Criando um Controller	334
7.8.2. Criando uma View.....	340
Pontos principais	343
Teste seus conhecimentos.....	345
Mãos à obra!	349
Capítulo 8 – Single Page App	397
8.1. Introdução	398
8.2. Introdução a Knockout, MVVM e Observer.....	399
8.2.1. MVVM (Model–View–ViewModel)	400
8.2.2. Biblioteca Knockout	401
8.2.3. Observable.....	403
8.2.4. Observable Array	406
8.3. O modelo SPA	409
8.4. Executando o modelo SPA.....	415
Pontos principais	418
Teste seus conhecimentos	419
Mãos à obra!	423
Capítulo 9 – Testes unitários	439
9.1. Introdução	440
9.2. Tipos de teste	441
9.3. TDD – Test-Driven Development.....	442
9.4. Projeto de teste.....	442
9.5. Classe e método de teste	445
9.6. Teste de unidade nos modelos.....	449
Pontos principais	451
Teste seus conhecimentos	453
Mãos à obra!	457

Informações sobre este treinamento

Para o melhor aproveitamento do curso **Visual Studio 2015 – ASP.NET com C# Recursos Avançados**, é imprescindível ter participado do curso Visual Studio 2015 – ASP.NET com C# Acesso a Dados, ou possuir conhecimentos equivalentes.

1

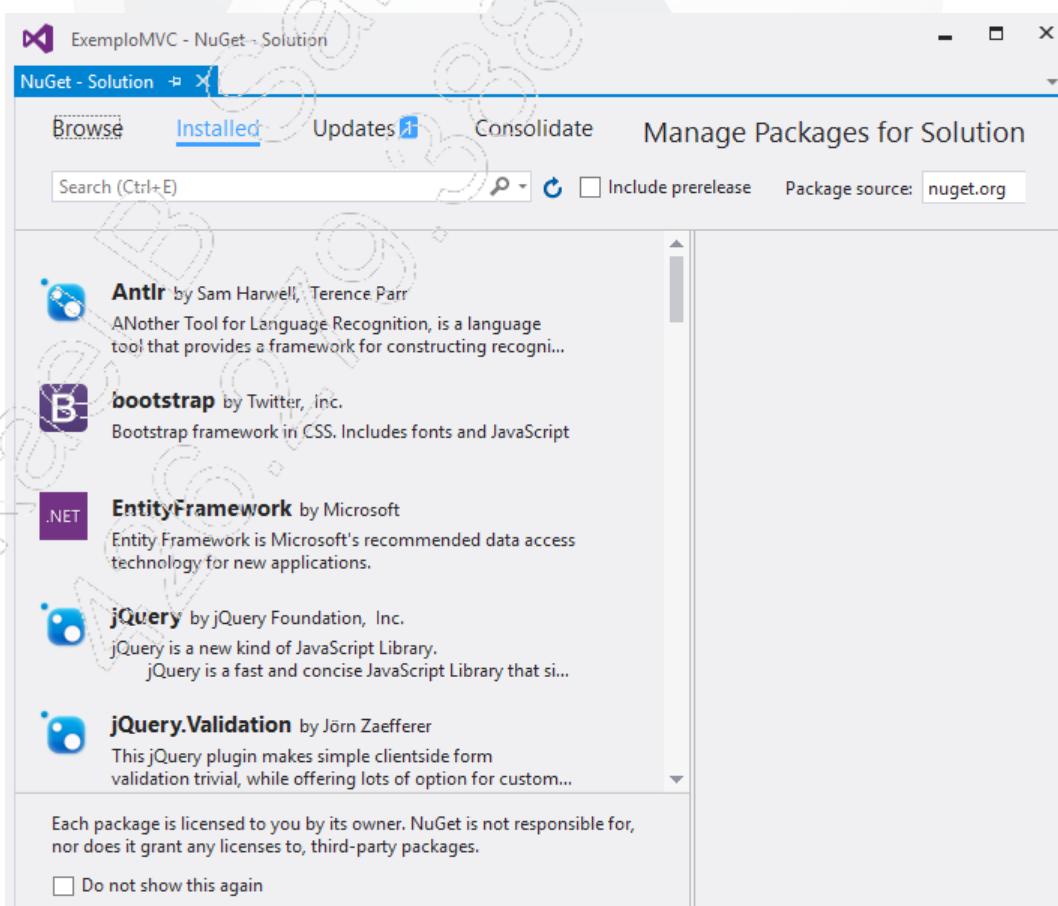
Componentes

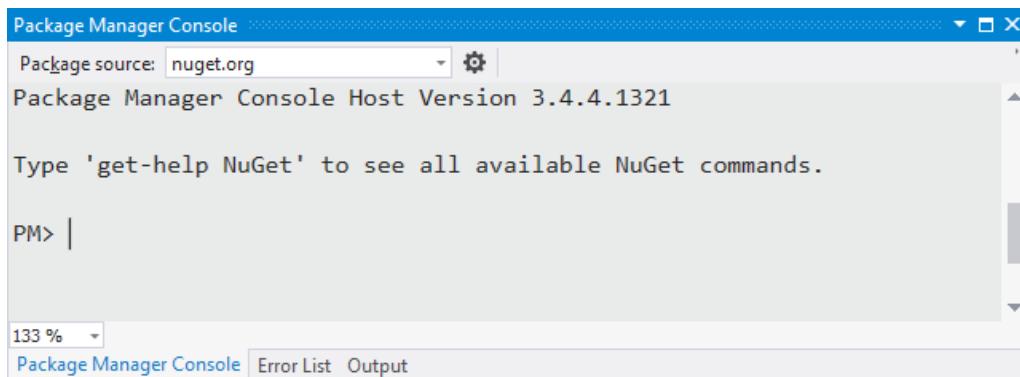
- ✓ Componentes do modelo MVC;
- ✓ Fluxo de execução do modelo MVC.

1.1. Introdução

Nos templates disponibilizados pelo Visual Studio, diversos componentes são utilizados em conjunto com os recursos nativos do .NET Framework e do ASP.NET. O objetivo deste capítulo é apresentar todos os componentes utilizados no modelo ASP.NET MVC e em que parte do aplicativo exemplo esses componentes são utilizados.

Alguns desses componentes dependem de outros e muitas vezes essa referência exige versões específicas. Administrar esse tipo de ligação entre componentes pode se tornar uma tarefa complicada. O gerenciador NuGet facilita muito a instalação e o controle de versão de componentes dentro de um projeto. O conjunto de arquivos e configurações necessárias no projeto para a utilização de um componente é chamado pelo NuGet de **pacote (Package)**. A utilização do NuGet pode ser feita por meio do gerenciador de pacotes, pelo menu **Tools / NuGet Package Manager / Manage NuGet Packages** ou por meio do console **Tools / NuGet Package Manager / Package Manager Console**.





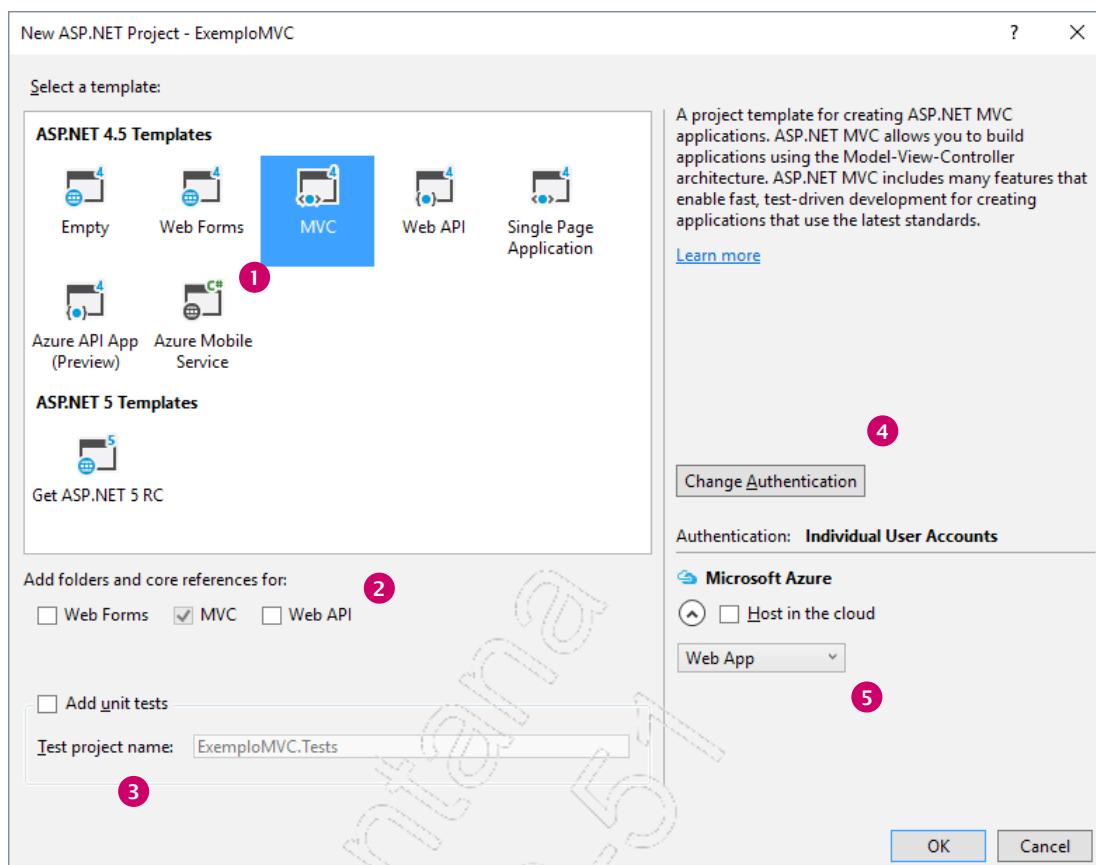
1.2. Componentes do modelo MVC

Um projeto ASP.NET MVC, ao ser criado por meio do comando de menu **File / New Project** e escolhendo a categoria **Web** e **ASP.NET**, apresenta as seguintes opções iniciais, que definem o conjunto de componentes que será incluído:

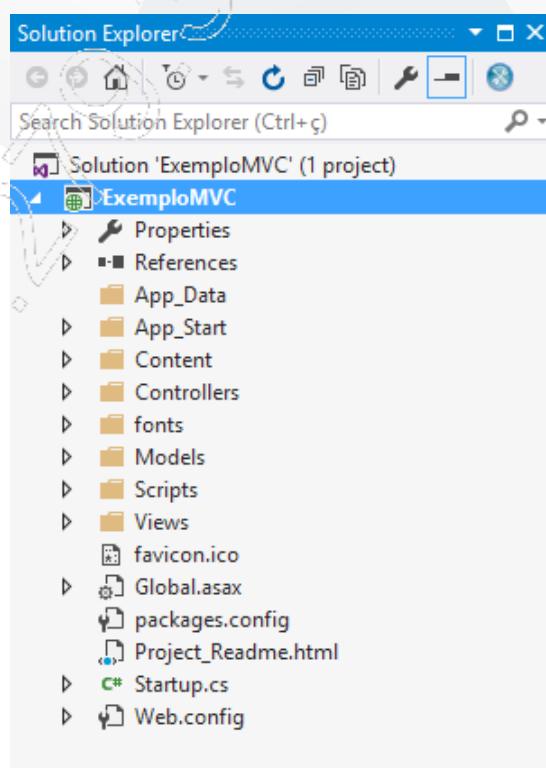
1. Modelos de projetos;
2. Referências a bibliotecas: **Web Forms**, **MVC** e **Web API**;
3. Opção de adicionar uma unidade de testes;
4. Opção de criar autenticação via Windows, formulários, programas externos ou contas empresariais;

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

5. Opção de hospedar a aplicação Web no Azure, no IIS ou no servidor local.



Escolhendo um projeto MVC com as opções padrão, a seguinte estrutura é criada:



Na raiz do site, estão presentes os seguintes arquivos:

1. **favicon.ico**;
2. **Global.asax**;
3. **packages.config**;
4. **Project_Readme.html**;
5. **Startup.cs**;
6. **Web.config**.

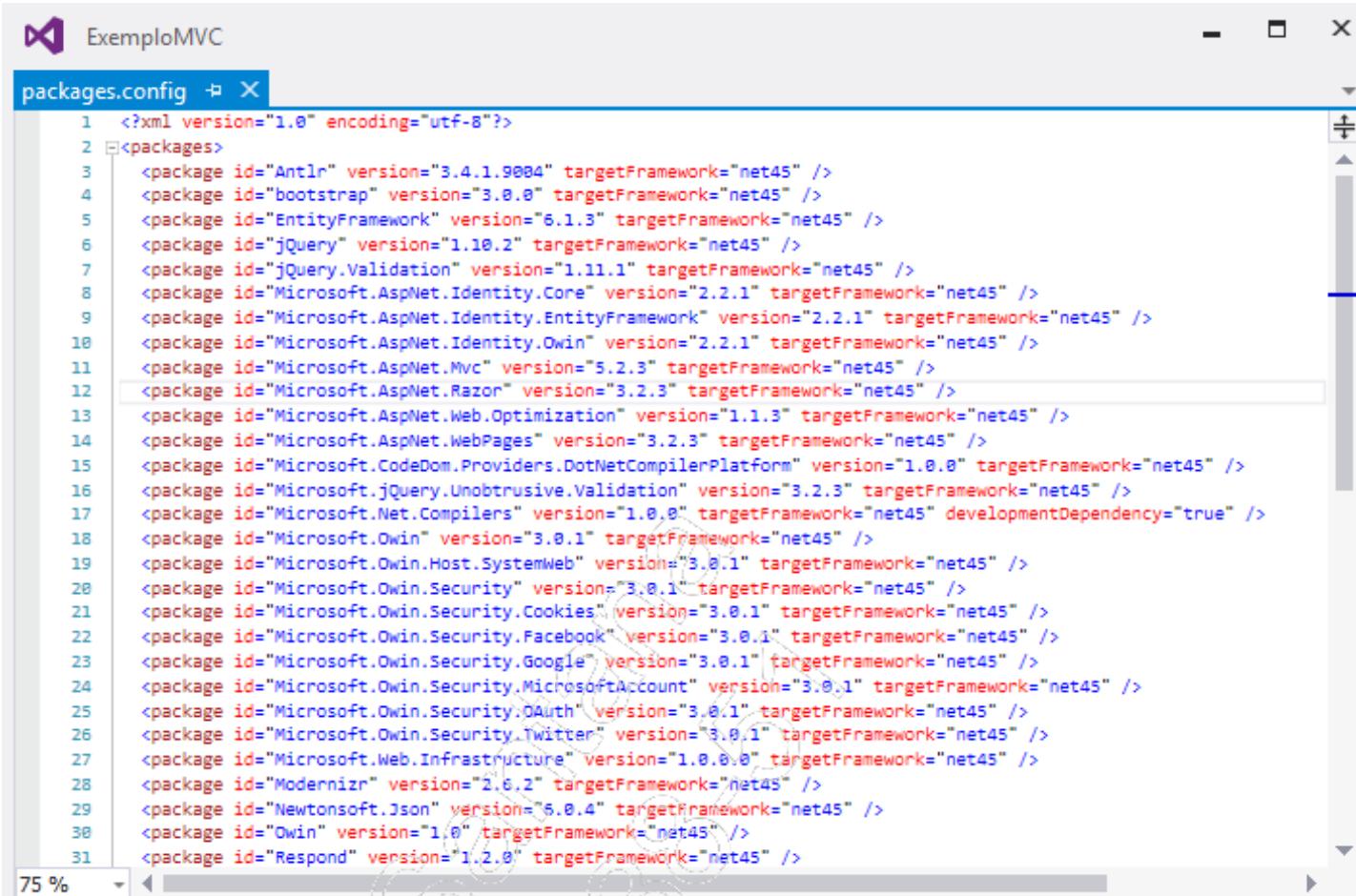
Nessa lista, os arquivos **Web.config**, **Global.asax** e **Startup.cs** estão relacionados ao processo de inicialização do aplicativo e serão analisados mais adiante.

O arquivo **favicon.ico** é uma convenção da Internet: uma imagem que aparece ao lado da URL, na maioria dos navegadores.

Project_Readme.html é um arquivo do Visual Studio com informações sobre o projeto e as tecnologias envolvidas.

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

O arquivo **packages.config** contém a lista de componentes e versões inseridos no projeto por meio do NuGet.



The screenshot shows the Visual Studio 2015 interface with a project named "ExemploMVC". In the Solution Explorer, the "packages.config" file is selected. The code editor displays the XML content of the package manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="Antlr" version="3.4.1.9004" targetFramework="net45" />
  <package id="bootstrap" version="3.0.0" targetFramework="net45" />
  <package id="EntityFramework" version="6.1.3" targetFramework="net45" />
  <package id="jQuery" version="1.10.2" targetFramework="net45" />
  <package id="jQuery.Validation" version="1.11.1" targetFramework="net45" />
  <package id="Microsoft.AspNet.Identity.Core" version="2.2.1" targetFramework="net45" />
  <package id="Microsoft.AspNet.Identity.EntityFramework" version="2.2.1" targetFramework="net45" />
  <package id="Microsoft.AspNet.Identity.Owin" version="2.2.1" targetFramework="net45" />
  <package id="Microsoft.AspNet.Mvc" version="5.2.3" targetFramework="net45" />
  <package id="Microsoft.AspNet.Razor" version="3.2.3" targetFramework="net45" />
  <package id="Microsoft.AspNet.Web.Optimization" version="1.1.3" targetFramework="net45" />
  <package id="Microsoft.AspNet.WebPages" version="3.2.3" targetFramework="net45" />
  <package id="Microsoft.CodeDom.Providers.DotNetCompilerPlatform" version="1.0.0" targetFramework="net45" />
  <package id="Microsoft.jQuery.Unobtrusive.Validation" version="3.2.3" targetFramework="net45" />
  <package id="Microsoft.Net.Compilers" version="1.0.0" targetFramework="net45" developmentDependency="true" />
  <package id="Microsoft.Owin" version="3.0.1" targetFramework="net45" />
  <package id="Microsoft.Owin.Host.SystemWeb" version="3.0.1" targetFramework="net45" />
  <package id="Microsoft.Owin.Security" version="3.0.1" targetFramework="net45" />
  <package id="Microsoft.Owin.Security.Cookies" version="3.0.1" targetFramework="net45" />
  <package id="Microsoft.Owin.Security.Facebook" version="3.0.1" targetFramework="net45" />
  <package id="Microsoft.Owin.Security.Google" version="3.0.1" targetFramework="net45" />
  <package id="Microsoft.Owin.Security.MicrosoftAccount" version="3.0.1" targetFramework="net45" />
  <package id="Microsoft.Owin.Security.OAuth" version="3.0.1" targetFramework="net45" />
  <package id="Microsoft.Owin.Security.Twitter" version="3.0.1" targetFramework="net45" />
  <package id="Microsoft.Web.Infrastructure" version="1.0.0.0" targetFramework="net45" />
  <package id="Modernizr" version="2.6.2" targetFramework="net45" />
  <package id="Newtonsoft.Json" version="6.0.4" targetFramework="net45" />
  <package id="Owin" version="1.0" targetFramework="net45" />
  <package id="Respond" version="1.2.0" targetFramework="net45" />

```

Os componentes instalados são os seguintes:

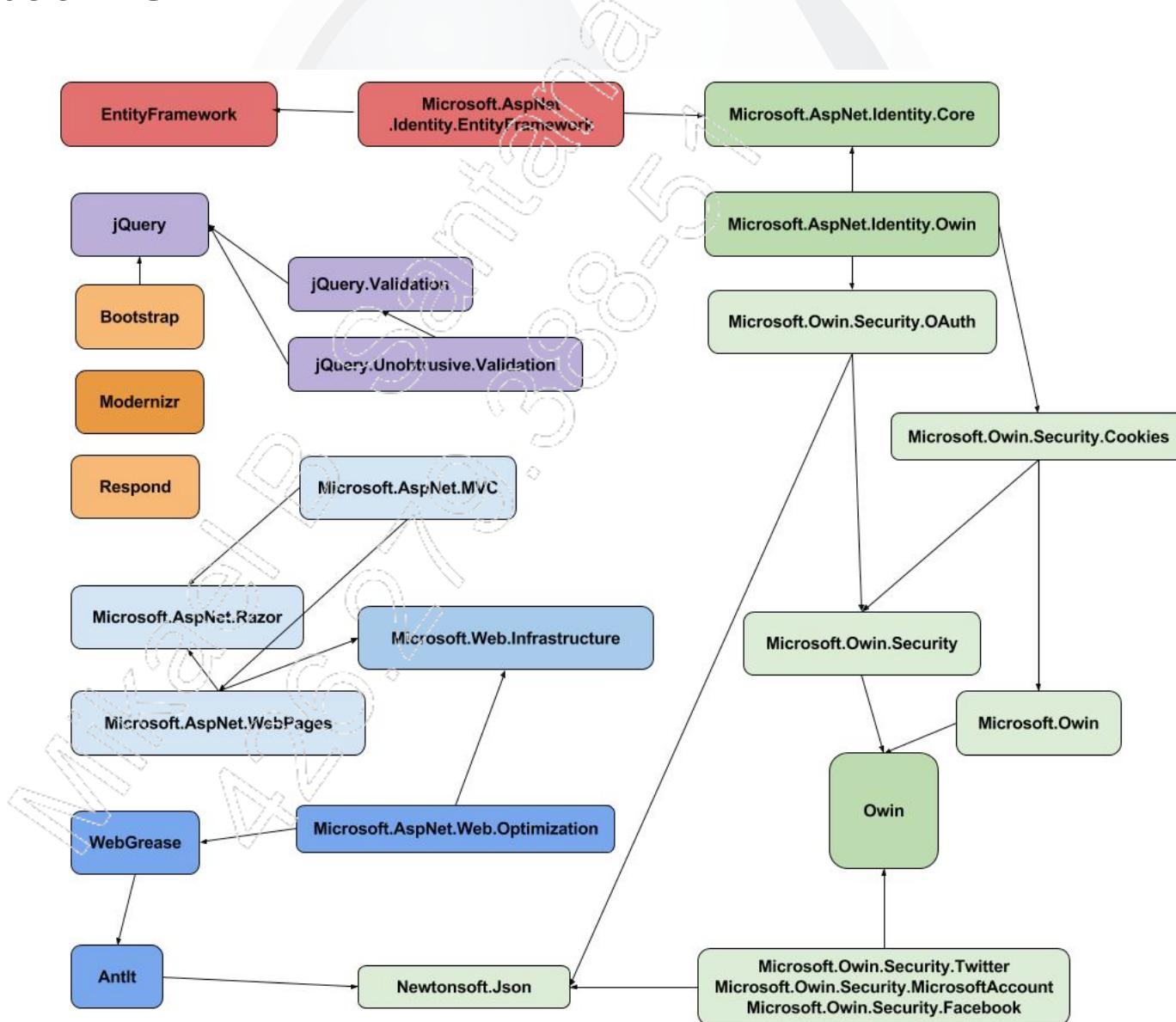
- **jQuery**: Biblioteca JavaScript que facilita a criação de códigos para manipular o documento HTML;
- **EntityFramework**: Framework para acesso a dados;
- **bootstrap**: Framework CSS para criar páginas responsivas (que se adaptam à resolução do dispositivo). O Bootstrap utiliza o jQuery;
- **Owin**: Open Web Interface for .NET é uma especificação de interface entre servidores .NET e aplicações Web;
- **Microsoft.AspNet.Razor**: Mecanismo razor de renderização de páginas;

- **Microsoft.AspNet.WebPages**: Mecanismo de criação de Web Pages. Este recurso é compartilhado com MVC;
- **Microsoft.AspNet.Mvc**: Framework MVC;
- **Microsoft.Web.Infrastructure**: Assembly que permite registrar módulos HTTP em tempo de execução;
- **Microsoft.AspNet.Web.Optimization**: Biblioteca para otimizar arquivos JavaScript e CSS, agrupando-os e reduzindo o tamanho dos arquivos;
- **Microsoft.AspNet.Identity.Owin**: Implementação do recurso de gerenciamento de usuários usando a especificação OWIN;
- **Microsoft.AspNet.Identity.EntityFramework**: Implementação do recurso de persistência de dados de usuários usando o Entity Framework;
- **Microsoft.Owin.Security**: Classes para implementar autenticação de usuários usando a especificação OWIN;
- **Microsoft.Owin.Security.Cookies**: Classes para implementar autenticação de usuários usando a especificação OWIN e cookies para armazenar dados de usuários;
- **Microsoft.Owin.Security.Facebook**: Classes para implementar autenticação de usuários por meio do Facebook;
- **Microsoft.Owin.Security.MicrosoftAccount**: Classes para implementar autenticação de usuários por meio do Windows;
- **Microsoft.Owin.Security.OAuth**: Classes para implementar autenticação de usuários por meio de diversos provedores;
- **Microsoft.Owin.Security.Twitter**: Classes para implementar autenticação de usuários por meio do Twitter;
- **Respond**: Biblioteca JavaScript para criar páginas responsivas;

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

- **WebGrease**: Biblioteca JavaScript para otimizar arquivos CSS e JavaScript;
- **AntlIt**: Ferramenta para reconhecimento de linguagem de programação e estruturas gerais. Usado para otimizar JavaScript;
- **jQuery.Validation**: Biblioteca JavaScript para validação de campos;
- **Newtonsoft.Json**: Framework para ler ou gravar dados em formato JSON;
- **Modernizr**: Biblioteca JavaScript para criar páginas compatíveis com antigos navegadores.

O quadro adiante mostra algumas relações entre os componentes incluídos no modelo MVC:



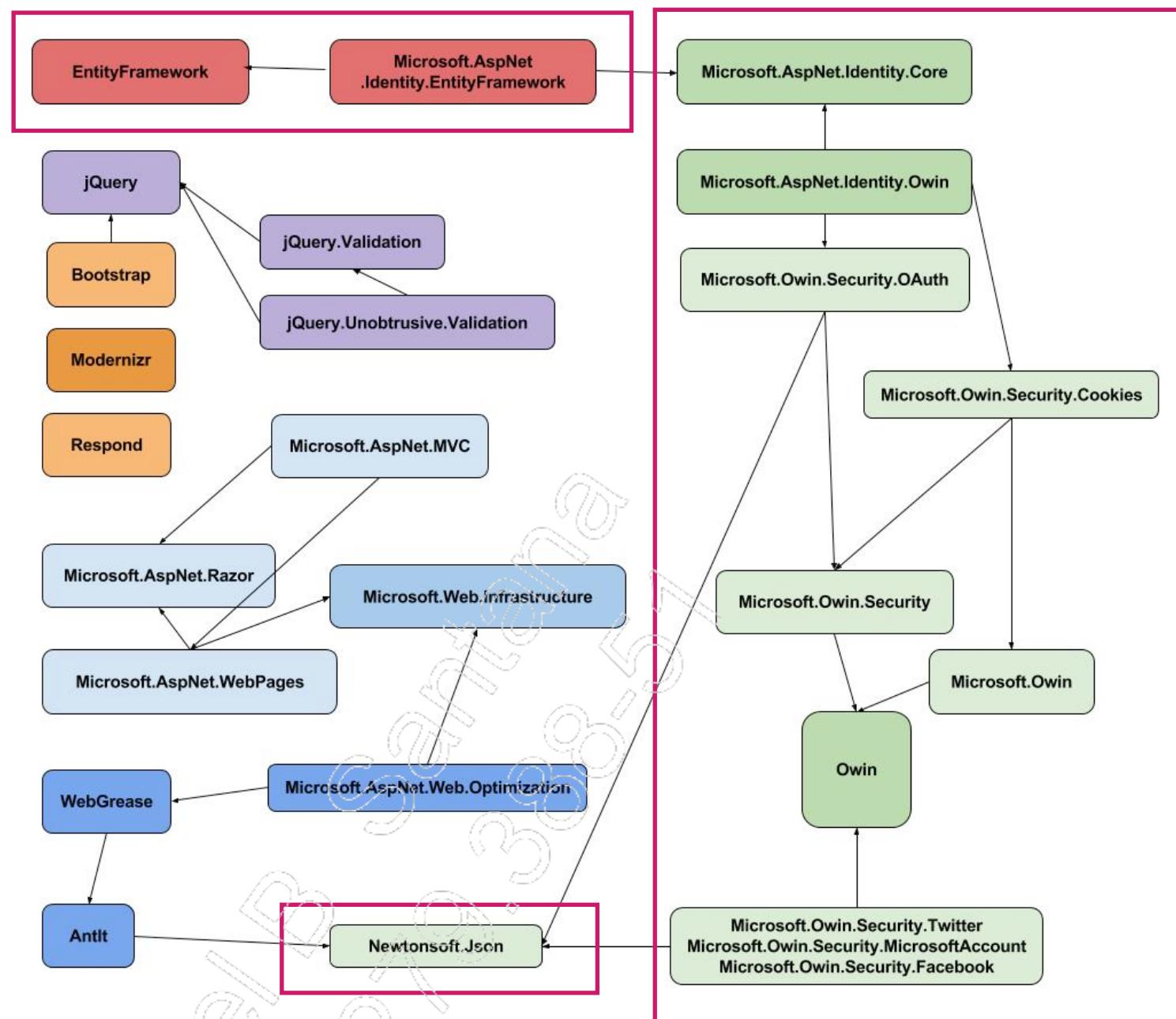
Quando o NuGet gerencia os pacotes, o relacionamento entre componentes fica em segundo plano, de modo que o programador possa se concentrar no código da aplicação em si, e não nos detalhes de funcionamento dos componentes.

Por exemplo, o **Microsoft.AspNet.Web.Optimization** fornece classes para otimizar arquivos JavaScript e folhas de estilo CSS. Para reduzir o tamanho de um arquivo JavaScript, é necessário interpretar o código, analisar as variáveis, encontrar os comentários inseridos pelo programador etc. Parte desse processo é feito pelo componente **Antlr**. O **Antlr** disponibiliza recursos para análise sintática de um texto de acordo com regras estabelecidas. Esse tipo de dependência (**Microsoft.AspNet.Web.Optimization->Antlr**) fica transparente para o programador com o uso do NuGet.

Cada biblioteca ou grupo se refere a uma característica de um projeto Web. Por exemplo, os componentes relacionados à segurança são, entre outros, os seguintes:

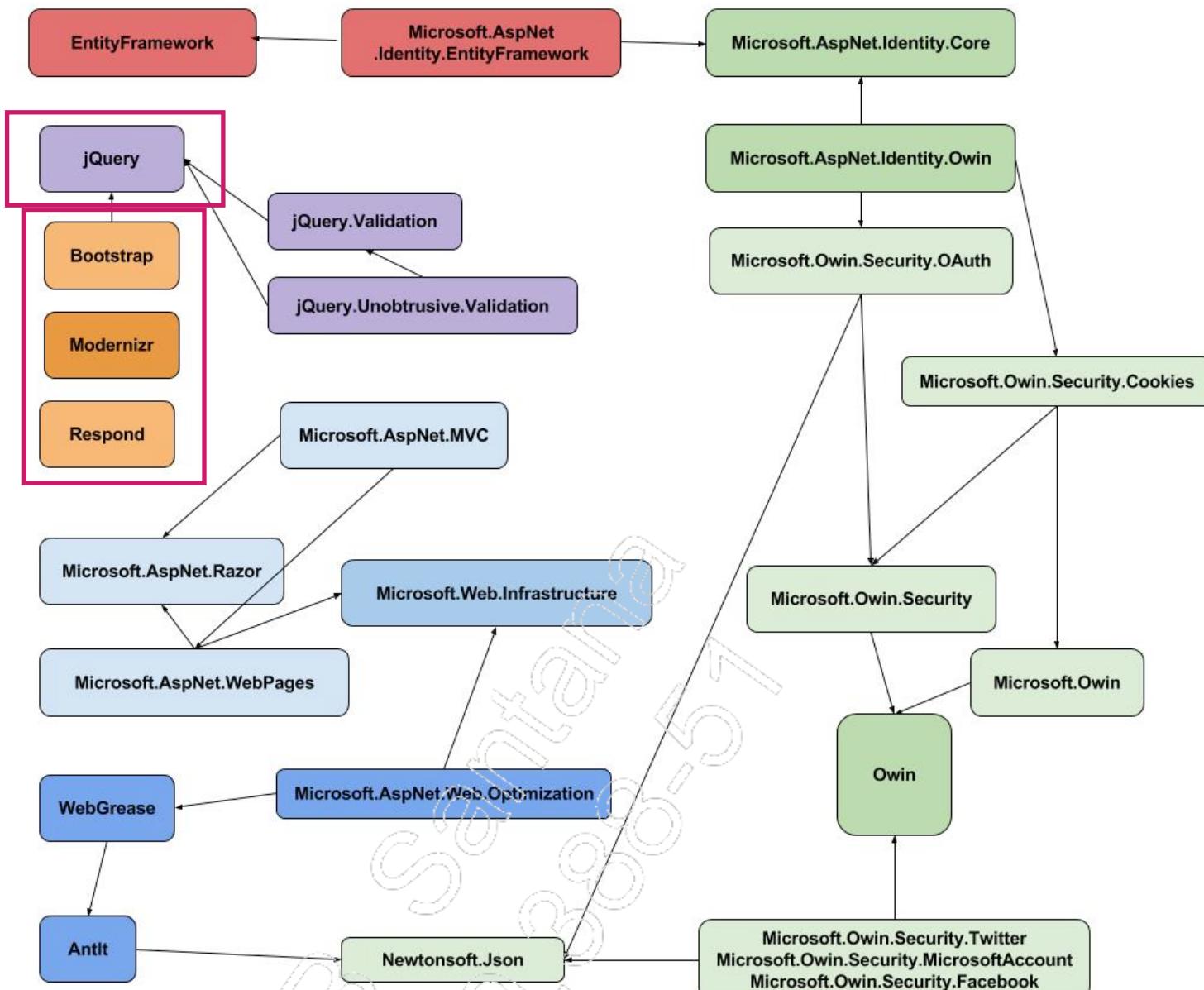
- **Owin;**
- **Microsoft.AspNet.Identity.Owin;**
- **Microsoft.AspNet.Identity.EntityFramework;**
- **Microsoft.Owin.Security;**
- **Microsoft.Owin.Security.Cookies;**
- **Microsoft.Owin.Security.Facebook;**
- **Microsoft.Owin.Security.MicrosoftAccount;**
- **Microsoft.Owin.Security.OAuth;**
- **Microsoft.Owin.Security.Twitter;**
- **EntityFramework;**
- **Microsoft.AspNet.Identity.EntityFramework.**

Visual Studio 2015 - ASP.NET com C# Recursos Avançados



O quadro adiante mostra os componentes relacionados, direta ou indiretamente, à aparência do site:

- **Bootstrap;**
- **Modernizr;**
- **Respond;**
- **jQuery.**



A melhor maneira de conhecer os componentes é utilizando-os em um projeto. Os templates do Visual Studio são perfeitos para o estudo inicial. Esses modelos foram criados pela mesma equipe que criou o ASP.NET e nos fornecem uma boa visão de como deve ser utilizado determinado recurso, ou, pelo menos, como o criador do componente espera que seja usado.

1.3. Fluxo de execução do modelo MVC

O ponto de partida da aplicação Web MVC do modelo disponível pelo Visual Studio (até o framework 4.6) é o arquivo **Global.asax**, no qual se encontra a classe **MvcApplication**. O método **Application_Start()** é o primeiro método a ser executado pelo mecanismo do MVC.

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

A seguir, cada um desses itens será analisado.

1.3.1. AreaRegistration

O primeiro comando da inicialização da aplicação MVC é a chamada ao método **RegisterAllAreas()** da classe **AreaRegistration**. Uma área é uma região organizada por pastas (**Models**, **Views** e **Controllers**) que são acionadas a partir de uma URL. Todo projeto MVC está dentro de uma área padrão.

A classe **AreaRegistration** é uma classe abstrata, ou seja, não é possível criar instâncias dela diretamente. Esse modelo foi construído assim para que as áreas inseridas criem sua própria instância herdada de **AreaRegistration**. Por exemplo, ao ser criada uma área chamada **Vendas**, uma classe chamada **VendasAreaRegistration** será responsável por configurar os detalhes como rotas de URLs e métodos padrão. Esse processo todo é criado automaticamente pelo Visual Studio ao adicionar uma nova área.

O motivo principal para usarmos áreas é a organização. Em um projeto complexo, a divisão em áreas permite o agrupamento de controllers, views e models por assunto, como no exemplo adiante:

\Raiz da aplicação

- \Controllers
- \Views
- \Models

\Áreas

\Financeiro

- \Controllers
- \Views
- \Models

\Vendas

- \Controllers
- \Views
- \Models

\Estoque

- \Controllers
- \Views
- \Models

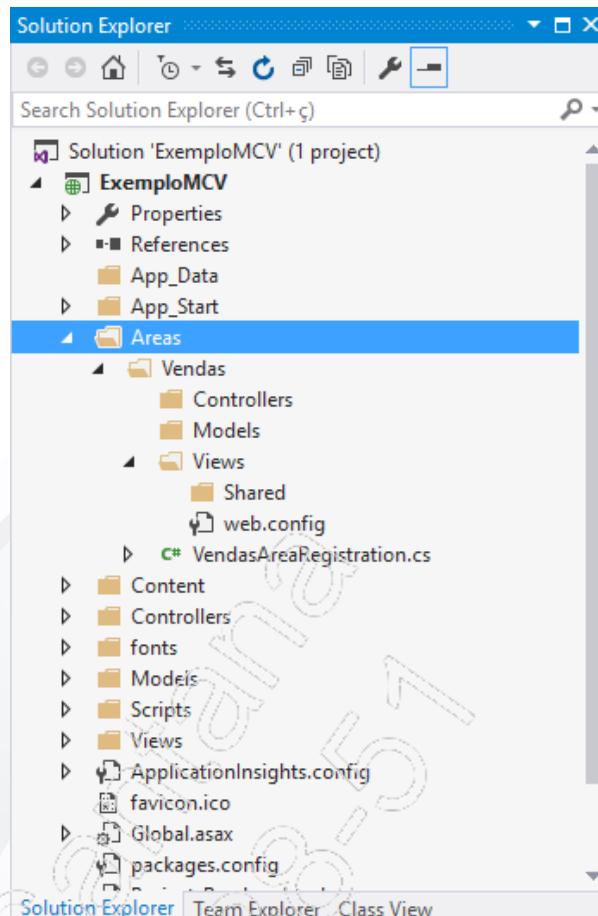
A URL para acessar um método de um controller da área **Financeiro**, por exemplo, seria a seguinte:

`http://servidor/webApp/Financeiro/Relatorios/FechamentoDoMes`

Em que **Financeiro** é o nome da área, **Relatórios** é o controller e **FechamentoDoMes** é o método.

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

Para adicionar uma área, use o menu de contexto do projeto e escolha **Add / Area**. Uma vez definido o nome, a área é adicionada, com as pastas essenciais e a classe de registro criadas automaticamente:



```
public class VendasAreaRegistration : AreaRegistration
{
    public override string AreaName
    {
        Get { return "Vendas"; }
    }

    public override void RegisterArea(
        AreaRegistrationContext context)
    {
        context.MapRoute(
            "Vendas_default",
            "Vendas/{controller}/{action}/{id}",
            new { action = "Index", id = UrlParameter.
Optional })
    }
}
```

1.3.2. FilterConfig

O segundo comando é a chamada ao método **RegisterGlobalFilters** da classe **FilterConfig**:

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

A classe **filterConfig** está definida no arquivo **/AppStart/FilterConfig.cs** e contém apenas esse método, que registra os filtros da aplicação. Um filtro é um atributo aplicado a um método ou a uma classe e que modifica ou complementa o modo como esse método ou essa classe são executados. Por exemplo, o filtro **OutputCache** armazena o retorno de um método por um tempo determinado. Isso torna a aplicação muito mais rápida, pois o método é processado apenas uma vez durante o tempo definido.

No exemplo adiante, o método **Index** é armazenado na memória por 60 segundos, usando o filtro **OutputCache**:

```
public class TesteController : Controller
{
    [OutputCache(Duration =60)]
    public ActionResult Index()
    {
        return View();
    }
}
```

Existem quatro tipos de filtros: **Authorization**, **Action**, **Result** e **Exception**.

Os filtros do tipo **Authorization** são utilizados nos processos de autenticação e autorização. Os do tipo **Action** são executados antes ou depois que um método é executado e antes ou depois que o resultado é gerado. Os do tipo **Result** são executados antes ou depois de as views serem renderizadas. E os do tipo **Exception** são executados quando ocorre uma **Exception**.

As interfaces que definem esses métodos são, respectivamente: **IAuthorizationFilter**, **IActionFilter**, **IResultFilter** e **IExceptionFilter**.

Durante a inicialização do aplicativo, a classe **FilterConfig** registra um filtro para o tratamento de exceptions:

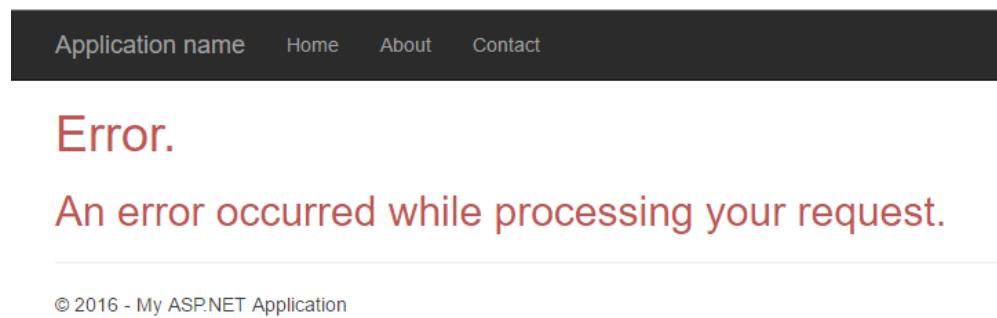
```
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
    }
}
```

A classe **HandleErrorAttribute** diz ao MVC framework para exibir a página de erro **/Views/Shared/Error.cshtml**. Para testar isso dentro do Visual Studio no servidor local, é necessário habilitar o erro personalizado no Web.Config:

```
<?xml version="1.0" encoding="utf-8"?>

<configuration>
    ...
    <system.web>
        <customErrors mode="On"></customErrors>
    ...
    </system.web>
    ...
</configuration>
```

Desse modo, a tela exibida (`/Views/Shared/Error.cshtml`) será a seguinte:



No lugar da tela de erro do ASP.NET:

Server Error in '/' Application.

Teste

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated.

Exception Details: System.Exception: Teste

Source Error:

```
Line 12:         public ActionResult Index()
Line 13:     {
Line 14:         throw new Exception("Teste");
Line 15:         return View();
Line 16:     }
```

Source File: C:\estudolasaspnet\ExemploMCV\ExemploMCV\Controllers\TesteController.cs **Line:** 14

Stack Trace:

```
[Exception: Teste]
   ExemploMCV.Controllers.TesteController.Index() in C:\estudo\aspnet\ExemploMCV\ExemploMCV\Controllers\TesteController.cs:14
   lambda_method(Closure , ControllerBase , Object[] ) +62
   System.Web.Mvc.ActionMethodDispatcher.Execute(ControllerBase controller, Object[] parameters) +14
   System.Web.Mvc.ReflectedActionDescriptor.Execute(ControllerContext controllerContext, IDictionary`2 parameters) +35
   System.Web.Mvc.ControllerActionInvoker.InvokeActionMethod(ControllerContext controllerContext, ActionDescriptor actionDescriptor) +39
   System.Web.Mvc.Async.AsyncControllerActionInvoker.BeginInvokeSynchronousActionMethod() +39/TAsyncResult+Start
```

1.3.3. RouteConfig

O terceiro e mais importante comando é a chamada ao método **RegisterRoutes** da classe **RouteConfig**:

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

A classe **RouteConfig** encapsula os métodos **MapRoute** e **IgnoreRoute** da classe **RouteCollection**. A classe **RouteTable** possui um método estático chamado **Routes** que retorna a coleção de **Routes** da aplicação.

```
public class RouteTable
{
    public RouteTable()
    {
        public static RouteCollection Routes { get; }
    }
}

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

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

O mapeamento das rotas é o responsável por definir que determinadas URLs retornem conteúdo vindo de métodos.

O padrão **{controller}/{action}/{id}** vai retornar o resultado do método **Controller.Metodo()**. Por exemplo, a URL **http://servidor/loja/produto/1** vai retornar o resultado do método **LojaController.produto(int id)**. Como, normalmente, os métodos retornam views, é necessário uma view chamada **Views/Loja/Produto.cshml**. Se essa view receber ou exibir uma instância de uma classe chamada **Models/Produto**, temos o framework MVC com seu fluxo de dados completo.

1.3.4. BundlesConfig

E, finalmente, na inicialização de um projeto MVC sem autenticação, o quarto comando é para registrar os bundles (vistos anteriormente no módulo I), que minimizam e agrupam arquivos.

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

A classe **BundleConfig** adiciona os arquivos JavaScript e CSS em grupos:

```
public class BundleConfig
{
    public static void RegisterBundles(BundleCollection bundles)
    {
        bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
                    "~/Scripts/jquery-{version}.js"));

        bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(
                    "~/Scripts/jquery.validate.*"));
        bundles.Add(new ScriptBundle("~/bundles/modernizr").Include(
                    "~/Scripts/modernizr-*"));

        bundles.Add(new ScriptBundle("~/bundles/bootstrap").Include(
                    "~/Scripts/bootstrap.js",
                    "~/Scripts/respond.js"));

        bundles.Add(new StyleBundle("~/Content/css").Include(
                    "~/Content/bootstrap.css",
                    "~/Content/site.css")));
    }
}
```

E as view que fazem uso desses arquivos declaram o local onde os arquivos serão inseridos:

_Layout.cshtml

...

```
@Styles.Render("~/Content/css")
```

```
@Scripts.Render("~/bundles/modernizr")
```

...

```
@Scripts.Render("~/bundles/jquery")
```

```
@Scripts.Render("~/bundles/bootstrap")
)
```

Se for escolhido o template ASP.NET com autenticação, outros itens serão acrescentados usando o arquivo **Startup.cs**. Nesse item, é utilizada outra técnica: a classe **Partial**.

```
public partial class Startup
{
    public void Configuration(IAppBuilder app)
    {
        ConfigureAuth(app);
    }
}
```

O arquivo `\App_Start\Startup.Auth.cs` contém o código que define os componentes de autenticação (esse tópico será visto nos próximos capítulos).

```
public partial class Startup
{
    public void ConfigureAuth(IAppBuilder app)
    {
        app.CreatePerOwinContext(...)

        app.UseCookieAuthentication(...)

        app.UseExternalSignInCookie(...);

    }
}
```

Para esse processo, por padrão, são utilizados os seguintes componentes:

- **Owin;**
- **Microsoft.AspNet.Identity.Owin;**
- **Microsoft.AspNet.Identity.EntityFramework;**
- **Microsoft.Owin.Security;**
- **Microsoft.Owin.Security.Cookies;**
- **Microsoft.Owin.Security.Facebook;**
- **Microsoft.Owin.Security.MicrosoftAccount;**
- **Microsoft.Owin.Security.OAuth;**
- **Microsoft.Owin.Security.Twitter;**
- **EntityFramework;**
- **Microsoft.AspNet.Identity.EntityFramework.**

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- **NuGet** é o gerenciador de pacotes para instalação, atualização e desinstalação de componentes;
- O arquivo **global.asax** contém o método **Application_Start**, que é o primeiro método executado no aplicativo;
- No método **Application_Start**, diversos itens de configuração são executados;
- Área é uma série de arquivos agrupados com a mesma estrutura do site raiz MVC;
- **Filtro** é um recurso que permite inserir processamento antes ou depois de um método ser executado;
- **Route** é o processo que decide qual URL vai acionar qual método de uma classe controller;
- **Bundles** é o processo de agrupar arquivos, realizar menos downloads do servidor e com isso agilizar o programa;
- O modelo MVC conta com mais de 20 componentes que podem ser gerenciados via NuGet.

1

Componentes

Teste seus conhecimentos

Mikael B
Santana
426.279.57



IMPACTA
EDITORA

1. Qual ferramenta do Visual Studio é usada para gerenciar os pacotes de aplicativos que são utilizados em uma aplicação?

- a) OWIN
- b) ASP.NET
- c) NuGet
- d) jQuery
- e) OAuth

2. Qual recurso do MVC permite incluir processamento extra antes ou depois de um método ser executado?

- a) Filtro
- b) Bundle
- c) Autenticação
- d) Routes
- e) Autorização

3. Qual é o primeiro método a ser executado em uma aplicação MVC?

- a) Index da classe HomeController
- b) Start da classe HomeController
- c) Application_Start da classe MVCAplication
- d) Configuration da classe MVCAplication
- e) Home da classe Controller

4. Qual mecanismo do MVC é responsável por instanciar uma classe Controller e executar um método baseado em uma URL?

- a) Filtro
- b) Bundle
- c) Autenticação
- d) Routes
- e) Autorização

5. Que recurso pode ser utilizado para organizar as URLs por categorias, evitando criar muitos nomes diferentes para controllers e métodos?

- a) Área
- b) Solution Explorer
- c) Filtro
- d) WebRoutes
- e) Refatoração

1

Componentes

Mãos à obra!

Mikael B. Sennhauser
426.279.388-0



IMPACTA
EDITORA

Laboratório 1

A – Criando um Web site institucional simples para apresentar uma empresa, usando o modelo MVC sem autenticação com duas áreas

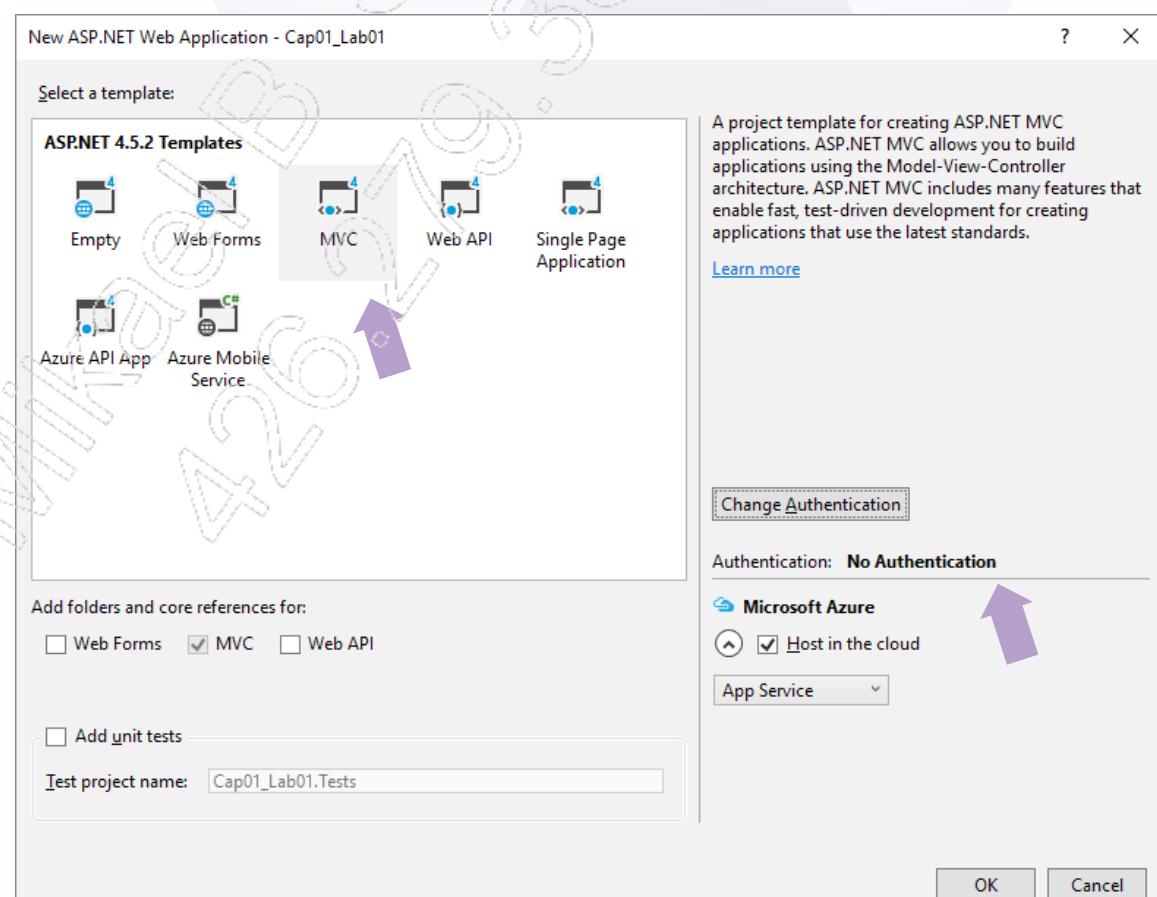
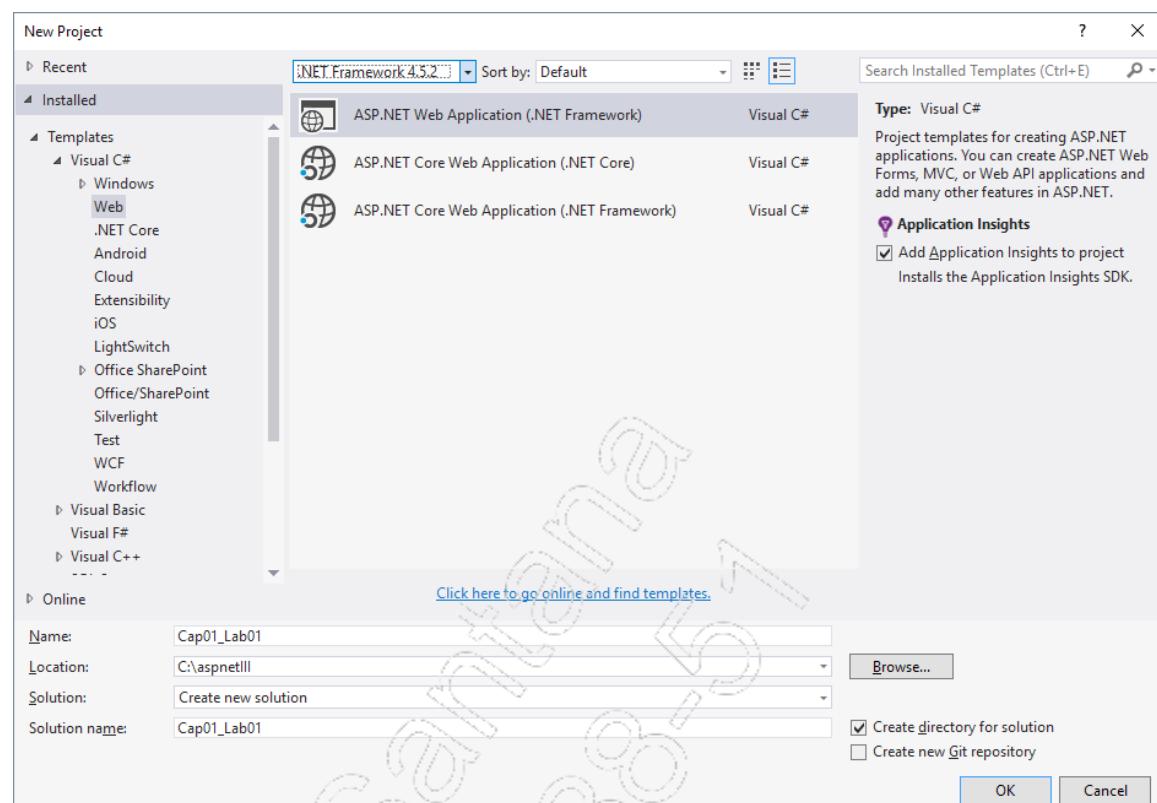
Neste laboratório, você criará um Web site institucional simples para apresentar uma empresa, usando o modelo MVC sem autenticação. Esse Web site terá quatro páginas: a página inicial com um resumo da empresa, uma página com um perfil mais detalhado, uma página de contato, na qual o visitante preenche um formulário e os dados são gravados em um arquivo de texto, e uma página de um futuro catálogo de produtos da empresa.

As telas são as seguintes:

- Tela inicial;
- Tela **Quem Somos**;
- Formulário de contato e resposta;
- Área de produtos.

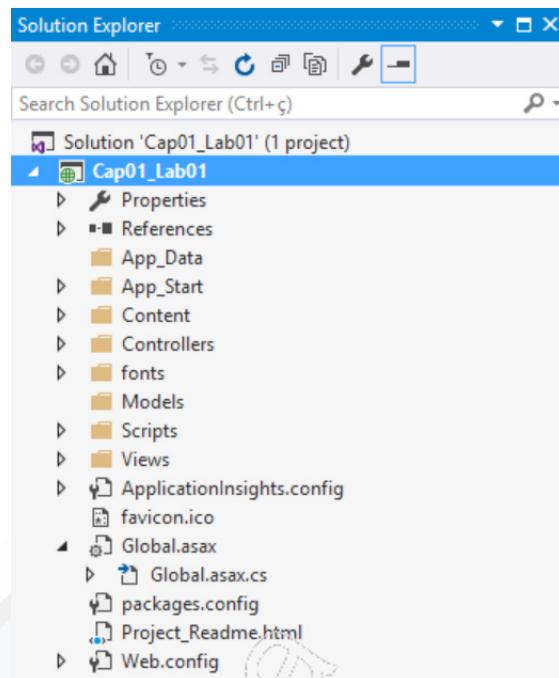
Para isso, siga os passos adiante:

1. O novo Web site terá como base o template MVC do Visual Studio. Crie um novo projeto **ASP.NET Web Application, MVC**, sem autenticação (**No Authentication**), chamado **Cap01_Lab01**:

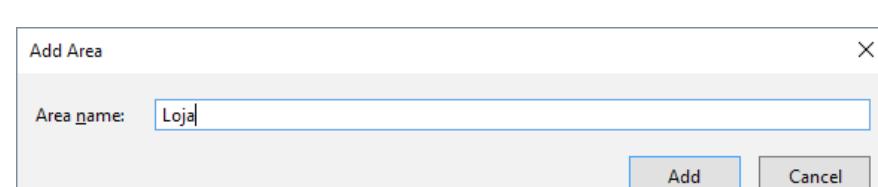
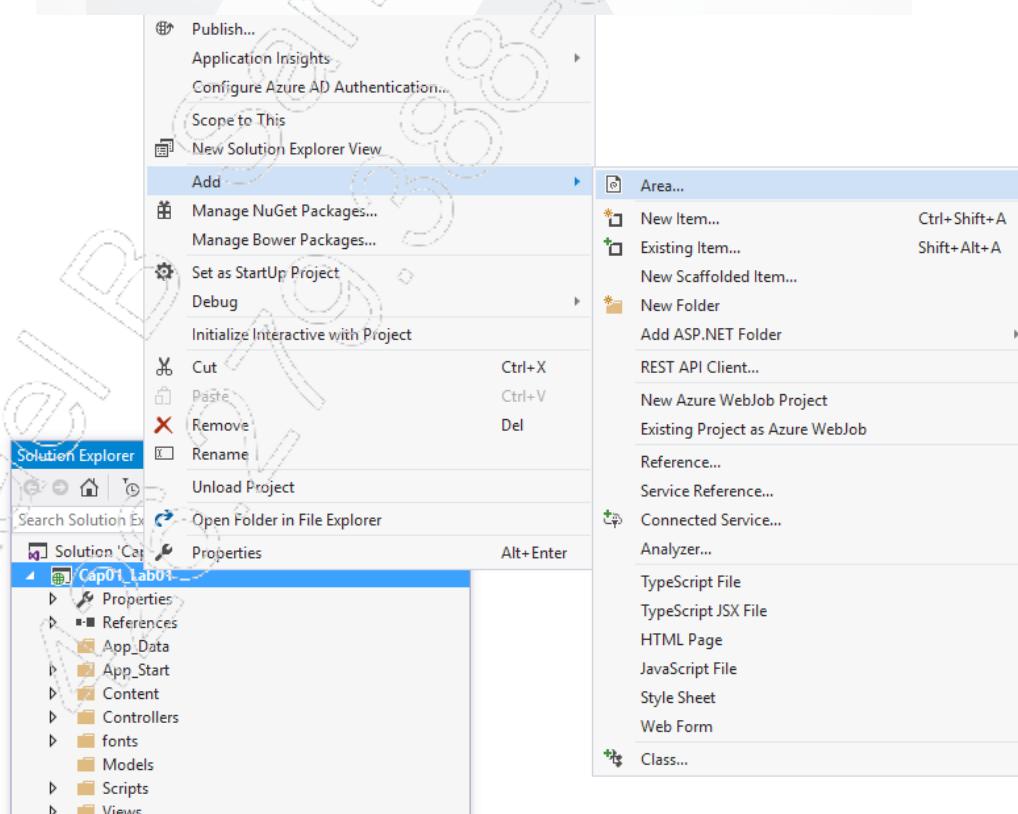


Visual Studio 2015 - ASP.NET com C# Recursos Avançados

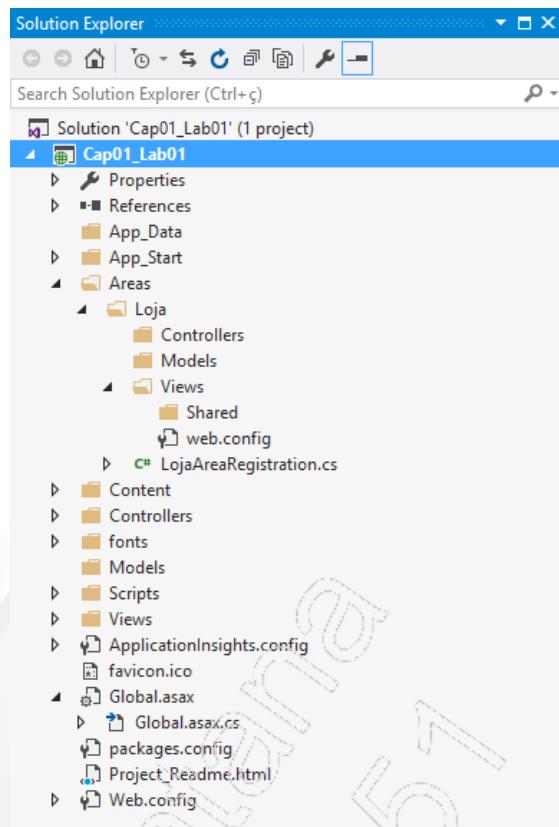
2. Verifique os arquivos criados:



3. Com o cursor no nome do projeto, abra o menu de contexto e escolha Add / Área. Defina o novo nome como Loja:



4. Repare que foram criadas várias pastas: **Areas**, **Loja**, **Models**, **Controllers**, **Views** e os arquivos **web.config** e **LojaAreaRegistration.cs**:



5. Abra o arquivo **LojaAreaRegistration.cs** e observe o arquivo criado. Repare que foi definida uma route: **Servidor/Loja/Controller/Action/Id**:

```
public class LojaAreaRegistration : AreaRegistration
{
    public override string AreaName
    {
        Get
        {
            return "Loja";
        }
    }
}
```

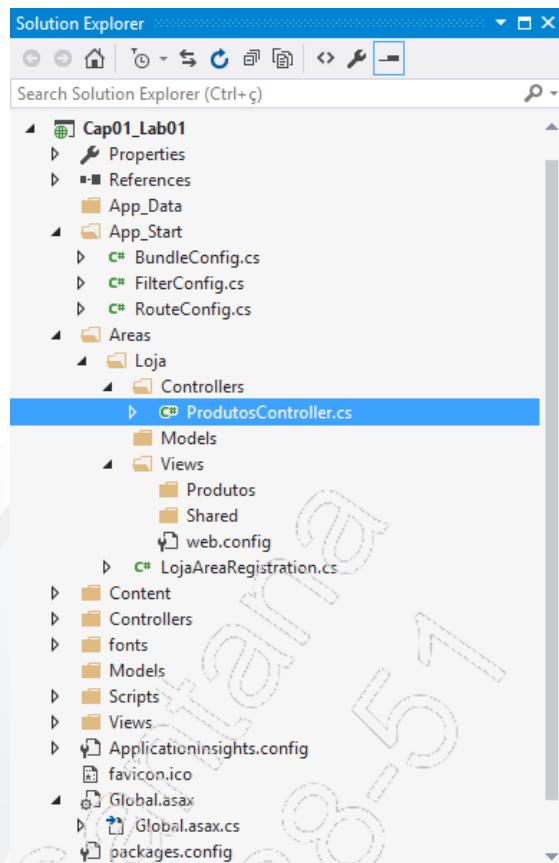
```
public override void RegisterArea(AreaRegistrationContext context)
{
    context.MapRoute(
        "Loja_default",
        "Loja/{controller}/{action}/{id}",
        new { action = "Index",
              id = UrlParameter.Optional }
    );
}
```

6. Abra o arquivo **/App_Start/RouteConfig.cs** e observe as routes padrão:

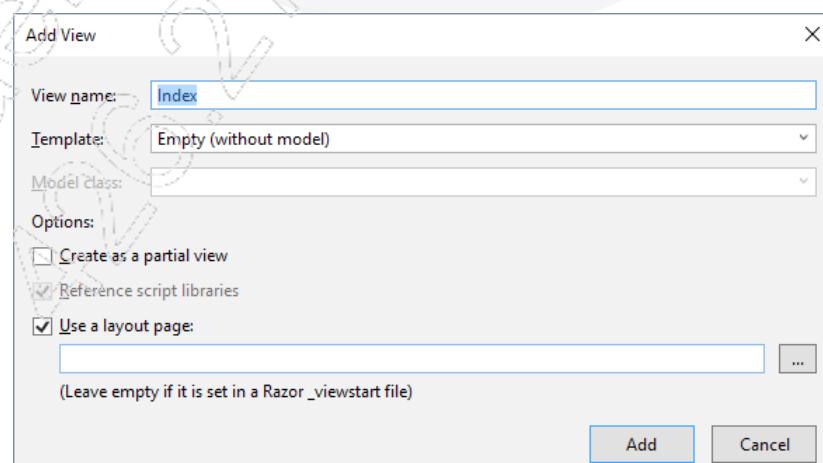
```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

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

7. Adicione um controller chamado **ProdutosController** na pasta **Areas / Loja / Controllers**:



8. Adicione uma view para o método **Index** da classe **ProdutosController**. Dentro do método **Index**, use o comando do menu de contexto **Add-View** e adicione a view vazia:



- **/Areas/Loja/Views/Produtos/index**

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

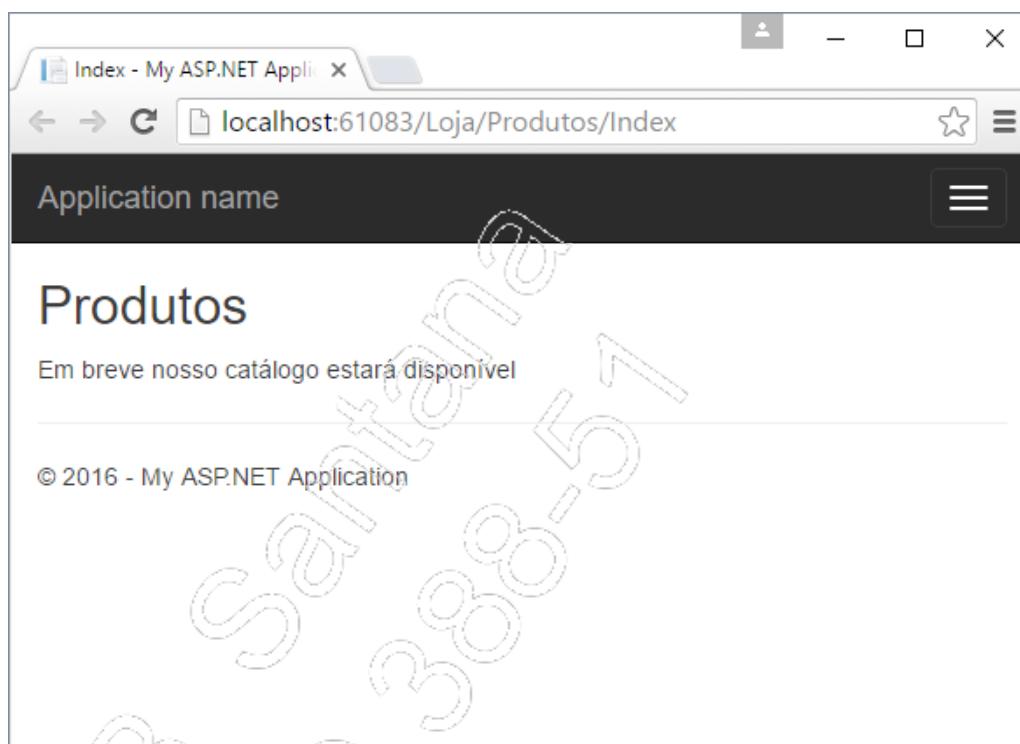

## Produtos



Em breve nosso catálogo estará disponível


```

9. Teste a URL **http://servidor/loja/produtos;**



10. Dentro página **/View/Shared/_Layout.cshtml**, altere o navegador:

```
<!-- Navegação -->  
<div class="navbar navbar-inverse navbar-fixed-top">  
  
    <!-- Container para o menu -->  
<div class="container">  
  
        <!-- Link no canto e botão -->  
<div class="navbar-header">
```

```
<!-- botão em uma tela pequena -->
<button type="button" class="navbar-toggle"
data-toggle="collapse"
data-target=".navbar-collapse">

    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
</button>

<!-- Link do canto superior esquerdo -->
@Html.ActionLink("Empresa ABC", "Index", "Home",
    new { area = "" },
    new { @class = "navbar-brand" })
</div>
<!-- Menu -->
<div class="navbar-collapse collapse">

    <!-- lista de itens do menu-->
    <ul class="nav navbar-nav">
        <li>
            @Html.ActionLink("Inicio", "Index", "Home")
        </li>
        <li>
            @Html.ActionLink("Quem Somos", "QuemSomos")
        </li>
        <li>
            @Html.ActionLink("Produtos", "Index",
                "Produtos", new { Area = "Loja" }, new { })
        </li>

        <li>
            @Html.ActionLink("Entre em Contato", "Contato")
        </li>
    </ul>

```

```
</div>

</div> <!-- final do container -->

</div><!-- final do menu de navegação -->
```

11. Altere a página **/Views/Home/Index.cshtml** para exibir dados da empresa:

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

<!-- Quadro Principal -->


<!-- Título -->
    <h1>Empresa ABC</h1>

    <!-- Subtítulo -->
    <p class="lead">A Empresa ABC tem como compromisso
        pesquisar novas tecnologias que
        facilitem o desenvolvimento de aplicações WEB
    </p>

    <!-- Saiba Mais -->
    <p>
        <a href="QuemSomos"
            class="btn btn-primary btn-lg">Saiba mais&raquo;
        </a>
    </p>


</div>
```

12. Execute e veja a página inicial:



13. Exclua as actions **About** e **Contact** do controller **Home**:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

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

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

14. No lugar dessas actions, crie dois métodos: **QuemSomos** e **Contato**:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    public ActionResult QuemSomos()
    {
        return View();
    }

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

15. Crie a view **QuemSomos**. Para isso, use o menu de contexto dentro no método **Quem Somos** e escolha o layout **empty**. Use texto HTML à vontade. A seguir, é apresentada apenas uma sugestão:

```
<h2> Empresa ABC</h2>

<h3> Indo onde ninguém jamais esteve...</h3>

<p>A empresa ABC tem como missão pesquisar e descobrir novas tecnologias que tornem a experiência do programador que cria aplicações Web mais interessante e instigante. </p>

<p> O número de bibliotecas, frameworks, padrões, ferramentas, protocolos e modelos disponíveis para o desenvolvimento de aplicações aumenta a cada ano. </p>
```

<p>Nosso trabalho é testar e validar cada ferramenta que surge no mercado. Cada uma é analisada quanto à sua relevância, facilidade de uso, integração com padrões existentes e possibilidade de extensão e adaptação. </p>

<p>Solicite a visita de um representante de nossa empresa sem compromisso, em uma visita real ou em vídeo conferência.</p>

16. Visualize a página:



17. Na página de contato, o usuário vai preencher um formulário. Para isso, vamos usar um modelo. Dentro da pasta **Models** (da raiz) crie a classe **ContatoViewModel**:

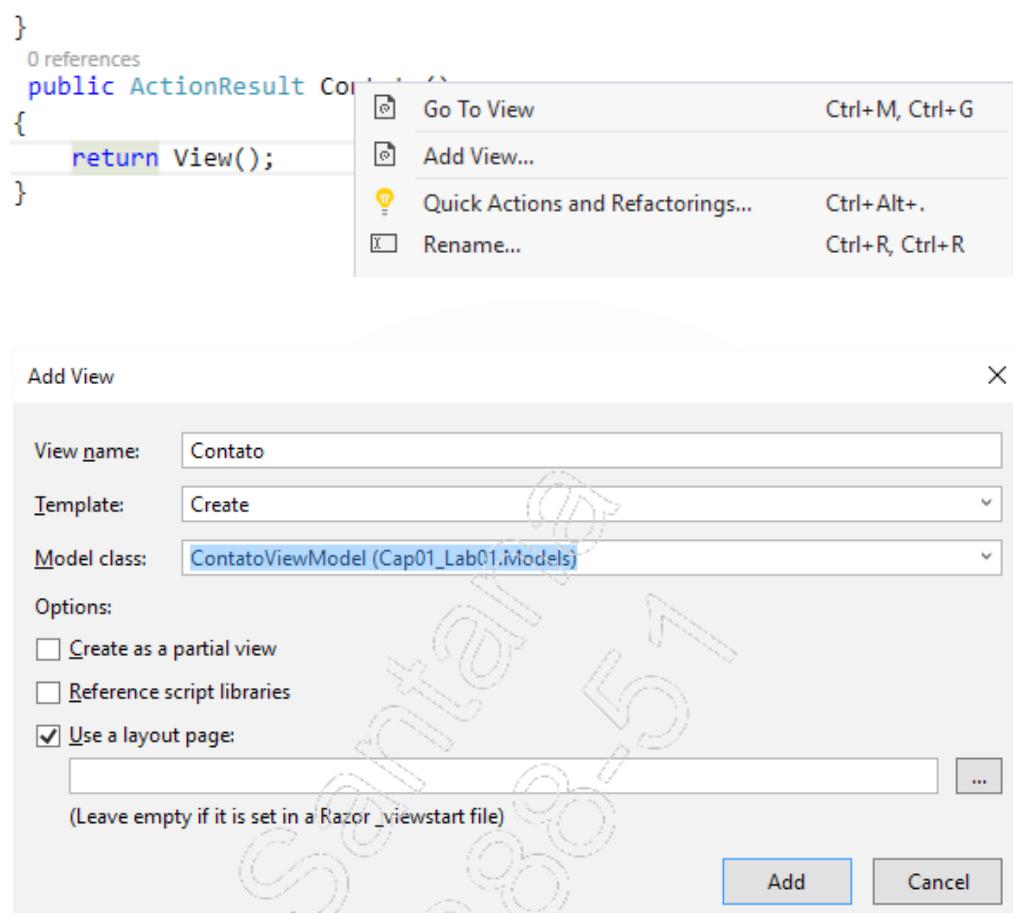
```
public class ContatoViewModel
{
    public string Nome { get; set; }
    public string Email { get; set; }
    public string Assunto { get; set; }
    public string Mensagem { get; set; }
}
```

18. Para gravar os dados, crie uma classe estática dentro da pasta **Controllers** (da raiz). A classe vai se chamar **RotinasWeb**:

```
public static class RotinasWeb
{
    public static void ContatoGravar(
        ContatoViewModel contato)
    {
        string arquivo = HttpContext
            .Current
            .Server
            .MapPath(
                "~/App_Data/Contatos.txt");

        using (var sw = new StreamWriter(
            arquivo, true, Encoding.UTF8))
        {
            sw.WriteLine(DateTime.Now);
            sw.WriteLine(contato.Nome);
            sw.WriteLine(contato.Email);
            sw.WriteLine(contato.Assunto);
            sw.WriteLine(contato.Mensagem);
            sw.WriteLine(new string('-', 30));
        }
    }
}
```

19. Crie a view **Contato**, usando o assistente para gerar um formulário automático. Para isso, abra o menu de contexto no método **Contato** da classe **HomeController** e escolha **Add View**. Escolha o template **Create** e a classe **ContatoViewModel** como modelo. Clique em **Add** para criar a view;



20. Pouca alteração é necessária no código criado pelo assistente. Apenas o subtítulo, o texto do botão e o link de retorno:

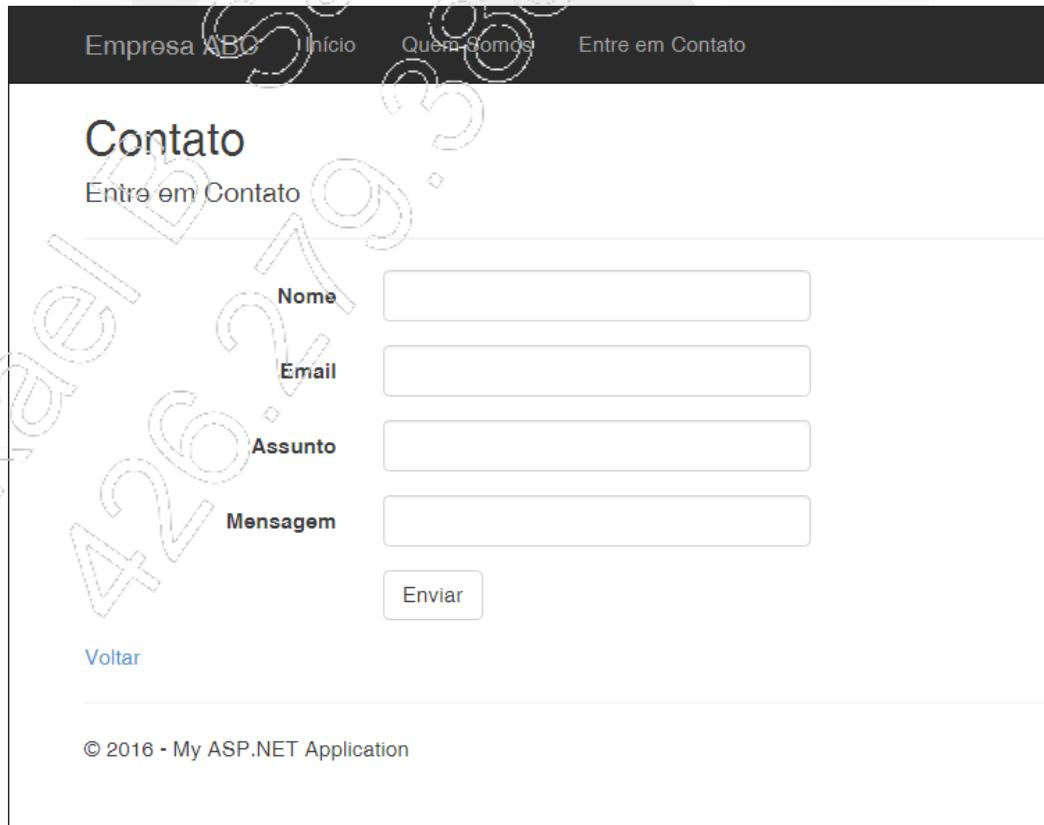
```
@model Cap01_Lab01.Models.ContatoViewModel  
...  
<h2>Contato</h2>  
@using (Html.BeginForm())  
{  
    @Html.AntiForgeryToken()  
  
    <div class="form-horizontal">  
        <h4>Entre em Contato</h4>
```

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

.... Campos do Formulário (Label, Editor e ValidationMessage)

```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Enviar" class="btn btn-default" />
    </div>
    ....
<div>
    @Html.ActionLink("Voltar", "Index")
</div>
```

21. Execute o programa para testar:



22. Crie o método que vai receber o formulário, na classe **ContatoController**:

```
[HttpPost]
public ActionResult Contato(ContatoViewModel contato)
{
    if (string.IsNullOrEmpty(contato.Nome))
    {
        ModelState.AddModelError("", "O nome deve ser
preechido");
    }

    if (ModelState.IsValid)
    {
        RotinasWeb.ContatoGravar(contato);
        return View("ContatoGravarOK");
    }
    else
    {
        return View(contato);
    }
}
```

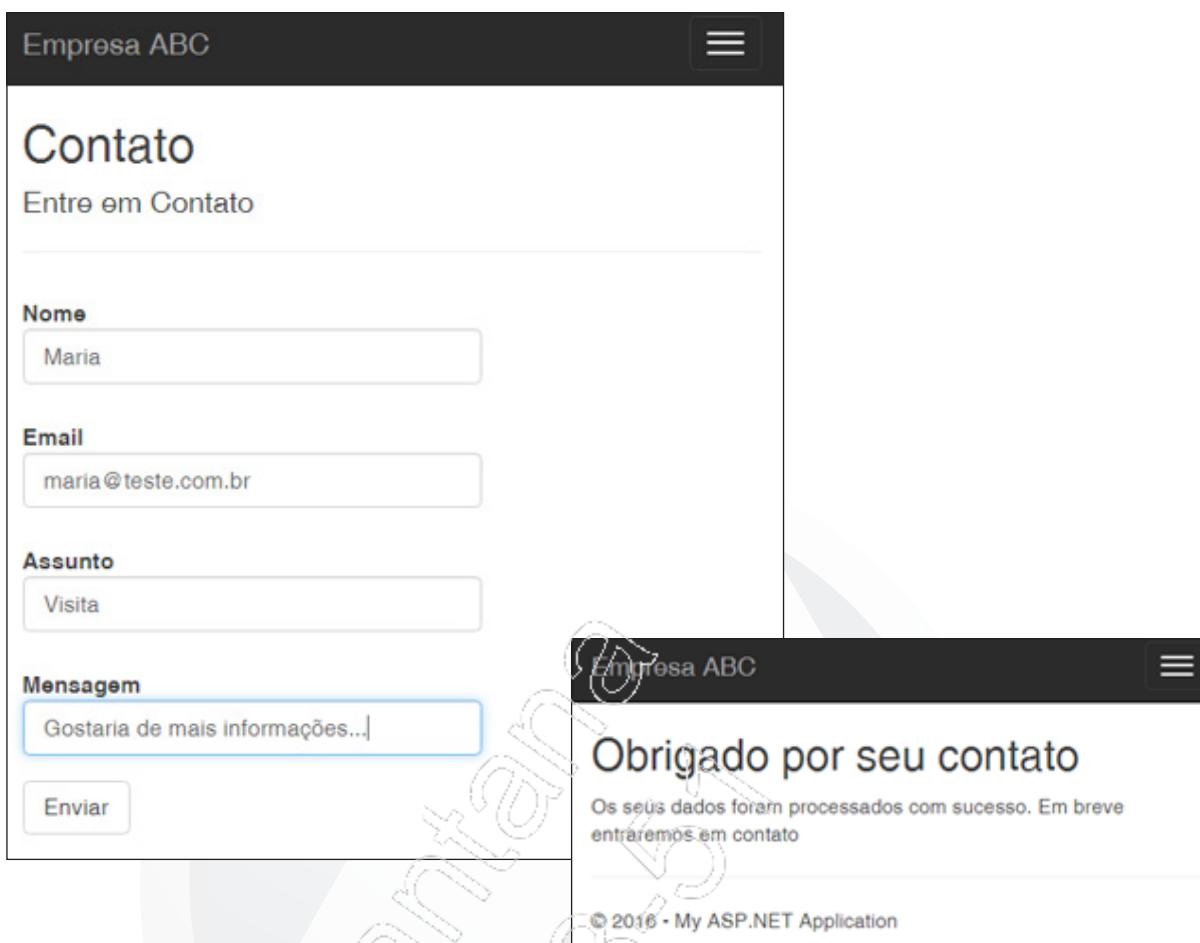
23. Crie uma view chamada **ContatoGravarOK**:

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

<h2>Obrigado por seu contato</h2>

<p>
Os seus dados foram processados com sucesso.
Em breve entraremos em contato
</p>
```

24. Teste a inclusão:



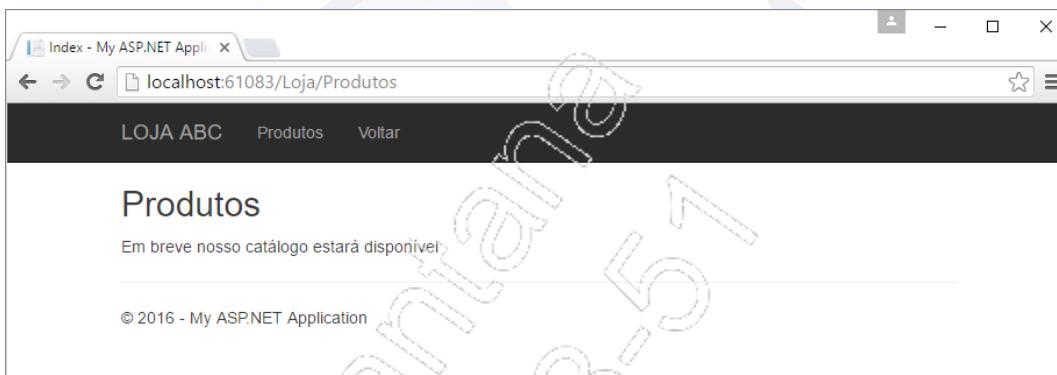
25. Altere a view /Areas/Loja/Shared/_Layout.cshtml para refletir o menu da loja e visualize:

```
<div class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle">
                ...
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            @Html.ActionLink("LOJA ABC", "Index", "Home",
                new { area = "Loja" },
                new { @class = "navbar-brand" })
        </div>
    </div>
```

```
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">

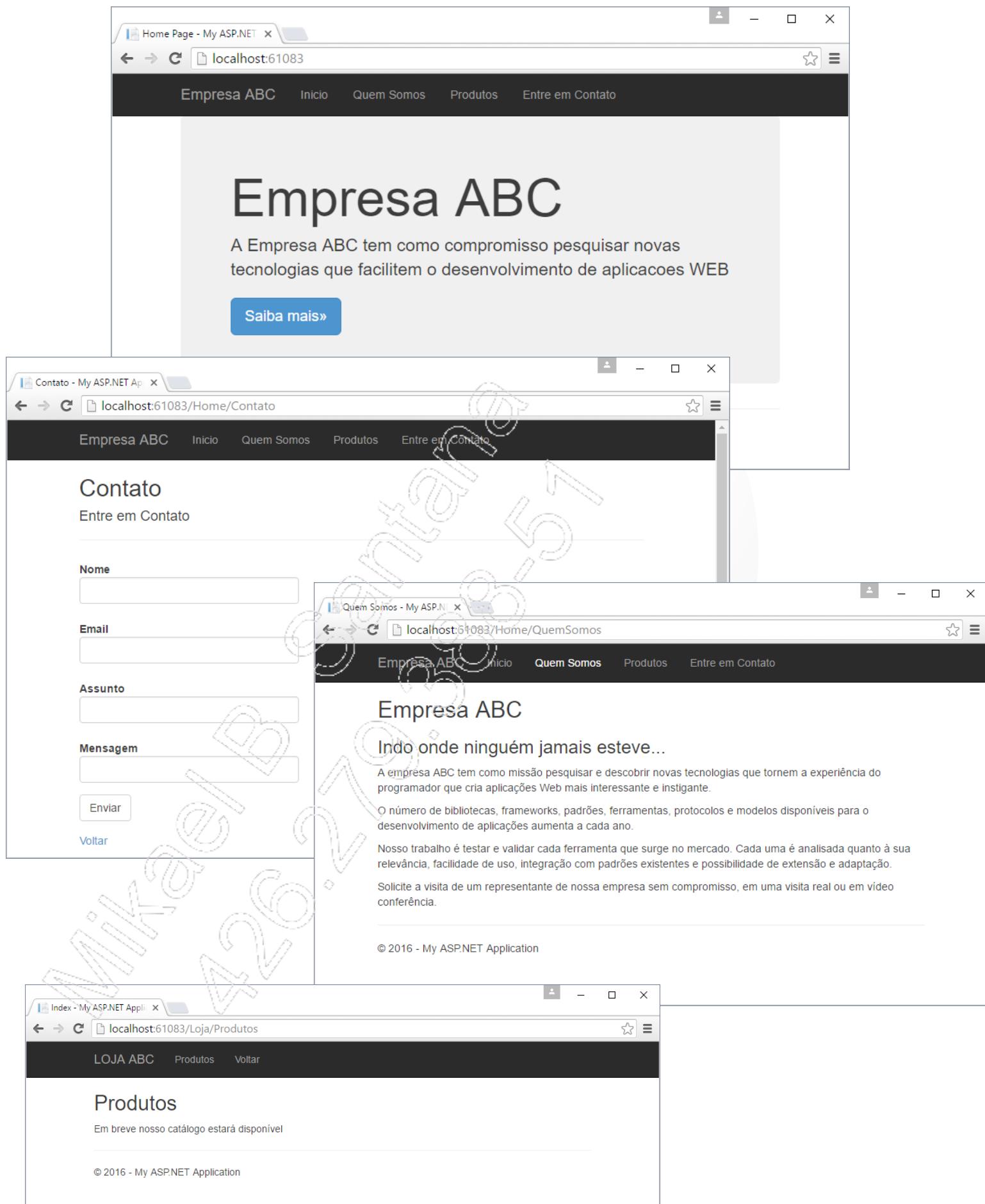
        <li>@Html.ActionLink("Produtos", "Index", "Produtos") </li>

        <li>@Html.ActionLink("Voltar", "Index", "Home",
            new { area = "" }, new { }) </li>
    </ul>
</div>
</div>
```



Visual Studio 2015 - ASP.NET com C# Recursos Avançados

26. Teste a aplicação completa:



2

ASP.NET Core

- ✓ .NET Core;
- ✓ Ferramentas de desenvolvimento;
- ✓ Aplicação Console;
- ✓ Aplicação App Web;
- ✓ Fluxo de execução.



IMPACTA
EDITORA

2.1. Introdução

Até o último instante, a versão que seria a sucessora da versão 4.6 do ASP.NET estava sendo chamada de ASP.NET 5. Poucos dias antes do lançamento oficial, a Microsoft anuncia que a nova versão terá o nome de ASP.NET Core 1.0. Essa mudança reflete exatamente o que houve com a nova versão da plataforma: uma reformulação geral, partindo para um modelo completamente novo.

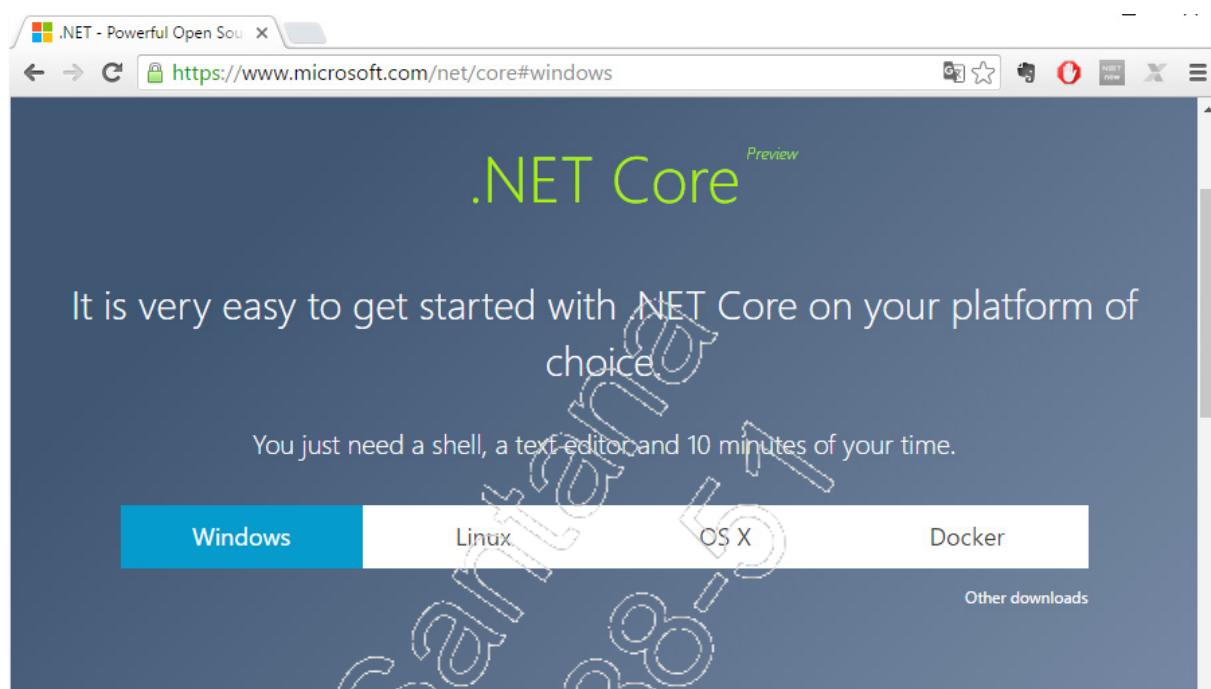
A nova plataforma .NET apresenta três características principais:

1. Arquitetura modular;
2. Multiplataforma;
3. Open source.

Essa nova direção abre muitas possibilidades de desenvolvimento e integração de sistemas. Até a versão 4.6, os recursos do ASP.NET e do .NET como um todo estavam limitados a servidores ou computadores desktop utilizando sistema operacional Windows. A partir da versão Core 1.0, qualquer sistema operacional, biblioteca, ferramenta de programação ou framework pode ser utilizado para desenvolver, executar ou hospedar aplicativos .NET. Na prática, isso significa que agora é possível criar um aplicativo para dispositivos Android, iPhone ou para um computador executando Linux. Além disso, é possível distribuir o ASP.NET como parte da aplicação, criando servidores virtuais e serviços de gerenciamento de informações da Internet que são distribuídos junto com o aplicativo.

2.2..NET Core

A plataforma .NET Core fornece os componentes necessários para executar programas em diversos sistemas operacionais. A página oficial fornece os links para a instalação do kit de desenvolvimento (SDK). Para os usuários Windows, é preferível utilizar o instalador do Visual Studio, que já faz o download e configura o ambiente de desenvolvimento automaticamente.



A instalação do SDK é muito rápida e instala as bibliotecas necessárias para a execução de aplicativos .NET:



Visual Studio 2015 - ASP.NET com C# Recursos Avançados

Uma vez instalado o SDK, é possível, usando o prompt de comando, criar e executar um aplicativo .NET Core. O exemplo adiante está descrito na página de download e serve de teste das bibliotecas. Neste caso, é o passo a passo para criar um pequeno aplicativo no Windows:

1. Usando o prompt de comando (Console): Crie uma pasta para o projeto;

```
>md \TesteNet + ENTER
```



```
C:\>md \TesteNet
```

2. Navegue até a pasta criada:

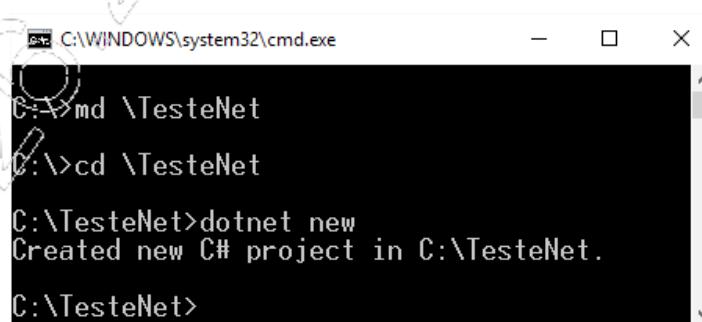
```
>cd \TesteNet + ENTER
```



```
C:\>md \TesteNet
C:\>cd \TesteNet
C:\TesteNet>
```

3. Crie um projeto de exemplo:

```
>dotnet new + ENTER
```

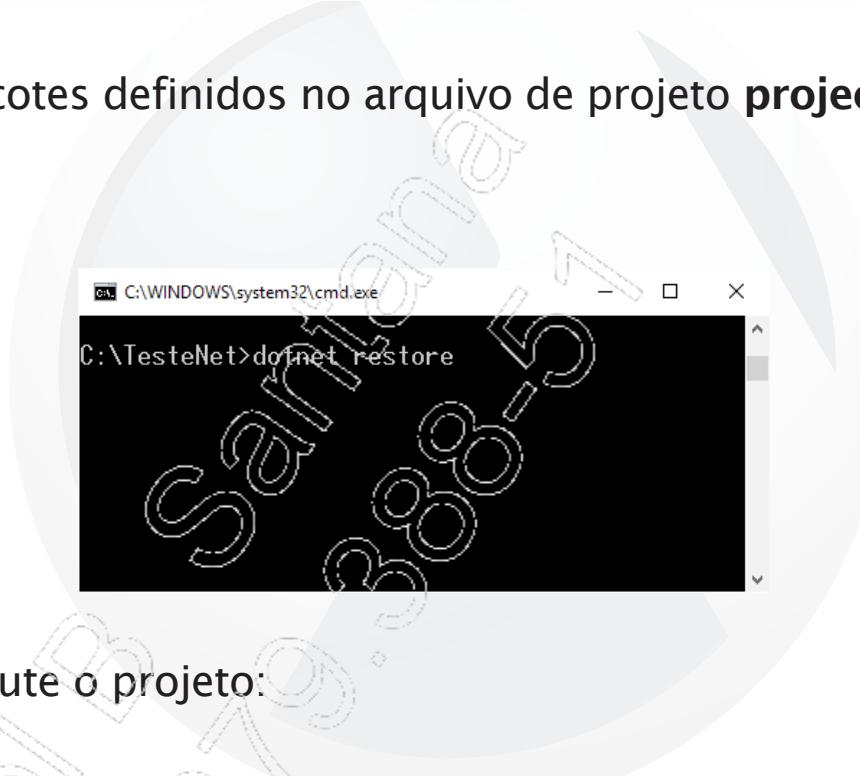


```
C:\>md \TesteNet
C:\>cd \TesteNet
C:\TesteNet>dotnet new
Created new C# project in C:\TesteNet.

C:\TesteNet>
```

4. Veja os arquivos criados (**Program.cs** e **project.json**);

>dir + ENTER



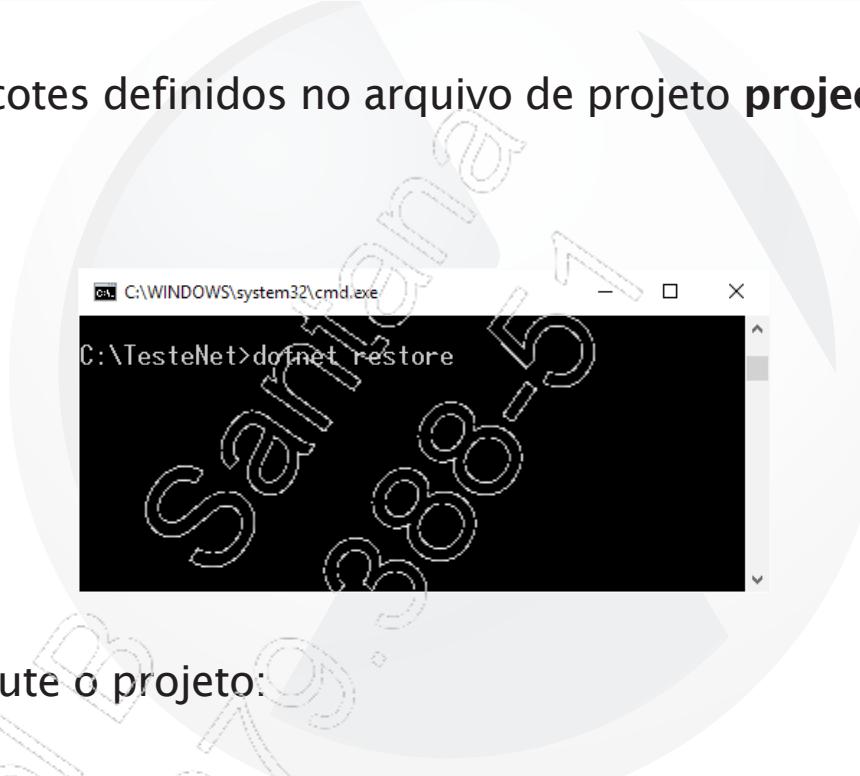
```
C:\TesteNet>dir
 0 volume na unidade C é OS
 0 Número de Série do Volume é B0D9-D549

  Pasta de C:\TesteNet
21/06/16 13:15    <DIR>    .
21/06/16 13:15    <DIR>    ..
21/06/16 13:15           214 Program.cs
21/06/16 13:15           302 project.json
                           2 arquivo(s)   516 bytes
                           2 pasta(s)  590.103.175.168 bytes disponíveis

C:\TesteNet>
```

5. Restaure os pacotes definidos no arquivo de projeto **project.json**:

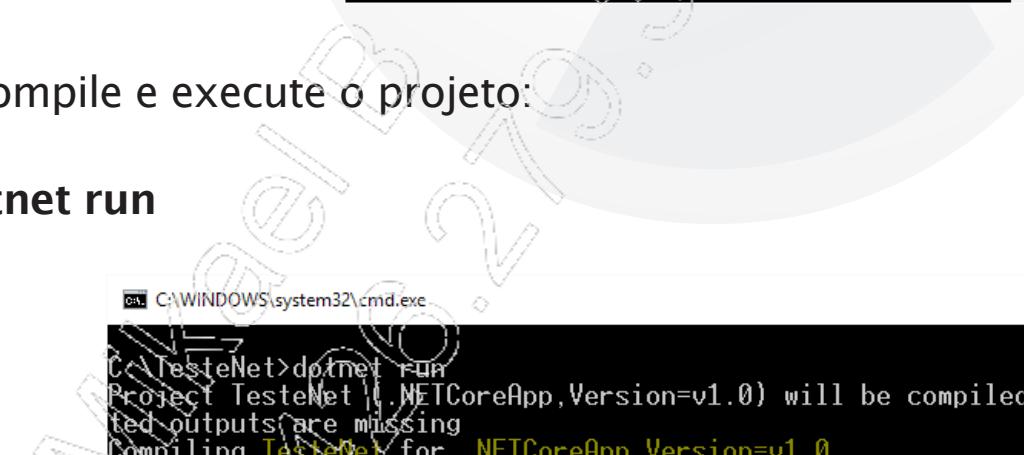
>dotnet restore



```
C:\TesteNet>dotnet restore
```

6. Compile e execute o projeto:

>dotnet run

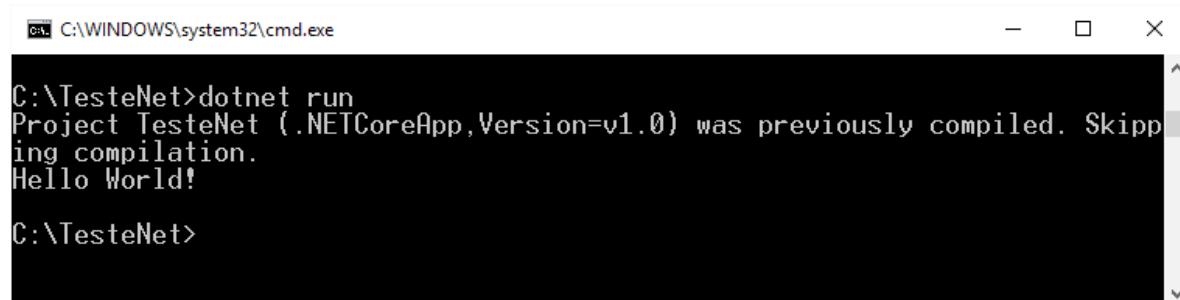


```
C:\TesteNet>dotnet run
Project TesteNet (.NETCoreApp,Version=v1.0) will be compiled because expected outputs are missing
Compiling TesteNet for .NETCoreApp,Version=v1.0
Compilation succeeded.
  0 Warning(s)
  0 Error(s)

Time elapsed 00:00:03.2203469

Hello World!
C:\TesteNet>
```

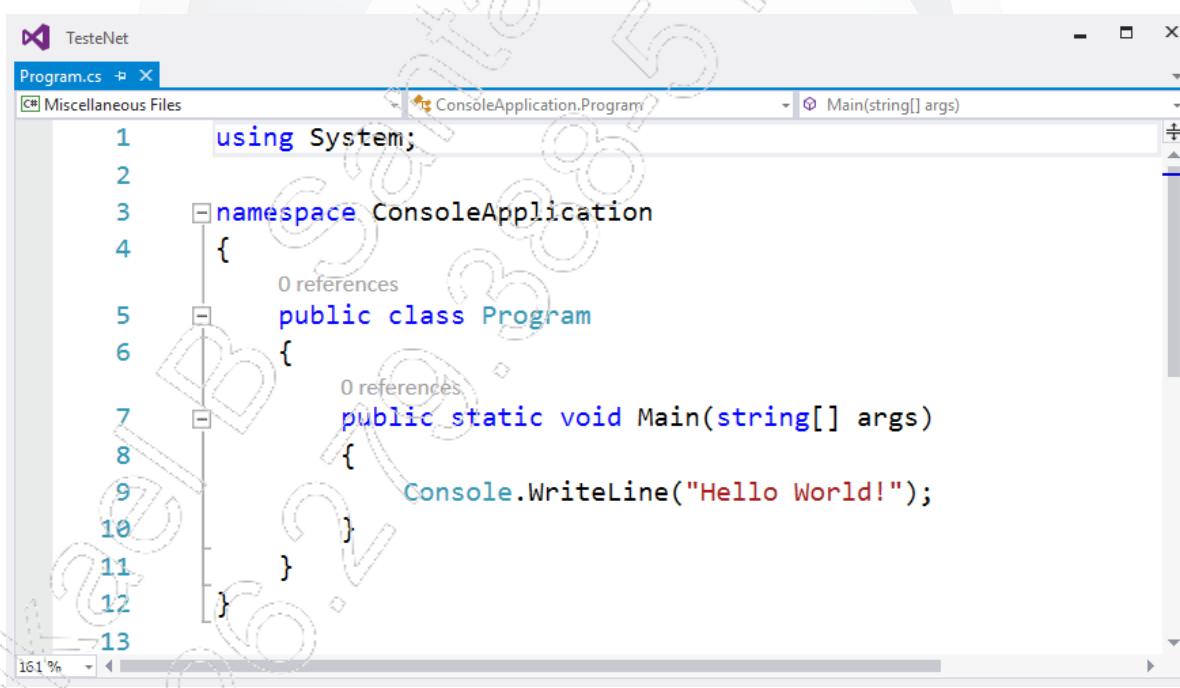
7. Ao pedir para executar novamente, o .NET Core não precisa compilar novamente:



```
C:\WINDOWS\system32\cmd.exe
C:\TesteNet>dotnet run
Project TesteNet (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
Hello World!
C:\TesteNet>
```

Os arquivos criados pelo comando **dotnet new** são os seguintes:

- **Program.cs**: Este arquivo não tem nada de novo em relação a uma aplicação Console da versão 4.6 ou inferior do .Net Framework;



```
TesteNet
Program.cs  ✎
Miscellaneous Files  ConsoleApplication.Program
1  using System;
2
3  namespace ConsoleApplication
4  {
5      public class Program
6      {
7          public static void Main(string[] args)
8          {
9              Console.WriteLine("Hello World!");
10         }
11     }
12 }
13
```

- **project.json**: Este arquivo de projeto é completamente novo. O formato é JSON, e não XML. A extensão não é **.csproj** ou **.vbproj** como nos projetos tradicionais. Apenas três características do projeto são definidas: versão, dependências e frameworks. Os detalhes definidos neste arquivo serão vistos adiante. O importante agora é ficar familiarizado com os arquivos que compõem um projeto.



```
project.json
Schema: http://json.schemastore.org/tslint
1 {  
2   "version": "1.0.0-*",  
3   "buildOptions": {  
4     "emitEntryPoint": true  
5   },  
6   "dependencies": {  
7     "Microsoft.NETCore.App": {  
8       "type": "platform",  
9       "version": "1.0.0-rc2-3002702"  
10    }  
11  },  
12  "frameworks": {  
13    "netcoreapp1.0": {  
14      "imports": "dnxcore50"  
15    }  
16  }  
17}  
18
```

2.3. Ferramentas de desenvolvimento

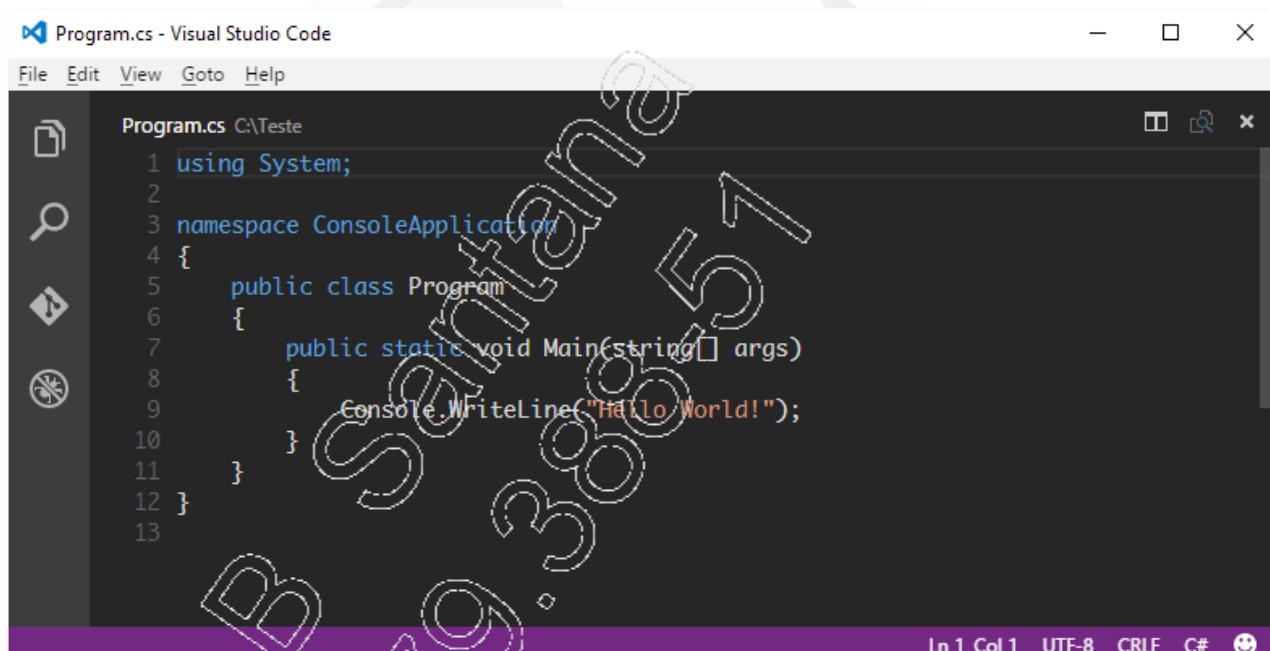
A versão inicial do .NET Core permite criar aplicações Console ou Web. A versão 4.6 e inferiores do .Net Framework permitem criar diversos tipos de aplicações, como Windows Forms, Console, Web Services, WCF e WPF, entre outros. Existe uma comunidade de programadores, analistas e engenheiros desenvolvendo modelos de programação para o .NET Core e a tendência é termos todos os tipos de modelos usados em todas as plataformas.

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

Quanto mais complexa é a tecnologia de um aplicativo, mais necessário é ter em mãos uma boa ferramenta de desenvolvimento. É possível desenvolver aplicativos apenas com um editor de texto. O uso, porém, de uma ferramenta de desenvolvimento adequada aumenta em muito a produtividade. As principais IDEs (Integrated Development Environment – ambiente integrado de desenvolvimento) são Visual Studio, Visual Code e Sublime Text.

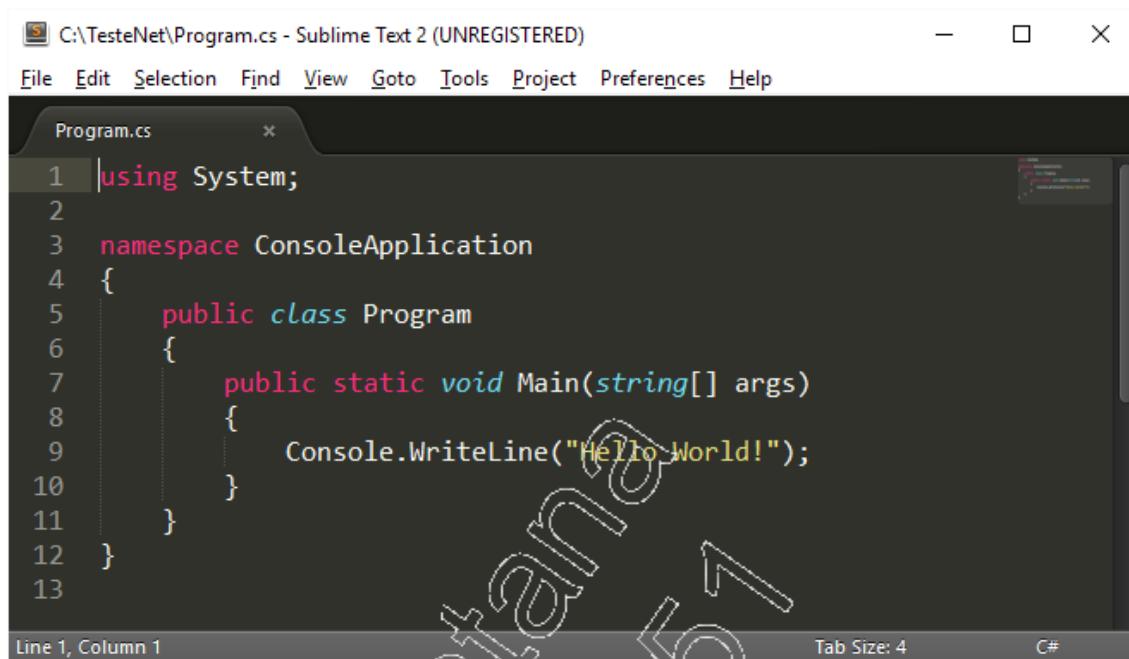
- **Visual Code**

Ambiente multiplataforma com suporte para diversos tipos de arquivos e linguagens de programação.



- **Sublime Text**

No mesmo estilo do Visual Code, o Sublime Text é uma opção simples, mas que trabalha muito bem diversos tipos de arquivos.



A screenshot of the Sublime Text 2 interface. The window title is "C:\TesteNet\Program.cs - Sublime Text 2 (UNREGISTERED)". The menu bar includes File, Edit, Selection, Find, View, Goto, Tools, Project, Preferences, and Help. A tab labeled "Program.cs" is open, showing the following C# code:

```
1 using System;
2
3 namespace ConsoleApplication
4 {
5     public class Program
6     {
7         public static void Main(string[] args)
8         {
9             Console.WriteLine("Hello World!");
10        }
11    }
12 }
13
```

The status bar at the bottom shows "Line 1, Column 1", "Tab Size: 4", and "C#".

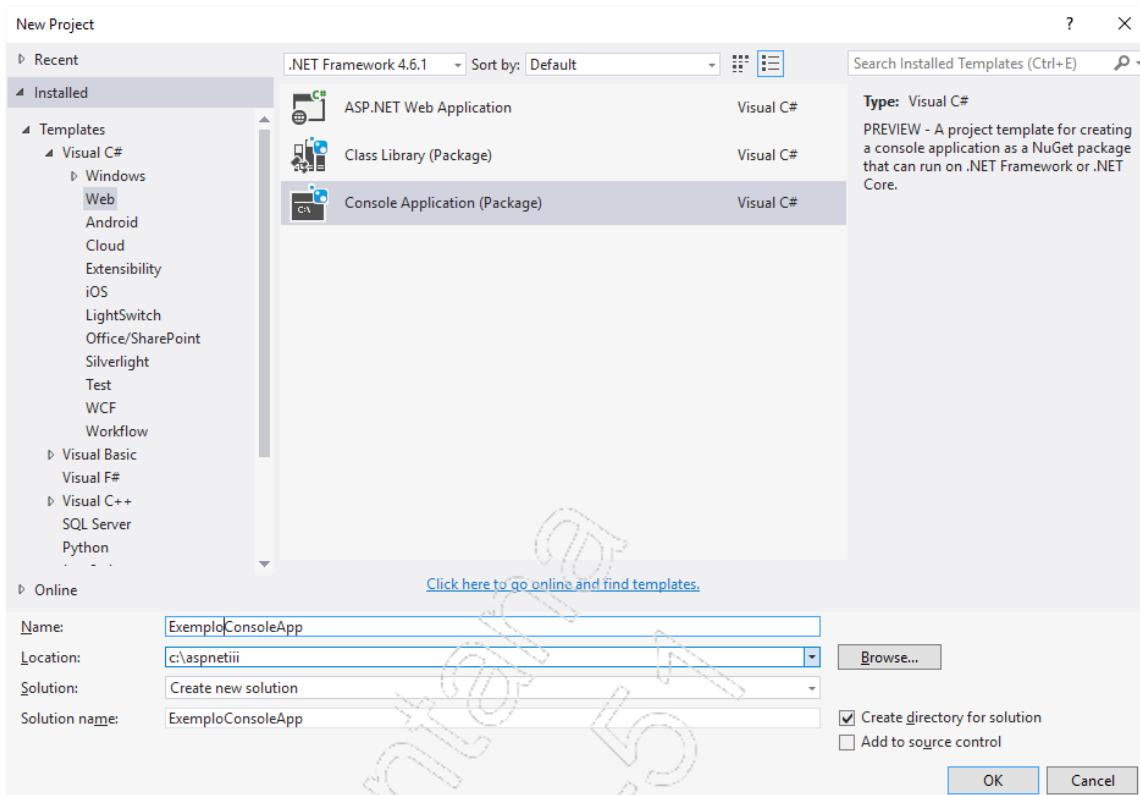
Um dos recursos interessantes do Sublime e que depois foi copiado para outros editores é o modo sem distração (**Distraction Free**): acionado com F11, remove tudo da tela, para o programador se concentrar apenas no código que está escrevendo.



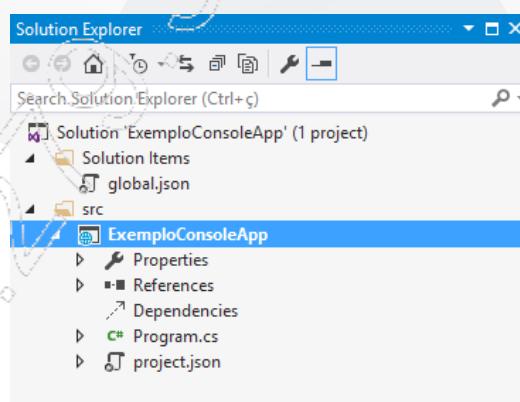
- **Visual Studio**

A ferramenta oficial da plataforma .NET. Para iniciar um projeto do tipo Console usando o ASP.NET Code, é necessário utilizar o comando de menu **File / New Project / Web** e escolher a opção **Console Application (Package)**.

2.4. Aplicação Console



O projeto cria a seguinte estrutura:



O arquivo **project.json** contém os metadados do projeto, assim como referências, versões e dependências.

```
{  
  "version": "1.0.0-*",  
  "description": "ExemploConsoleApp Console Application",  
  "authors": [ "Grasso" ],  
  "tags": [ "" ],  
  "projectUrl": "",  
  "licenseUrl": "",  
  
  "compilationOptions": {  
    "emitEntryPoint": true  
  },  
  
  "dependencies": {  
  },  
  
  "commands": {  
    "ExemploConsoleApp": "ExemploConsoleApp"  
  },  
  
  "frameworks": {  
    "dnx451": { },  
    "dnxcore50": {  
      "dependencies": {  
        "Microsoft.CSharp": "4.0.1-beta-23516",  
        "System.Collections": "4.0.11-beta-23516",  
        "System.Console": "4.0.0-beta-23516",  
        "System.Linq": "4.0.1-beta-23516",  
        "System.Threading": "4.0.11-beta-23516"  
      }  
    }  
  }  
}
```

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

O arquivo **Program.cs** contém o código-fonte do programa principal:

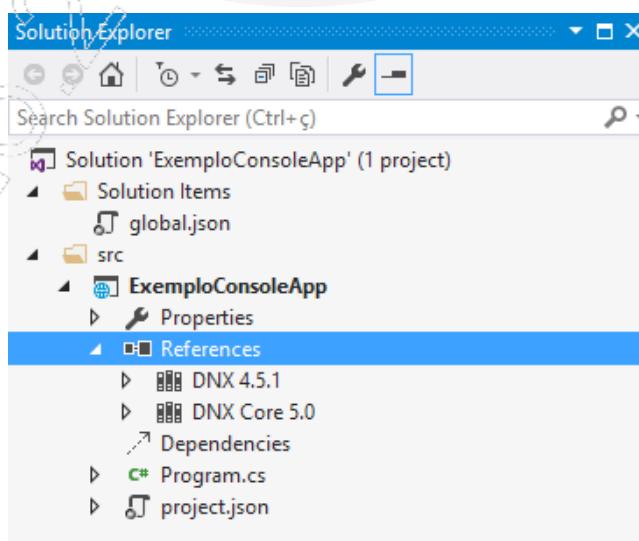
```
namespace ExemploConsoleApp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Olá para todos");
        }
    }
}
```

O arquivo **global.json** define o nome dos projetos e a versão do aplicativo:

```
{
    "projects": [ "src", "test" ],

    "sdk": {
        "version": "1.0.0-rc1-update1"
    }
}
```

Duas referências são adicionadas: **DNX 4.5.1** e **DNX Core 5.0**.

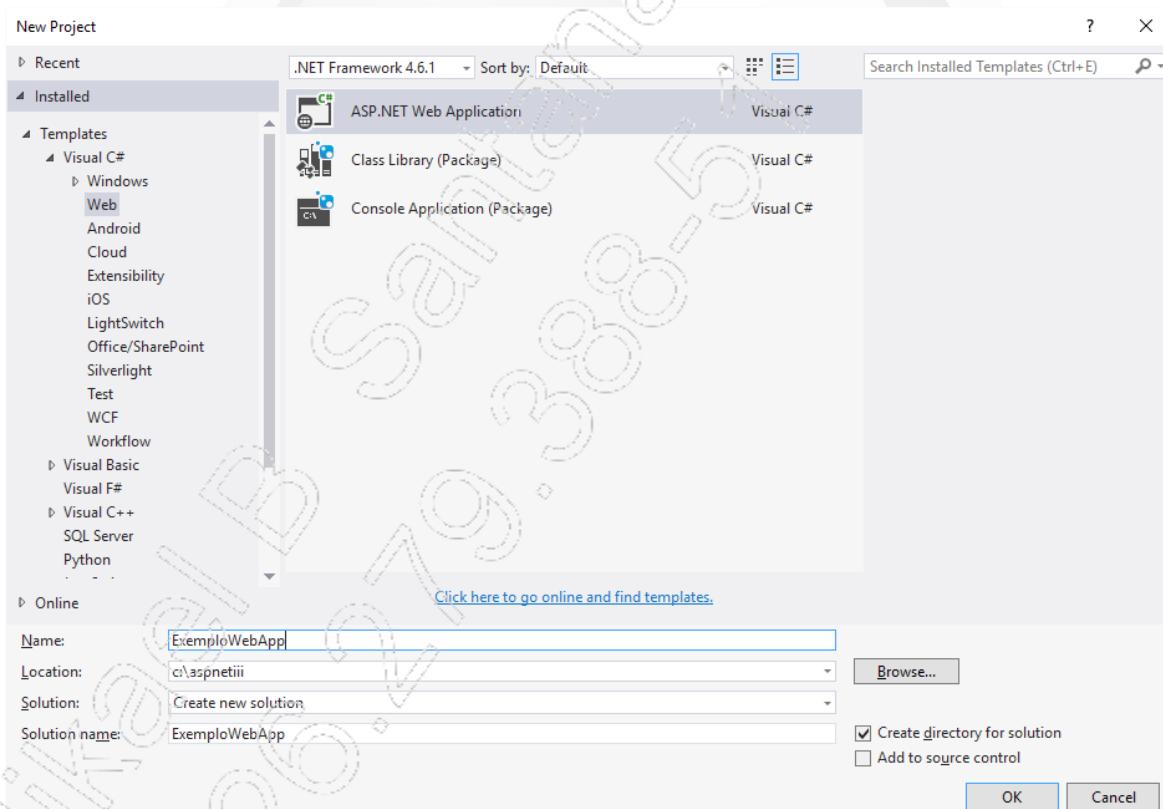


Para executar o programa, pressione CTRL + F5.

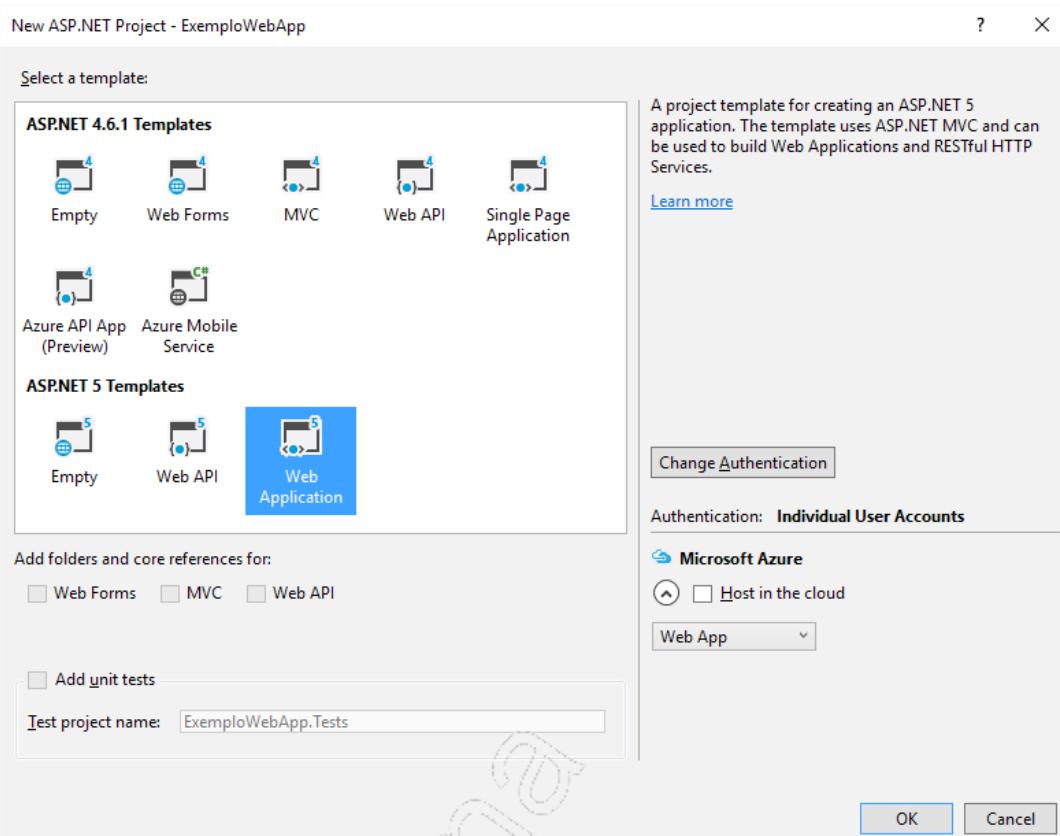


2.5. Aplicação App Web

O modelo de Web App para o .NET Core é um bom exemplo de como configurar e implementar os componentes ASP.NET Code. Para incluir uma Web App usando .NET Core, é necessário escolher **File / New Project / Web / ASP.NET Web Application**.



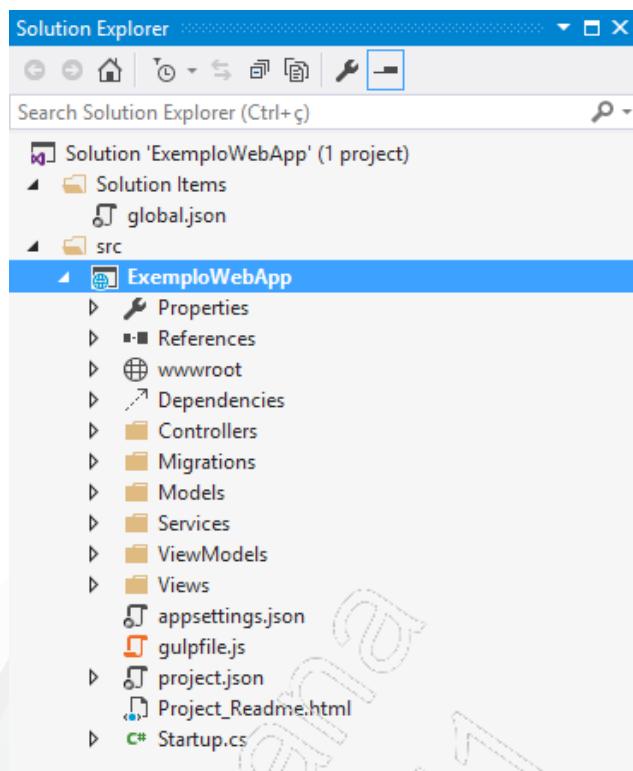
Visual Studio 2015 - ASP.NET com C# Recursos Avançados



Utilizando a janela Solution Explorer é fácil de observar que a estrutura é bem diferente da utilizada pelo ASP.NET 4.6. Algumas importantes alterações estruturais são as seguintes:

- O formato XML não é utilizado mais para arquivos de sistema. O formato agora é JSON;
- Não existe mais **Web.Config**. O arquivo de configuração agora se chama **appsettings.json**;
- O arquivo **Global.asax** foi substituído pelo arquivo **Solution Items / global.json**;

- As dependências agora são armazenadas no arquivo de projeto, **project.json**.



2.5.1. **project.json**

O item **dependencies** é o equivalente ao arquivo **packages** da versão 4.6 e nos informa quais componentes estão sendo utilizados:

```
{  
  "userSecretsId": "aspnet5-ExemploWebApp-e4d40e56-fcb8-  
40c3-bf5b-5074273674b2",  
  "version": "1.0.0-*",  
  "compilationOptions": {  
    "emitEntryPoint": true  
  },  
  
  "dependencies": {  
    "EntityFramework.Commands": "7.0.0-rc1-final",  
    "EntityFramework.MicrosoftSqlServer": "7.0.0-rc1-final",  
    "Microsoft.AspNet.Authentication.Cookies": "1.0.0-rc1-  
final",  
    "Microsoft.AspNet.Diagnostics.Entity": "7.0.0-rc1-  
final",  
    "Microsoft.AspNet.Identity.EntityFramework": "3.0.0-rc1-  
final",  
  }  
}
```

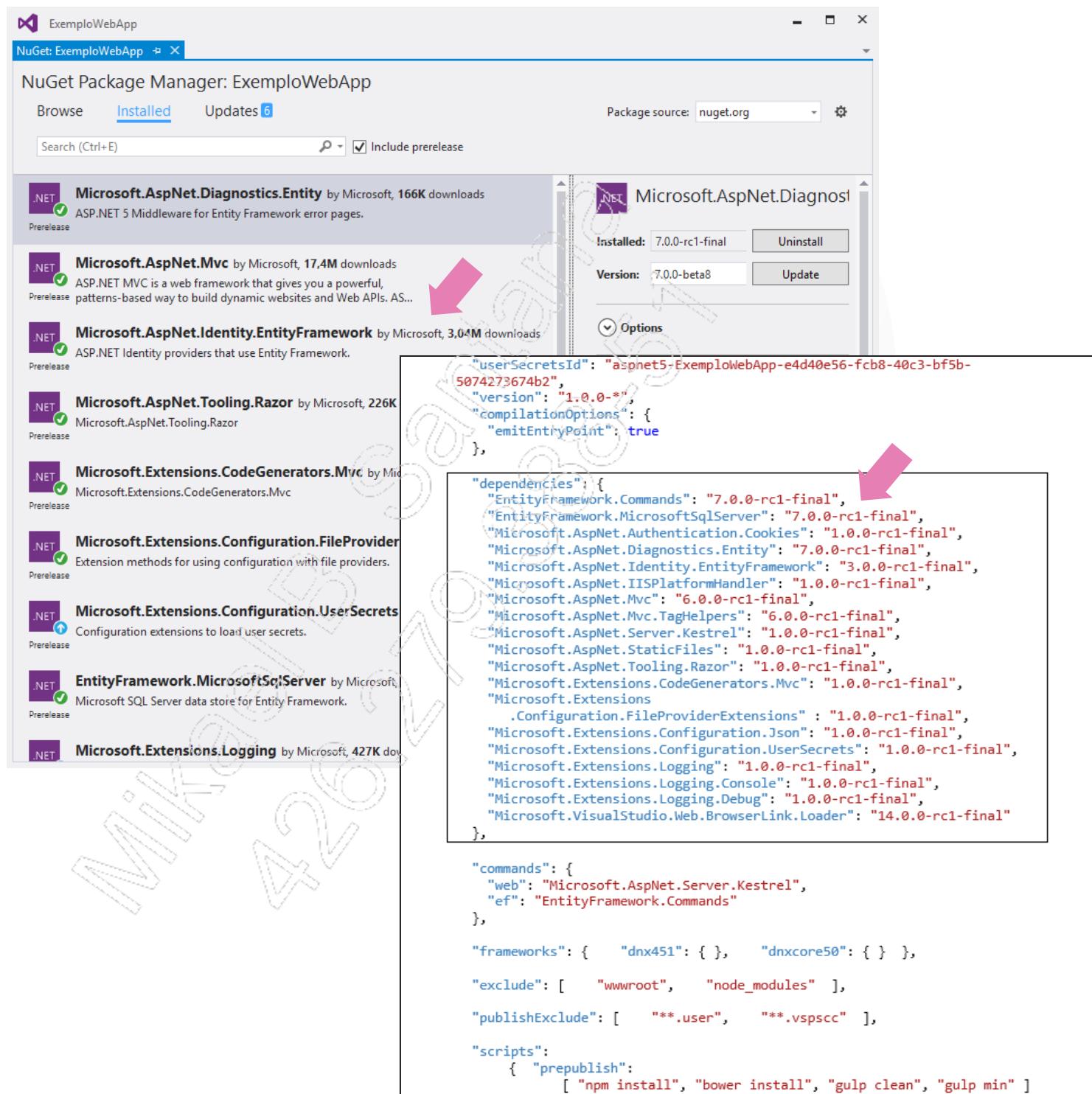
Visual Studio 2015 - ASP.NET com C# Recursos Avançados

```
        "Microsoft.AspNet.IISPlatformHandler": "1.0.0-rc1-final",
        "Microsoft.AspNet.Mvc": "6.0.0-rc1-final",
        "Microsoft.AspNet.Mvc.TagHelpers": "6.0.0-rc1-final",
        "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
        "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final",
        "Microsoft.AspNet.Tooling.Razor": "1.0.0-rc1-final",
        "Microsoft.Extensions.CodeGenerators.Mvc": "1.0.0-rc1-final",
        "Microsoft.Extensions.Configuration.FileProviderExtensions": "1.0.0-rc1-final",
        "Microsoft.Extensions.Configuration.Json": "1.0.0-rc1-final",
        "Microsoft.Extensions.Configuration.UserSecrets": "1.0.0-rc1-final",
        "Microsoft.Extensions.Logging": "1.0.0-rc1-final",
        "Microsoft.Extensions.Logging.Console": "1.0.0-rc1-final",
        "Microsoft.Extensions.Logging.Debug": "1.0.0-rc1-final",
        "Microsoft.VisualStudio.Web.BrowserLink.Loader": "14.0.0-rc1-final"
    },
}
```

```
    "commands": {
        "web": "Microsoft.AspNet.Server.Kestrel",
        "ef": "EntityFramework.Commands"
    },
    "frameworks": {
        "dnx451": { },
        "dnxcore50": { }
    },
    "exclude": [
        "wwwroot",
        "node_modules"
    ],
    "publishExclude": [
        "**.user",
        "**.vspscc"
    ],
    "scripts": {
        "prepublish": [
            "npm install",
            "bower install",
            "gulp clean",
            "gulp min"
        ]
    }
}
```

Entrando no menu de contexto da aplicação e escolhendo **Manage NuGet Packages**, percebe-se que os mesmos componentes estão listados. Isso é uma alteração estrutural muito importante: os componentes do .NET Framework são packages NuGet e, portanto, podem ser "montados" de diversas maneiras e distribuídos em bibliotecas.

No .Net Framework tradicional isso não acontece. Ou é adicionada uma referência a uma biblioteca, por exemplo, **System.Data.dll**, ou não é adicionada.

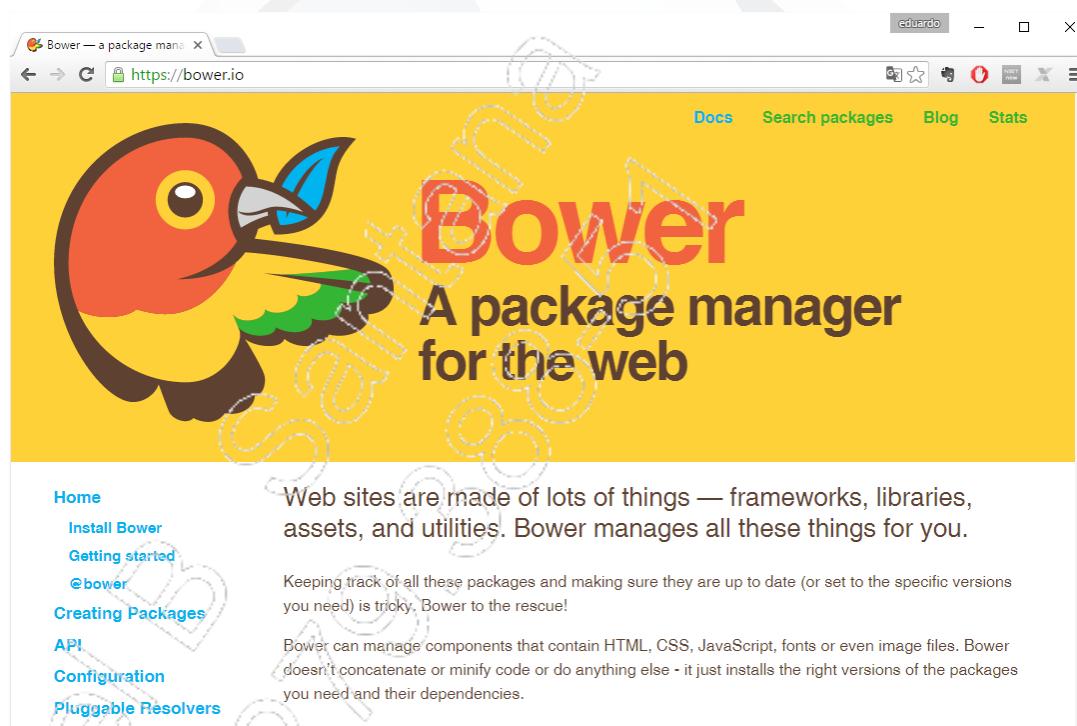


Não existe maneira de importar uma parte de uma biblioteca. Por exemplo, não há como importar um pacote com a classe **DataSet** sem que venha junto a classe **SqlConnection**. Essas classes não estariam obrigatoriamente juntas em toda aplicação, mas não tem como separá-las.

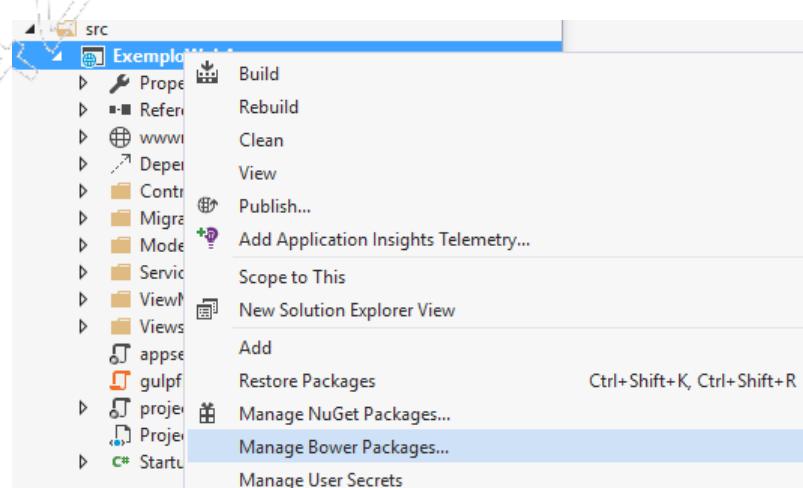
Esse tipo de problema não acontece no .NET Core. É possível montar packages NuGet usando componentes menores e mais independentes.

2.5.2. Bower

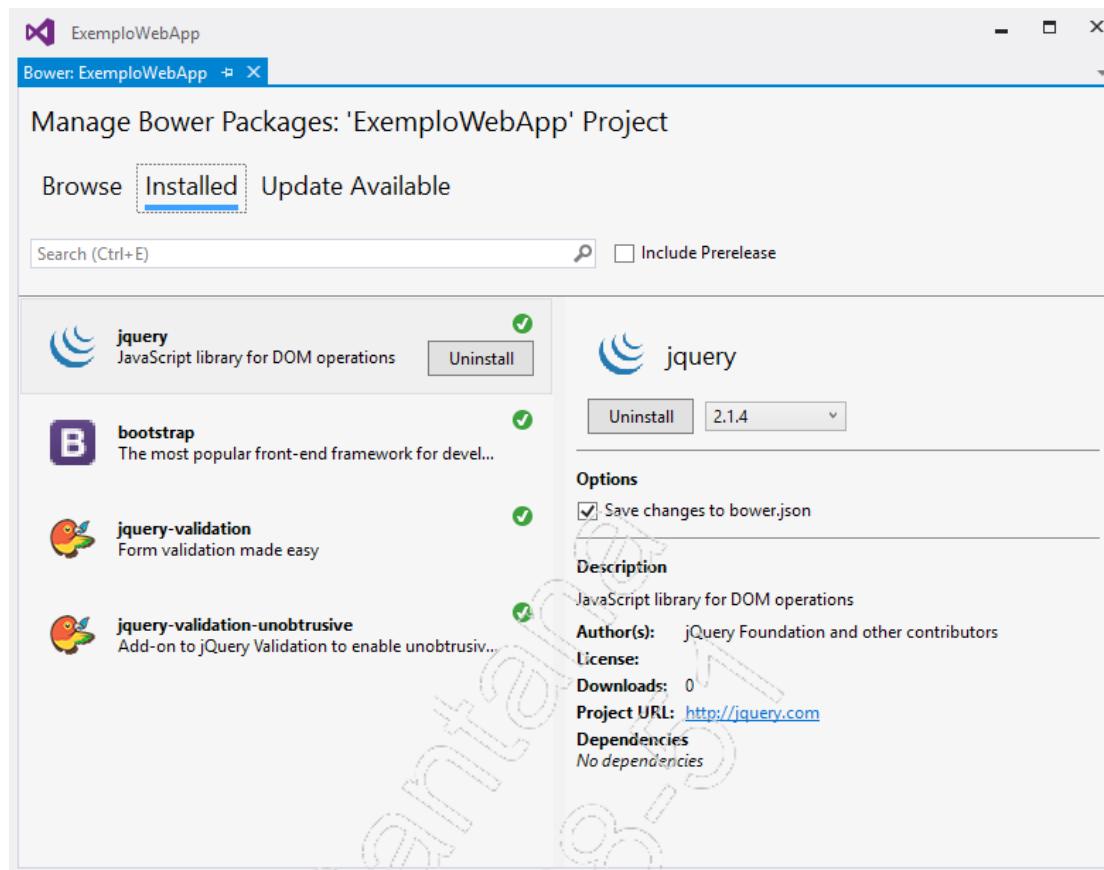
Outra novidade é o Bower, usado para administrar os componentes da aplicação.



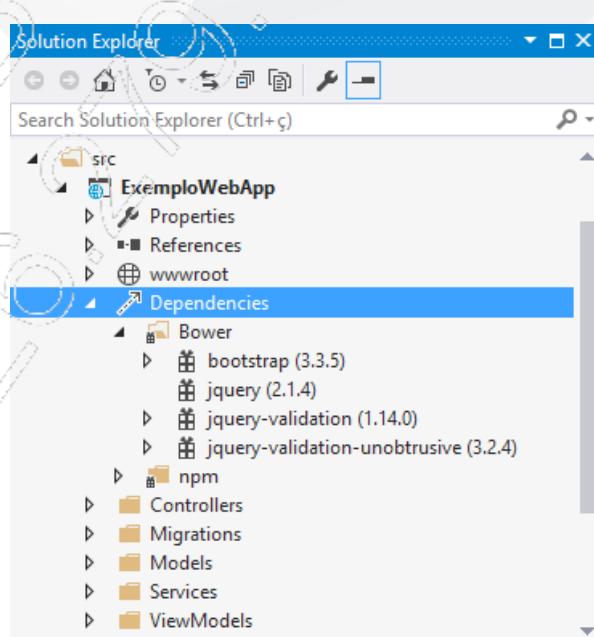
Por meio do menu de contexto, é possível entrar no configurador do componente Bower.



A interface para gerenciamento dos pacotes é igual ao NuGet. Opta-se pelo uso do componente Bower porque é um projeto open source, com uma comunidade de usuários muito ativa e utilizado em muitos projetos.

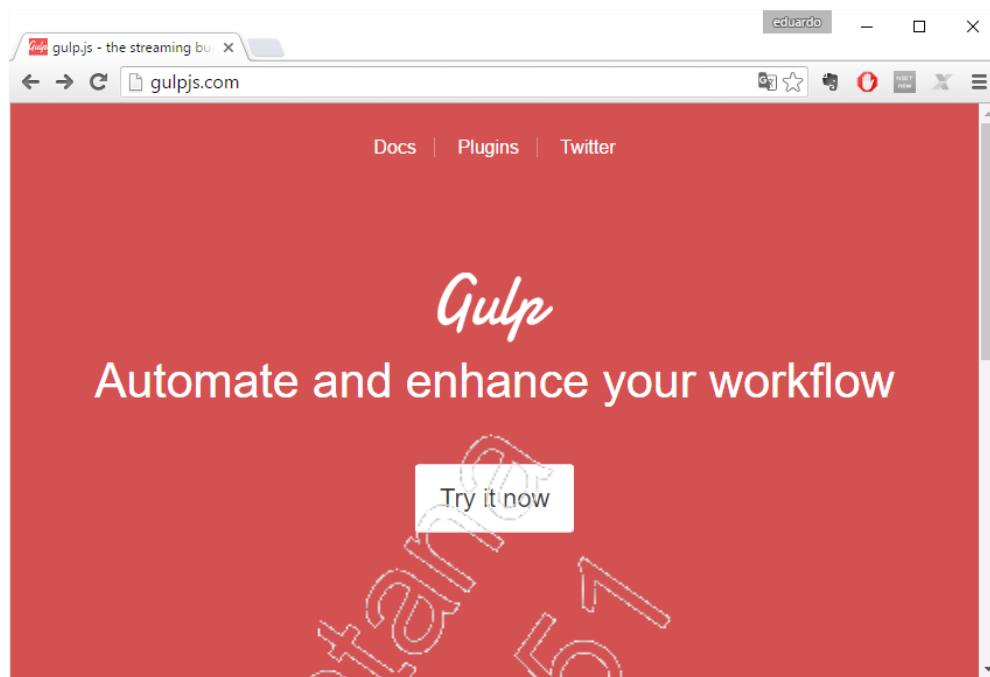


As configurações ficam armazenadas na pasta **Dependencies**:



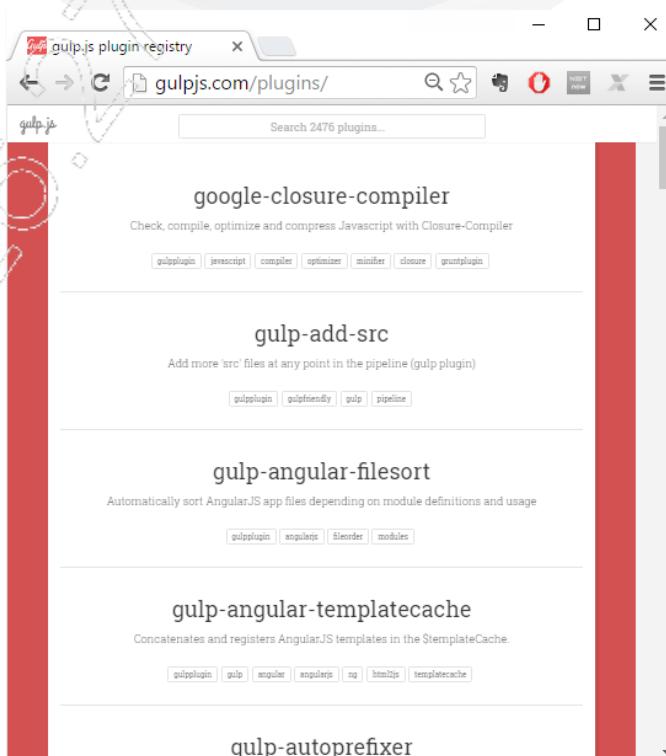
2.5.3. Gulp

Outro componente utilizado é a biblioteca JavaScript Gulp, que automatiza tarefas como otimizar arquivos, testar bibliotecas e preparar arquivos para publicação.



O Gulp fornece uma série de funcionalidades internas, mas também funciona com o conceito de plugins. Desse modo, é possível criar um processamento (por exemplo, retirar os acentos de um texto) e incluir esse processamento nas tarefas definidas no fluxo de trabalho do Gulp.

No site oficial, existem mais de dois mil plugins para utilizar:



O arquivo **gulpfile.js** contém a configuração das tarefas a serem executadas. Nesse caso, são configurações para otimizar os arquivos de script (JavaScript) e as folhas de estilo (CSS):

```
/// <binding Clean='clean' />
"use strict";

var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");

var paths = {
    webroot: "./wwwroot/"
};

paths.js = paths.webroot + "js/**/*.*";
paths.minJs = paths.webroot + "js/**/*.min.*";
paths.css = paths.webroot + "css/**/*.*";
paths.minCss = paths.webroot + "css/**/*.min.*";
paths.concatJsDest = paths.webroot + "js/site.min.js";
paths.concatCssDest = paths.webroot + "css/site.min.css";

gulp.task("clean:js", function (cb) {
    rimraf(paths.concatJsDest, cb);
});

gulp.task("clean:css", function (cb) {
    rimraf(paths.concatCssDest, cb);
});

gulp.task("clean", ["clean:js", "clean:css"]);

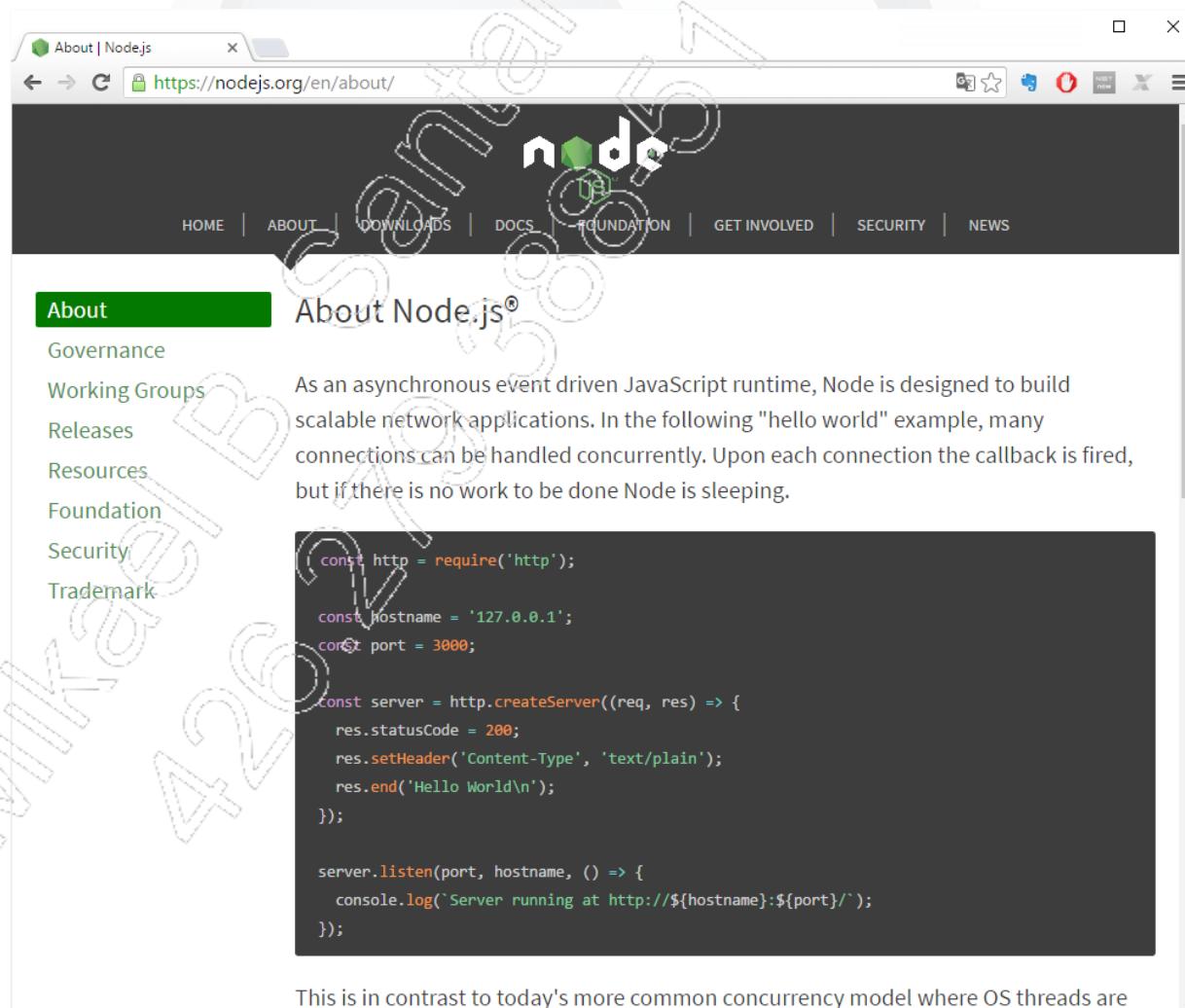
gulp.task("min:js", function () {
    return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
        .pipe(concat(paths.concatJsDest))
        .pipe(uglify())
        .pipe(gulp.dest("."));
});
```

```
gulp.task("min:css", function () {
    return gulp.src([paths.css, "!" + paths.minCss])
        .pipe(concat(paths.concatCssDest))
        .pipe(cssmin())
        .pipe(gulp.dest("."));
});

gulp.task("min", ["min:js", "min:css"]);
```

2.5.4. Node.js e npm

Os frameworks Gulp e seus plugins utilizam uma infraestrutura baseada no componente Node.js. O Node.js é um componente que permite executar JavaScript no servidor.



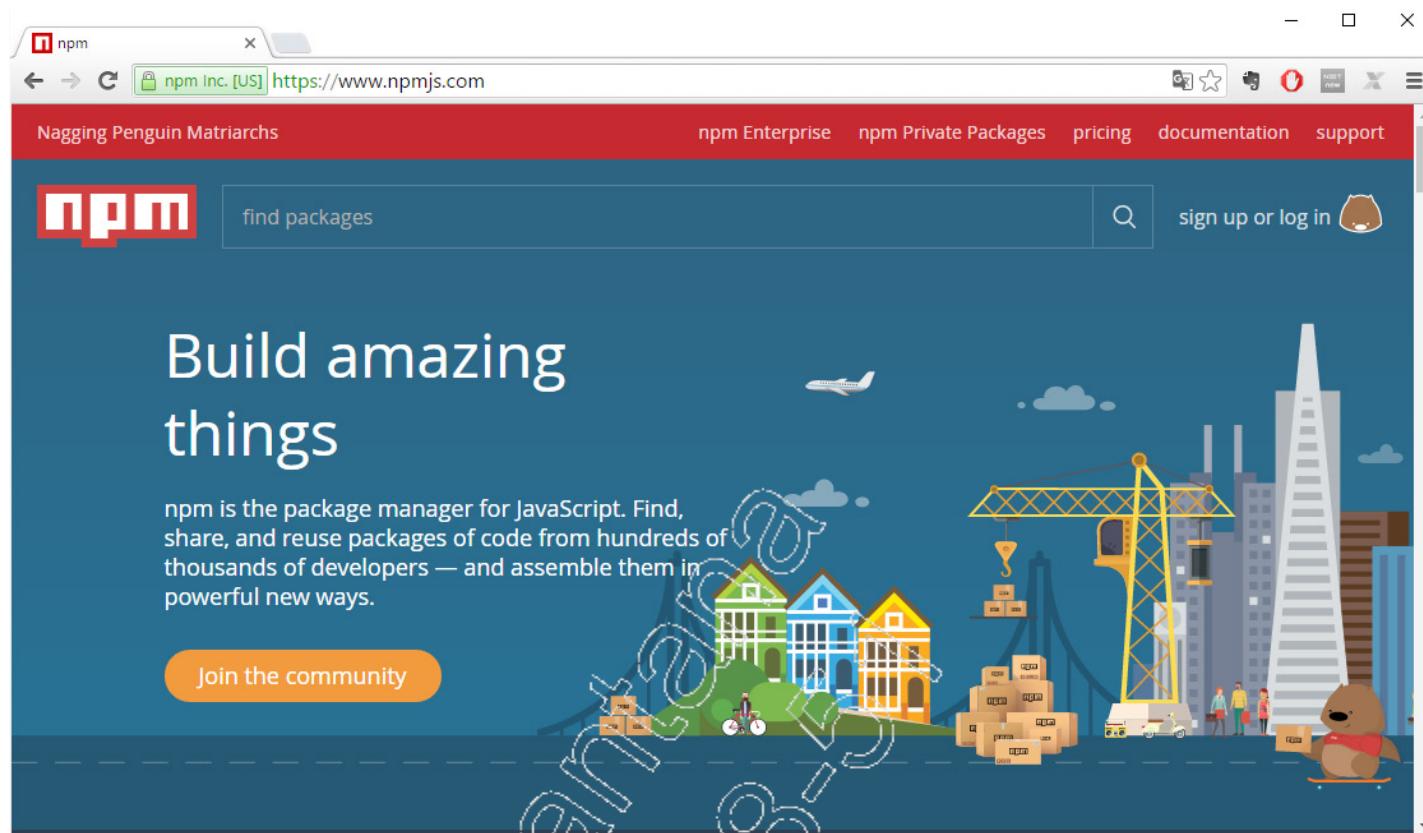
```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

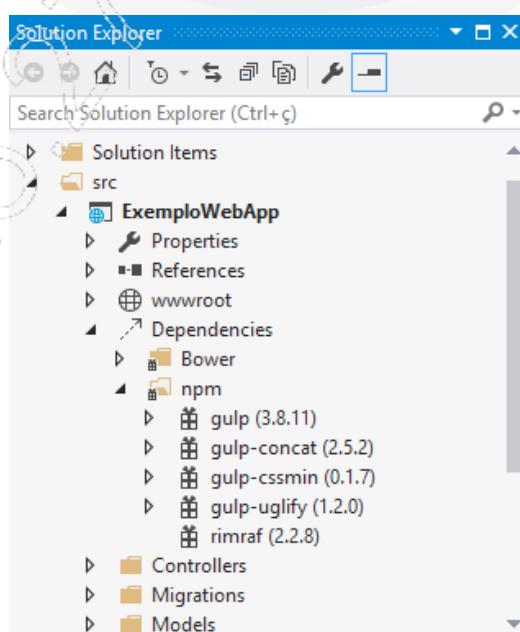
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}`);
});
```

This is in contrast to today's more common concurrency model where OS threads are

npm é um gerenciador de pacotes JavaScript (Node.js Package Manager) e é parte integrante do Node.js. Como são mais frequentes as atualizações do npm do que as atualizações no Node.js, é possível fazer o download das atualizações do npm diretamente do site.



Dentro da pasta **wwwroot / Dependencies**, encontra-se a pasta **npm** com os pacotes gerenciados por ele: **gulp**, **gulp-concat**, **gulp-cssmin**, **gulp-uglify** e **rimraf**. Todos esses componentes são otimizadores de arquivos JavaScript e folhas de estilo CSS.



2.5.5. Outros componentes

Outros componentes que fazem parte do projeto padrão são os mesmos usados na versão 4.6 e inferiores no ASP.NET:

- **Bootstrap;**
- **jQuery;**
- **jQuery-Validation;**
- **Entity Framework;**
- **Microsoft.AspNet.Mvc;**
- **Razor.**

2.5.6. Microsoft.AspNet.Mvc.TagHelpers

Algumas novidades foram introduzidas na geração do código HTML. A classe **TagHelpers** define alguns novos elementos HTML.

- **environment**

Este elemento renderiza ou não um código HTML baseado no valor da propriedade **Microsoft.AspNet.Hosting.IHostingEnvironment.EnvironmentName**, que é um valor inserido automaticamente quando a aplicação é iniciada.

No exemplo adiante, os elementos link serão renderizados apenas se o ambiente de execução estiver definido como **Development**.

```
<environment names="Development">
    <link rel="stylesheet" href="css/bootstrap.css" />
    <link rel="stylesheet" href("~/css/site.css" />
</environment>
```

- **asp-controller**

Este atributo define o nome de um controlador. É utilizado para criar uma âncora HTML (hiperlink).

```
<a asp-controller="Home" asp-action="Index">Home</a>
```

- **asp-action**

Este atributo define o nome de um método de um controlador. É utilizado para criar uma âncora HTML (hiperlink).

```
<a asp-controller="Home" asp-action="Index">Home</a>
```

2.6. Fluxo de execução

Quando uma aplicação ASP.NET Core Web App inicia, os comandos seguintes são executados, na sequência adiante:

1. Classe **Startup** – Método de entrada **Main**

```
public static void Main(string[] args)
=> WebApplication.Run<Startup>(args);
```

2. Classe **Startup** – Método construtor

Uma instância da classe **ConfigurationBuilder** é criada e associada à propriedade **Configuration** da classe **Startup**. O método **AddUserSecrets** cria o ambiente necessário para armazenar informações sem compartilhar com ninguém, por meio de um utilitário Console. O método **AddEnvironmentVariables** cria as variáveis do ambiente administrativo. E, finalmente, o método **Build** inicia as classes.

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .AddJsonFile("appsettings.json")
        .AddJsonFile(
            $"appsettings.{env.EnvironmentName}.json",
            optional: true);

    if (env.IsDevelopment())
    {
        builder.AddUserSecrets();
    }

    builder.AddEnvironmentVariables();

    Configuration = builder.Build();
}
```

3. Classe Startup - Método ConfigureServices

A classe executa diversos métodos para iniciar os serviços.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddEntityFramework()

        .AddSqlServer()

        .AddDbContext<ApplicationContext>(options =>
            options.UseSqlServer(
                Configuration["Data:DefaultConnection:ConnectionString"]));
}
```

```
services.AddIdentity<ApplicationUser, IdentityRole>()  
    .AddEntityFrameworkStores<ApplicationContext>()  
        .AddDefaultTokenProviders();
```

```
services.AddMvc();
```

```
services.AddTransient<IEmailSender, AuthMessageSender>();
```

```
services.AddTransient<ISmsSender, AuthMessageSender>();
```

```
}
```

4. Classe Startup – Método Configure

A classe executa diversos métodos para iniciar os serviços.

```
public void Configure(IApplicationBuilder app,  
    IHostingEnvironment env,  
    ILoggerFactory loggerFactory)  
{  
    loggerFactory.AddConsole(  
        Configuration.GetSection("Logging"));  
    loggerFactory.AddDebug();  
  
    if (env.IsDevelopment()) {  
        app.UseBrowserLink();  
        app.UseDeveloperExceptionPage();  
        app.UseDatabaseErrorResponse();  
    }  
    else {  
        app.UseExceptionHandler("/Home/Error");  
    }
```

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

```
try
{ using (var serviceScope =
    app. ApplicationServices
    .GetRequiredService
    <IServiceScopeFactory>()
    .CreateScope()) {
    ServiceScope.ServiceProvider
    .GetService
    <ApplicationDbContext>()
    .Database.Migrate();
}
catch { }
}

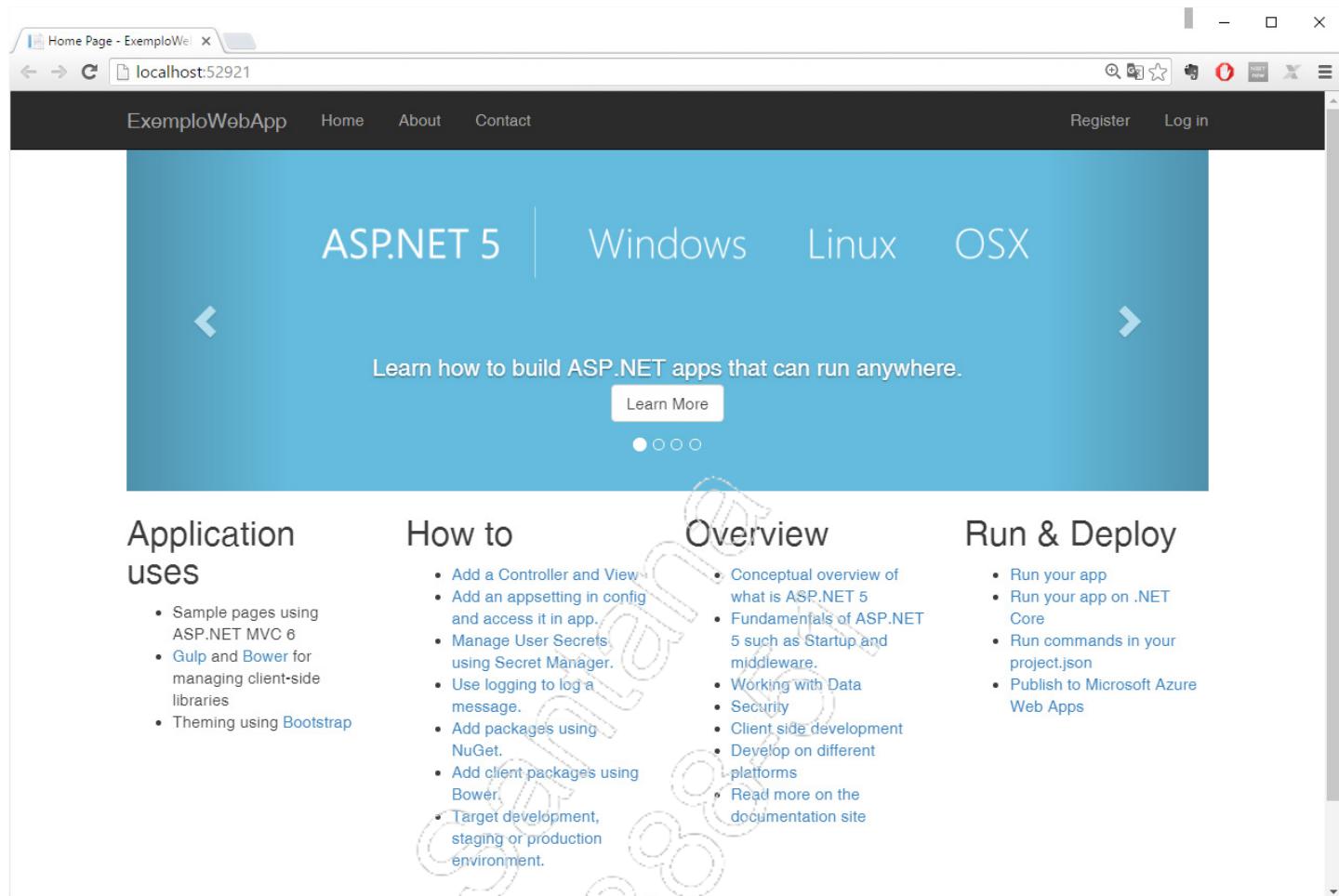
app.UseIISPlatformHandler(options =>
options
    .AuthenticationDescriptions.Clear());

app.UseStaticFiles();

app.UseIdentity();

app.UseMvc(routes =>{routes.MapRoute(
    name: "default",
    template:
    "{controller=Home}/{action=Index}/{id?}");
});
```

5. A View Home / Index é exibida:



How to

- Add a Controller and View
- Add an appsetting in config and access it in app
- Manage User Secrets using Secret Manager
- Use logging to log a message
- Add packages using NuGet
- Add client packages using Bower
- Target development, staging or production environment

Overview

- Conceptual overview of what is ASP.NET 5
- Fundamentals of ASP.NET 5 such as Startup and middleware
- Working with Data
- Security
- Client side development
- Develop on different platforms
- Read more on the documentation site

Run & Deploy

- Run your app
- Run your app on .NET Core
- Run commands in your project.json
- Publish to Microsoft Azure Web Apps

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- As principais características do .Net Core são: modularidade, open source e multiplataforma;
- Node.js é uma biblioteca que permite executar código JavaScript no servidor;
- Gulp é uma biblioteca JavaScript que automatiza tarefas;
- Bower é uma biblioteca para gerenciar pacotes;
- O arquivo de inicialização no ASP.NET Core é o arquivo **Startup.cs**;
- Bootstrap é uma biblioteca para gerenciar folhas de estilo CSS;
- npm significa Node Package Manager e é o gerenciador de pacotes do Node;
- jQuery é uma biblioteca básica JavaScript;
- Os aplicativos .NET Core rodam no Windows, Linux e OSX.

2

ASP.NET Core

Teste seus conhecimentos

Mikael B
426.279.57

1. Qual característica não faz parte do ASP.NET Core?

- a) Arquitetura modular
- b) Multiplataforma
- c) Open source
- d) Hospedagem de aplicativos Web apenas no IIS
- e) Administração via Console

2. Qual linha de comando cria uma aplicação DotNet do tipo Console?

- a) >dotnet new Console
- b) >dotnet bundle App
- c) >dotnet new
- d) >dotnet Console
- e) >dotnet new ConsoleApp

3. No ASP.NET Core, como se chama um arquivo de projeto?

- a) Statup.cs
- b) project.cs
- c) project.json
- d) json.project
- e) global.asax

4. Qual biblioteca é usada para gerenciar os componentes de uma aplicação MVC?

- a) Perls
- b) Bower
- c) DOM
- d) Entity Framework
- e) Gulp

5. O que é o componente Gulp?

- a) É um automatizador de tarefas.
- b) É um componente responsável por autenticar usuários.
- c) É uma biblioteca CSS.
- d) É um arquivo compilado.
- e) É um compilador para código-fonte.

2

ASP.NET Core

Mãos à obra!

Mikael B
426.279.57



IMPACTA
EDITORA

Laboratório 1

A – Criando um Web site institucional simples para apresentar uma empresa, usando o modelo MVC sem autenticação e ASP.NET Core 1.0

O resultado final é exatamente igual ao do laboratório anterior e o objetivo é ver claramente o que mudou e que adaptações teriam que ser feitas para migrar uma aplicação .NET 4.6 para .NET Core 1.0.

Neste laboratório, você criará um Web site institucional simples para apresentar uma empresa, usando o modelo Web Form sem autenticação. Esse Web site terá três páginas: uma página inicial com um resumo da empresa, uma página com um perfil mais detalhado e uma página de contato, na qual o visitante preenche um formulário e os dados são gravados em um arquivo de texto.

As telas são as seguintes:

- **Tela inicial**



- Tela Quem Somos

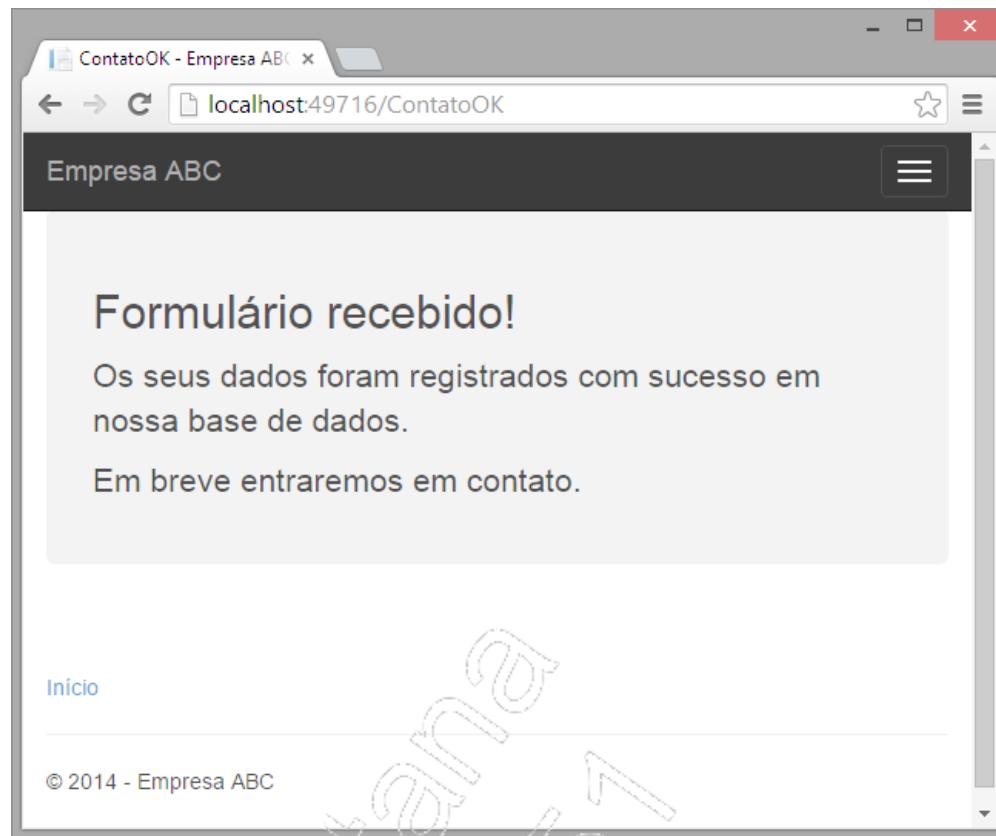


- Formulário de contato e resposta

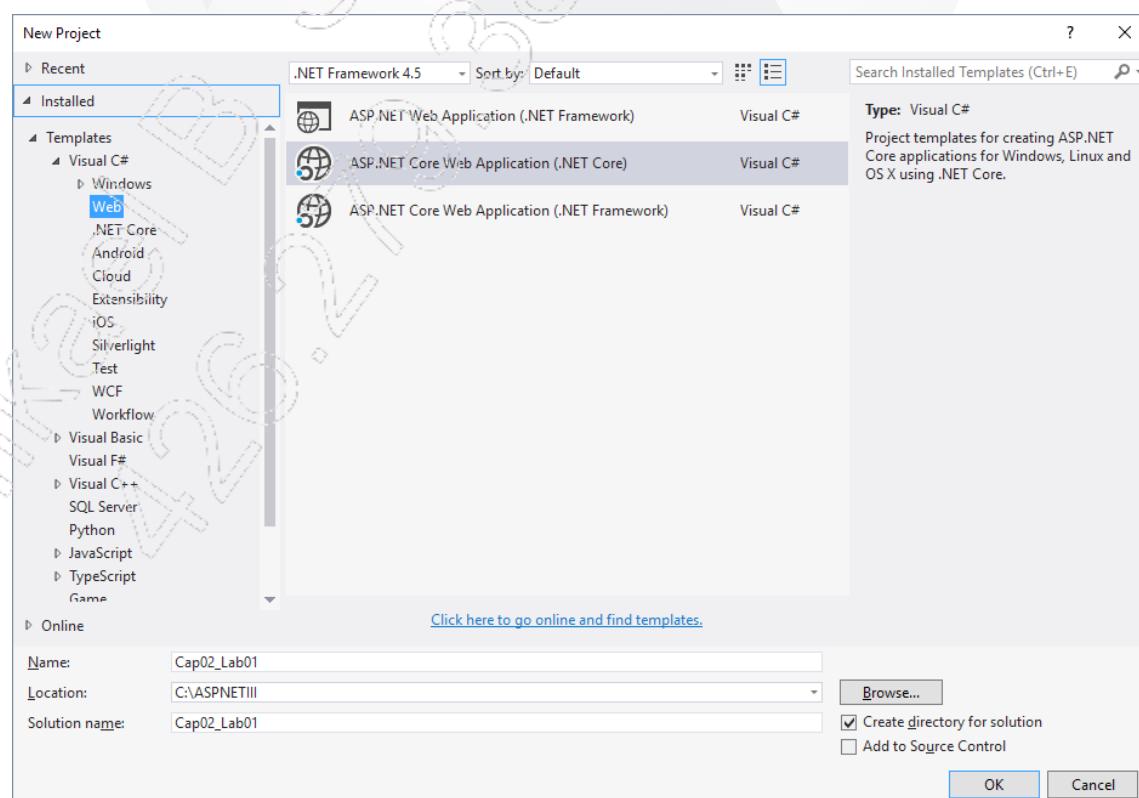
The screenshot shows a web browser window titled 'Entre em Contato - Empresa ABC'. The URL in the address bar is 'localhost:49716/Contato'. The page has a dark header with the text 'Empresa ABC'. The main content area features a heading 'Entre em Contato' and a sub-instruction 'Por favor, preencha o formulário abaixo para entrar em contato'. Below this, there are input fields for 'Seu Nome' (with 'Harison Ford' typed in), 'Seu Email' (with 'hans.solo@star.was' typed in), and 'Assunto' (with 'Informações' selected). There is also a larger text area for 'Mensagem' containing the text 'Quem, Quando, Como e Porque?'. At the bottom, there is a 'Enviar' (Send) button.

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

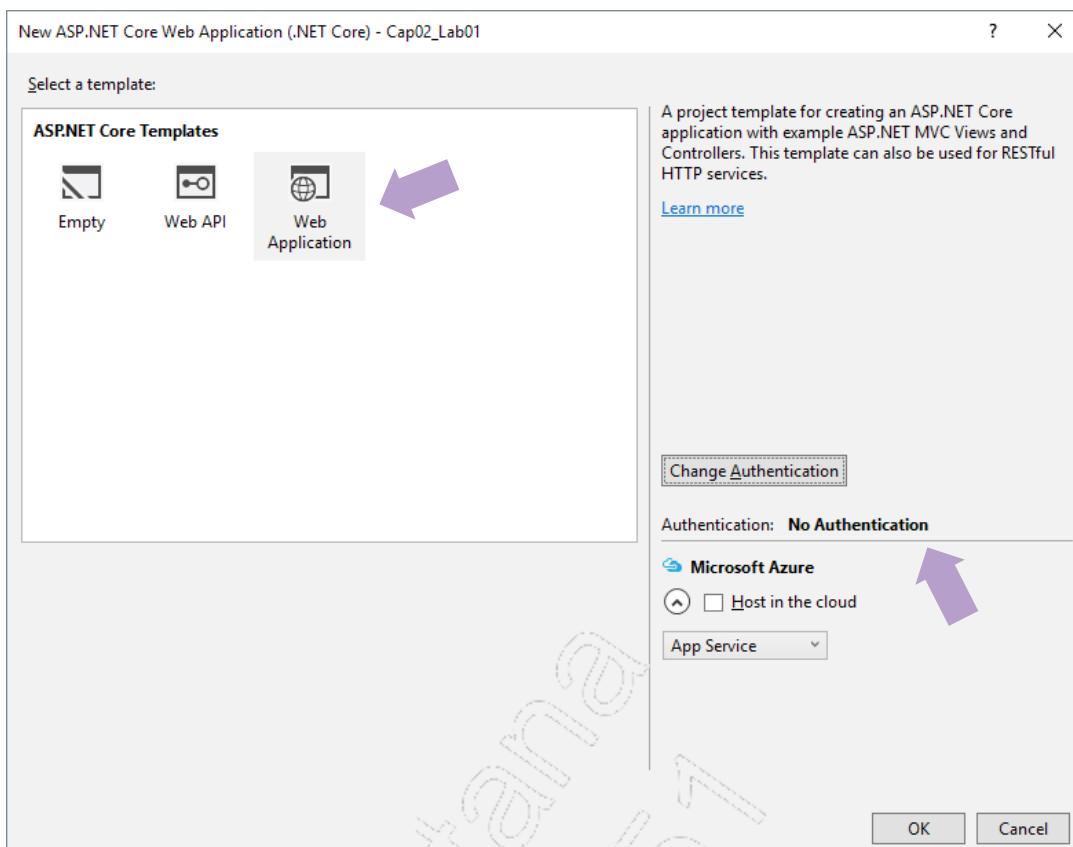
- Quando os dados são digitados corretamente



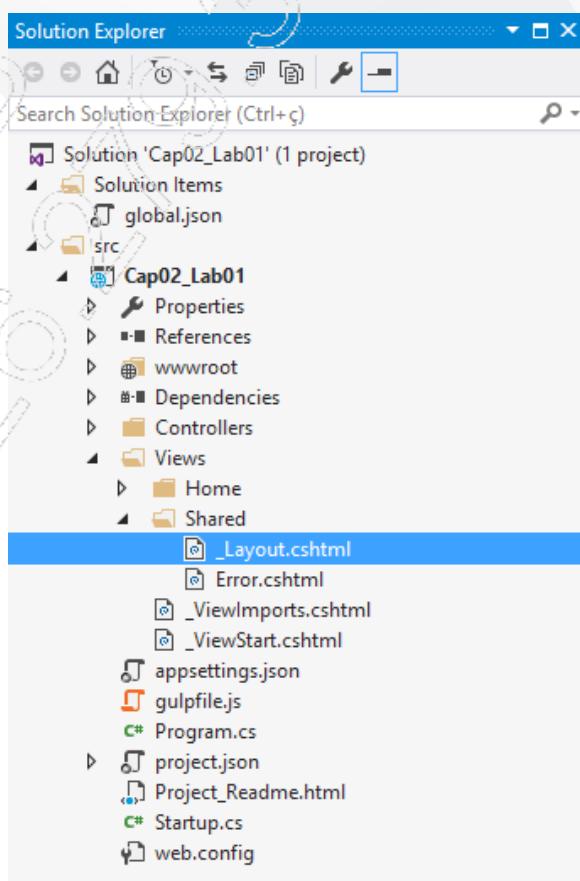
1. O novo Web site terá como base o template MVC do Visual Studio. Crie um novo projeto do tipo **ASP.NET Core Web Application (.NET Core)**:



2. Escolha o template Web Application e Authentication: No Authentication;



3. O primeiro passo é identificar cada elemento da view **Views / Shared / _Layout.cshtml** e descobrir sua finalidade. Faremos isso colocando um comentário HTML(`<!-- -->`) em cima de cada elemento identificado. Abra a view:



4. No cabeçalho <head>, comente a tag do viewport. Essa tag contém informação para o Bootstrap renderizar corretamente as páginas de celulares. O comando é para definir o zoom para 1.0, ou seja, sem zoom:

```
<!-- Bootstrap -->
<meta name="viewport"
      content="width=device-width, initial-scale=1.0" />
```

5. Comente o título da página. Repare que o título é a concatenação do título que é definido em cada página com o nome da empresa:

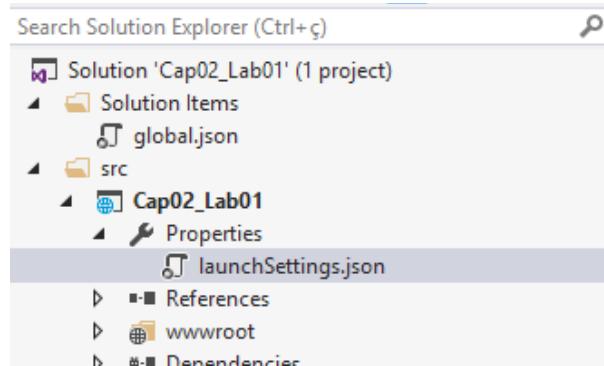
```
<!-- Título da Página -->
<title>@ViewData["Title"] - Cap02_Lab01</title>
```

6. A tag **environment** é equivalente a **bundles** e **minification** da versão 4.6. As variáveis de ambiente **environment** são definidas nas propriedades do projeto e no arquivo **launchSettings.json**. Coloque o comentário nessas tags:

```
<!-- tag renderizada apenas em ambiente de desenvolvimento
-->
<environment names="Development">
    <link rel="stylesheet" ...
    <link rel="stylesheet" href("~/css/site.css" />
</environment>

<!-- tag renderizada apenas em ambiente de homologação e
produção -->
<environment names="Staging,Production">
    <link rel="stylesheet" ...
    <link rel="stylesheet" href "~/css/site.min.css" ...
</environment>
```

Para ver onde estão definidas as variáveis de ambiente, abra o arquivo **Properties / launchSettings.json**:



```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:52506/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "Cap02_Lab01": {
      "commandName": "Project",
      "launchBrowser": true,
      "launchUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

7. Dentro da seção **body**, a primeira div é o navegador. Coloque os comentários para cada item. Repare no uso de tags como **asp-controller**, **asp-action**, **data-toggle**, **data-target**: é o recurso **Tag Helpers**, que permite criar elementos HTML inserindo tags especiais que fornecem informações sobre o que deve ser gerado. É uma alternativa à sintaxe padrão **razor @html.XXX;**

```
<!-- container para o menu-->
<div class="container">

    <!-- cabeçalho do navegador -->
    <div class="navbar-header">

        <!-- botão -->
        <button type="button" class="navbar-toggle"
            data-toggle="collapse"
            data-target=".navbar-collapse">
            <span class="sr-only">Toggle
            navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
        </button>

        <!-- Nome do Projeto-->
        <a asp-controller="Home" asp-
            action="Index"
            class="navbar-brand">Empresa ABC</a>
    </div>

    <!-- menu-->
    <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
            <li><a asp-controller="Home"
                asp-action="Index">Inicio</a></li>
            <li><a asp-controller="Home"
                asp-action="QuemSomoa">Quem Somos</a></li>
            <li><a asp-controller="Home"
                asp-action="COnタato">Entre em Contato</a></li>
        </ul>
    </div>
</div>
</div>
```

8. Comente a tag que guarda lugar para o corpo da página e a tag do rodapé:

```
<!-- Corpo -->
<div class="container body-content">

    <!-- Renderiza a página html -->
    @RenderBody()

    <!-- Rodapé -->
    <hr />
    <footer>
        <p>&copy; <%: DateTime.Now.Year %> - Empresa
ABC</p>
    </footer>
</div>
```

9. Altere a página **Views / Home / Index.cshtml** para exibir dados da empresa:

```
@{
    ViewData["Title"] = "Home Page";
}

<!-- Quadro Principal -->
<div class="jumbotron">

    <!-- Título -->
    <h1>Empresa ABC</h1>

    <!-- Subtítulo -->
    <p class="lead">A Empresa ABC tem como compromisso
        pesquisar novas tecnologias que
        facilitem o desenvolvimento de aplicações WEB
    </p>

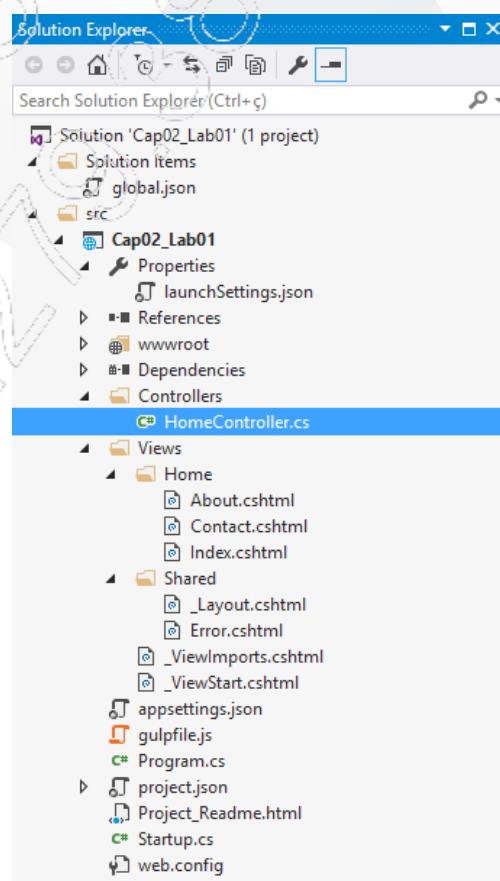
    <!-- Saiba Mais -->
    <p>
        <a href="QuemSomos"
            class="btn btn-primary btn-lg">Saiba mais&raquo;
        </a>
    </p>
</div>
```

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

10. Execute e veja a página inicial:



11. Abra o controller Home:



12. Exclua as actions **About** e **Contact**. Nesta versão (.NET Core 1.0), o retorno é uma interface (**IActionResult**), e não uma classe abstrata (**Action Result**). Isso cria menos dependência entre os componentes. Repare, também, que existe um método a mais: **Error**. Pode deixar esse método, pois será usado no futuro;

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
```

```
public IActionResult About()
{
    ViewData["Message"] = "Your application description
page.";

    return View();
}
```

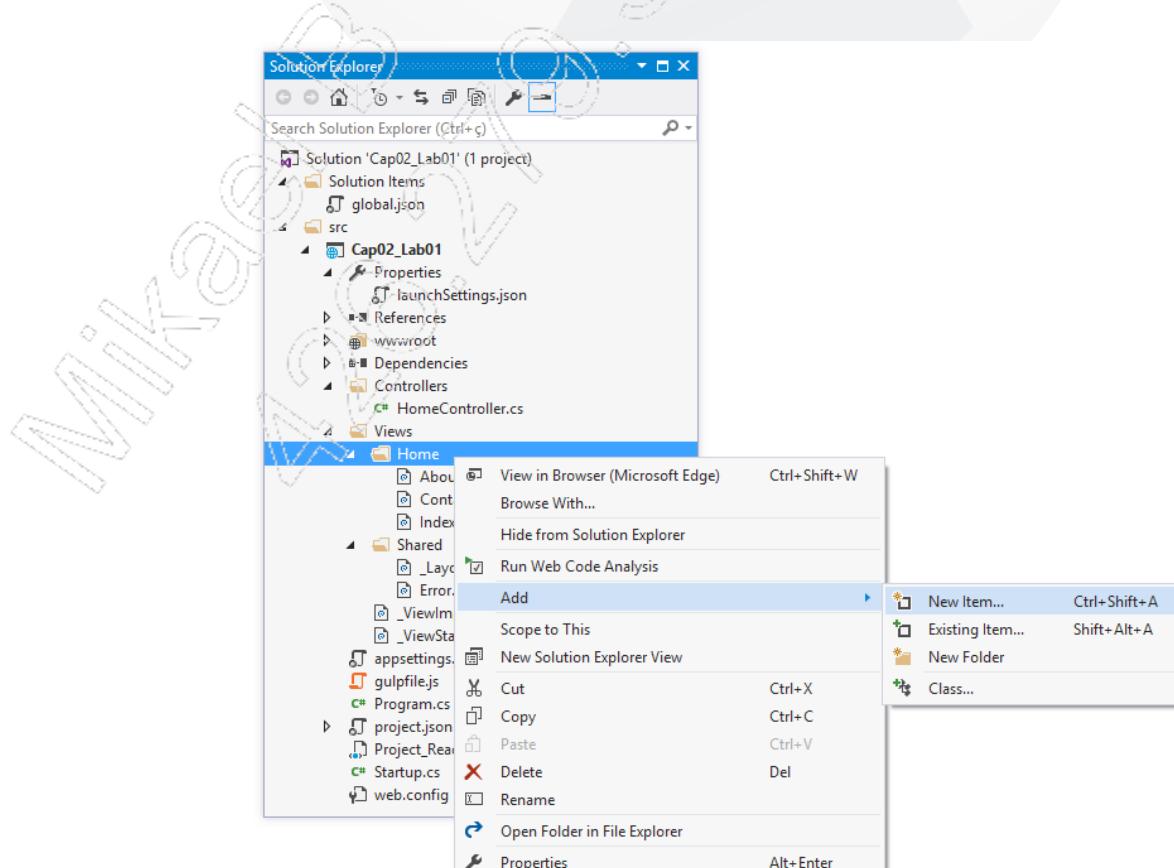
```
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";
    return View();
}
```

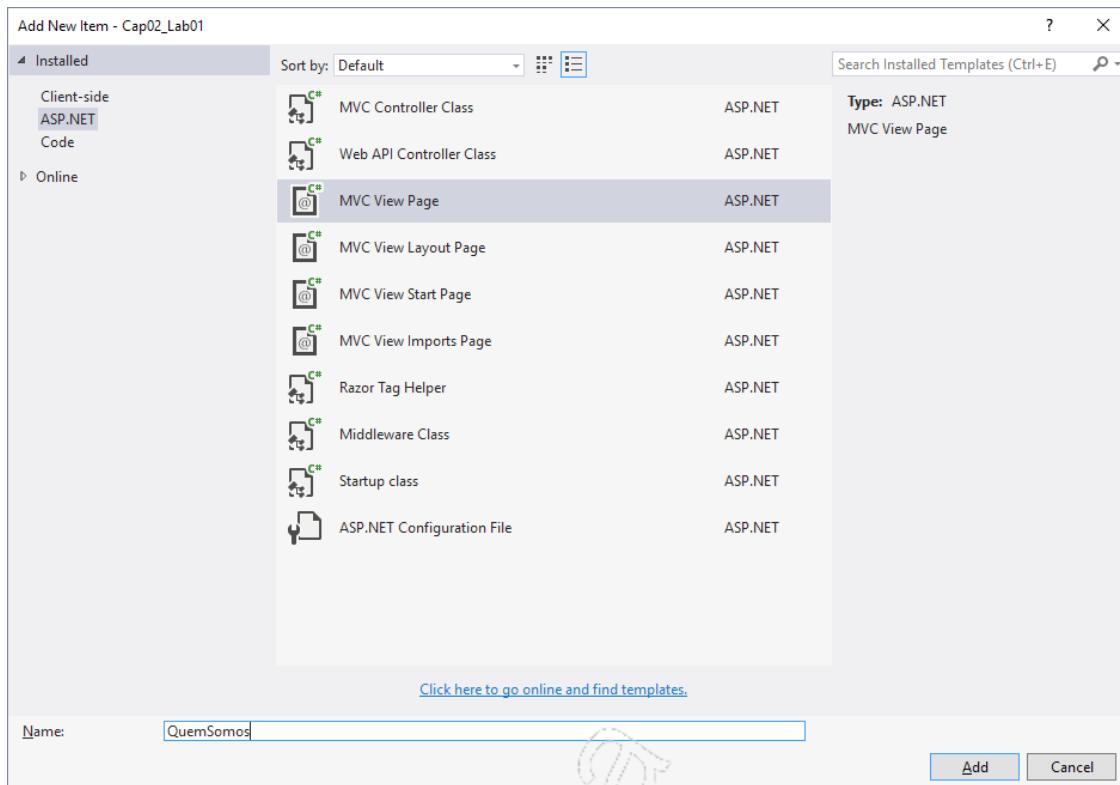
```
public IActionResult Error()
{
    return View();
}
```

13. No lugar dessas actions, crie dois métodos: **QuemSomos** e **Contato**:

```
public class HomeController : Controller
{
    public IActionResult Index()....  
  
    public IActionResult QuemSomos()
    {
        return View();
    }
  
  
    public IActionResult Contato()
    {
        Return View();
    }
  
  
    public IActionResult Error()...
}
```

14. Crie a view **QuemSomos**. Para isso use o menu de contexto no Solution Explorer, na pasta **View / Home**, e escolha **Add New Item**. Na próxima janela, escolha **ASP.NET** na categoria, depois **MVC View Page** e nomeie-a como **QuemSomos**:





15. Use o texto HTML à vontade. Veja, adiante, uma sugestão:

<h2> Empresa ABC</h2>

<h3> Indo onde ninguém jamais esteve...</h3>

<p>A empresa ABC tem como missão pesquisar e descobrir novas tecnologias que tornem a experiência do programador que cria aplicações Web mais interessante e instigante. </p>

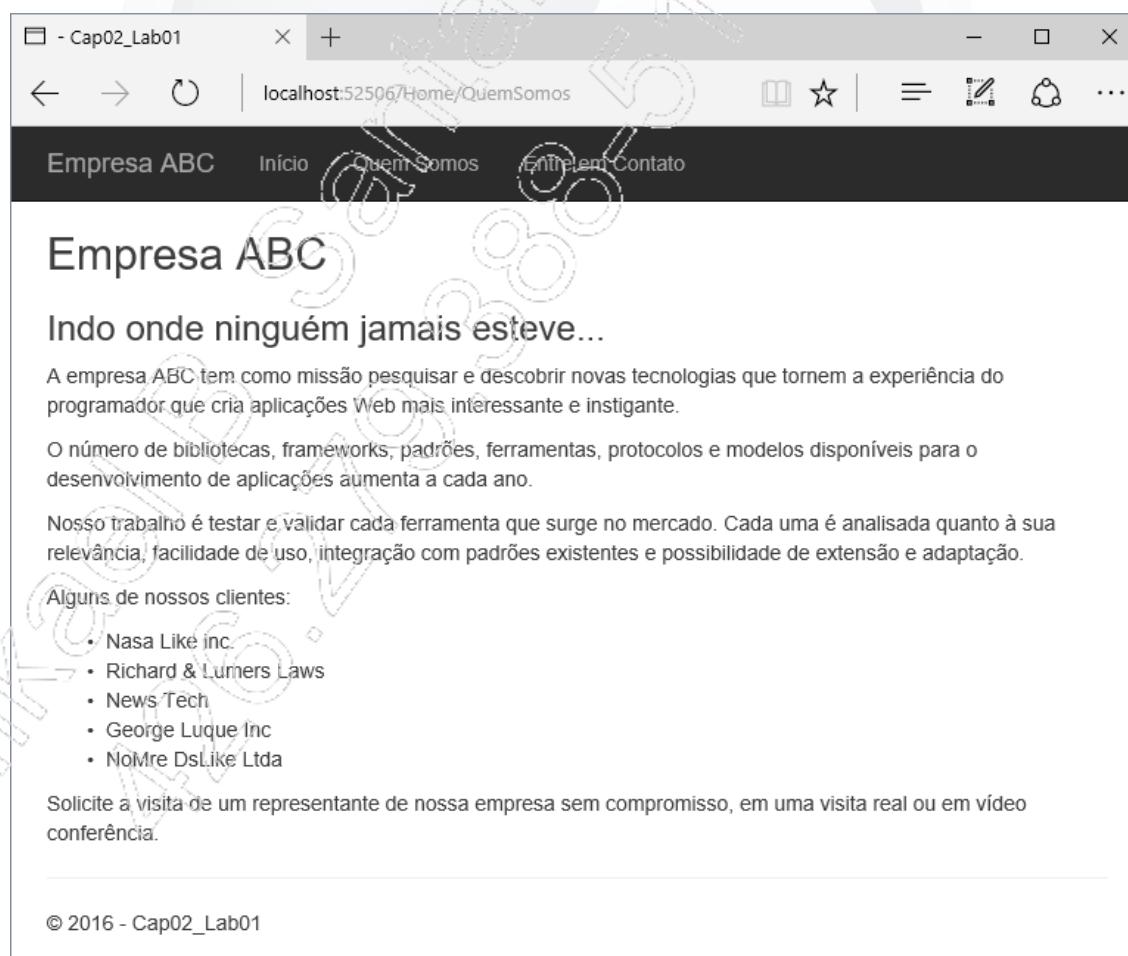
<p>O número de bibliotecas, frameworks, padrões, ferramentas, protocolos e modelos disponíveis para o desenvolvimento de aplicações aumenta a cada ano.</p>

<p>Nosso trabalho é testar e validar cada ferramenta que surge no mercado. Cada uma é analisada quanto à sua relevância, facilidade de uso, integração com padrões existentes e possibilidade de extensão e adaptação.</p>

```
<p>Alguns de nossos clientes:</p>
<ul>
    <li>Nasa Like inc.</li>
    <li>Richard & Lumers Laws</li>
    <li>News Tech</li>
    <li>George Luque Inc</li>
    <li>NoMre DsLike Ltda</li>
</ul>
```

```
<p>Solicite a visita de um representante de nossa
empresa sem compromisso, em uma visita real ou em vídeo
conferência.
</p>
```

16. Visualize a página:



17. Na página de contato, o usuário vai preencher um formulário. Para isso, vamos usar um modelo. Crie uma pasta chamada **Models** e, dentro dela, crie a classe **ContatoViewModel**:

```
public class ContatoViewModel
{
    public string Nome { get; set; }
    public string Email { get; set; }
    public string Assunto { get; set; }
    public string Mensagem { get; set; }
}
```

18. Para gravar os dados, crie uma classe estática dentro da pasta **Controllers**. A classe vai se chamar **RotinasWeb**:

```
public static class RotinasWeb
{
    public static void ContatoGravar(ContatoViewModel contato)
    {
        string pathInicial = PlatformServices
            .Default
            .Application
            .ApplicationBasePath;

        string arquivo = pathInicial + @"\Contatos.txt";

        using (var sw = File.CreateText(arquivo))
        {
            sw.WriteLine(DateTime.Now);
            sw.WriteLine(contato.Nome);
            sw.WriteLine(contato.Email);
            sw.WriteLine(contato.Assunto);
            sw.WriteLine(contato.Mensagem);
            sw.WriteLine(new string('-', 30));
        }
    }
}
```

19. Crie a view **Contato**. Nas primeiras versões do ASP.NET Core, não foram incluídos os assistentes para gerar formulários. Mas é fácil construir um formulário usando o mesmo estilo da aplicação exemplo:

```
@model Cap02_Lab01.Models.ContatoViewModel
@{
    ViewData["Title"] = "Entre em Contato";
}

<h2>Entre em Contato</h2>
<form method="post" class="form-horizontal">

    <hr />
    @{
        if (ViewData["erros"] != null) {
            var erros = (List<string>)ViewData["erros"];
            if (erros.Count > 0) {
                <ul class="text-danger">
                    @foreach (var erro in erros)
                    { <li>@erro</li> }
                </ul>
            }
        }
    }

    <div class="form-group">
        <label asp-for="Nome" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Nome" class="form-control" />
        </div>
    </div>

    <div class="form-group">
        <label asp-for="Email" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Email" class="form-control" />
        </div>
    </div>
```

```
<div class="form-group">
    <label asp-for="Assunto" class="col-md-2 control-
label"></label>
    <div class="col-md-10">
        <input asp-for="Assunto" class="form-control"
/>
    </div>
</div>
<div class="form-group">
    <label asp-for="Mensagem" class="col-md-2 control-
label"></label>
    <div class="col-md-10">
        <input asp-for="Mensagem" class="form-control"
/>
    </div>
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <button type="submit" class="btn btn-
default">Enviar</button>
    </div>
</div>
</form>
```

20. Crie o método que vai receber o formulário, na classe **ContatoController**:

```
[HttpPost]
public IActionResult Contato(Models.ContatoViewModel
contato)
{
    var erro = new List<string>();

    if (string.IsNullOrEmpty(contato.Nome))
    {
        erro.Add("O nome deve ser informado");
    }
}
```

```
if (string.IsNullOrEmpty(contato.Mensagem) )
{
    erro.Add("O Email deve ser informado");
}

if (erro.Count == 0)
{
    RotinasWeb.ContatoGravar(contato);
    return View("ContatoOK");
}
else
{
    ViewData["erros"] = erro;
    return View(contato);
}
}
```

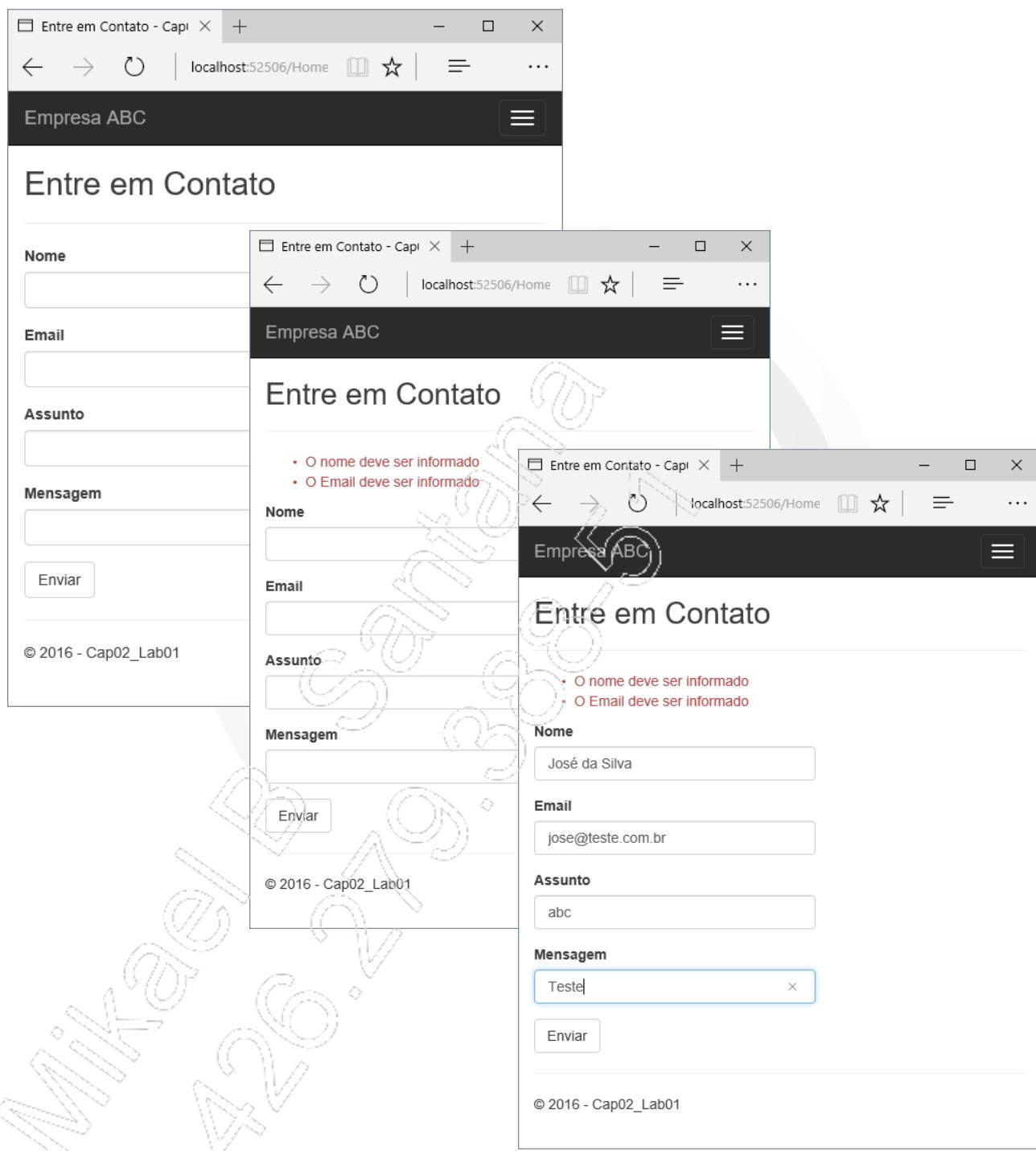
21. Crie uma view chamada **ContatoGravarOK**:

```
<h2>Obrigado por seu contato</h2>

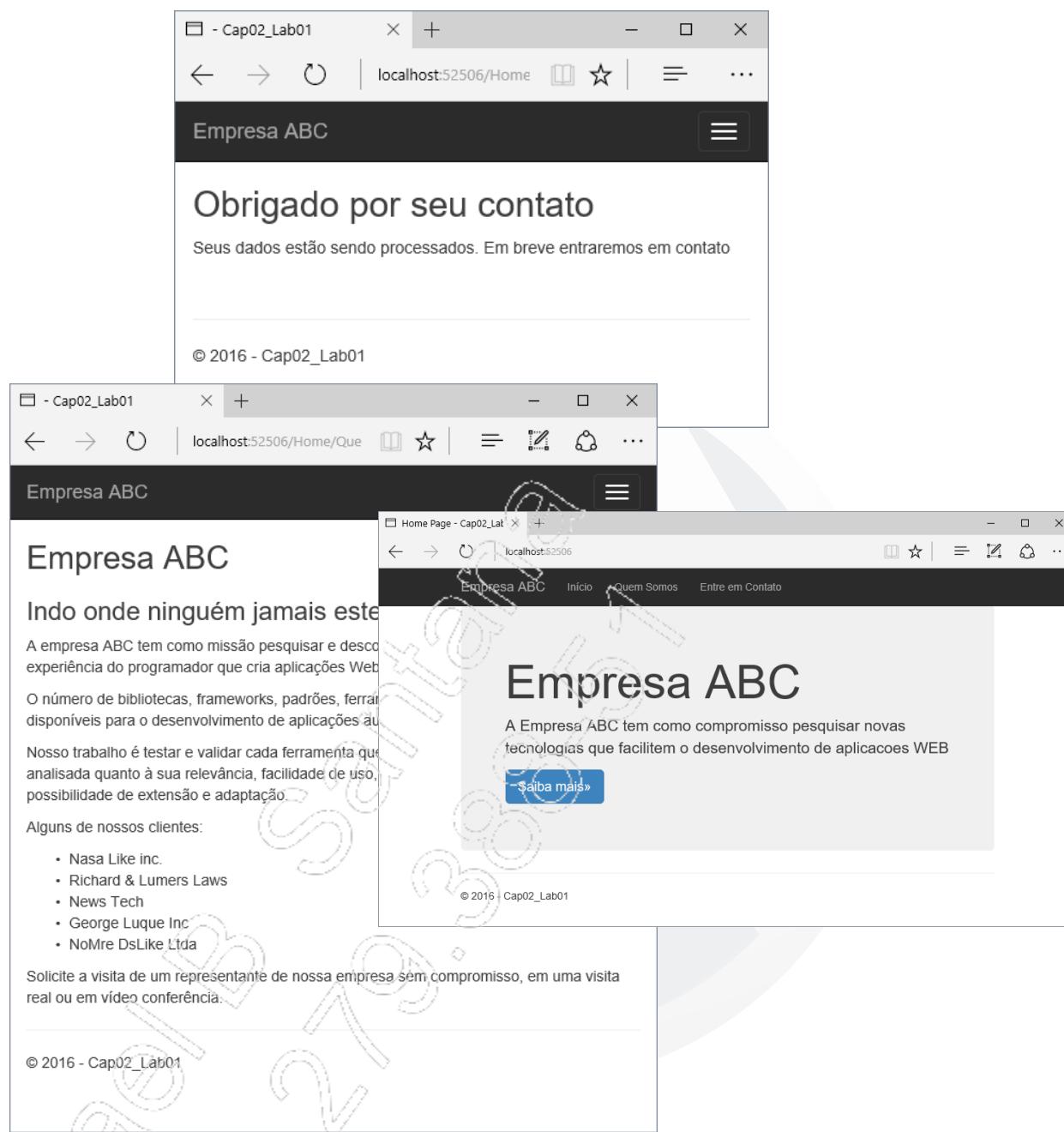
<p>Seus dados estão sendo processados. Em breve entraremos
em contato</p>

<br/>
```

22. Projeto completo! Teste o contato e aplicação completa.



Visual Studio 2015 - ASP.NET com C# Recursos Avançados



3

SignalR

- ✓ WebSockets e outros protocolos;
- ✓ Implementando uma comunicação SignalR;
- ✓ Nome da função em JavaScript;
- ✓ Escopo dos clientes.

3.1. Introdução

Uma das principais características do ambiente Web é o conceito de Cliente/Servidor ou Solicitação/Resposta: um software (chamado **cliente**) solicita uma informação a um outro software (chamado **servidor**). Este, ao receber a requisição, envia a resposta ao cliente, e o processo é encerrado. Em nenhuma hipótese o servidor chama o cliente sem ter sido solicitado.

A descrição anterior é o padrão de um navegador chamando um servidor pela Internet. ASP.NET SignalR é uma nova biblioteca que quebra, até um certo ponto, essa definição, criando a possibilidade da comunicação em tempo real bidirecional: do cliente para o servidor e do servidor para o cliente.

ASP.NET SignalR é muito útil em aplicações que necessitam de atualização constante, como chats, quadros de avisos, painéis indicadores, controle de estoque, controle de posicionamento global, suporte técnico, jogos, entre outros.

A comunicação bidirecional pela Web não é novidade. Muitas implementações foram feitas usando desde componentes e plugins instalados no navegador até programas em JavaScript que simulavam o efeito do bidirecionamento chamando o servidor em intervalos regulares.

A novidade, no caso da SignalR, é uma biblioteca voltada especialmente para isso, fácil de usar, sem componentes externos, funcionando totalmente em ambiente gerenciado e que se adapta a qualquer tipo de browser, até aos mais antigos.

SignalR trabalha com o conceito de Hub, que é um programa que controla a transmissão de dados. Do lado do servidor existe um Hub que é responsável pela conexão com diversos clientes, e cada cliente tem um HubProxy que é responsável pela conexão com um único Hub. A conexão fica diretamente aberta. Isso é completamente diferente de uma comunicação HTTP, em que a conexão é aberta e fechada a cada requisição.

3.2. WebSockets e outros protocolos

O protocolo que torna a comunicação bidirecional possível se chama **WebSockets**. É uma tecnologia relativamente nova e que depende da plena implementação por parte dos navegadores e dos servidores. O que torna a biblioteca SignalR interessante é que, quando não é possível utilizar essa tecnologia, o framework usa automaticamente outras técnicas, por meio de eventos e outros recursos. O SignalR gerencia automaticamente o protocolo possível de ser utilizado e escolhe um entre quatro disponíveis:

- **WebSockets**: O único realmente bidirecional;
- **Long Polling**: Utiliza uma técnica na qual a conexão fica sempre aberta, aumentando o tempo de resposta entre cliente e servidor;
- **Server-Sent Events**: Utiliza o recurso de eventos no servidor para chamadas em intervalos regulares;
- **Forever Frame**: Utiliza uma antiga técnica de um iframe oculto com um código em JavaScript que mantém requisições constantes com o servidor, dando a impressão de que está sendo atualizado em tempo real.

3.3. Implementando uma comunicação SignalR

Existem várias maneiras de implementar a comunicação SignalR. Uma das mais simples e funcionais é a descrita a seguir:

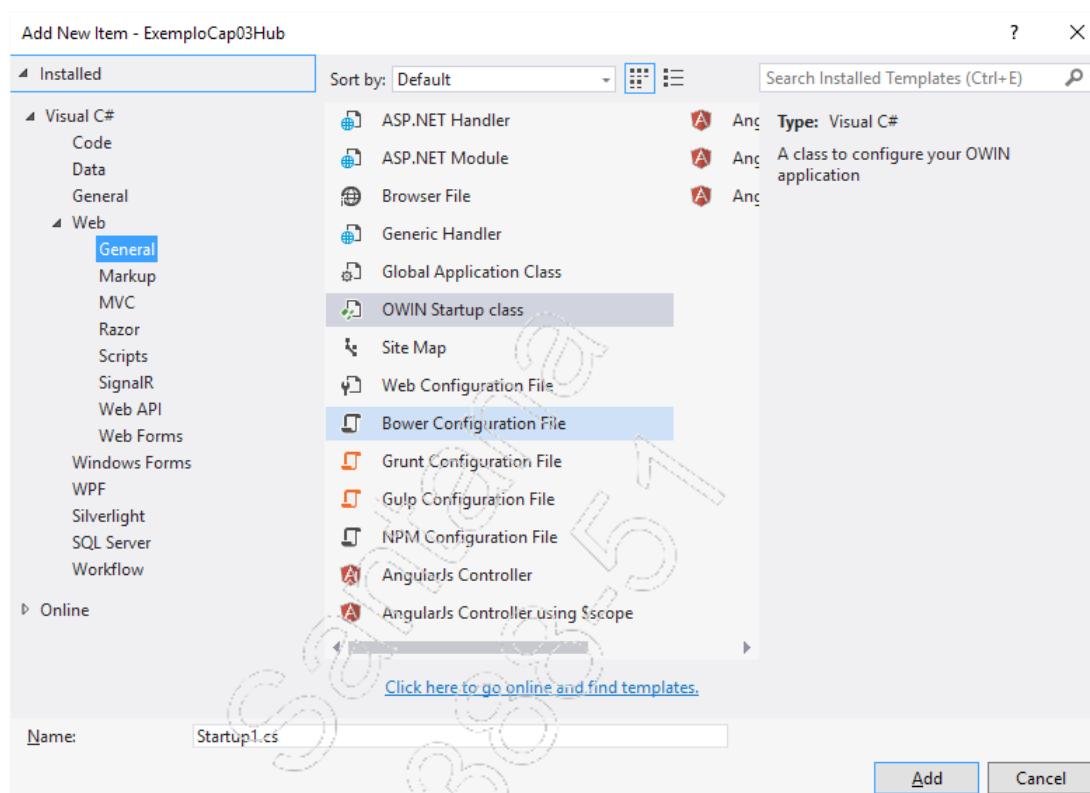
1. Adicione um arquivo de inicialização OWIN;
2. Crie uma classe derivada da classe **Microsoft.AspNet.SignalR.Hub**;
3. Crie métodos públicos nesta classe, que serão chamados pelos clientes;
4. Por meio da propriedade **Clients** da classe **Hub**, realize chamadas que serão executadas nos clientes;
5. Crie uma página HTML com o JavaScript que se conecte ao Hub.

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

Veja a implementação passo a passo:

- **Adicionando um arquivo de inicialização OWIN**

Para realizar isso, clique em **Add New Item**, selecione **Web** e, em seguida, **OWIN Startup class**:



O próximo passo é chamar o método de extensão **MapSignalR()**. Esse método cria uma rota para as chamadas do SignalR, por padrão, como **/signalR**. Essa URL não será chamada diretamente do código, mas é necessária para a infraestrutura funcionar.

```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;

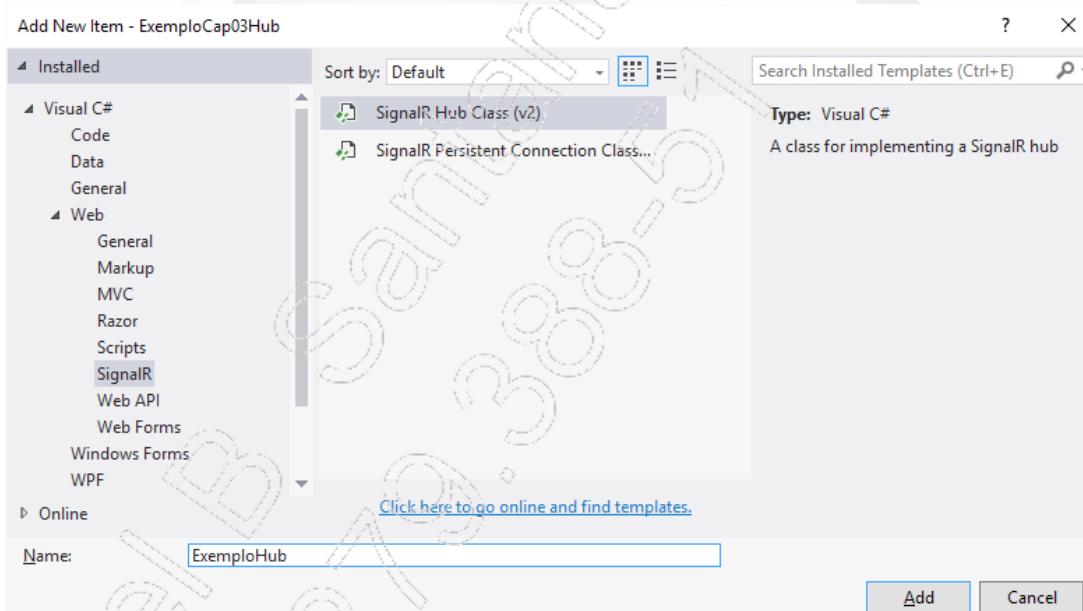
[assembly: OwinStartup(typeof(ExemploSR.Startup))]

namespace ExemploSR
{
```

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        app.MapSignalR();
    }
}
```

- Criando uma classe derivada da classe `Microsoft.AspNet.SignalR.Hub`

É possível criar isso manualmente, porém, a melhor maneira de fazê-lo é usar **Add New Item**, escolher **SignalR** e a opção **SignalR Hub Class**. Isso inclui automaticamente as bibliotecas necessárias.



Esta é a classe criada pelo modelo do Visual Studio. **Hello** é o método que pode ser chamado pelos clientes JavaScript. A propriedade **Clients.All** chama um método dinâmico de nome **hello** (com letras minúsculas, não definido aqui) que deve ser definido na classe JavaScript, servindo como Proxy.

Para que a classe **Hub** tenha mais funcionalidade e seja mais clara para estudo, ela foi reescrita com um método que escreve uma mensagem. Essa classe tem as seguintes características:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Microsoft.AspNet.SignalR;

namespace ExemploSignalR

{
    public class ExemploHub : Hub
    {
        public void EnviarMensagemParaHub(string mensagem)
        {
            Clients.All.enviarMensagemParaCliente(mensagem);
        }
    }
}
```

- 1 – Deve herdar de Hub;
- 2 – Os clientes podem chamar esses métodos públicos;
- 3 – Por meio da propriedade **Clients**, o Hub chama todos os clientes. É necessário criar um JavaScript no cliente com um método chamado, neste exemplo, **enviarMensagemParaCliente**.

O próximo passo é criar a página HTML com o JavaScript necessário para chamar o Hub. Neste exemplo, será apenas um TextBox com um botão para enviar mensagens.

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>

    <input type="text" id="mensagem" />
    <input type="button" id="enviar" value="Enviar" />
    <br/>
    <div id="mensagens"></div>

</body>
</html>
```

É necessário incluir algumas referências a arquivos JavaScript. É interessante notar a terceira referência, que é para um script inexistente. O script **signalr/hubs** será criado dinamicamente.

```
<!DOCTYPE html>
<html>
<head>
    <title>Exemplo SignalR</title>

        <script src="Scripts/jquery-1.10.2.min.js"></script>
        <script src="Scripts/jquery.signalR-2.0.2.min.js"></script>
        <script src="signalr/hubs"></script>

</head>

<body>
    <input type="text" id="mensagem" />
    <input type="button" id="enviar" value="Enviar" />
    <br/>
    <div id="mensagens"></div>
</body>
</html>
```

- **JavaScript – Criando o Proxy Hub**

O próximo passo é escrever o script que recebe e envia comandos para o servidor por meio do Hub:

```
<!DOCTYPE html>
<html>
<head><title>SignalR Simple Chat</title>
<script src="Scripts/jquery-1.10.2.min.js"></script>
<script src="Scripts/jquery.signalR-2.0.2.min.js"></script>
<script src="signalr/hubs"></script>

<script type="text/javascript">

$(function () {

    var h = $.connection.exemploHub;

    h.client.enviarMensagemParaCliente = function (mensagem) {
        $('#mensagens').append(mensagem + '<br />');
    };

    function enviarMensagemParaHub() {
        h.server.enviarMensagemParaHub($('#mensagem').val());
        $('#mensagem').val('').focus();
    }

    function addEventoClick() {
        $('#enviar').click(enviarMensagemParaHub);
    }

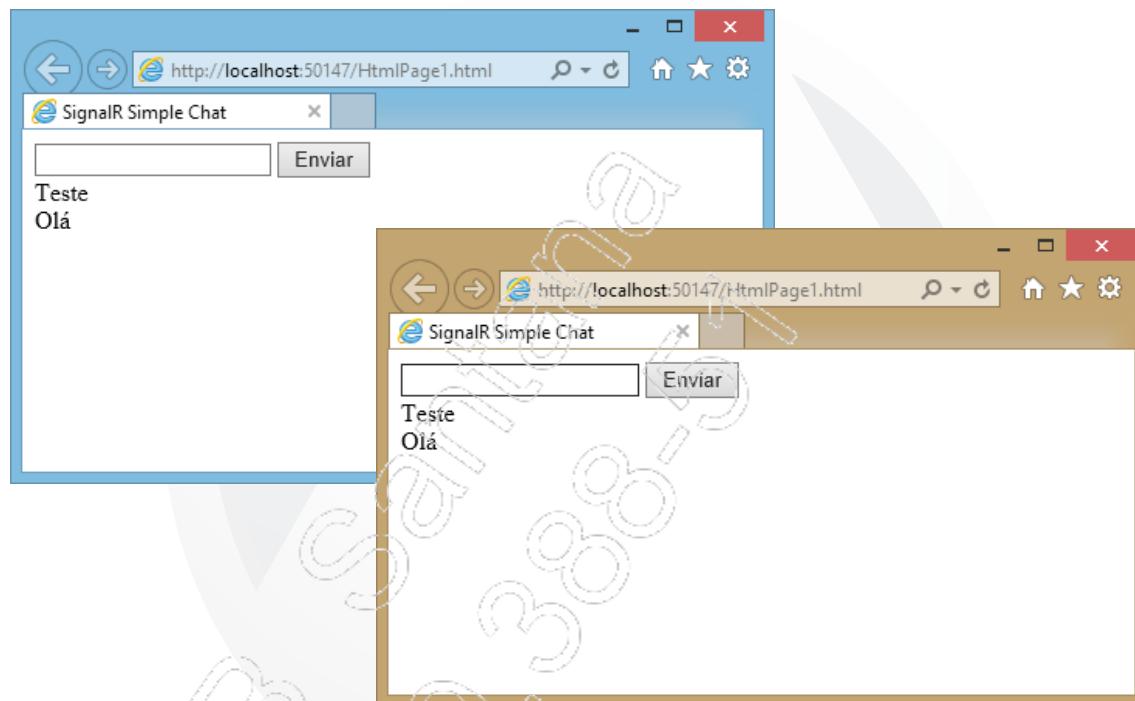
    $.connection.hub.start().done(addEventoClick);

    $('#mensagem').focus();

}) ;
</script>
```

```
</head>
<body>
    <input type="text" id="mensagem" />
    <input type="button" id="enviar" value="Enviar" />
    <br/>   <div id="mensagens"></div>
</body>
</html>
```

Ao executar em vários navegadores, todos são notificados quando um deles altera uma informação.



A seguir, vejamos uma explicação, passo a passo, do script anterior:

```
var h = $.connection.exemploHub;
```

Esta linha obtém uma instância da classe **ExemploHub** criada anteriormente. Aqui entra uma convenção: o nome da classe em JavaScript deve usar **camelCase**, enquanto o nome da classe no sistema usa **PascalCase**.

```
h.client.enviarMensagemParaCliente = function (mensagem) {  
    $('#mensagens').append(mensagem + '<br />');  
};
```

Esta parte define a função em JavaScript que será executada quando for chamada a partir do servidor. A classe **Hub** fornece a propriedade **Clients**, que permite executar um método por meio de uma propriedade dinâmica. No caso, todos os clientes conectados (**all**) terão o método chamado.

```
public class ExemploHub : Hub  
{  
    public void EnviarMensagemParaHub(string mensagem)  
    {  
        Clients.All.enviarMensagemParaCliente(mensagem);  
    }  
  
    function enviarMensagemParaHub() {  
        h.server.enviarMensagemParaHub($('#mensagem').val());  
        $('#mensagem').val('').focus();  
    }  
}
```

Esta função é a que envia a mensagem para o servidor por meio da propriedade **server**. O problema é que ela só pode ser chamada depois que a conexão com o servidor tiver sido estabelecida. Ela está declarada aqui, mas será utilizada posteriormente.

```
function addEventoClick() {  
    $('#enviar').click(enviarMensagemParaHub);  
}
```

Mesmo caso que o anterior. O servidor só pode ser chamado quando a conexão estiver completa. Esta função adiciona o evento **click** do botão **Enviar** ao método **enviarMensagemParaHub**, mas aqui ela está apenas declarada. Também será usada depois.

```
$.connection.hub.start().done(addEventoClick);
```

Aqui é chamado o método **start** da classe **Hub**. Quando o método é completado (**done**), o evento **click** é adicionado com a função que envia a mensagem para o servidor. A função **addEventoClick**, então, é chamada, e, por sua vez, chama a função **enviarMensagemParaHub**.

```
$( '#mensagem' ).focus();
```

Este passo posiciona o cursor no TextBox.

É importante perceber que tanto o servidor chama o cliente quanto o cliente chama o servidor. A implementação não precisa ter a lógica apresentada aqui. Por exemplo, nem sempre o cliente chama um método no servidor e este atualiza todos os clientes.

O programador fica livre para definir métodos no JavaScript e no servidor e chamá-los a qualquer momento. A grande diferença em relação à Web tradicional é a chamada que o servidor executa por meio da propriedade **Clients**.

3.4. Nome da função em JavaScript

Todas as convenções utilizadas pelo SignalR podem ser sobrepostas. Por exemplo, o nome da classe **Hub** é escrita em **PascalCase**, ou seja, em maiúsculas:

```
public class ExemploHub : Hub
{}
```

A classe correspondente em JavaScript é em **camelCase**, ou seja, a primeira letra em minúscula e as internas em maiúscula:

```
var h = $.connection.exemploHub;
```

Na verdade, a classe em JavaScript pode ter qualquer nome. Basta informar o nome desejado por meio do atributo **HubName**:

```
[HubName ("ExemploJavascriptHub")]
public class ExemploHub : Hub
{
    public void EnviarMensagemParaHub (string mensagem)
    {
        Clients.All.enviarMensagemParaCliente (mensagem);
    }
}
```

Nesse caso, o comando em JavaScript ficaria da seguinte forma:

```
var h = $.connection.ExemploJavascriptHub;
```

3.5. Escopo dos clientes

O modelo padrão criado pelo Visual Studio usa a propriedade dinâmica **All** para executar métodos em todos os clientes conectados no Hub. Essa propriedade está definida na interface **IHubConnectionContext**, que é uma interface indireta da propriedade **Clients**.

```
public abstract class Hub : IHub, IDisposable
{
    protected Hub ();

    public IHubCallerConnectionContext Clients { get; set; }

    public HubCallerContext Context { get; set; }
    public IGroupManager Groups { get; set; }
    public void Dispose ();
    protected virtual void Dispose (bool disposing);
    public virtual Task OnConnected ();
    public virtual Task OnDisconnected ();
    public virtual Task OnReconnected ();
}
```

```
interface IHubCallerConnectionContext : IHubConnectionContext
{
    [Dynamic]
    dynamic Caller { get; }

    [Dynamic]
    dynamic Others { get; }

    dynamic OthersInGroup(string groupName);

    dynamic OthersInGroups(IList<string> groupNames);

}
```

```
public interface IHubConnectionContext
{
    [Dynamic]
    dynamic All { get; }

    dynamic AllExcept(params string[] excludeConnectionIds);
    dynamic Client(string connectionId);
    dynamic Clients(IList<string> connectionIds);
    dynamic Group(string groupName,
                  params string[] excludeConnectionIds);
    dynamic Groups(IList<string> groupNames,
                  params string[] excludeConnectionIds);
    dynamic User(string userId);
}
```

A classe que executa métodos nos clientes utiliza as propriedade **Clients** e **All**:

```
public class ExemploHub : Hub
{
    public void EnviarMensagemParaHub(string mensagem)
    {
        Clients.All.enviarMensagemParaCliente(mensagem);
    }
}
```

As possibilidades, vindas da interface **IhubConnectionContext** ou **IhubCallerConnectionContext**, usadas com mais frequência são as seguintes:

- **Clients.Caller.funcao(params)**: Envia mensagem apenas para quem chamou o Hub;
- **Clients.Others.funcao(params)**: Envia para todos, exceto para quem chamou o Hub;
- **Clients.Client(Context.ConnectionId) .funcao(params)**: Envia para um determinado cliente;
- **Clients.AllExcept(Context.ConnectionId) .funcao(params)**: Envia para todos, exceto para o cliente identificado;
- **Clients.AllExcept(Conn1, Conn2, ...).funcao(params)**: Envia para todos, exceto para as conexões identificadas;
- **Clients.Group(nomeGrupo) .funcao(params)**: Envia para todos de um grupo. Para adicionar a conexão atual a um grupo, é necessário usar a propriedade **Groups** herdada da classe **Hub**:

```
public class ExemploHub : Hub
{
    public void Inicio()
    {
        this.Groups.Add(Context.ConnectionId, "Administradores");
    }
}
```

- **Clients.User(UserId) .funcao(params)**: Envia para um usuário específico. Dados do usuário atual podem ser obtidos por meio de **this.Context.User**;
- **Clients.Users(array de strings) .funcao(params)**: Envia para diversos usuários.

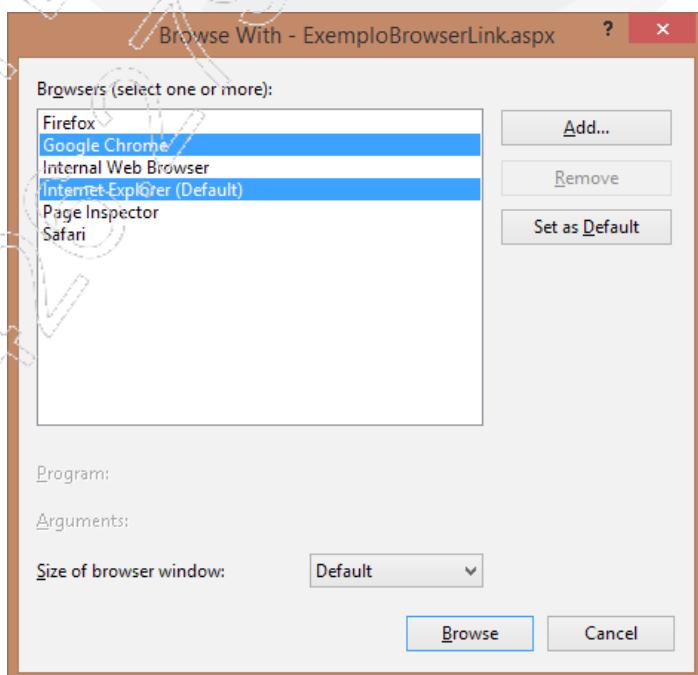
Grande parte das informações sobre uma conexão podem ser obtidas por meio da propriedade **Context** da classe **Hub**, que é do tipo **HubCallerContext**:

```
public class HubCallerContext
{
    public string ConnectionId { get; }
    public INameValueCollection Headers { get; }
    public INameValueCollection QueryString { get; }
    public IRequest Request { get; }
    public IDictionary<string, Cookie> RequestCookies { get; }
    public IPrincipal User { get; }
}
```

Por exemplo, para obter o nome do usuário usamos o seguinte código:

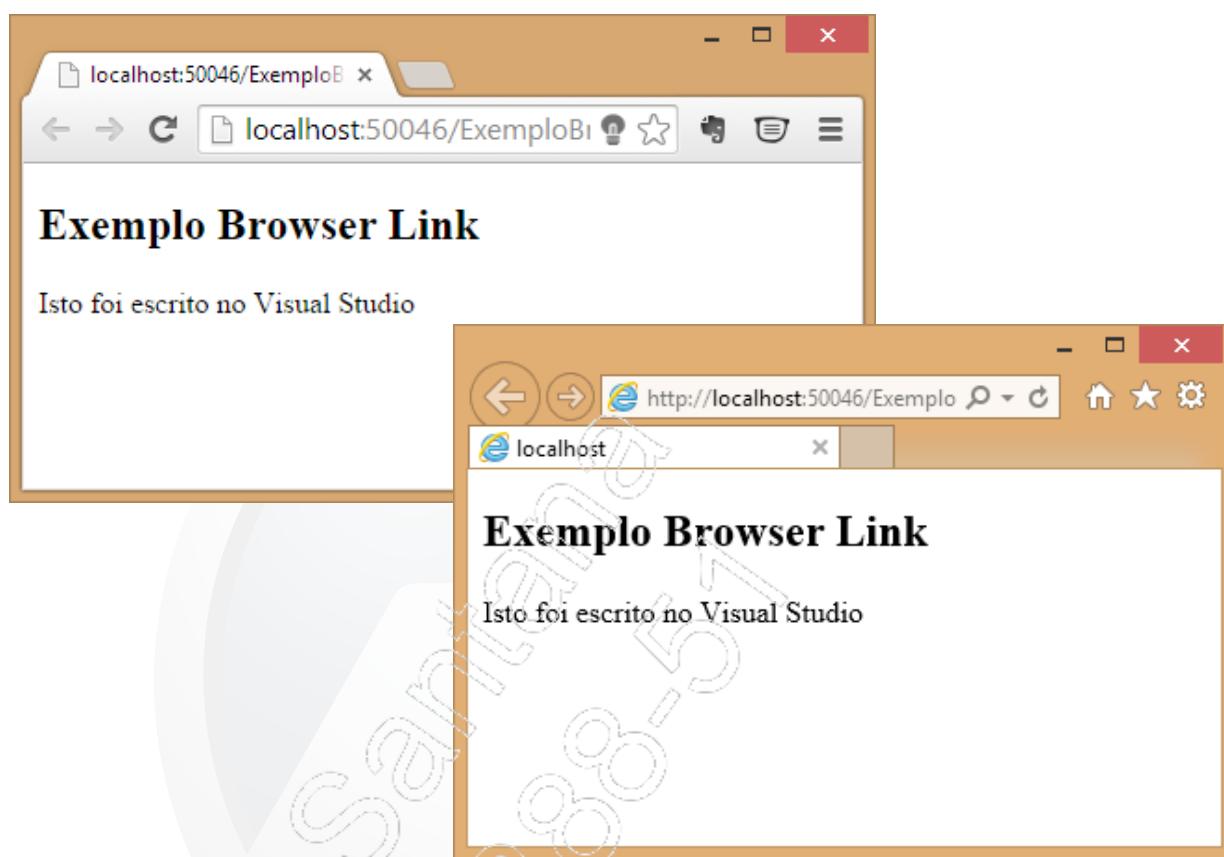
```
string usuario=Context.User.Identity.Name
```

O modelo SignalR tem muitas aplicações, principalmente aplicações financeiras, chats, jogos, entre outras que necessitam de atualização constante. O próprio Visual Studio utiliza os mesmos recursos do SignalR para atualizar a visualização das páginas ASP.NET em diversos navegadores, e possibilita, ainda, que uma alteração de código HTML seja feita tanto no Visual Studio quanto no navegador. Esse recurso é chamado de **Browser Link**. Para acioná-lo, é necessário entrar na opção **File / Browse With** e escolher os navegadores:

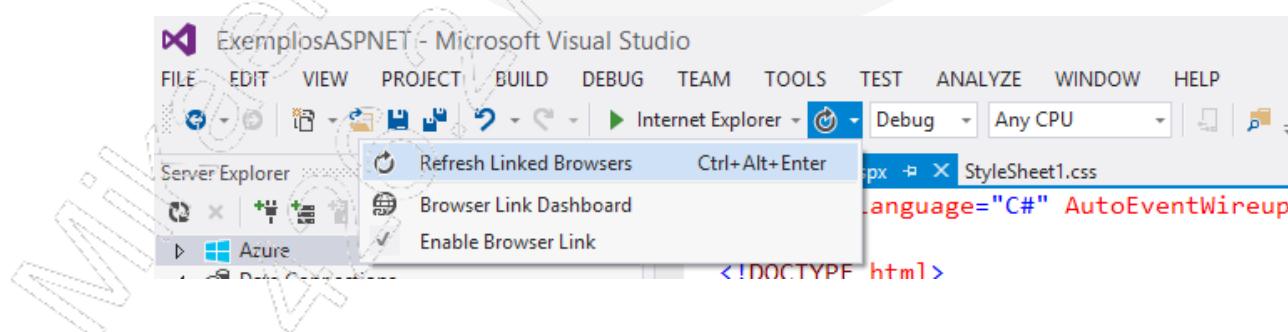


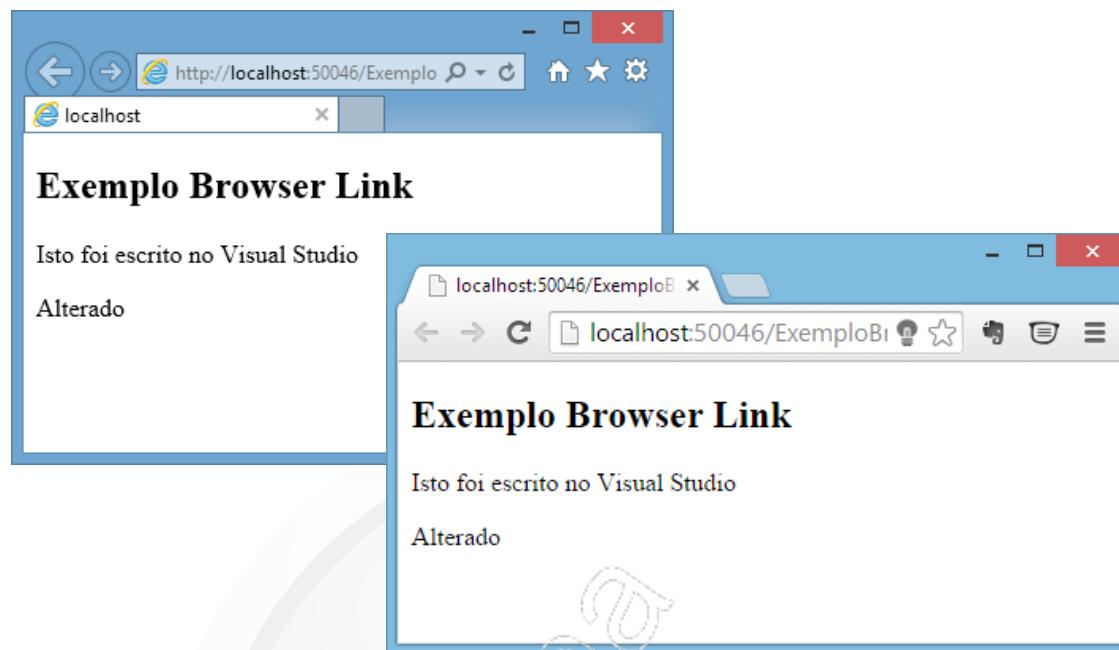
Visual Studio 2015 - ASP.NET com C# Recursos Avançados

A página é renderizada nos navegadores selecionados e estes são conectados ao Visual Studio como **Clients** de um **Hub** disponibilizado para esse fim.

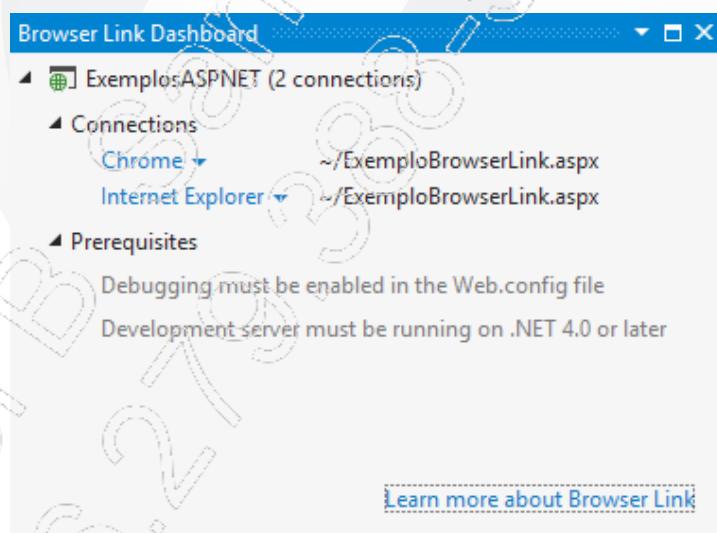


Ao atualizar a página e clicar em **Refresh Linked Browsers** na barra de ferramentas, os browsers são atualizados ao mesmo tempo.





A janela **Browser Link Dashboard** exibe os browsers que estão conectados.



Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- SignalR é um framework para implementação de comunicação em tempo real nas aplicações .NET;
- WebSockets é a principal tecnologia que permite a comunicação bidirecional entre cliente e servidor;
- SignalR utiliza o conceito de Hub, que é uma classe que controla a conexão;
- Para iniciar a implementação, é necessário criar uma classe derivada da classe **Hub**, que está no namespace **Microsoft.AspNet.SignalR**;
- É necessário, na raiz do site, criar um arquivo de configuração OWIN, mapeando o endereço do SignalR;
- A classe derivada da classe **Hub** pode ter métodos públicos que serão chamados pelos clientes;
- A classe derivada da classe **Hub** pode chamar funções definidas nos clientes, por meio das propriedades **Clients** e **All/Other/Client/Group**;
- Os scripts em JavaScript contidos nas páginas podem chamar os métodos públicos expostos na classe derivada da classe **Hub** e devem definir as funções que podem ser chamadas a partir do **Hub**;
- A propriedade **Context** retorna informações sobre a conexão atual na classe **Hub**;
- O SignalR utiliza uma convenção de letras em **PascalCase** para os nomes de métodos na classe **Hub** e **camelCase** para funções correspondentes no código em JavaScript da página. Essa convenção pode ser alterada usando o atributo **HubName**.

3

SignalR

Teste seus conhecimentos

Mikael B
426.279.9857
Santana



IMPACTA
EDITORA

1. Qual o nome do protocolo que torna a comunicação bidirecional real possível?

- a) HTTP
- b) AJAX
- c) iFrame
- d) OWIN
- e) WebSockets

2. Para implementar o SignalR, um dos passos é criar uma classe que é derivada de qual classe base?

- a) Microsoft.AspNet.SignalR.Hub
- b) Microsoft.AspNet.SignalR.Proxy
- c) Microsoft.AspNet.Identity
- d) System.Web.Hub
- e) System.net.SignalR.Protocol

3. Qual é a URL padrão do script que é criado dinamicamente quando implementada uma comunicação SignalR entre cliente e servidor?

- a) signalr/proxy
- b) signalr/web/api
- c) signalr/hubs
- d) signalr/server
- e) signalr/web.services

4. Qual propriedade deve ser usada para chamar o método JavaScript de todos os clientes, exceto quem chamou o servidor?

- a) Clients.Caller.funcao(...)
- b) Clients.Sender.funcao(...)
- c) Clients.All.ExceptThis.funcao(...)
- d) Clients.Others.funcao(...)
- e) Clientes.All.ButCaller.funcao(...)

5. Como se chama o recurso do Visual Studio que utiliza a tecnologia SignalR para atualizar instantaneamente diversos browsers quando o código-fonte de uma página muda?

- a) Degub
- b) Trace
- c) Browser SignalR
- d) BrowserLink
- e) Page Inspector

SignalR

Mãos à obra!

3

Mikael B
426.279.3857
Santana



IMPACTA
EDITORA

Laboratório 1

A – Criando um protótipo de sistema de leilão on-line

Neste laboratório, você criará um protótipo de sistema de leilão on-line, usando SignalR para atualizar todos os participantes. Um administrador inicia um leilão registrando um produto com foto, lance mínimo e nome do produto, além de poder enviar mensagens. Já os participantes assistem e participam do leilão em tempo real.

As telas serão as seguintes:

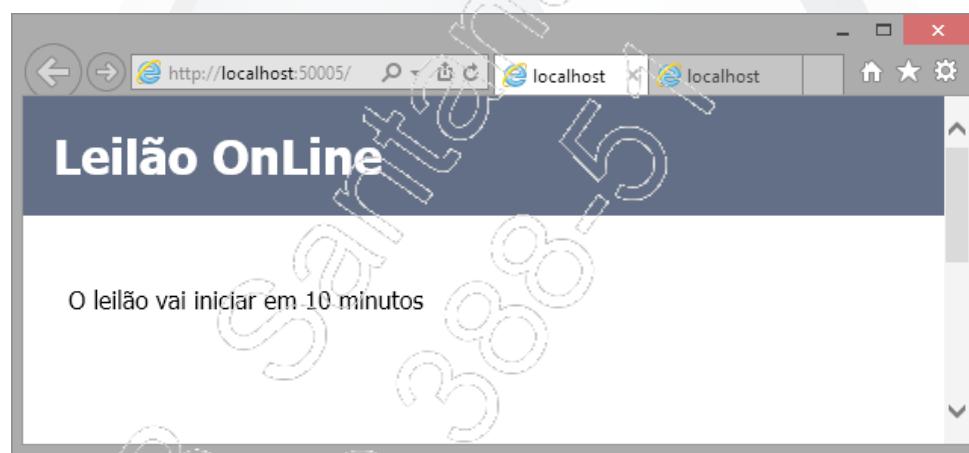
- O participante visualiza esta tela ao entrar:



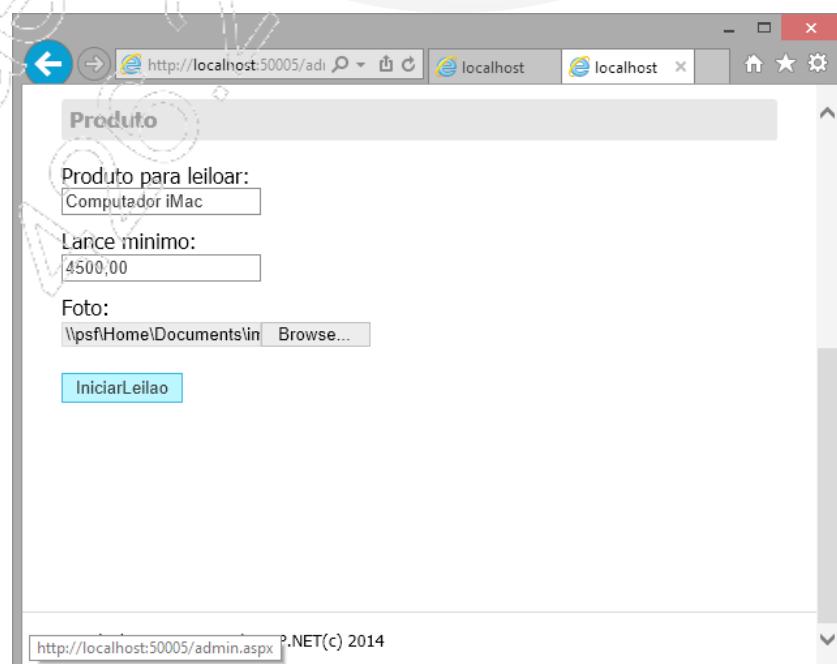
- O administrador envia uma mensagem ao entrar:



- A mensagem é exibida aos participantes:



- O administrador inicia o leilão registrando um produto:

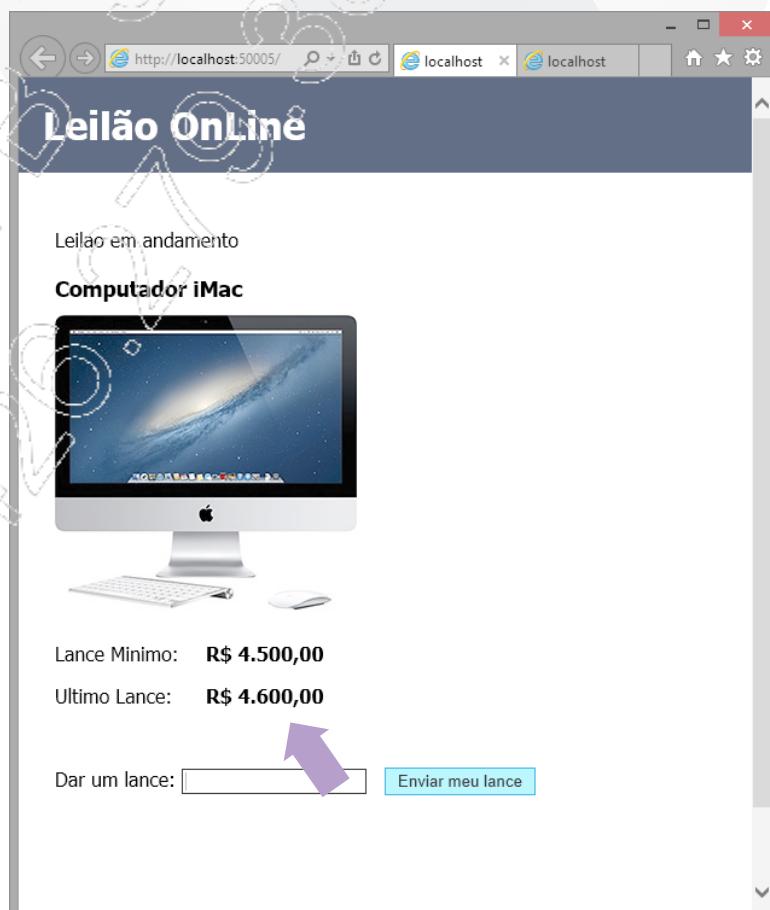


Visual Studio 2015 - ASP.NET com C# Recursos Avançados

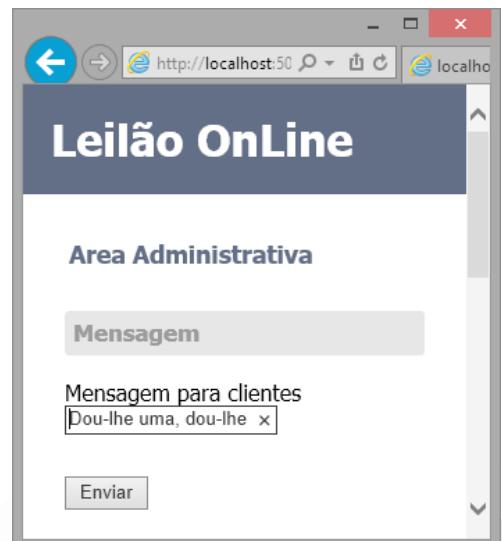
- Todos os participantes visualizam o produto ao começar o leilão:



- Os participantes podem fazer um lance cujo valor aparecerá no campo Último Lance, na tela do produto:



- O administrador pode enviar mensagens enquanto os participantes estão interagindo com o sistema:



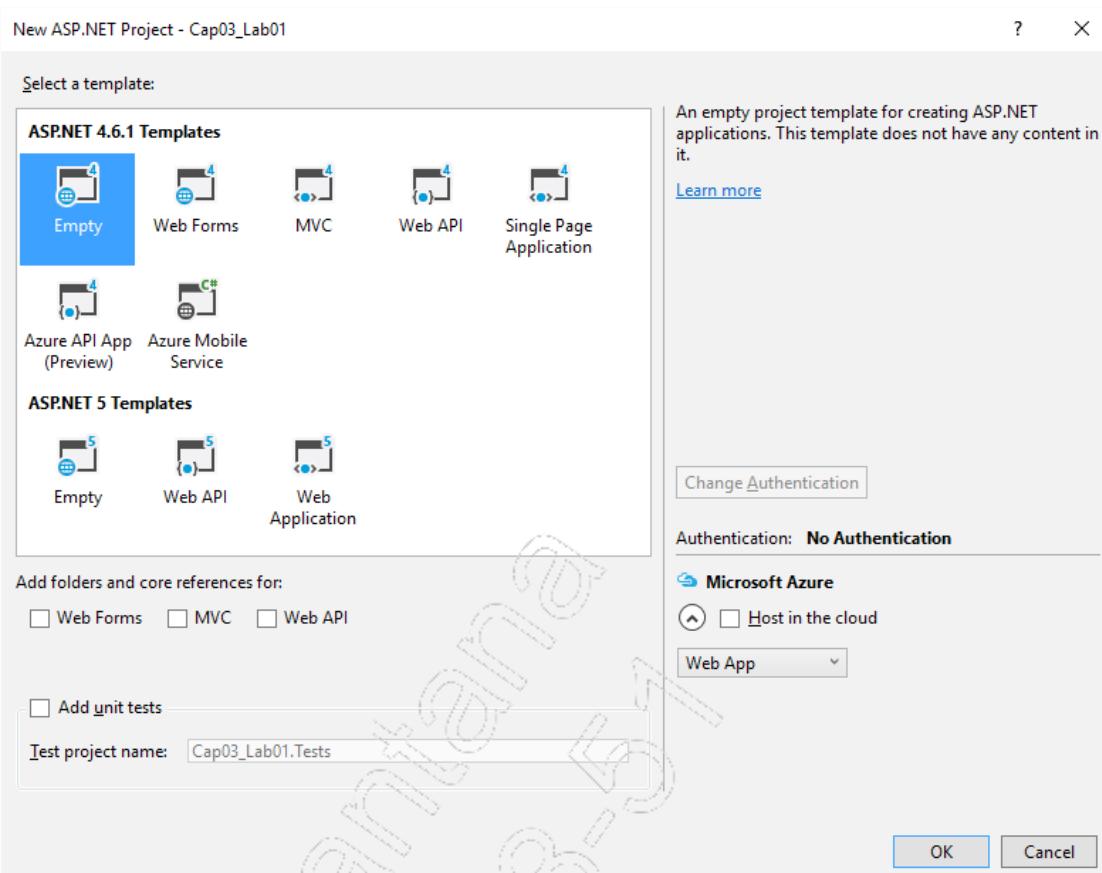
- As mensagens são exibidas para todos os participantes:



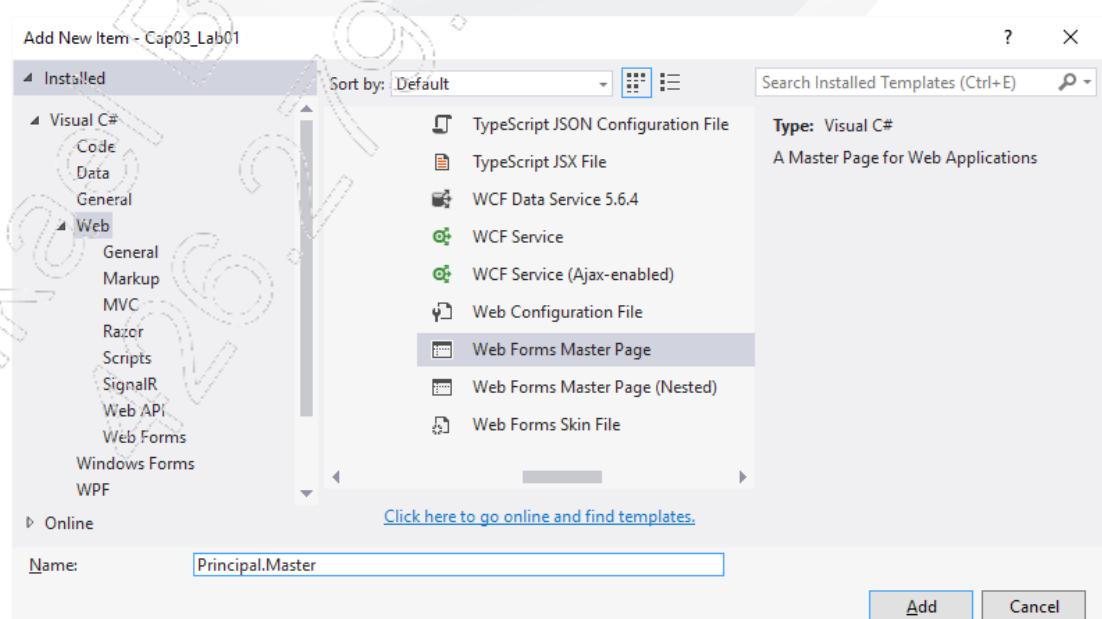
Visual Studio 2015 - ASP.NET com C# Recursos Avançados

Para isso, siga o passo a passo adiante:

1. Crie um novo projeto vazio (.NET 4.6) chamado Cap03_Lab01:



2. Adicione uma Web Forms Master Page ao seu projeto, chamada Principal.Master;



3. Adicione uma folha de estilos chamada **estilos.css**. Associe-a à Master Page;

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <link href="estilos.css" rel="stylesheet" />
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:ContentPlaceHolder ID="ContentPlaceHolder1"
runat="server">
                </asp:ContentPlaceHolder>
            </div>
        </form>
    </body>
</html>
```

4. Defina, no corpo da Master Page, as seguintes áreas: **Header**, **conteúdo** e **Footer**. Inicie com **Header**:

```
<header>
    <h1>Leilão OnLine</h1>
</header>
```

5. Abaixo do **Header**, defina a seção de conteúdo que engloba o Web Control **ContentPlaceholder**:

```
<section class="conteudo">
    <asp:ContentPlaceholder ...>
    </asp:ContentPlaceholder>
</section>
```

6. Abaixo da seção de conteúdo, defina a área de rodapé (**footer**):

```
<footer>
    <p>Desenvolvido para o curso de ASP.NET (c) 2016</p>
</footer>
```

7. Esta é a listagem completa da Master Page:

```
<%@ Master Language="C#" AutoEventWireup="true"  
    CodeBehind="Principal.master.cs"  
    Inherits="Cap03_Lab01.Principal" %>  
  
<!DOCTYPE html>  
  
<html xmlns="http://www.w3.org/1999/xhtml">  
  
    <head runat="server">  
        <title></title>  
        <link href="estilos.css" rel="stylesheet" />  
    </head>  
  
    <body>  
        <form id="form1" runat="server">  
            <div>  
                <header>  
                    <h1>Leilão OnLine</h1>  
                </header>  
  
                <section class="conteudo">  
                    <asp:ContentPlaceHolder ID="conteudo" runat="server">  
                    </asp:ContentPlaceHolder>  
                </section>  
  
                <footer>  
                    <p>Desenvolvido para o curso de ASP.NET (c) 2016</p>  
                </footer>  
            </div>  
        </form>  
    </body>  
</html>
```

8. Na folha de estilos, adicione os seguintes estilos:

```
body {  
    margin:0px;  
    font-family:Tahoma, Arial;  
}
```

```
header{  
    background-color:#445068;  
    color:#ffffff;  
    padding:20px;  
}
```

```
h1 {  
    margin:0px;  
}
```

```
h2 {  
    font-size: 110%;  
    margin:3px;  
    color:#445068;  
}
```

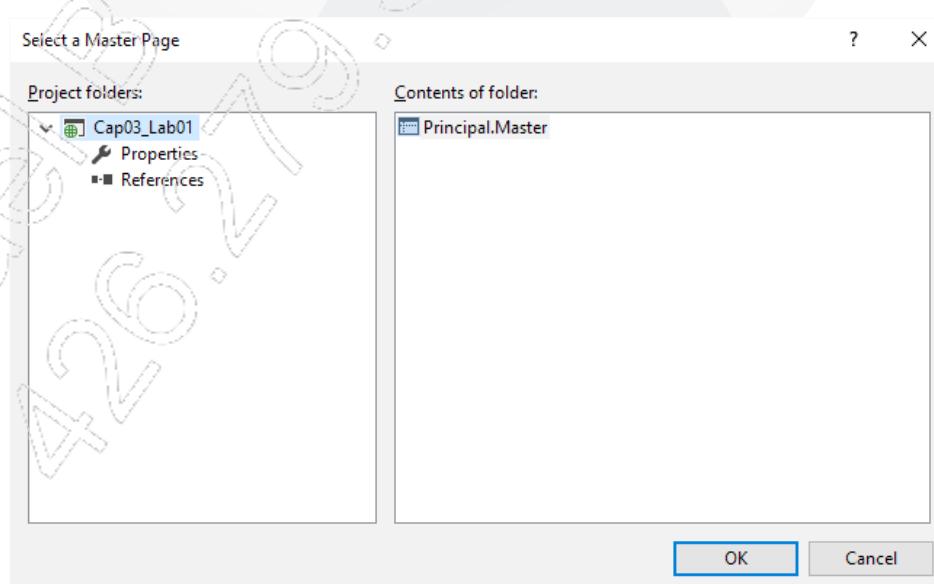
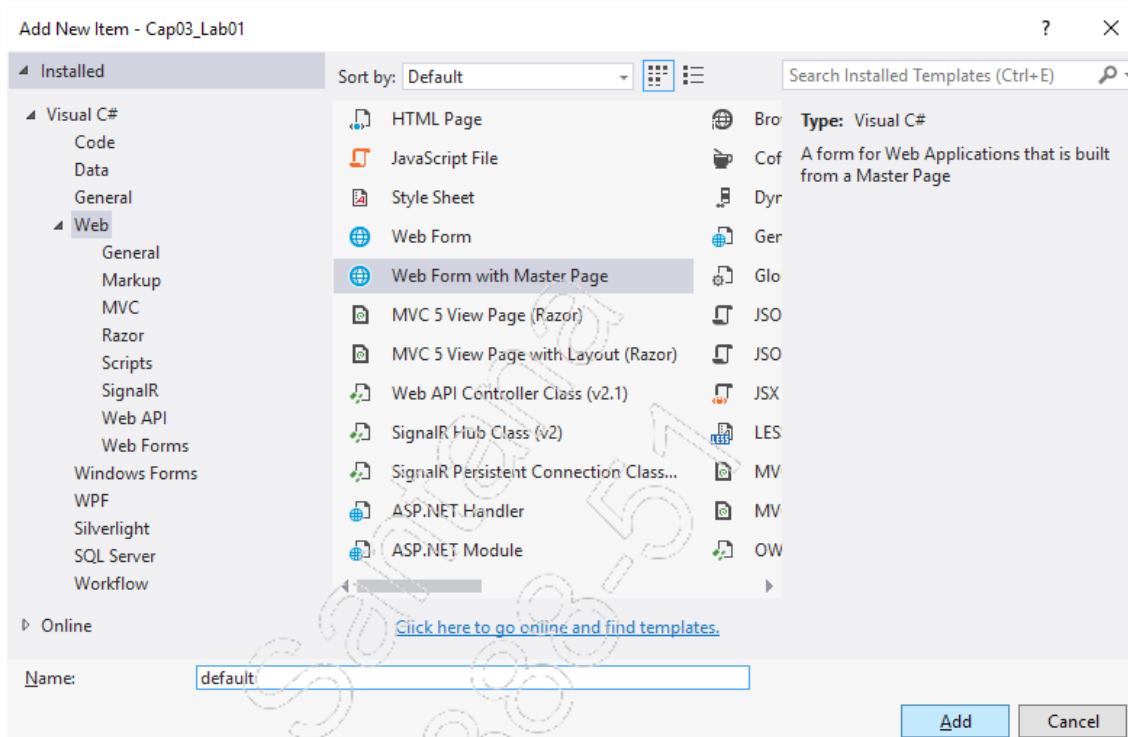
```
.conteudo {  
    padding:30px;  
    min-height:200px;  
}
```

```
footer {  
    margin-top:100px;  
    border-top:1px solid #ccc;  
}
```

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

```
footer p {  
    padding-left:10px;  
    font-size:80%;  
}
```

9. Crie uma página baseada na Master Page, chamada **default.aspx**:



10. Adicione a seguinte linha:

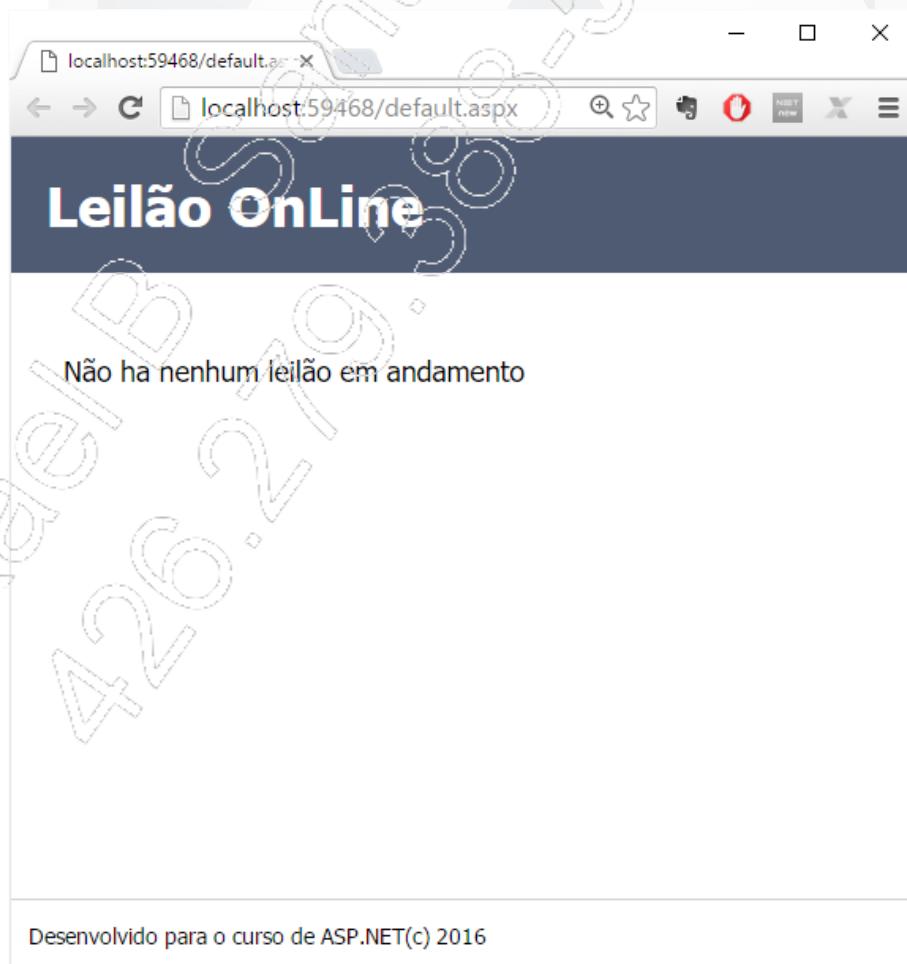
```
<%@ Page Title="" Language="C#" MasterPageFile("~/Principal.Master"
    AutoEventWireup="true" CodeBehind="default.aspx.cs"
    Inherits="Cap03_Lab01._default" %>

<asp:Content ID="Content1" ContentPlaceHolderID="conteudo"
runat="server">

    <p id="mensagem">Não ha nenhum leilão em andamento</p>

</asp:Content>
```

11. Teste o layout inicial:



12. Coloque os outros elementos na página Default.aspx:

```
<p id="mensagem">Nao ha nenhum leilao em andamento</p>

<div id="produto" style="display:none">
    <div id="produtoNome"></div>

    <div id="produtoFoto"></div>

    <div class="produtoCampo">
        <div class="produtoLegenda">Lance Minimo:</div>
        <div class="produtoValor" id="produtoMinimo"></div>
    </div>

    <div class="produtoCampo">
        <div class="produtoLegenda">Ultimo Lance:</div>
        <div class="produtoValor" id="produtoUltimoLance"></div>
    </div>

    <div id="meuLance" style="display:none">
        <label for="meuLanceText">Dar um lance:</label>
        <input id="meuLanceText" />&nbsp;
        <button type="button" id="meuLanceButton">
            Enviar meu lance</button>
    </div>
```

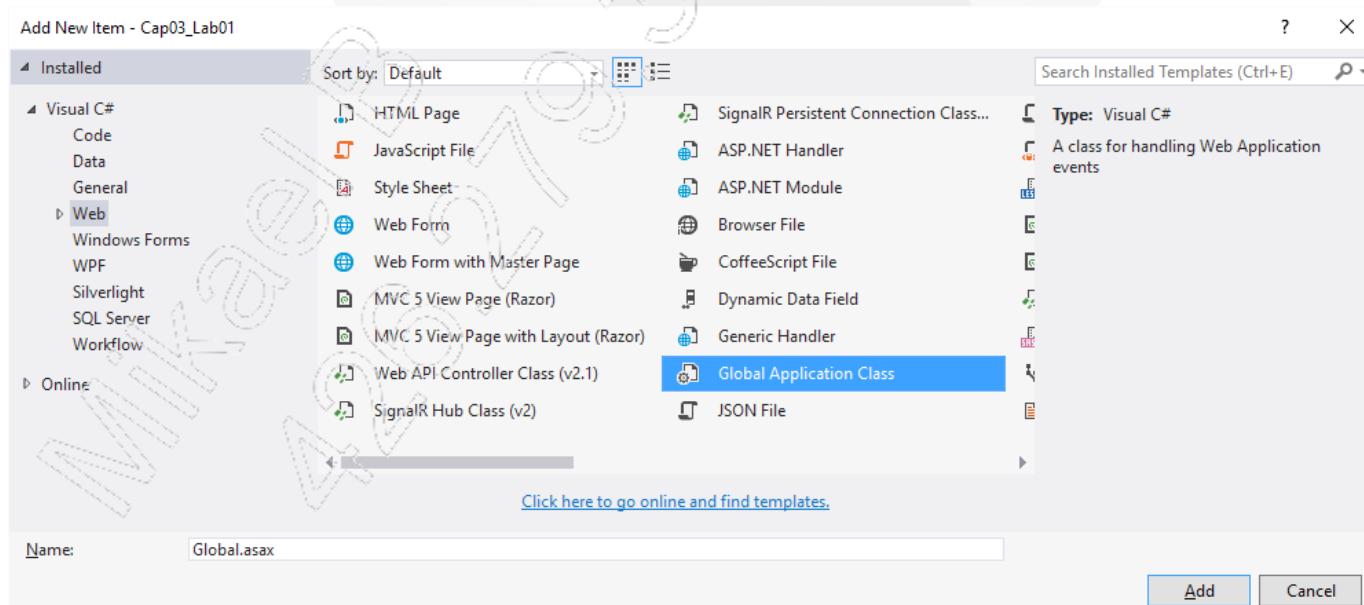
13. Adicione uma pasta chamada **Models** e, dentro dela, a classe **Leilao.cs** ao projeto. Essa classe armazena dados sobre o leilão atual:

```
[Serializable]
public class Leilao
{
    public string Produto { get; set; }
    public string Imagem { get; set; }
    public decimal LanceMinimo { get; set; }
    public decimal UltimoLance { get; set; }
    public string LanceMinimoFormatado {
        get{ return LanceMinimo.ToString("c"); } }

    public string UltimoLanceFormatado{
        get { return UltimoLance.ToString("c"); } }

}
```

14. Adicione um arquivo **Global.asax (Global Application Class)** no projeto:



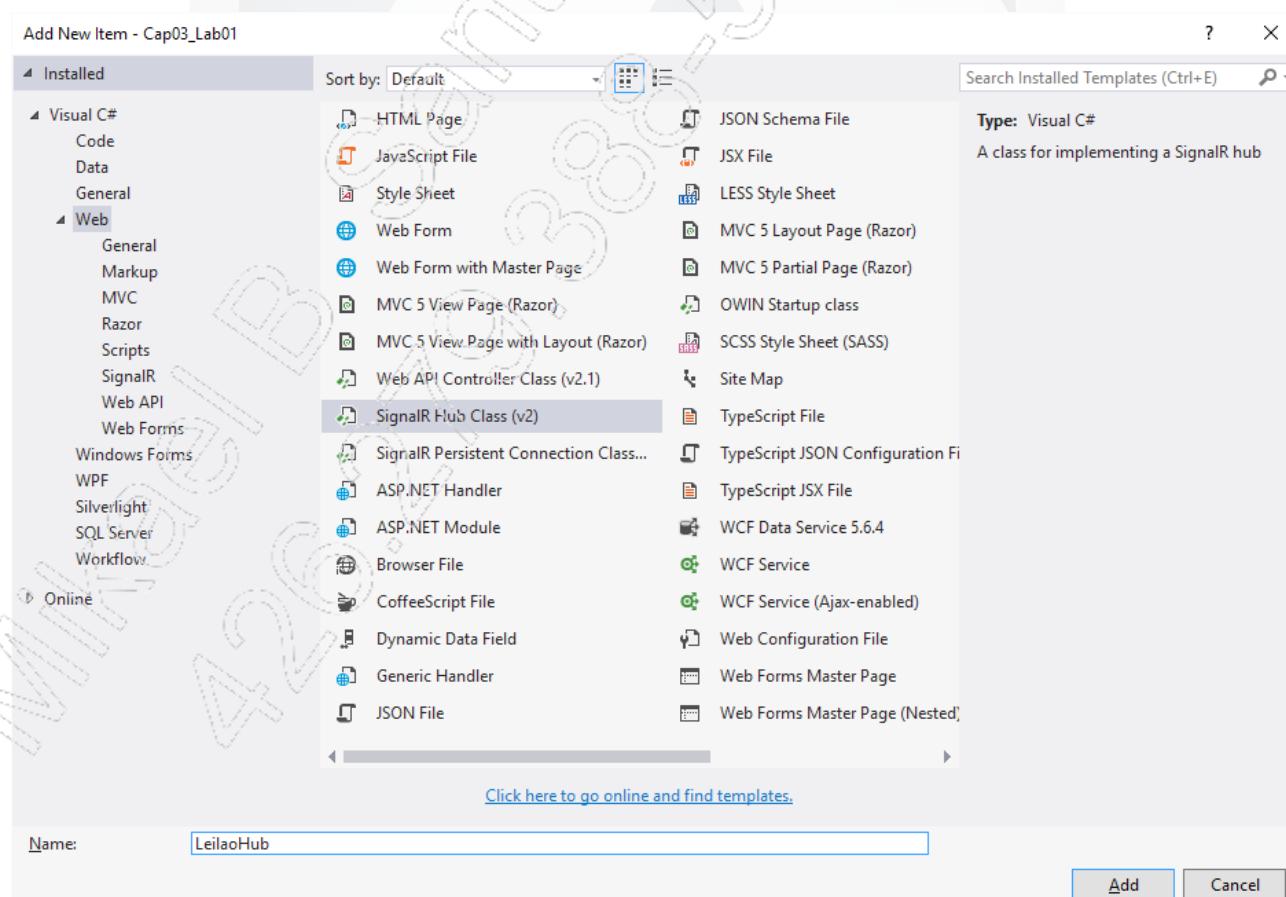
15. Adicione uma variável compartilhada para criar uma variável que armazene o último leilão:

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(...)

    {
        Application["ultimoLance"] = Convert.ToDecimal(0);
        Application["leilao"] = null;
    }

}
```

16. Adicione uma classe do tipo **SignalR Hub** chamada **LeilaoHub**:



17. Crie o primeiro método, que é o responsável por exibir mensagens do servidor:

```
namespace ASPNETIII_CAP03_LAB01
{
    public class LeilaoHub : Hub
    {
        //EnviarMensagensParaTodos
        // Envia uma mensagem para todos os clientes
        // No cliente, será executada
        // uma função em JavaScript
        // chamada receberMensagemDoServidor()
        public void EnviarMensagemParaTodos(
            string mensagem)
        {
            Clients
                .All
                .ReceberMensagemDoServidor(mensagem);
        }
    }
}
```

18. Crie o segundo método, que inicia um leilão no servidor. Todos os clientes exibem este leilão:

```
//IniciarLeilao
//Ao iniciar um leilão na área admin, este método
//deve ser chamado para exibir em todos os clientes
public void IniciarLeilao(Leilao leilao)
{
    Clients.All.exibirLeilaoAtual(leilao);
}
```

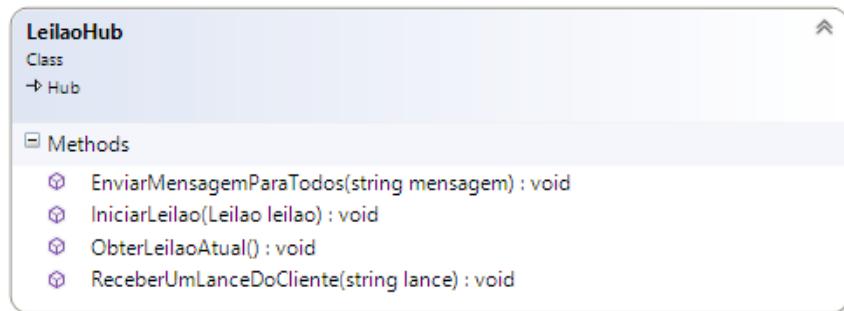
19. Crie o terceiro método, que obtém uma instância do leilão atual. Isso é necessário quando um usuário chega e está havendo um leilão. Nesse caso, é necessário exibi-lo:

```
//ObterLeilaoAtual  
//Obtém o leilão atual a partir da  
//variável global de aplicação  
//Se existir, exibe o leilão em todos os clientes  
public void ObterLeilaoAtual()  
{  
    if (HttpContext.Current  
        .Application["leilao"] != null)  
    {  
        var leilao = (Leilao)HttpContext  
            .Current  
            .Application["leilao"];  
  
        Clients.All.exibirLeilaoAtual(leilao);  
    }  
}
```

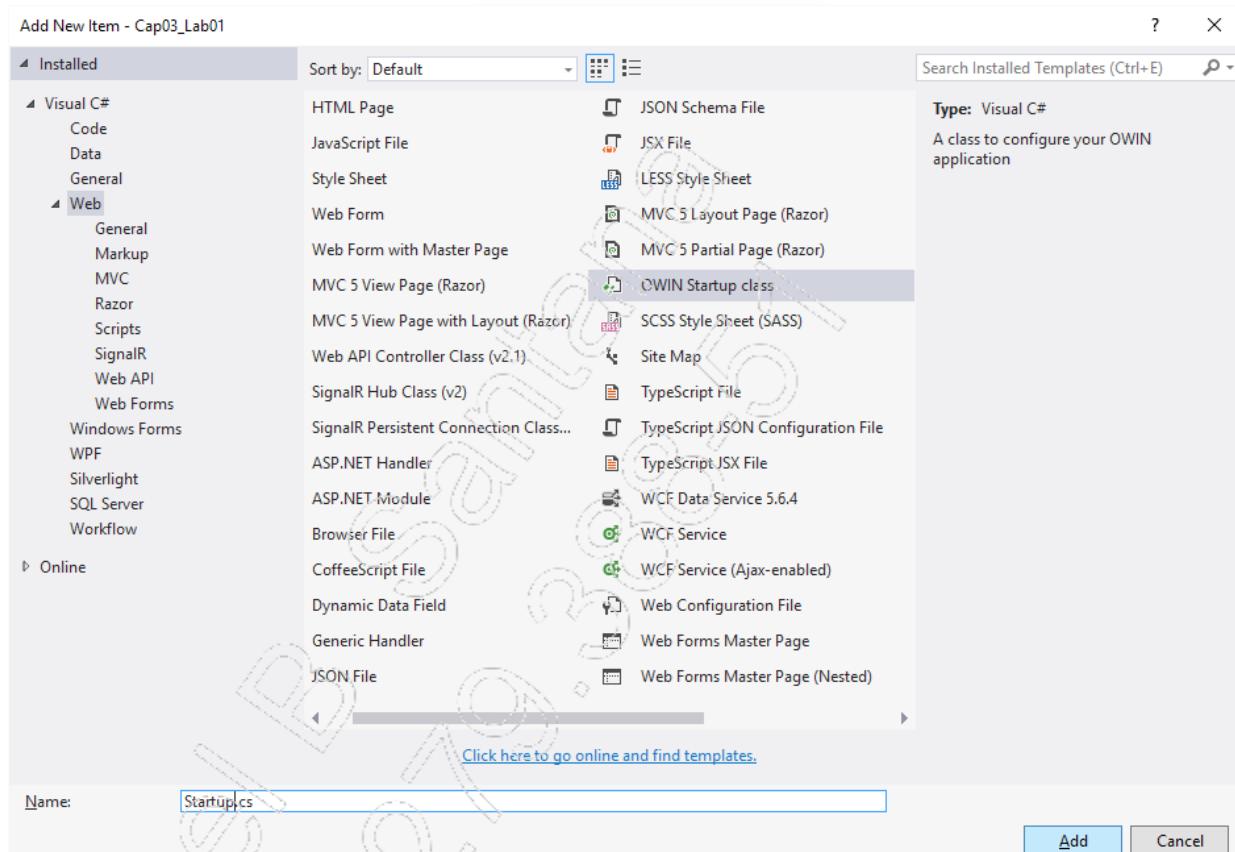
20. Crie o quarto método, que recebe um lance e exibe para todos os participantes. Fica gravado no campo **UltimoLance**:

```
//ReceberUmLanceDoCliente  
//Recebe um valor vindo do cliente  
//em relação ao leilão atual.  
public void ReceberUmLanceDoCliente(string lance)  
{  
    decimal valor = 0;  
    if (decimal.TryParse(lance, out valor))  
    {  
        Clients  
            .All  
            .exibirUltimoLance(valor.ToString("c"));  
  
        var leilao = (Leilao)HttpContext.Current  
            .Application["leilao"];  
  
        leilao.UltimoLance = valor;  
    }  
}
```

Este é o esquema completo da classe **LeilaoHub**:



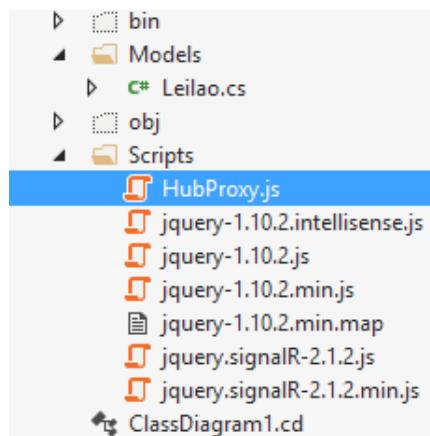
21. Insira o arquivo OWIN de configuração:



22. Insira o registro do serviço SignalR na inicialização:

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        app.MapSignalR();
    }
}
```

23. Agora, é necessário criar o JavaScript que executará as funções de atualização. Adicione um arquivo de script chamado **HubProxy.js** na pasta **Scripts**:



24. O arquivo de script define as funções que foram referenciadas nos métodos da classe **LeilaoHub**. Comece criando uma função dentro da marca do jQuery para ser executada quando a página estiver carregada:

```
$(function () {  
});
```

25. O framework **SignalR** cria o objeto **connection**, no qual é definida a classe que faz a ligação entre o cliente e o servidor. Use a convenção **camelCase** para identificar a classe:

```
$(function () {  
    var h = $.connection.leilaoHub;  
});
```

26. Crie a função que exibe uma mensagem:

```
$(function () {  
    var h = $.connection.leilaoHub;  
  
    h.client.receberMensagemDoServidor =  
        function (mensagem) {  
            $('#mensagem').text(mensagem);  
        };  
});
```

27. Crie a função que exibe um leilão:

```
$(function () {  
  
    var h = $.connection.leilaoHub;  
  
    h.client.receberMensagemDoServidor =  
        function (mensagem) {  
            $('#mensagem').text(mensagem);  
        };  
  
    h.client.exibirLeilaoAtual = function (leilao) {  
        $('#mensagem').text('Leilão em andamento');  
  
        $('#produto').show();  
  
        $('#produtoNome').text(leilao.Produto);  
  
        $('#produtoFoto').html('');  
  
        $('#produtoMinimo').text(  
            leilao.LanceMinimoFormatado);  
  
        $('#ultimoLance').text(  
            leilao.UltimoLanceFormatado);  
  
        $('#meuLance').show();  
    };  
});
```

28. Crie a função que exibe o último lance, abaixo da função anterior:

```
h.client.exibirUltimoLance = function (ultimoLance) {  
    $('#ultimoLance').text(ultimoLance);  
}
```

29. Crie o método que chama o servidor por meio do evento de um botão, para enviar o lance:

```
function enviarLanceParaHub() {  
    h.server.receberUmLanceDoCliente(  
        $('#meuLanceText').val());  
  
    $('#meuLanceText').val('').focus();  
}  
  
function addEventoClick() {  
    $('#meuLanceButton').click(enviarLanceParaHub);  
    h.server.obterLeilaoAtual();  
}
```

30. Crie o último item do script, em que o Hub é inicializado. Quando finalizado esse processo inicial, o método **addEventoClick** é executado para associar o método **enviarLanceParaHub** ao evento **Click** do botão:

```
$.connection.hub.start().done(addEventoClick);
```

Veja, a seguir um resumo do script:

```
$(function () {  
  
    var h = $.connection.leilaoHub;  
    h.client.receberMensagemDoServidor ...  
    h.client.exibirLeilaoAtual...  
    h.client.exibirUltimoLance ...  
    function enviarLanceParaHub() ...  
    function addEventoClick() ...  
    $.connection.hub.start().done(addEventoClick);  
  
});
```

31. Na Master Page, insira as referências aos scripts:

```
<head runat="server">

    <title></title>

    <link href="estilos.css" rel="stylesheet" />

    <script src="Scripts/jquery-1.10.2.js"></script>

    <script src="Scripts/jquery.signalR-2.1.2.js"></script>

    <script src="signalr/hubs"></script>

</head>
```

32. Na página **Default**, insira a referência ao script:

```
...

<asp:Content ID="Content1" ContentPlaceHolderID="conteudo"
runat="server">

    <script src="Scripts/HubProxy.js"></script>
```

```
<p id="mensagem">Não ha nenhum leilão em andamento</p>

<div id="produto" style="display: none">

    <div id="produtoNome"></div>
```

...

33. A folha de estilos precisa de alguns itens a mais. Configure os seguintes itens:

```
body { margin:0px; font-family:Tahoma, Arial; }

header{
    background-color:#445068;color:#ffffff; padding:20px;
}

h1 { margin:0px; }

h2 { font-size: 110%; margin:3px; color:#445068; }

.conteudo { padding:30px; min-height:200px; }

.legenda { display:block; }

.botao { margin-top:20px; }

.campo { display: block; margin-bottom:10px; }

footer { margin-top:100px; border-top:1px solid #ccc; }

footer p {padding-left:10px; font-size:80%; }

.adminItem { padding-top:10px; padding-bottom:30px; }

h3 { font-size:105%; padding:6px; color:#808080;
    background-color:#e0dede; border-radius:3px; }

#produto { margin-top:20px; margin-bottom:50px; }

#produtoNome { font-weight:bold; font-size:110%;
    margin-bottom:10px; margin-top:10px; }

#produtoFoto { margin-bottom:20px; }

.produtoLegenda { display: inline-block; width:120px; }

.produtoValor { display:inline-block; font-weight:bold; }

.produtoCampo { display:block; margin-bottom:15px; }
```

34. Insira uma página baseada na Master Page chamada **Admin.aspx** com o código HTML listado adiante. Essa página é responsável por fazer o processo de criar os leilões e enviar mensagens:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/MasterPage.Master" AutoEventWireup="true" CodeBehind="Admin.aspx.cs" Inherits="ASPNETIII_CAP03_LAB01.Admin" %>

<asp:Content ID="Content1" ContentPlaceHolderID="conteudo" runat="server">

    <h2>Area Administrativa</h2>

    <!-- Envio de Mensagens -->
    <div class="adminItem">

        <h3>Mensagem</h3>

        <asp:Label runat="server"
            AssociatedControlID="mensagemTextBox"
            Text="Mensagem para clientes"
            CssClass="legenda"></asp:Label>

        <asp:TextBox CssClass="campo"
            runat="server"
            ID="mensagemTextBox"></asp:TextBox>

        <asp:Button CssClass="botao"
            runat="server"
            ID="enviarButton" Text="Enviar" />

    </div>
```

```
<!-- Envio de Mensagens -->
<div class="adminItem">

    <h3>Produto</h3>

    <asp:Label runat="server"
        AssociatedControlID="produtoTextBox"
        Text="Produto para leiloar:"
        CssClass="legenda"></asp:Label>

    <asp:TextBox CssClass="campo" runat="server"
        ID="produtoTextBox"></asp:TextBox>

    <asp:Label runat="server"
        AssociatedControlID="lanceMinimoTextBox"
        Text="Lance minimo:"
        CssClass="legenda"></asp:Label>

    <asp:TextBox CssClass="campo"
        runat="server"
        ID="lanceMinimoTextBox"></asp:TextBox>

    <asp:Label runat="server"
        AssociatedControlID="fotoUpload"
        Text="Foto:"
        CssClass="legenda"></asp:Label>

    <asp:FileUpload runat="server" ID="fotoUpload" />

    <br />
    <br />

    <asp:Button runat="server"
        ID="iniciarLeilaoButton"
        Text="IniciarLeilao" />

</div>

</asp:Content>
```

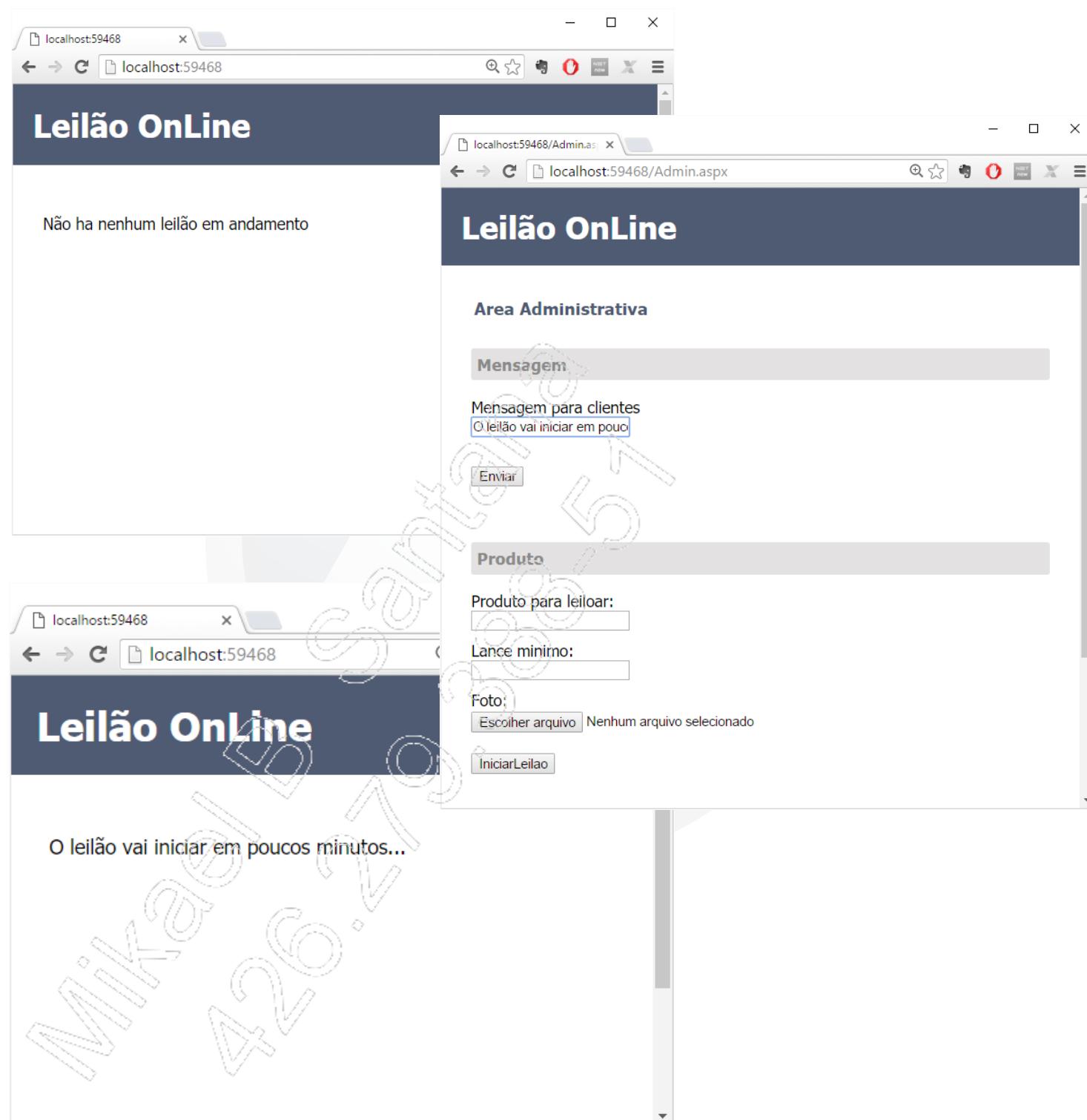
35. Agora, escreva o code-behind da página **admin** e o método do evento **Click** do botão **enviarButton**. É necessário obter o contexto atual de execução da classe **Hub**:

```
protected void enviarButton_Click(...)  
{  
    var ctx = GlobalHost  
        .ConnectionManager  
        .GetHubContext<LeilaoHub>();  
  
    ctx  
        .Clients  
        .All  
        .receberMensagemDoServidor(mensagemTextBox.Text);  
}
```

36. Crie o método que inicia um leilão. Adicione no projeto uma pasta chamada **Imagens**. A classe **GlobalHost** está no namespace **Microsoft.AspNet.SignalR**. A classe **Leilao** está no namespace **Cap01Lab01.Models**:

```
protected void iniciarLeilaoButton_Click(...)  
{  
    var ctx = GlobalHost.ConnectionManager  
        .GetHubContext<LeilaoHub>();  
  
    string nomeArquivo = fotoUpload.FileName;  
    string pathVirtual = "imagens/" + nomeArquivo;  
    string pathFisico = Server.MapPath(pathVirtual);  
    fotoUpload.SaveAs(pathFisico);  
  
    var leilao = new Leilao()  
    {  
        Produto = produtoTextBox.Text,  
        Imagem = pathVirtual,  
        LanceMinimo =  
            Convert.ToDecimal(lanceMinimoTextBox.Text),  
        UltimoLance = 0  
    };  
  
    Application["leilao"] = leilao;  
    ctx.Clients.All.exibirLeilaoAtual(leilao);  
}
```

37. Teste o sistema de mensagens:



38. Teste o leilão:

The image displays two side-by-side browser windows. The left window shows an 'Admin' interface titled 'Produto' with fields for 'Produto para leiloar:' (camera), 'Lance minimo:' (3000), and a file input for 'Foto:' containing 'Camera.jpg'. A 'IniciarLeilao' button is present. The right window shows a user interface titled 'Leilão OnLine' featuring a large image of a Canon EOS 7D camera with a 28-135mm lens. The page indicates the auction is 'Leilao em andamento' (Auction in progress) for the item 'camera'. It shows the 'Lance Minimo: R\$ 3.000,00' and 'Ultimo Lance: R\$ 0,00'. Below this is a form with 'Dar um lance:' and 'Enviar meu lance' buttons.

4

Segurança (ASP.NET Core)

- ✓ Autenticação e autorização;
- ✓ ASP.NET Identity no .NET Core.



IMPACTA
EDITORA

4.1. Introdução

Existem dois processos básicos envolvidos na segurança dos dados disponibilizados por uma aplicação: **autenticação** e **autorização**. A **autenticação** é o processo de identificar a pessoa, software ou serviço que realizará alguma tarefa no programa. Já a **autorização** é o processo de permitir ou proibir o acesso às funcionalidades do sistema pelo usuário identificado.

É comum criarmos grupos de usuários e atribuirmos permissões a esses grupos, incluindo sempre o usuário identificado em um ou mais grupos definidos no sistema. Por exemplo, um sistema de controle de uma escola teria grupos como **Professores**, **Alunos** e **Funcionários**. Já um sistema de comércio eletrônico poderia ter grupos como **Clientes**, **Fornecedores**, **Administradores** ou **Visitantes**.

Esses grupos teriam permissões diferentes. Por exemplo, um fornecedor poderia incluir um novo produto, um administrador poderia liberar o produto para venda, um cliente poderia visualizar o seu (e apenas o seu) carrinho de compras, um fornecedor não poderia visualizar informações de venda de produtos de outros fornecedores, e assim por diante.



Repare que o processo de autorização é sempre uma operação de **permitir** ou **proibir** determinadas ações.

4.2. Autenticação e autorização: História

A Microsoft implementou diversos modelos de autenticação e de autorização desde o surgimento da plataforma .NET. Em 2005, na versão 2.0 do ASP.NET, foi apresentado um modelo chamado **Membership Provider**. Esse modelo é baseado em armazenamento de informações de usuários em tabelas do SQL Server e em uma classe para gerenciamento das informações chamada **Membership**. O problema desse modelo é o forte acoplamento com o modelo relacional de banco de dados, e a única implementação fornecida era, evidentemente, a do SQL Server.

É possível implementar qualquer forma de armazenamento, mas a estrutura de dados é muito rígida. Esse fato contribuiu para que programadores e analistas implantassem modelos de autenticação personalizados, adequados à realidade de cada empresa.

Em 2010, uma versão simplificada do Membership foi introduzida, chamada **ASP.NET Simple Membership**. Ela fornecia uma versão mais limpa em relação ao armazenamento de dados. Porém, não era facilmente estendida e nem funcionava com padrões Web de terceiros. Continuava com a mesma estrutura do Membership.

Naquele mesmo ano, uma estrutura chamada **ASP.NET Universal Providers** foi criada, usando o mecanismo do bem-sucedido **Entity Framework Code First**. Apesar de ser um avanço em relação ao acoplamento de banco de dados, a infraestrutura ainda era a mesma do primeiro Membership Provider.

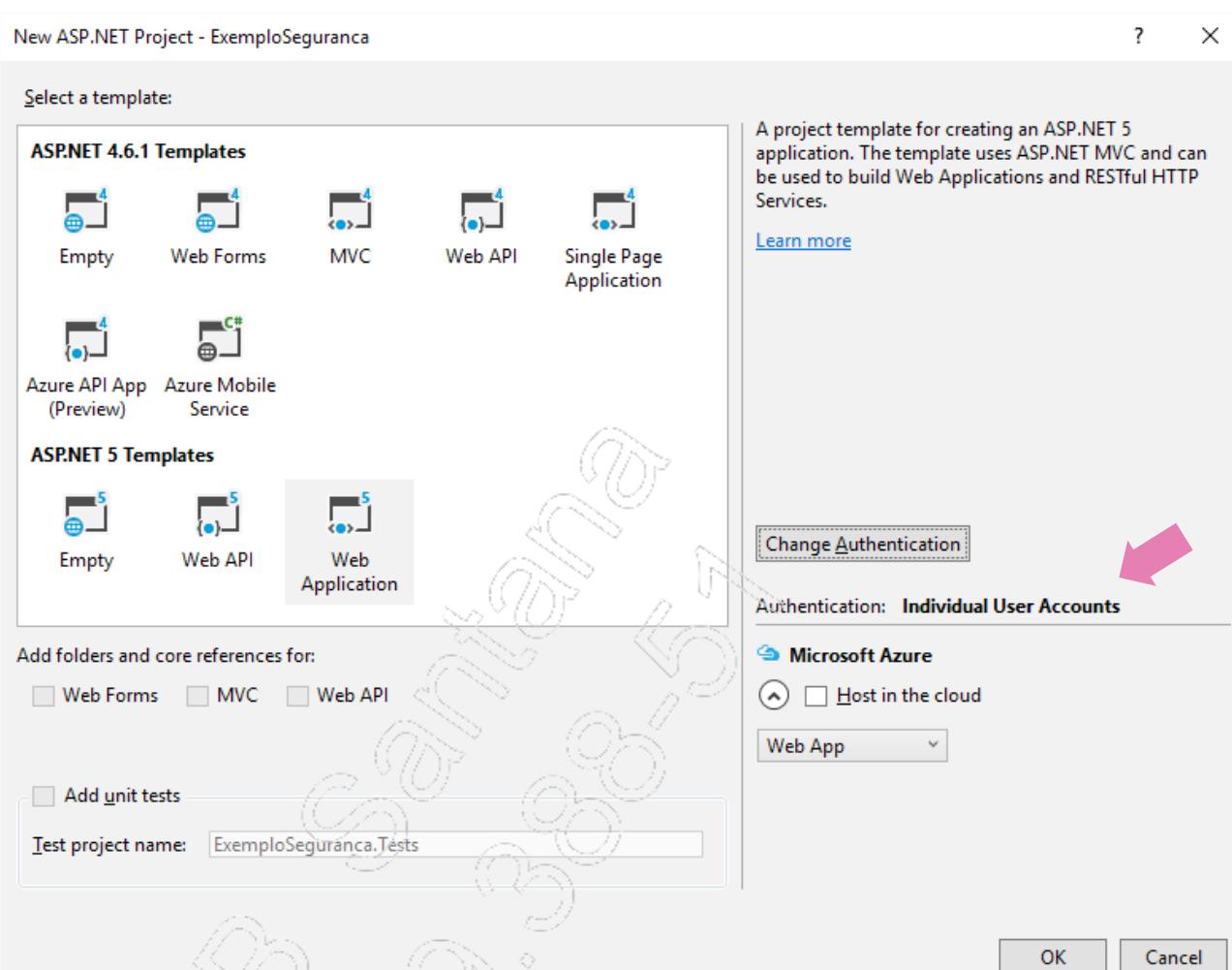
Em 2012, apostando em um modelo totalmente diferente, a Microsoft lança o **ASP.NET Identity**, que permite usar qualquer modelo de dados ou provedor de autenticação, como Facebook, Twitter ou Windows Live. O esquema de dados é totalmente controlado pela aplicação, e todo o processo é modular, permitindo substituir ou modificar qualquer componente do processo.

Toda a modelagem do ASP.NET Identity faz uso de interfaces. Existe interface para **Usuário**, **Gerenciamento de Usuário**, **Grupos de Usuários**, **Senhas** e todo tipo de informação que seja necessária à implementação de um sistema de segurança.

Em 2015, com o lançamento do ASP.NET Core, algumas alterações foram introduzidas nos componentes do ASP.NET Identity. Como o ASP.NET Core utiliza outro sistema de configuração baseado em JSON, o processo completo de implementação de uma sistema de segurança é substancialmente diferente do que era na versão 4.6 do ASP.NET.

4.3. ASP.NET Identity no .NET Core

O modelo de projeto padrão do ASP.NET Core inclui a autenticação por meio de contas individuais armazenadas em um banco de dados SQL Server.



O processo de identificação começa na classe **Startup**, no método **ConfigureServices**, que é chamado automaticamente em tempo de execução. Esta é a hora e o local para adicionar os serviços que fazem parte da aplicação.

```
class Startup
{
    ...
    public void ConfigureServices(IServiceCollection services)
    {
        ...
        services.AddIdentity< ApplicationUser, IdentityRole>()
        ...
    }
}
```

O método **AddIdentity** é um método de extensão presente na biblioteca **Microsoft.AspNetCore.Identity**, na classe **Microsoft.Extensions.DependencyInjection.IdentityServiceCollectionExtensions**.

Esse método (**AddEntity**) espera receber dois parâmetros de tipos: **User** e **Role** (usuário e papel). A declaração desse método é a seguinte:

```
public static IdentityBuilder AddIdentity<TUser, TRole>(
    this IServiceCollection services)
    where TUser : class
    where TRole : class;
```

Repare que os tipos **TUser** e **TRole** devem ser uma classe qualquer, nem mesmo uma interface está definida. A implementação é completamente livre. Na hora certa, o sistema vai instanciar essa classe. Esse processo se chama **injeção de dependência** e por isso está no namespace **xxx.DependencyInjection**.

Para o tipo **TUser** é passado o tipo **ApplicationUser**, e para o tipo **TRole**, o tipo **IdentityRole**.

A classe **ApplicationUser** está definida na aplicação, e a classe **IdentityRole**, no Entity Framework.

Adiante, o método completo **ConfigureServices**. Depois de o método **AddIdentity** ser executado, o método **AddEntityFrameworkStores** é chamado e, depois dele, o método **AddDefaultTokenProviders**.

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddEntityFramework()
        .AddSqlServer()
        .AddDbContext<ApplicationDbContext>(
            options =>options.UseSqlServer(
                Configuration[
                    "Data:DefaultConnection:ConnectionString"]));
}
```

```
services.AddIdentity< ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores< ApplicationDbContext >()
    .AddDefaultTokenProviders();

services.AddMvc();

// Add application services.
services.AddTransient< IEmailSender, AuthMessageSender >();
services.AddTransient< ISmsSender, AuthMessageSender >();

}
```

AddEntityFrameworkStores é um método de extensão da classe **IdentityBuilder**, que é o tipo retornado quando uma identidade é adicionada. Esse método espera uma parâmetro do tipo ou derivado de **DbContext**. Veja a declaração desse método de extensão:

```
namespace Microsoft.Extensions.DependencyInjection
{
    public static class IdentityEntityFrameworkBuilderExtensions
    {
        public static IdentityBuilder
            AddEntityFrameworkStores< TContext >(
                this IdentityBuilder builder)
            where TContext : DbContext;

        ...
    }
}
```

E, finalmente, o método **AddDefaultTokenProviders** é invocado na saída do método **AddEntityFrameworkStores**:

```
services.AddIdentity< ApplicationUser, IdentityRole >()
    .AddEntityFrameworkStores< ApplicationDbContext >()
    .AddDefaultTokenProviders();
```

AddDefaultTokenProviders é um método definido na classe **IdentityBuilder** e está no namespace **Microsoft.AspNet.Identity**. Token é uma informação usada para validar algumas ações, como trocar a senha ou alterar o e-mail.

O processo de identificação continua na classe **Startup**, no método **Configure**, que também é chamado automaticamente em tempo de execução. Esse método recebe uma instância da classe que implementa a interface **IApplicationBuilder**, e o método **UseIdentity()** é chamado.

```
class Startup
{
    public void Configure( IApplicationBuilder app,
                          IHostingEnvironment env
                          ILoggerFactory loggerFactory)
    {
        ...
        app.UseIdentity();
        ...
    }
}
```

O método **UseIdentity** é um método de extensão da classe **BuilderExtensions**, que está no assembly **Microsoft.AspNet.Identity**.

```
namespace Microsoft.AspNet.Builder
{
    public static class BuilderExtensions
    {
        public static IApplicationBuilder UseIdentity(this
IApplicationBuilder app);
    }
}
```

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

As seguintes classes foram usadas até o momento:

```
namespace MeuApp  
    class Startup
```

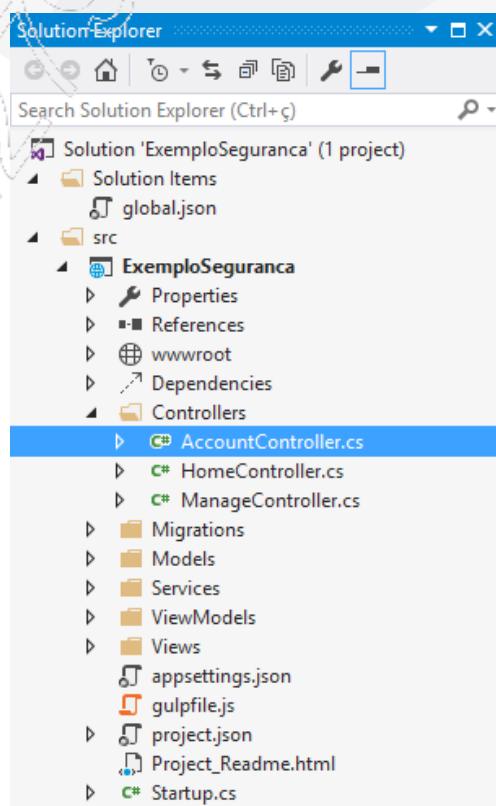
```
namespace MeuApp.Models  
    class ApplicationUser  
    class ApplicationDbContext
```

```
namespace Microsoft.AspNet.Identity.EntityFramework  
    class IdentityDbContext<TUser>  
    class IdentityRole
```

```
namespace Microsoft.Data.Entity  
    class DbContext
```

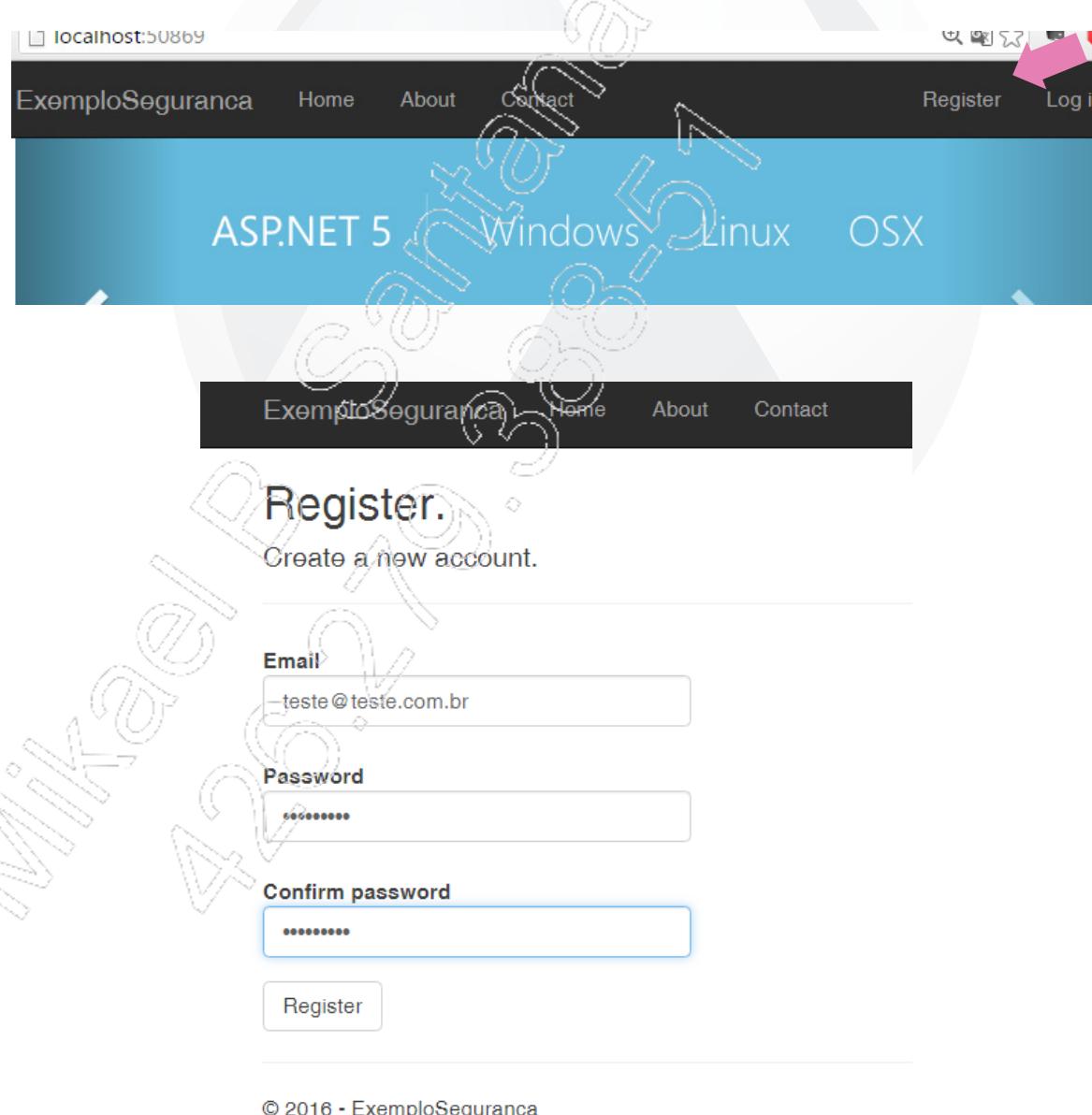
```
namespace Microsoft.AspNet.Identity  
    class IdentityBuilder  
    class BuilderExtensions
```

Para o registro de um usuário (**ApplicationUser**) é usado o método **Register** da classe **AccountController**:



```
public class AccountController : Controller  
{  
  
    ...  
    public IActionResult Register()  
    {  
        return View();  
    }  
  
    ...  
}
```

Quando o programa é executado, o item de menu **Register** direciona para a URL **Account/Register**, sendo executado o método correspondente e acionada a View:



Ao clicar no botão **Register**, o formulário HTML envia os dados usando o verbo POST, fazendo o MVC acionar o método correspondente:

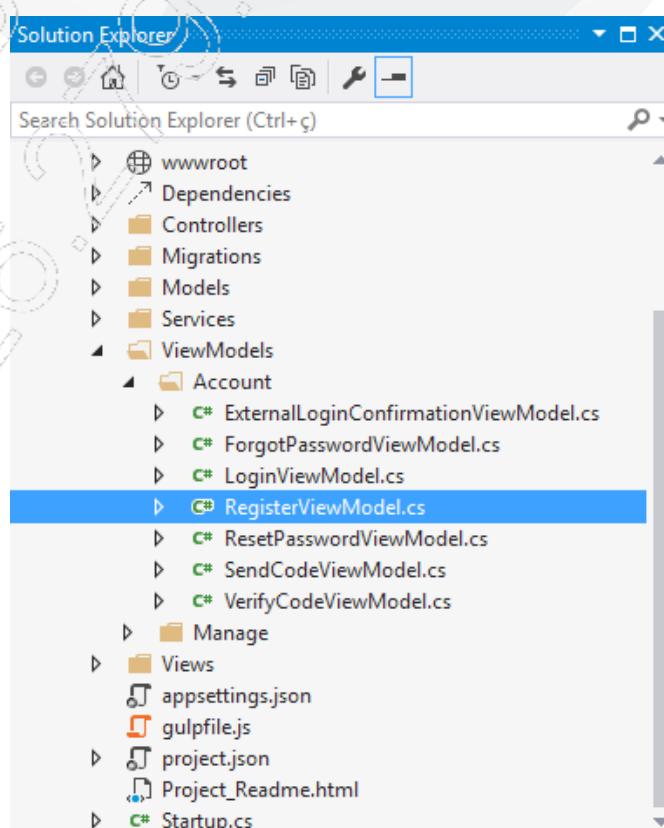
```
[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email,
                                         Email = model.Email };

        var result = await _userManager.CreateAsync(user,
model.Password);

        if (result.Succeeded) { ... (código omitido) }

        return View(model);
    }
}
```

O método **Register** recebe uma instância da classe **RegisterViewModel**, definida na pasta **ViewModels**:



A classe **RegisterViewModel** contém apenas o necessário ao cadastro de usuário: **Email**, **Senha** e **Confirmação da Senha**. As anotações foram omitidas na listagem adiante para maior clareza:

```
public class RegisterViewModel
{
    public string Email { get; set; }
    public string Password { get; set; }
    public string ConfirmPassword { get; set; }
}
```

Dentro do método **Register**, uma instância de **ApplicationUser** é criada usando os dados fornecidos pelo usuário:

```
[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    ...
    var user = new ApplicationUser { UserName = model.Email,
                                    Email = model.Email };
    ...
}
```

A linha seguinte grava os dados do usuário no banco de dados:

```
var result = await _userManager.CreateAsync(user, model.Password);
```

`_userManager` é uma variável local criada no construtor da classe **AccountController**. A classe **AccountController** recebe como parâmetro várias instâncias de classes que serão usadas para gravar o usuário, realizar a autenticação, enviar e-mail e registrar um log de atividades.

De onde vêm essas instâncias? É aí que entra a injeção de dependência. A classe **AccountController** não sabe quais classes concretas vai utilizar, mas conhece as interfaces. O mecanismo do ASP.NET é responsável por instanciar essas classes de acordo com as configurações feitas no início da aplicação, na classe **Startup**.

A classe **AccountController** não é responsável por autenticar, gravar ou enviar e-mails. Ela recebe instâncias dos responsáveis e apenas executa os dados. Esse mecanismo (injeção de dependência) é muito poderoso, pois permite interligar componentes com segurança e é a base de todo o .NET Core.

Para visualizar o processo de injeção de dependência, vamos analisar o código relativo ao envio de e-mails. Adiante, está a listagem da classe **AccountController** apenas com o construtor. Veja a declaração da interface **IEmailSender**.

```
public class AccountController : Controller
{
    private readonly UserManager< ApplicationUser > _userManager;
    private readonly SignInManager< ApplicationUser > _signInManager;
    private readonly IEmailSender _emailSender;
    private readonly ISmsSender _smsSender;
    private readonly ILogger _logger;

    public AccountController(
        UserManager< ApplicationUser > userManager,
        SignInManager< ApplicationUser > signInManager,
        IEmailSender emailSender,
        ISmsSender smsSender,
        ILoggerFactory loggerFactory)
    {
        _userManager = userManager;
        _signInManager = signInManager;
        _emailSender = emailSender;
        _smsSender = smsSender;
        _logger = loggerFactory.CreateLogger< AccountController >();
    }
}
```

O que faz uma instância que implementa a interface **IEmailSender** cair dentro do construtor da classe **Account** é este método da classe **Startup**:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        ...
        services.AddTransient<IEmailSender,
AuthMessageSender>();
        ...
    }
}
```

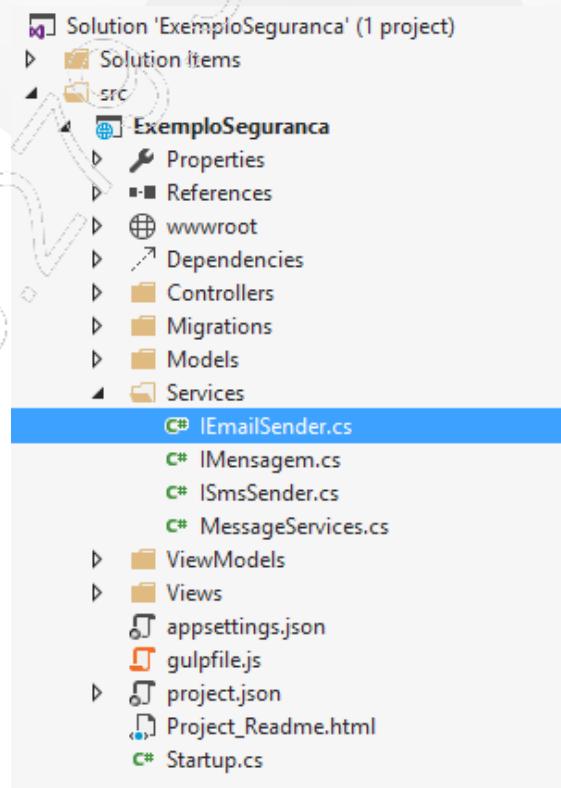
O método **ConfigureServices** recebe como parâmetro uma classe que implementa a interface **IServiceCollection**. O objetivo dessa interface é gerenciar os componentes da aplicação e permitir que o sistema crie instâncias de classes que serão passadas para o construtor de um **Controller** se esse construtor esperar um parâmetro do tipo de algum serviço definido nessa lista.

O método **AddTransient<T1, T2>()** adiciona um serviço à coleção de serviços. É necessário passar como parâmetro a interface (**IEmailSender**) e a classe que implementa essa interface (**AuthMessageSender**).

Isso é tudo. Automaticamente, se algum **Controller** tiver um construtor que espere um parâmetro do tipo **IEmailSender**, o mecanismo do MVC vai criar uma instância da classe **EmailSender** e passar essa instância como parâmetro quando criar a classe **Controller**. É o que acontece na classe **AccountController**:

```
public class AccountController : Controller
{
    ...
    public AccountController(
        UserManager< ApplicationUser > userManager,
        SignInManager< ApplicationUser > signInManager,
        IEmailSender emailSender,
        ISmsSender smsSender,
        ILoggerFactory loggerFactory)
    {
        ...
    }
}
```

A interface **IEmailSender** e a classe que a implementa (que na verdade não tem nenhuma implementação) estão definidas na pasta **Services**:



A interface e a classe são apenas declarações, sem nenhum implementação real:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ExemploSeguranca.Services
{
    public interface IEmailSender
    {
        Task SendEmailAsync(string email, string subject,
string message);
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ExemploSeguranca.Services
{
    public class AuthMessageSender : IEmailSender, ISmsSender
    {
        public Task SendEmailAsync(string email, string
subject, string message)
        {
            return Task.FromResult(0);
        }

        public Task SendSmsAsync(string number, string message)
        {
            return Task.FromResult(0);
        }
    }
}
```

Apenas para relembrar como está definido na classe **Startup**:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        ...
        services.AddTransient<IEmailSender,
AuthMessageSender>();
        ...
    }
}
```

- **Autenticação**

Uma vez identificado o usuário, é necessário autenticá-lo. É muito comum usar cookies para a autenticação. Logo após o registro do usuário no banco de dados, ele é autenticado no sistema e um cookie de autenticação é gerado.

```
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser ....
        var result = await _userManager.CreateAsync...
        if (result.Succeeded)
        {
            await _signInManager.SignInAsync(user, isPersistent: false);
            _logger.LogInformation(3, "Usuário criado.");
            return RedirectToAction(nameof(HomeController.Index), "Home");
        }
        AddErrors(result);
    }
    return View(model);
}
```

Uma instância de **SignInManager** é criada pelo ASP.NET Core.

- **Autorização**

A autorização consiste em proibir ou permitir acesso a recursos do sistema. Nesta versão, a autorização é definida usando **DataAnnotations** nas classes de controle. O atributo **Authorize** define que apenas usuários autenticados podem acessar:

```
[Authorize]  
public class AccountController : Controller
```

Para liberar um método dessa exigência, usa-se o atributo **AllowAnonymous**:

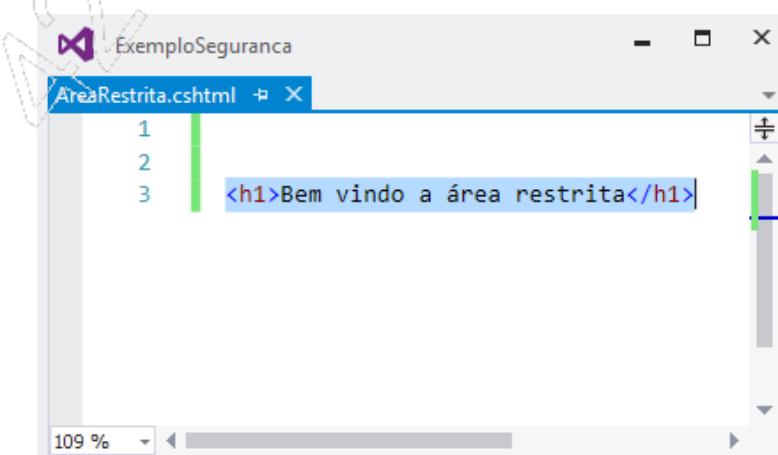
```
[AllowAnonymous]  
public IActionResult Login(string returnUrl = null)
```

- **Teste de login e autorização**

Para testar o processo de autorização, é necessário tentar entrar em uma página proibida para usuários anônimos. Depois, fazer o login e tentar novamente. Dentro da classe **AccountManager** foi criado este método:

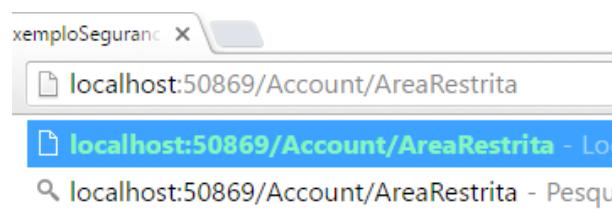
```
public ViewResult AreaRestrita()  
{  
    return View();  
}
```

E dentro da pasta **Views**, um documento razor chamado **AreaRestrita.cshtml**:



Visual Studio 2015 - ASP.NET com C# Recursos Avançados

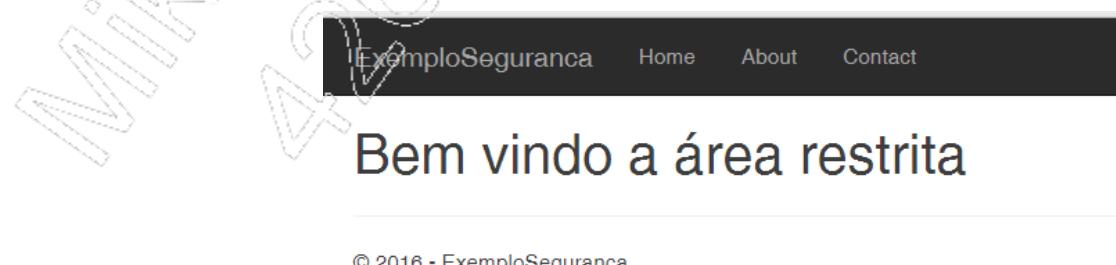
- Tentando visualizar (não é possível, sempre cai no login):



- Fazendo o login:

The screenshot shows a login page for 'ExemploSegurança'. At the top, there is a navigation bar with links to 'Home', 'About', and 'Contact'. The main content area has a heading 'Log in.' and a sub-instruction 'Use a local account to log in.' Below this, there are two input fields: 'Email' containing 'teste@teste.com.br' and 'Password' containing '*****'. There is also a checked 'Remember me?' checkbox and a 'Log in' button. To the left of the form, there is a faint watermark with the text 'Mikael B Santos' repeated multiple times. At the bottom of the page, there is a copyright notice '© 2016 - ExemploSegurança'.

- E agora está disponível:



Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Autenticação é o processo de identificar o usuário;
- Autorização é o processo de permitir ou proibir, ao usuário identificado, acesso a funcionalidades do sistema;
- ASP.NET Identity é o novo modelo da Microsoft para gerenciar a parte de segurança de aplicações Web;
- A classe **Startup.cs** é utilizada para definir as classes de autenticação e autorização de usuários;
- O método **ConfigureServices** é utilizado para registrar os componentes que serão utilizados na aplicação;
- A classe **UserManager** é responsável por adicionar, excluir ou alterar dados de um usuário;
- Usando **DataAnnotations**, é possível permitir ou negar acesso a classes ou métodos. O atributo **Authorize** proíbe o acesso a usuários não autenticados. O atributo **AllowAnonymous** permite que um usuário não autenticado tenha acesso a métodos de uma classe restrita.

4

Segurança (ASP.NET Core)

Teste seus conhecimentos

Mikael B
426.279.
57



IMPACTA
EDITORA

1. Em qual arquivo de um projeto ASP.NET Core é definido o serviço de identificação de usuário que será usado?

- a) global.asax
- b) Startup.cs
- c) project.json
- d) web.config
- e) setup.ini

2. Qual é o namespace das classes do .NET Core usado para identificar um usuário?

- a) Microsoft.User
- b) Microsoft.AspNet.Identity
- c) Microsoft.AspNet.User
- d) System.Identity
- e) System.MembershipProvider

3. Qual é o atributo que deve ser colocado em uma classe para proibir o acesso a usuários não autenticados?

- a) [deny]
- b) [Allow]
- c) [Block]
- d) [Required]
- e) [Authorize]

4. Qual é a classe usada para Adicionar, Excluir ou Alterar dados de um usuário?

- a) IUser
- b) UserManager
- c) LoginUser
- d) Principal
- e) Application

5. A classe `ApplicationDbContext` pertence a qual componente?

- a) ADO.NET
- b) AspNetIdentity
- c) EntityFramework
- d) SqlConnection
- e) IdentityContext

4

Segurança (ASP.NET Core)

Mãos à obra!

Mikael Braga
426.270-57

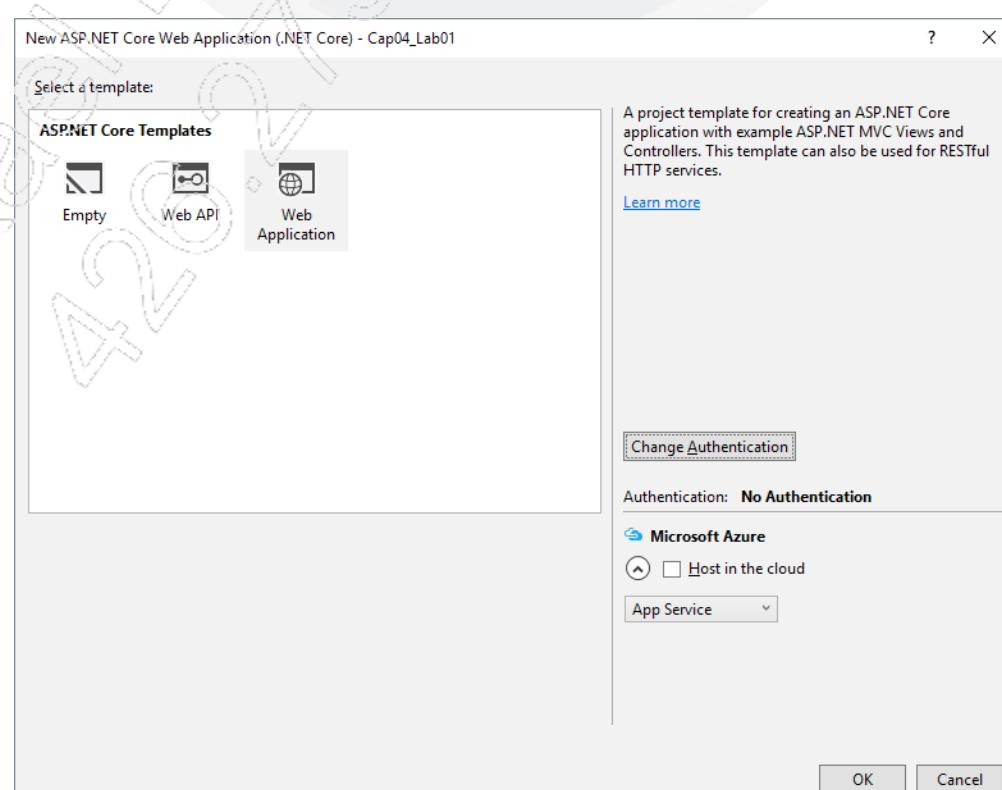
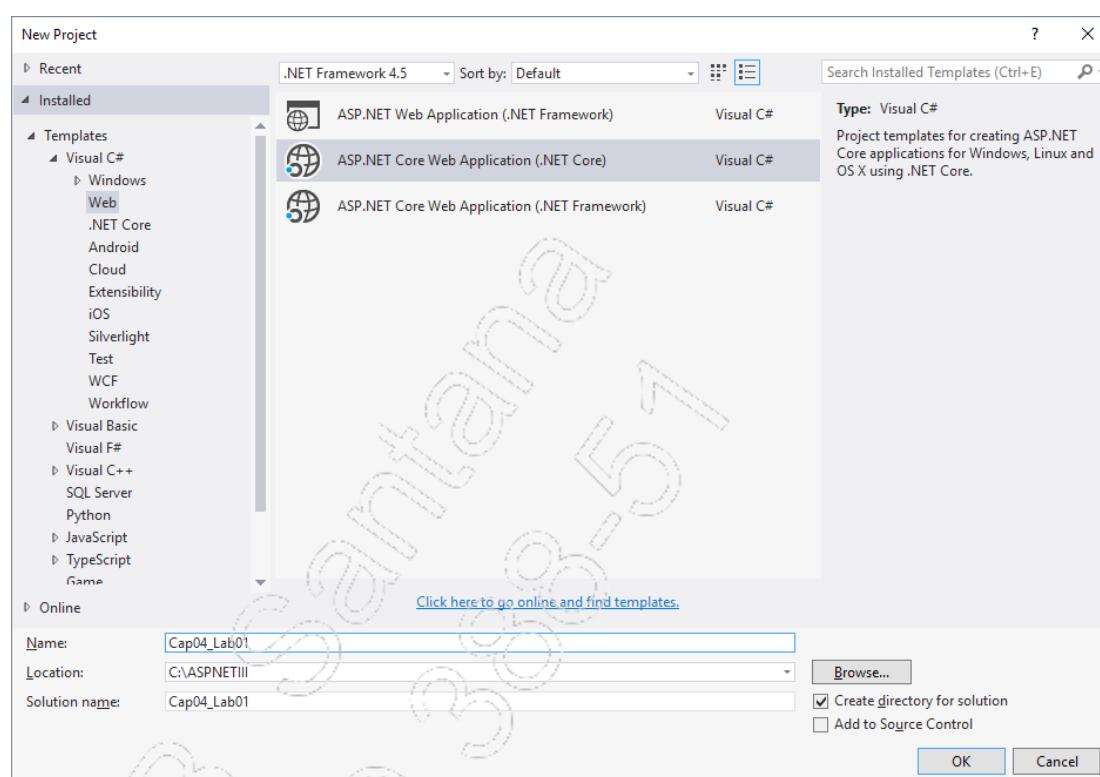


IMPACTA
EDITORA

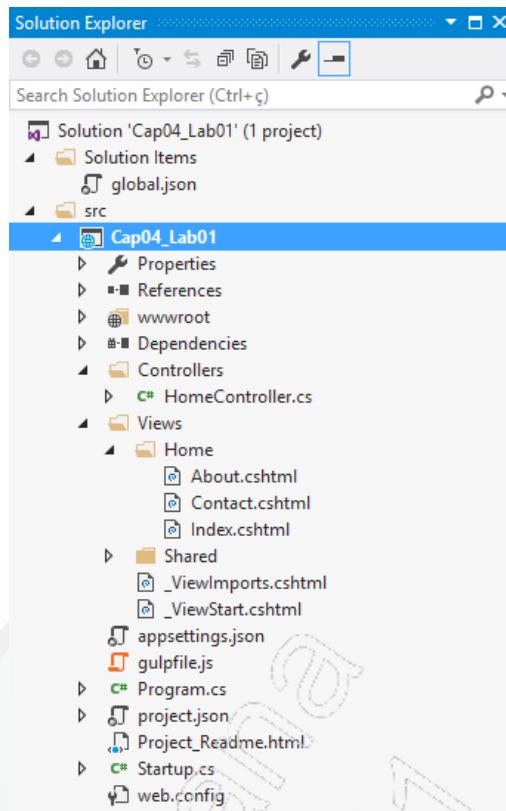
Laboratório 1

A – Criando um login simples com o ASP.NET Identity com autenticação por cookies

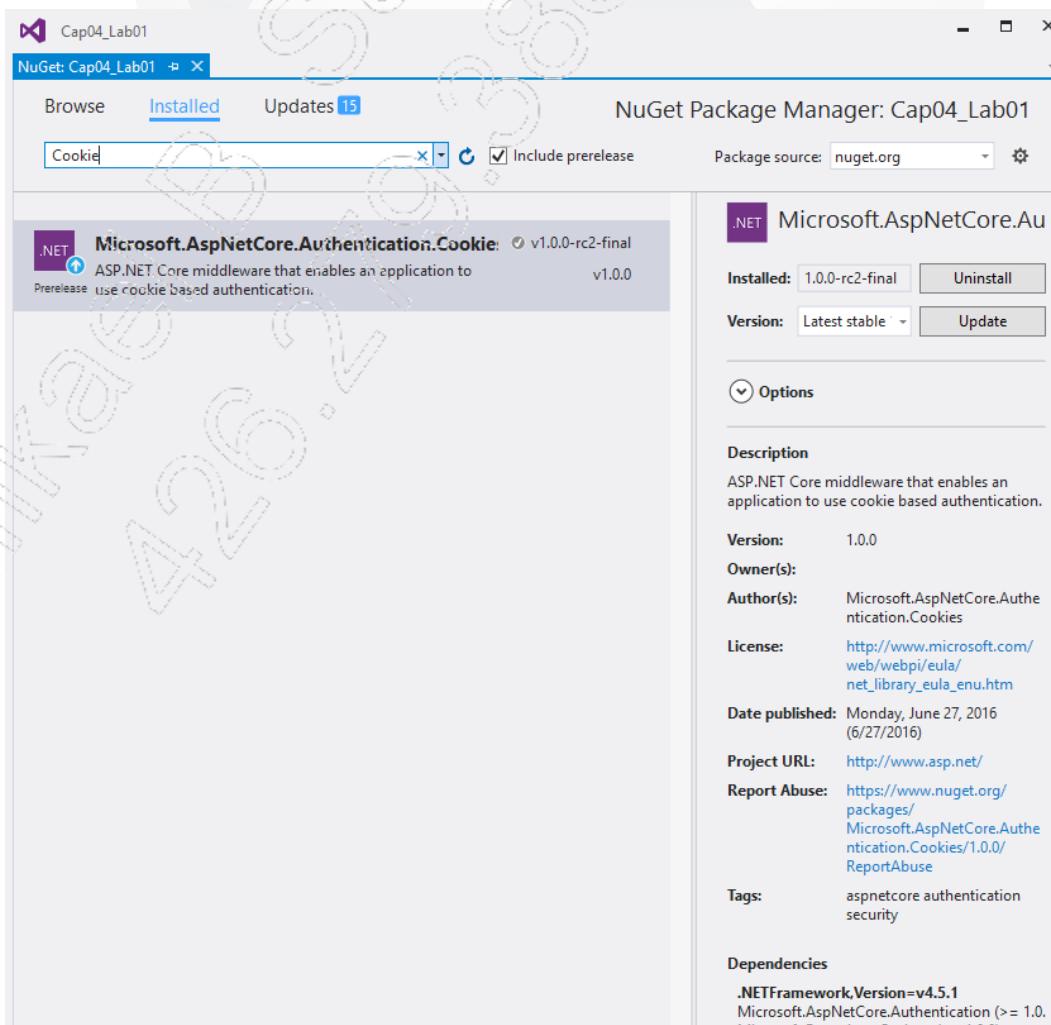
1. Crie um novo projeto do tipo **ASP.NET Core Web Application (.NET Core)**, sem autenticação (**No Authentication**), chamado **Cap04_Lab01**:



2. Verifique os arquivos gerados:



3. Usando NuGet, adicione o componente responsável por gerenciar autenticações por cookies: **Microsoft.AspNetCore.Authentication.Cookies**;



4. No arquivo **Startup.cs**, no método **Configure**, defina o modo como a aplicação vai utilizar autenticação por cookies:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.AspNetCore.Http;

namespace Cap04_Lab01
{
    public class Startup
    {
        public Startup(IHostingEnvironment env)...
        public IConfigurationRoot Configuration...
        public void ConfigureServices...

        public void Configure(IApplicationBuilder app...
        {
            loggerFactory.AddConsole(Configuration.
GetSection("Logging"));
            loggerFactory.AddDebug();

            if (env.IsDevelopment())
            {
                ...
            }
            else
            {
                ...
            }
        }
    }
}
```

```
    app.UseCookieAuthentication(new  
CookieAuthenticationOptions()  
    {  
        AuthenticationScheme =  
"AutenticacaoPorCookies",  
        LoginPath = new PathString("/Usuario/Login"),  
        AccessDeniedPath = new PathString("/Usuario/  
Login"),  
        AutomaticAuthenticate = true,  
        AutomaticChallenge = true  
    }) ;
```

```
    app.UseStaticFiles();  
  
    app.UseMvc(routes  
    {  
        }  
    }  
}
```

5. Crie uma pasta chamada **Models** e uma classe chamada **LoginViewModel**:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
  
namespace Cap04_Lab01.Models  
{  
    public class LoginViewModel  
    {  
        public string Email { get; set; }  
        public string Senha { get; set; }  
    }  
}
```

6. Na classe **HomeController**, exclua os métodos **Contact** e **About** e inclua os métodos **AreaLivre** e **AreaRestrita**, conforme o esquema adiante:

```
public class HomeController : Controller
{
```

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

```
    public IActionResult AreaLivre()
    {
        return View();
    }
```

```
    public IActionResult AreaRestrita()
    {
        return View();
    }
```

```
    public IActionResult Error()
    {
        return View();
    }
```

7. Para restringir o acesso à área restrita, coloque o atributo **Authorize** no método **AreaRestrita**. Quando o usuário tentar ter acesso a esse endereço, o sistema vai desviar para o login, como está definido no **Startup.cs**;

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

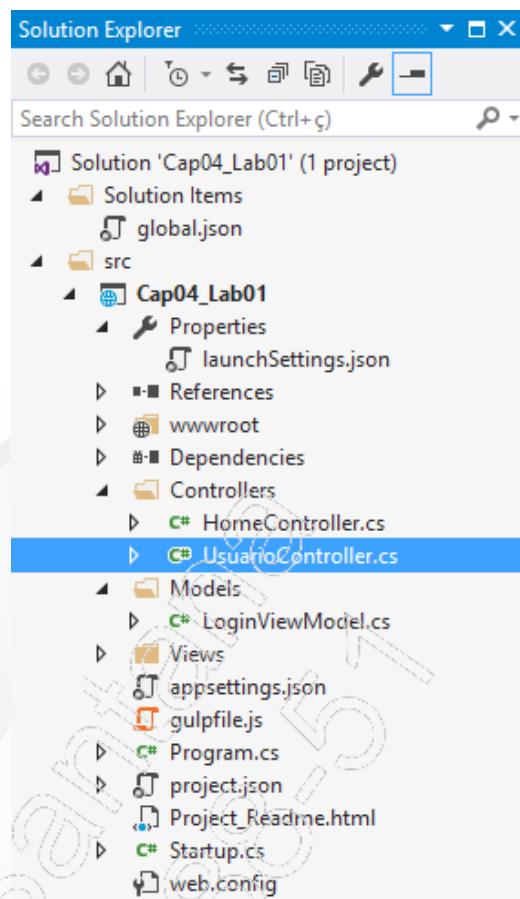
namespace Cap04_Lab01.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult AreaLivre()
        {
            return View();
        }

        [Authorize]
        public IActionResult AreaRestrita()
        {
            return View();
        }

        public IActionResult Error()
        {
            return View();
        }
}
```

8. Adicione uma classe de controle chamada **UsuarioController** e defina os métodos:



```
public class UsuarioController : Controller
{
    public IActionResult Login()
    {
        return View();
    }

    public async Task<IActionResult> Logout()
    {
        return View();
    }
}
```

9. Crie a pasta Views / Usuario e a página Login:

```
@model Cap04_Lab01.Models.LoginViewModel

<h2>Login</h2>
<hr/>

<form method="post">

    <div asp-validation-summary="All" class="text-danger"></div>

    <div class="form-group">
        <label asp-for="Email">Email:</label>
        <input asp-for="Email" name="Email" class="form-control" />
    </div>

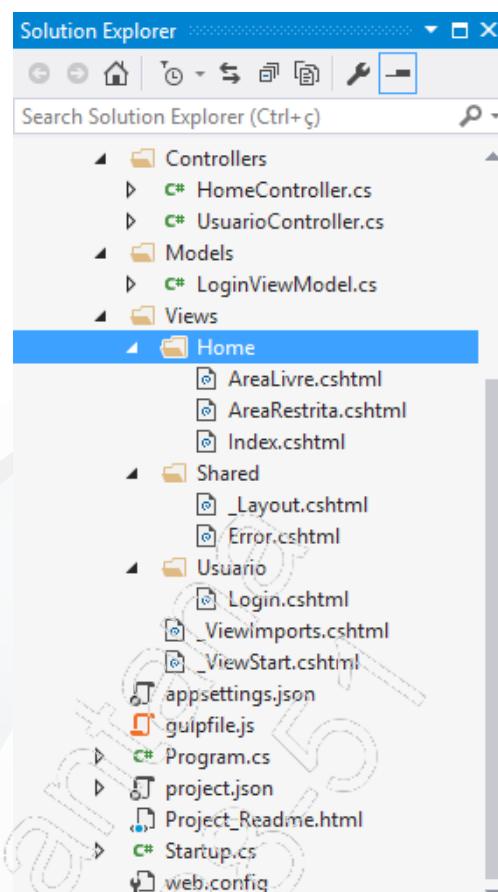
    <div class="form-group">
        <label asp-for="Senha">Senha:</label>
        <input asp-for="Senha" type="password" class="form-control" />
    </div>

    <div class="form-group">
        <input type="submit" value="Confirmar" class="btn btn-default" />
    </div>

    <br/>
    <hr/>
    <p>Use para o email: teste@teste.com.br e para a senha:123</p>

</form>
```

10. Na pasta **Views / Home**, crie duas páginas: **AreaLivre** e **AreaRestrita**. Pode apagar as views **About** e **Contact**, se houver;



11. Defina o conteúdo da página **AreaLivre**:

```
<h2>Área Livre</h2>

<hr/>
Usuário: @User.Identity.Name
```

12. Defina o conteúdo da página **AreaRestrita**:

```
<h2>Área Restrita</h2>

<hr/>
<p>Autorizado!</p>
<p>Usuário: @User.Identity.Name </p>
```

13. Defina o conteúdo da página **Index**:

```
<h2>Autenticação</h2>
<hr/>

<p>Exemplo de autenticação via Cookies</p>
```

14. No arquivo **Shared / _Layout.cshtml**, o menu deve ser alterado:

```
<ul class="nav navbar-nav">
    <li><a asp-controller="Home" asp-action="Index">Home</a></li>

    <li><a asp-controller="Home" asp-action="AreaLivre">Área Livre</a></li>

    <li><a asp-controller="Home" asp-action="AreaRestrita">Área Restrita</a></li>

    @if (User.Identity.IsAuthenticated)
    {
        <li><a asp-controller="Usuario" asp-action="Logout">Logout</a></li>
    }
    else
    {
        <li><a asp-controller="Usuario" asp-action="Login">Login</a></li>
    }
</ul>
```

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

15. Já é possível testar a navegação. Não vai ser possível entrar na ÁreaRestrita, sendo desviada para o login:

The image contains three screenshots of a web browser window titled "Home Page - Cap04_Lab01".

- Screenshot 1:** The browser shows the "Autenticação" page at `localhost:57279`. The URL bar shows "`localhost:57279`". The page content includes "Exemplo de autenticação via Cookies" and "© 2016 - Cap04_Lab01". The navigation bar at the top has links for "Cap04_Lab01", "Home", "Área Livre", "Área Restrita", and "Login".
- Screenshot 2:** The browser shows the "Área Livre" page at `localhost:57279/Home/ÁreaLivre`. The URL bar shows "`localhost:57279/Home/ÁreaLivre`". The page content includes "Usuário:" and "© 2016 - Cap04_Lab01". The navigation bar at the top has links for "Cap04_Lab01", "Home", "Área Livre", "Área Restrita", and "Login".
- Screenshot 3:** The browser shows the "Login" page at `localhost:57279/Usuario/Login?ReturnUrl=%2FHome%2FÁreaRestrita`. The URL bar shows "`localhost:57279/Usuario/Login?ReturnUrl=%2FHome%2FÁreaRestrita`". The page content includes fields for "Email:" and "Senha:", and a "Confirmar" button. The navigation bar at the top has links for "Cap04_Lab01", "Home", "Área Livre", "Área Restrita", and "Login".

16. Na classe **UsuarioController**, insira o método que vai validar o usuário. É o mesmo método do login, com o verbo POST:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Cap04_Lab01.Models;
using System.Security.Claims;
using Microsoft.AspNetCore.Http.Authentication;
namespace Cap04_Lab01.Controllers
{
    public class UsuarioController : Controller
    {
        public IActionResult Login()...
        
        [HttpPost]
        public async Task<IActionResult> Login(LoginViewModel
loginViewModel,
                                         string returnUrl = null)
        {
            if (string.IsNullOrEmpty(loginViewModel.Email))
            {
                ModelState.AddModelError("", "O Email deve ser
informado");
            }

            if (ModelState.IsValid)
            {
                //Validação apenas simbólica para simplificar
                //O objetivo é mostrar a autenticação por Cookies
                if (loginViewModel.Email == "teste@teste.com.br" &&
                    loginViewModel.Senha == "123")
            }
        }
    }
}
```

```
    var claim = new Claim(ClaimTypes.Name,
    "Administrador");
    var userIdentity = new
    ClaimsIdentity("IdentidadeAdmin");
    userIdentity.AddClaim(claim);
    var user = new ClaimsPrincipal(userIdentity);

    await HttpContext
        .Authentication
        .SignInAsync("AutenticacaoPorCookies", user);

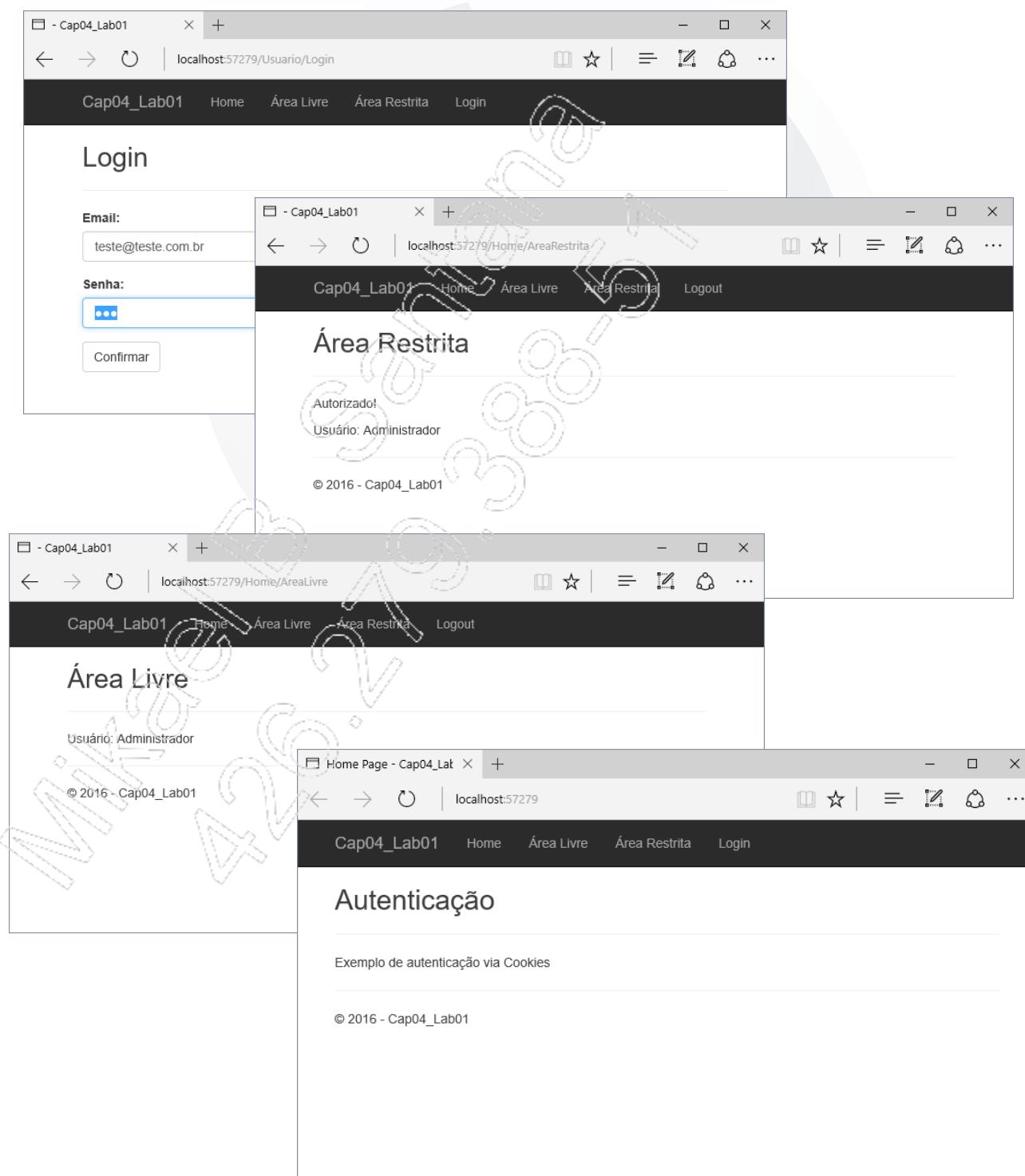
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        ModelState.AddModelError("", "Usuário ou Senha
        Inválida");
    }
}

if (ModelState.IsValid)
{
    return new RedirectToActionResult("Index", "Home",
    null);
}
else
{
    return View(loginViewModel);
}
}
```

17. Ainda na classe **UsuarioController**, crie o método **Logout**:

```
public async Task<IActionResult> Logout()
{
    await HttpContext.Authentication.
    SignOutAsync("AutenticacaoPorCookies");
    return new RedirectToActionResult("Index", "Home", null);
}
```

18. Teste o sistema completo: **Home**, **Área Livre**, **Área Restrita**, **Login** e **Logout**.



5

Segurança (.NET 4.6)

- ✓ ASP.NET Identity;
- ✓ OWIN;
- ✓ Implementações do Visual Studio.



IMPACTA
EDITORA

5.1. Introdução

O ASP.NET Core alterou significantemente o gerenciamento de componentes no ASP.NET. Porém, como toda tecnologia nova, será necessário um tempo de maturidade até que muitos sistemas tenham sido desenvolvidos na nova plataforma para que esta se torne robusta e confiável.

Tudo indica que o modelo tradicional no .NET Framework tem uma longa jornada ainda pela frente, porque é um sistema testado, confiável, com milhares de aplicações em pleno funcionamento e que por muito tempo não vão precisar de atualização. Por isso é importante conhecer, também, o sistema de segurança atual, ou seja, o ASP.NET Identity com os componentes OWIN e OAuth.

Existe uma clara tendência da Microsoft de passar a utilizar componentes open source para gerenciar a hospedagem de aplicativos na Web, mas isso é uma mudança gradual e adaptativa. Alguns componentes garantiram o seu lugar no ambiente de desenvolvimento, como jQuery, Knockout e Bootstrap.

Um bom indício de que um componente open source está sendo tratado como parte integrante do desenvolvimento .NET é quando esse componente passa a fazer parte dos modelos de projetos do Visual Studio. Com o lançamento do .NET Core, alguns componentes passaram a entrar nessa seleta lista: Node.js, Bower, Gulp, Angular, entre outros.

5.2. ASP.NET Identity

Antes do ASP.NET Core, a Microsoft lançou o ASP.NET Identity, que permite o uso de qualquer modelo de dados ou provedor de autenticação (como Facebook, Twitter ou Windows Live). O esquema de dados é totalmente controlado pela aplicação e todo o processo é modular, permitindo a substituição ou modificação de qualquer componente do processo. Esse componente é válido até a versão 4.6 no .NET Framework.

A modelagem do ASP.NET Identity utiliza interfaces para usuário, gerenciamento de usuário, grupos de usuários, senhas e todo tipo de informação que seja necessária à implementação de um sistema de segurança.

A principal interface é a que define um usuário, chamada **IUser**:

```
public interface IUser<out TKey>
{
    TKey Id { get; }
    string UserName { get; set; }
}

public interface IUser : IUser<string>
{ }

}
```

A interface **IUser** define apenas um **Id** e **Nome de Usuário**. O **Id** pode ser de qualquer tipo e uma sobrecarga é implementada, por padrão, definindo o tipo da chave como string. Repare que não há nenhuma outra implementação como **Senha**, **Nome Completo**, ou **Data de Cadastro**. O objetivo é exatamente este: manter o mais simples possível. Outras informações podem ser incluídas por meio de diferentes interfaces disponíveis. Uma implementação concreta dessa interface deve ser, no mínimo, conforme a listagem a seguir:

```
public class Usuario:IUser
{
    public string Id { get; set; }
    public string UserName { get; set; }
}
```

Nada impede, porém, que acrescentemos outras informações na classe básica:

```
public class Usuario:IUser
{
    public string Id { get; set; }
    public string UserName { get; set; }
    public string Cargo { get; set; }
    public DateTime DataNascimento { get; set; }
}
```

Para gravar as informações de um usuário, é necessário criar uma classe que implemente a interface **IUserStore**:

```
public interface IUserStore<TUser, in TKey> :  
    IDisposable where TUser : class,  
    IUser<TKey>  
{  
    Task CreateAsync(TUser user);  
    Task DeleteAsync(TUser user);  
    Task<TUser> FindByIdAsync(TKey userId);  
    Task<TUser> FindByNameAsync(string userName);  
    Task UpdateAsync(TUser user);  
}
```

A listagem adiante mostra o esqueleto de uma classe que implementa a interface **IUserStore**:

```
public class UsuarioStore:IUserStore<Usuario>  
{  
  
    public Task CreateAsync(Usuario user)  
    {  
        // implementação para criar usuário...  
    }  
  
    public Task DeleteAsync(Usuario user)  
    {  
        // implementação para excluir usuário...  
    }  
  
    public Task<Usuario> FindByIdAsync(string userId)  
    {  
        // implementação para encontrar usuário pelo id...  
    }  
  
    public Task<Usuario> FindByNameAsync(string userName)  
    {  
    }
```

```
// implementação para encontrar usuário pelo nome...
}

public Task UpdateAsync(Usuario user)
{
    // implementação para alterar usuário...
}

public void Dispose()
{
    // implementação para criar...
}

}
```

A última classe necessária para gerenciar os usuários se chama **UserManager** e está no namespace **Microsoft.AspNet.Identity**, o mesmo das interfaces **IUser** e **IUserStore**. Para instanciar, é necessário passar a classe de usuário, a chave e a classe que implementa a interface **IUserStore**. A listagem a seguir mostra apenas a declaração da classe, o método construtor e o método **Create**, que será usado neste exemplo:

```
public class UserManager<TUser, TKey> : IDisposable
    where TUser : class, IUser<TKey>
    where TKey : IEquatable<TKey>
{
    public UserManager(IUserStore<TUser, TKey> store);
    public virtual Task<IdentityResult> CreateAsync(TUser user);

    ... outros métodos omitidos
}
```

A classe **UserManager** contém vários métodos de extensão fornecidos pela biblioteca básica do ASP.NET Identity. Um dos métodos se chama **Create** e é utilizado para criar um novo usuário:

```
public static class UserManagerExtensions
{
    public static IdentityResult Create<TUser, TKey>(
        this UserManager<TUser, TKey> manager, TUser user)
        where TUser : class, IUser<TKey>
        where TKey : IEquatable<TKey>

    public static IdentityResult Create<TUser, TKey>
        (this UserManager<TUser, TKey> manager, TUser user)
        where TUser : class, IUser<TKey>
        where TKey : IEquatable<TKey>;
}

... outros métodos omitidos
```

O exemplo a seguir inclui um usuário:

```
1. var usuarioStore = new UsuarioStore();
2. var gerenciador = new UserManager<Usuario>(usuarioStore);
3. var usuario = new Usuario() { UserName = "Maria", Id="1" };
4. var resposta=gerenciador.Create(usuario);
5. if (resposta.Succeeded)
{
    //OK
}
```

Vejamos a explicação do código:

1. Uma instância da classe **UsuarioStore** é criada. Essa classe implementa a interface **IUserStore** e é necessária porque deve ser informada quando a classe **UserManager** for criada. Apenas para recapitular, a classe **UsuarioStore** executa as ações no banco de dados ou meio de armazenamento definido;
2. A classe **UserManager** é instanciada. Nessa classe, deve ser passado o parâmetro informando o tipo do usuário (aquele que implementa a interface **IUser**) e a classe **xxxStore**, que implementa a interface **IUserStore**. A classe **UserManager** chama os métodos para gerenciar usuários. Neste exemplo, apenas a parte de criação de dados foi implementada. Outras interfaces podem ser implementadas (**Senhas**, **Validações**, **Autenticadores**) e tudo fica reunido nessa classe;
3. Uma instância de **Usuário** com os dados do usuário a ser cadastrado é criada. Neste exemplo, para manter a simplicidade, os dados foram colocados manualmente;
4. Esta é a parte principal. O método **Create** é chamado, disparando os métodos apropriados na classe **xxxStore** e gravando o usuário. A classe **UserManager** chama o método **FindByName** e, se o nome do usuário for encontrado com um Id diferente do informado, a resposta retorna negativa. Caso contrário, o método **CreateAsync** é chamado para a criação do usuário;
5. A resposta do método **Create** da classe **UserManager** é uma instância da classe **IdentityResult**. Essa classe tem uma propriedade chamada **Succeeded**, do tipo booleana, para informar se a criação foi bem sucedida ou não. Caso tenha dado algum erro, a coleção de strings **Erros** retorna uma lista de erros.

A seguir, vejamos a lista de classes e interfaces utilizadas até o momento, que estão no namespace **Microsoft.AspNet.Identity**:

Nome	Tipo	Função
IUser	Interface	Define Id e nome de usuário. Opcionalmente, podem ser definidos outros dados.
IUserStore	Interface	Define os métodos para incluir, alterar, excluir e pesquisar nos dados do usuário.
UserManager	Classe	Chama os métodos das classes que implementam as funcionalidades do sistema.
IdentityResult	Classe	Contém a resposta às chamadas.

Adiante está a lista de classes criadas pelo aplicativo:

Nome	Tipo	Função
Usuario	Classe	Implementação da interface IUser.
UsuarioStore	Classe	implementação da interface IUserStore.

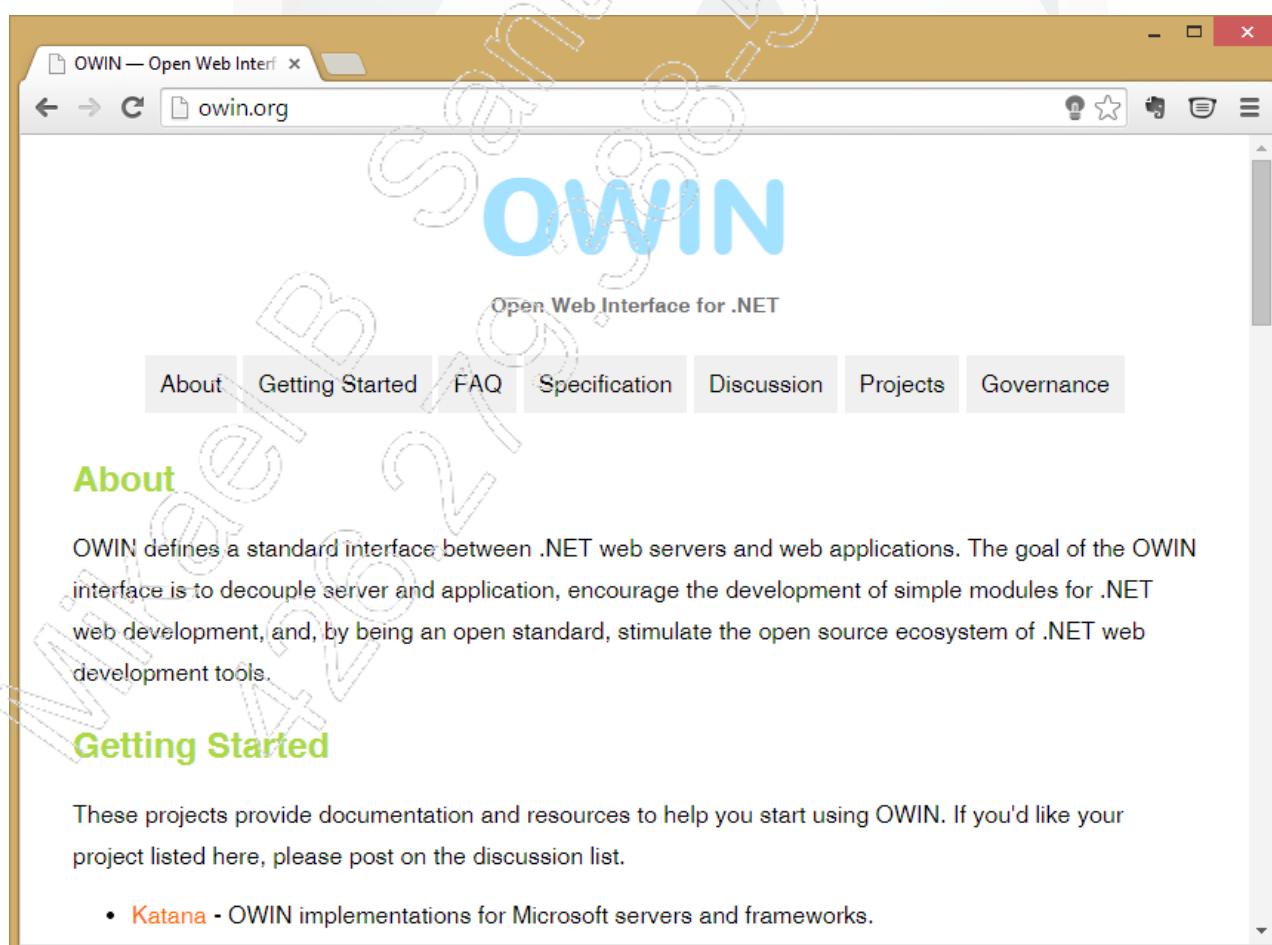
Todo o processo apresentado aqui implementa o cadastro de usuário manualmente. O ASP.NET contém uma implementação completa usando o Entity Framework e gravando o usuário em uma tabela do SQL Server, porém, essa implementação do Entity Framework é mais complexa e utiliza outros pacotes de funcionalidades. Para manter o foco nas classes principais do ASP.NET Identity, esta primeira parte será construída com o mínimo de componentes externos, mantendo, dentro do possível, apenas a biblioteca principal do ASP.NET Identity.

5.3. OWIN

Para realizar o processo de login, o ASP.NET utiliza componentes compatíveis com uma especificação chamada **OWIN**. Essa especificação define regras para que qualquer programa, em qualquer sistema operacional, possa hospedar aplicações do .NET Framework. Isso permite criar servidores mais leves do que o IIS e, também, utilizar autenticação de terceiros, como Facebook ou Google.

A sigla **OWIN** significa **Open Web Interface for .NET**. O propósito dessa especificação, segundo a própria definição em seu site, é o seguinte:

"OWIN define uma interface padrão entre os servidores Web .NET e as aplicações Web. O objetivo da interface OWIN é desacoplar servidores e aplicações, encorajar a criação de módulos simples para o desenvolvimento Web .NET e, por ser um padrão aberto, estimular um ecossistema open source de ferramentas de desenvolvimento para aplicações Web usando a plataforma .NET."



A ideia é ampliar e padronizar os recursos de integração de componentes criados com a plataforma .NET. Um dos projetos mais bem sucedidos desse tipo é chamado **Node.js**, que permite criar servidores e aplicações Web usando JavaScript.



A Microsoft utilizou muitos conceitos do Node.js, principalmente componentes específicos para cada tarefa em lugar de um grande conjunto de funcionalidades embutidas como no IIS. Porém, diferente do Node.js, que é um produto open source, o OWIN é uma especificação, ou seja, não existe um software determinado, apenas regras e padrões a serem adotados. Para servir de exemplo, um projeto foi criado utilizando os conceitos especificados no OWIN. Esse projeto se chama **Katana**.

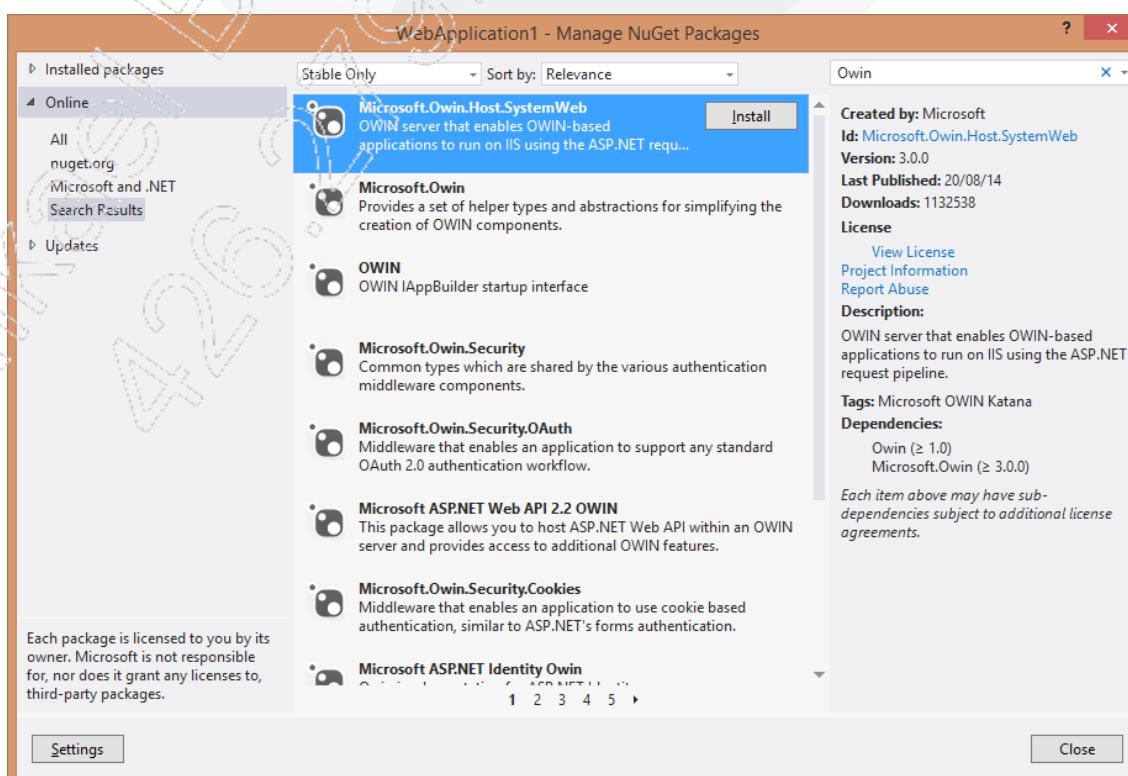
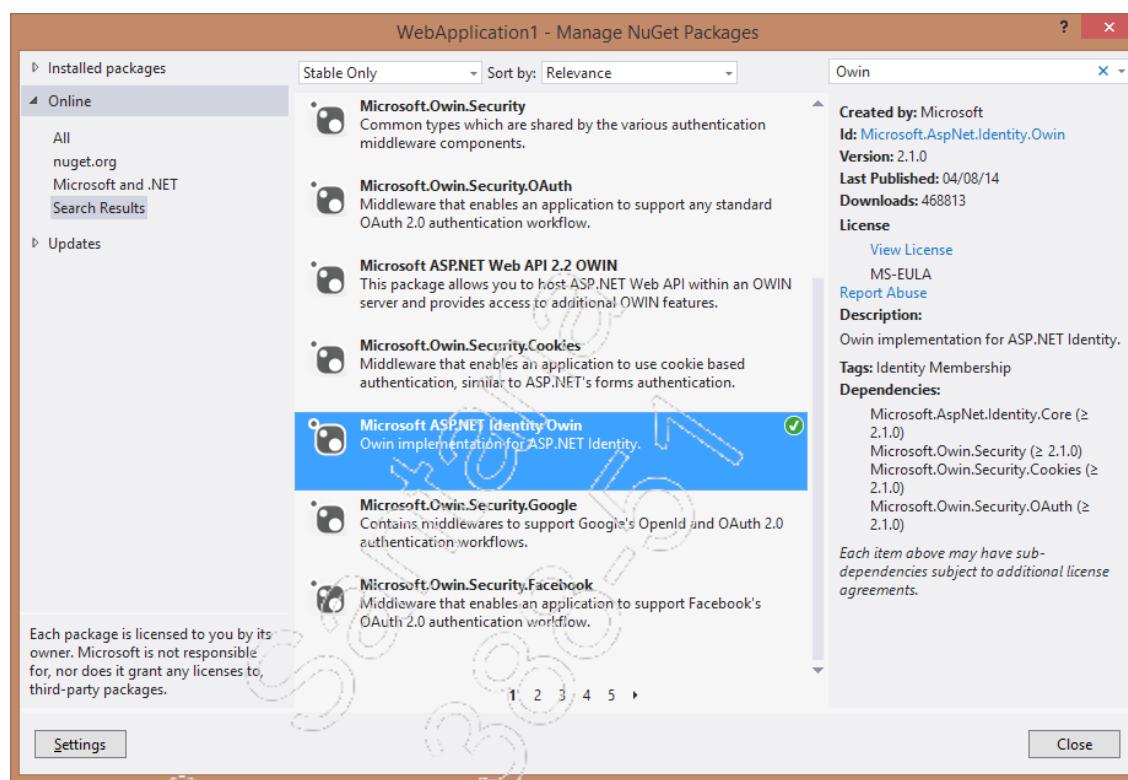


A especificação OWIN em si e a criação de servidores compatíveis fogem ao propósito deste capítulo. No entanto, é importante conhecer os componentes OWIN disponíveis para autenticação e como o ASP.NET Identity utiliza os recursos de autenticação definidos nessa especificação.

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

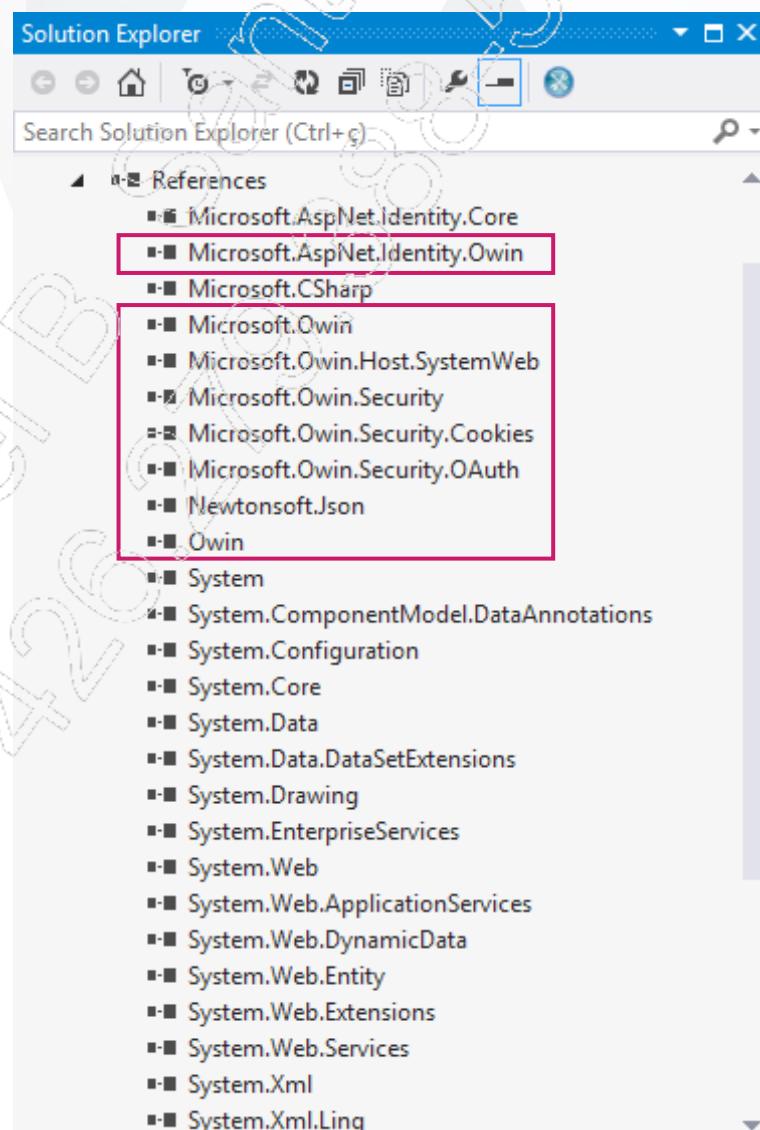
Por meio do NuGet, dois componentes devem ser instalados na aplicação:

- **Microsoft.AspNet.Identity.Owin**: Contém diversos métodos de extensão para que os componentes padrão do ASP.NET Identity utilizem o padrão OWIN;
- **Microsoft.Owin.Host.System.Web**: Permite que o IIS possa ser usado para rodar aplicações OWIN.



Esses pacotes dependem de outros pacotes de componentes, portanto, o número de itens adicionados à solução é maior:

- **Microsoft.AspNet.Identity.Owin;**
- **Microsoft.Owin;**
- **Microsoft.Owin.Host.SystemWeb;**
- **Microsoft.Owin.Security;**
- **Microsoft.Owin.Security.Cookies;**
- **Microsoft.Owin.Security.Oauth;**
- **Newtonsoft.Json;**
- **Owin.**



Para definir a autenticação OWIN é necessário incluir uma classe declarada com o atributo **OwinStartup**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;

[assembly: OwinStartup(typeof(WebApplication1.IniciarOwin))]1

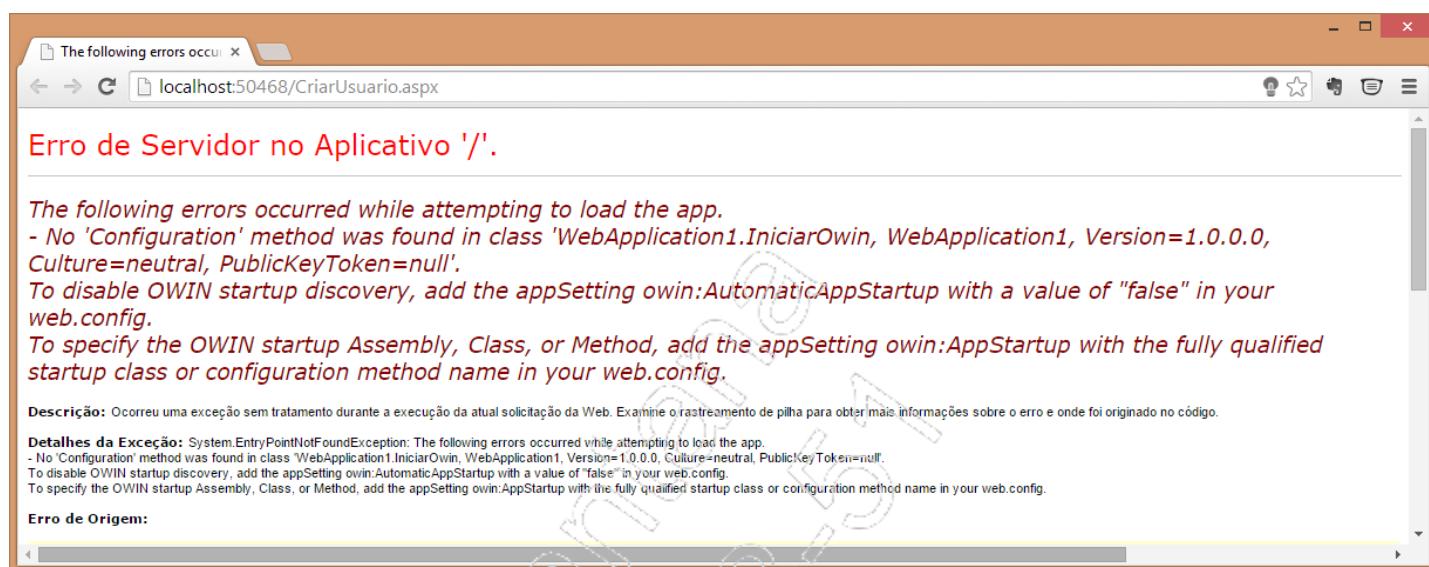
namespace WebApplication1
{
    public class IniciarOwin2
    {

        public void Configuration(IAppBuilder meuApp)3
        {
            var opcoes=new CookieAuthenticationOptions();
4

            opcoes.AuthenticationType=
                DefaultAuthenticationTypes.ApplicationCookie;

            opcoes.LoginPath = new PathString("/login");
5
            meuApp.UseCookieAuthentication(opcoes);
        }
    }
}
```

- 1 - O atributo **OwinStartup** recebe um parâmetro com o nome da classe (**namespace.nome**), contendo o método **Configuration**, que recebe um parâmetro do tipo **IAppBuilder**. Isso faz o mecanismo OWIN instanciar a classe e chamar o método no início da aplicação. Esse método define as configurações iniciais do aplicativo. Isso é uma **convenção**. Se não houver esse método, um erro ocorre:



- 2 - A classe chamada no item 1;
- 3 - O método exigido pelo OWIN. Ele deve se chamar **Configuration** e deve receber como parâmetro uma instância de uma classe que implemente a interface **IappBuilder**;
- 4 - O método principal desse processo se chama **UseCookieAuthentication**. Esse método espera receber uma instância da classe **CookieAuthenticationOptions**. Essa parte do código cria esta instância e define duas opções: o tipo de autenticação (**Cookies**) e a página de URL de Login;
- 5 - Finalmente, a instância passada como parâmetro (**meuApp**) é usada para definir as opções de autenticação, por meio do método **UserCookieAuthentication**.

Está completo o processo. O código que cria o usuário pode ser usado para autenticar esse usuário no sistema:

```
var usuarioStore = new UsuarioStore();
var gerenciador = new UserManager<Usuario>(usuarioStore);
var usuario = new Usuario() { UserName = "Maria", Id =
    Guid.NewGuid().ToString() };
var resposta = gerenciador.Create(usuario);

if (resposta.Succeeded)
{
    var authenticationManager = HttpContext.Current.
        GetOwinContext().Authentication;
    var tipoAutenticacao=DefaultAuthenticationTypes.
        ApplicationCookie;
    var userIdentity = gerenciador.CreateIdentity(usuario,
        tipoAutenticacao);
    authenticationManager.SignIn(userIdentity);

    Label1.Text = "OK";
}
```

O método de extensão **GetOwinContext** retorna informações sobre a execução atual. Uma das informações é a propriedade **Authentication**, um gerenciador de autenticação, que é uma instância de uma classe que implementa a interface **IAuthenticationManager**. Essa interface define o método **SignIn**, usado para autenticar o usuário.

As seguintes classes e interfaces foram usadas nesse processo de autenticação por cookies, usando componentes compatíveis com a especificação OWIN:

Namespace Owin		
Nome	Tipo	Função
IAppBuilder	Interface	Interface que deve ser passada (por meio de uma instância que a implemente) para o programa de inicialização de um site.

Namespace Microsoft.Owin		
Nome	Tipo	Função
OwinStartup	Classe	Atributo para definir uma classe de inicialização.
PathString	Struct	Manipula informações de uma URL.
IOwinContext	Interface	Permite extrair dados de ambiente OWIN usando tipos definidos (strong-typed accessors).

Namespace Microsoft.Owin.Security		
Nome	Tipo	Função
IAuthenticationManager	Interface	Usado para interagir como componentes OWIN.

Namespace Microsoft.Owin.Security.Cookies		
Nome	Tipo	Função
CookieAuthenticationOptions	Classe	Opções de autenticação por cookies.

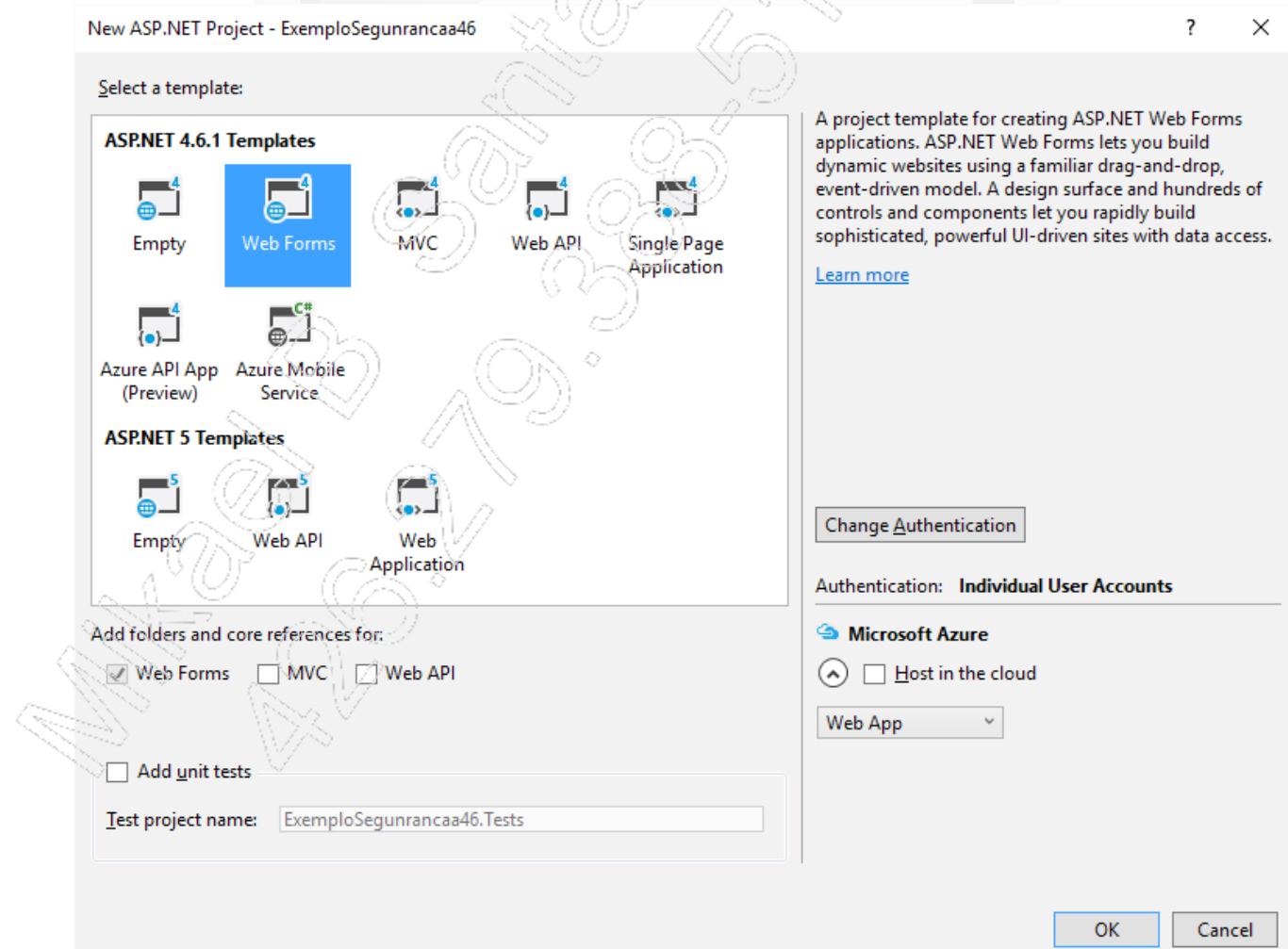
Namespace Microsoft.AspNet.Identity		
Nome	Tipo	Função
DefaultAuthenticationTypes	Classe	Tipos de autenticação.

5.4. Implementações do Visual Studio

O Visual Studio 2015 oferece uma série de implementações de segurança usando toda a funcionalidade das classes do ASP.NET Identity. Para armazenar os dados de usuário em um banco de dados, é possível usar o Entity Framework no modelo Code First, o que facilita bastante a personalização. Pode-se usar um provedor de autenticação externo, como Facebook ou Google.

5.4.1. Web Forms e contas individuais

Este é o modelo mais comum para quem utiliza Web Forms e armazena informações de usuário em um banco de dados da aplicação. Ao iniciar um novo projeto, basta selecionar **Web Forms e Individual User Accounts**:



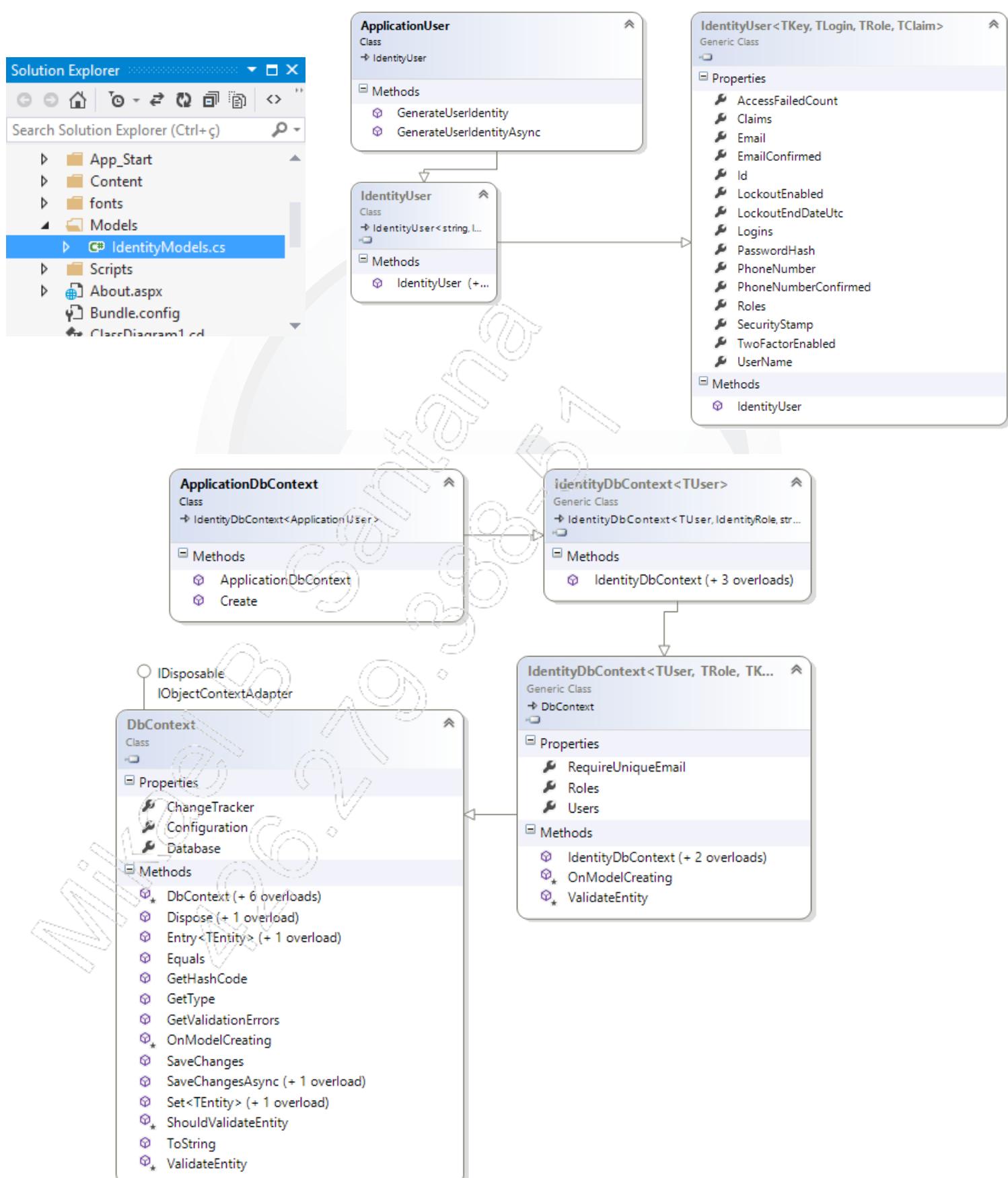
Além dos namespaces esperados (Asp.Net.Identity e OWIN), o modelo do Visual Studio adiciona referências às seguintes bibliotecas:

Biblioteca	Descrição
Microsoft.AspNet.Identity.EntityFramework	Responsável por gravar os dados em um banco de dados SQL Server.
Microsoft.Owin.Security.OAuth	
Microsoft.Owin.Security.Facebook	
Microsoft.Owin.Security.Google	Responsáveis pela autenticação de usuário em provedores externos.
Microsoft.Owin.Security.MicrosoftAccount	
Microsoft.Owin.Security.Twitter	



Visual Studio 2015 - ASP.NET com C# Recursos Avançados

O centro de todo o processo é o conjunto de classes definidas no arquivo **IdentityModels.cs**:



São apenas três classes utilizadas diretamente pelo sistema:

- **ApplicationUser**: Dados do usuário. As classes base **IdentityUser** e **IdentityUser<Tkey, Tlogin, Trole, Tclaim>** fornecem as propriedades que representam o usuário;
- **ApplicationDbContext**: Responsável pela comunicação com o Entity Framework. As classes base **IdentityContext<TUser, TRole, TKey, TUserRole, TUserClaim>** fornecem os métodos para obter e alterar informações do usuário e outros elementos de autenticação e autorização;
- **IdentityHelper**: Classe estática que apresenta alguns métodos para centralizar alguns processos, como executar o login, obter a URL de redirecionamento ou obter o nome do Provider de autenticação.

Ao iniciar a aplicação, a classe **Startup** é acionada, porque está configurada para ser a classe de configuração do modelo OWIN. Essa classe (marcada como **partial**) está dividida em dois arquivos:

- \Startup.cs

```
[assembly: OwinStartupAttribute(typeof(ExemploWebFormsContas.Startup))]  
public partial class Startup {  
    public void Configuration(IAppBuilder app) {  
        ConfigureAuth(app);  
    }  
}
```

- \App_Start\Startup.Auth.cs (parte do código omitida)

```
public partial class Startup
{
    public void ConfigureAuth(IAppBuilder app)
    {
        app.CreatePerOwinContext(ApplicationDbContext.Create);
        app.CreatePerOwinContext<ApplicationUserManager>...
        app.CreatePerOwinContext<ApplicationSignInManager>.....
        app.UseCookieAuthentication(...);
        app.UseExternalSignInCookie(...);
        app.UseTwoFactorSignInCookie(...);
        app.UseTwoFactorRememberBrowserCookie(...);
    }
}
```

O processo começa com o método **Configuration** chamando o método **ConfigureAuth** da mesma classe, mas que está definida no arquivo **/AppStart/Startup.Auth.cs**.

O método principal é o **ConfigureAuth**, que realiza algumas inicializações, chamando métodos da classe **Microsoft.Owin.Builder.AppBuilder**, que é passada como parâmetro.

Vejamos os métodos de extensão da classe **AppBuilder** chamados:

- **CreatePerOwinContext**

Chamado três vezes para criar um objeto de contexto para o banco de dados (**ApplicationContext.Create**), para o gerenciamento de usuário (**ApplicationUserManager.Create**) e para o gerenciamento de autenticação (**ApplicationSignInManager.Create**). O tipo de parâmetro que é passado para este método é um **delegate**, ou seja, uma variável que representa um método. No caso, cada classe oferece o método **Create** para o objeto que está manipulando.

```
app.CreatePerOwinContext(ApplicationDbContext.Create);
app.CreatePerOwinContext<ApplicationUserManager>(
    ApplicationUserManager.Create);
app.CreatePerOwinContext<ApplicationSignInManager>(
    ApplicationSignInManager.Create);
```

Este método é necessário porque, mais adiante, esse objeto será recuperado por meio do método **GetOwinContext**. Por exemplo, para criar um usuário, o início do código é este:

```
var manager = Context.GetOwinContext() .GetUserManager<  
    ApplicationUserManager>();
```

- **UseCookieAuthentication**

Este método de extensão habilita a aplicação a usar cookies para armazenar informação de um usuário autenticado.

```
app.UseCookieAuthentication(new  
CookieAuthenticationOptions....);
```

O parâmetro a ser passado para este método é uma instância da classe **CookieAuthenticationOptions** definida no módulo **Microsoft.Owin.Security.Cookies**.

- **UseExternalSignInCookie**

Configura a aplicação para usar cookies quando autenticado por provedores externos.

- **UseTwoFactorSignInCookie**

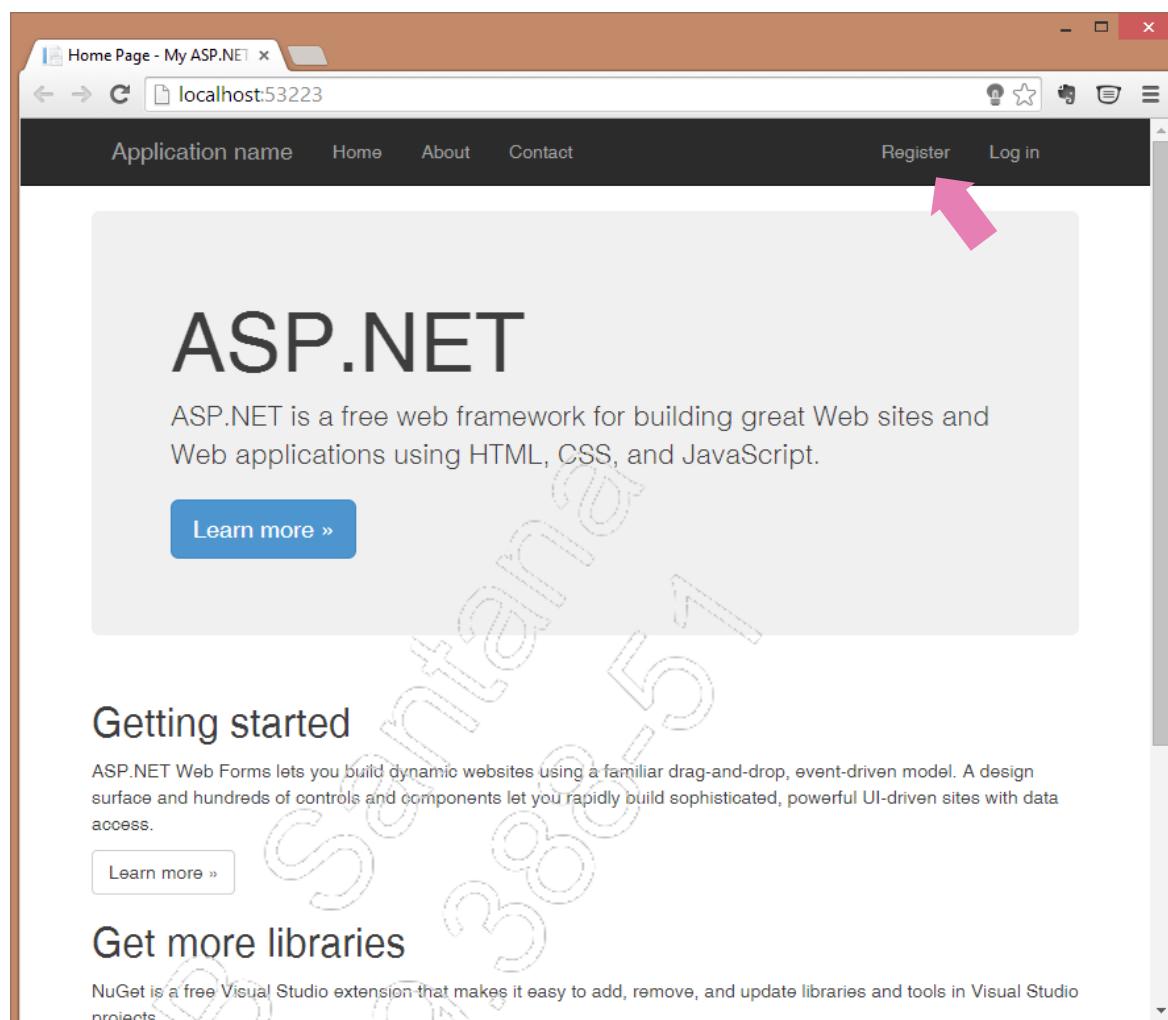
Configura a aplicação para usar cookies, armazenando parte da identificação do usuário para usar autenticação em dupla-verificação, que é um processo em que não apenas a senha é usada para autenticar o usuário mas também outra informação do seu cadastro.

- **UseTwoFactorRememberBrowserCookie**

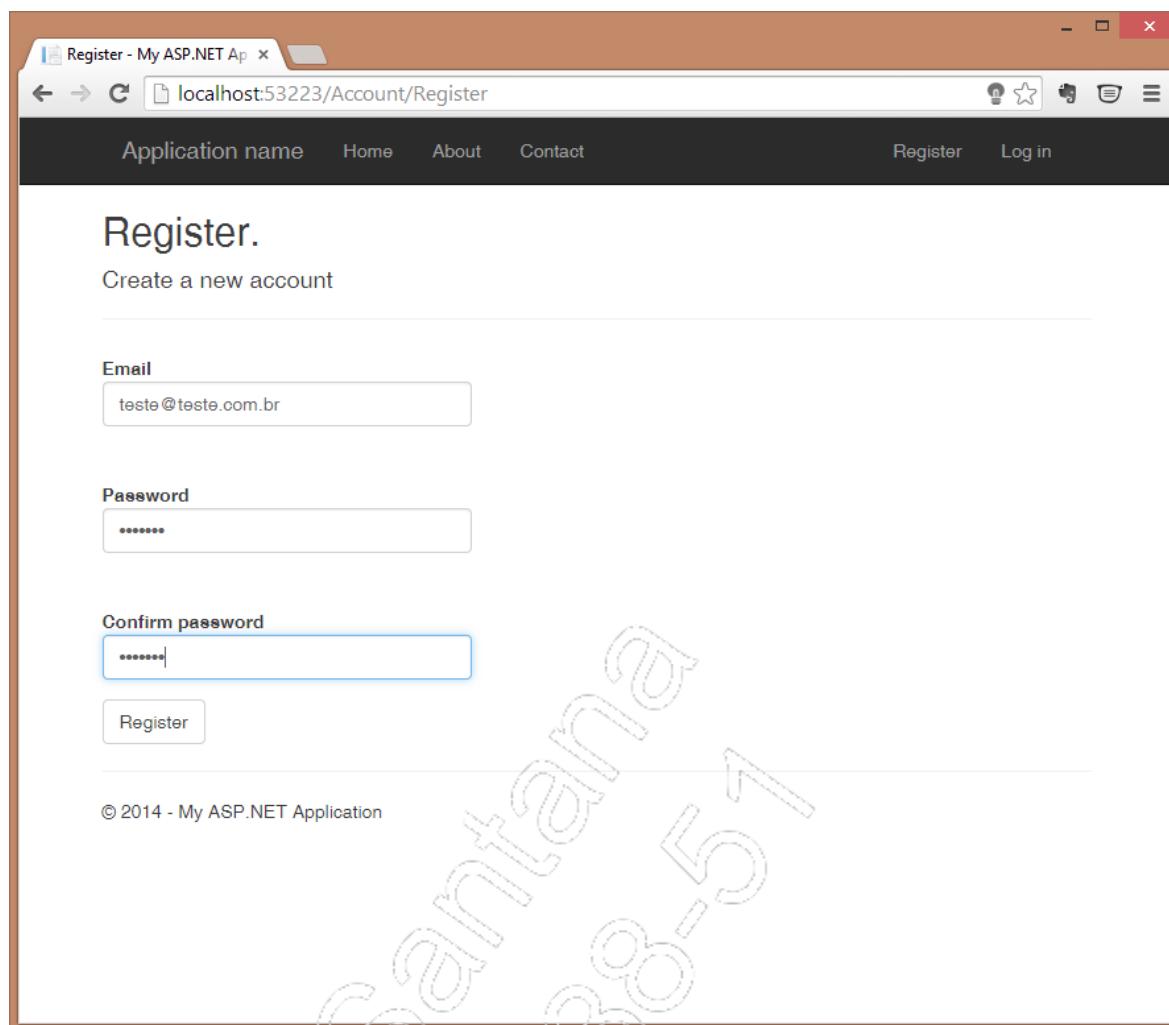
Configura a aplicação para usar cookies para lembrar as informações adicionais de um usuário quando é usada a autenticação em dupla-verificação.

5.4.2. Criando usuários

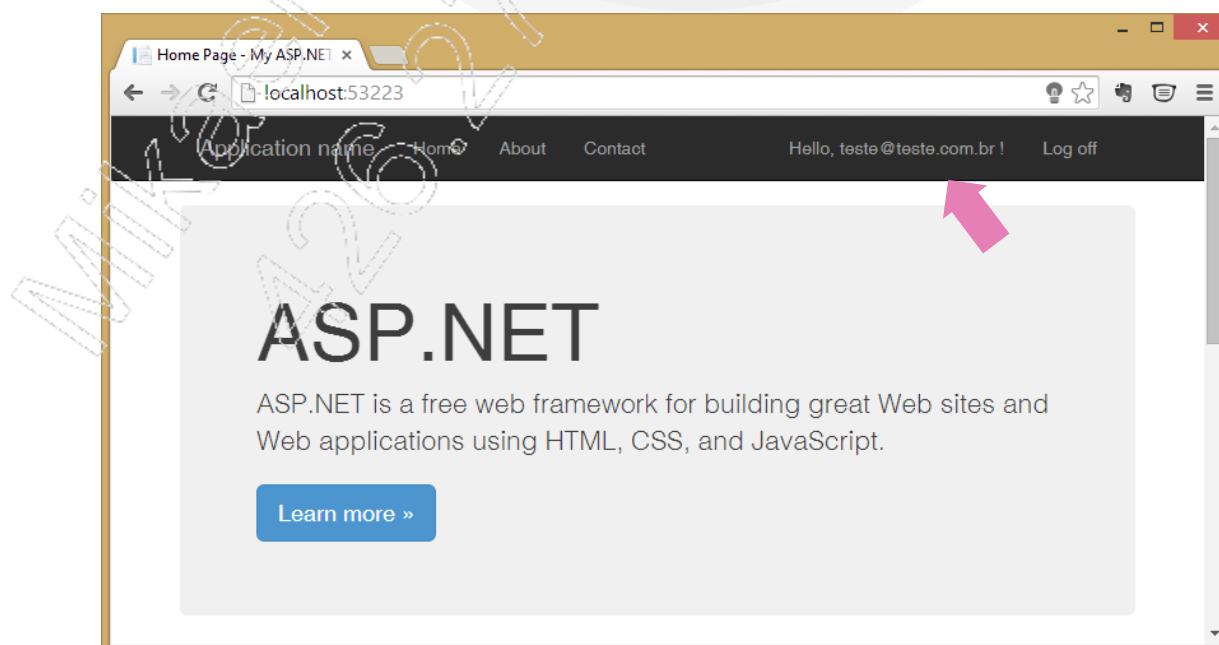
O primeiro passo é clicar em **Register** no menu principal. Isso nos levará para a página **Register.aspx**.



Neste modelo, apenas e-mail e senha são utilizados.



Uma vez registrado, o usuário é autenticado imediatamente. O e-mail aparece na barra de menus, bem como uma opção de **Log off**:



- **Entendendo a página Register.aspx**

O formulário de registro apresenta apenas os user controls **Label**, **TextBox**, **RequiredFieldValidator**, **CompareValidator** e **Button**. A listagem adiante foi simplificada para facilitar a visualização desses componentes:

```
<h4>Create a new account</h4>

<asp:ValidationSummary runat="server" />

<asp:Label runat="server">Email</asp:Label>
<asp:TextBox runat="server" ID="Email" />
<asp:RequiredFieldValidator runat="server" />

<asp:Label runat="server">Password</asp:Label>
<asp:TextBox runat="server" ID="Password" />
<asp:RequiredFieldValidator runat="server"/>

<asp:Label runat="server">Confirm password</asp:Label>
<asp:TextBox runat="server" ID="ConfirmPassword" />
<asp:RequiredFieldValidator runat="server" />
<asp:CompareValidator runat="server" />

<asp:Button runat="server" />
```

- **Code-behind**

```
protected void CreateUser_Click(object sender, EventArgs e)
{
    ① var manager = Context.GetOwinContext()
        .GetUserManager<ApplicationUserManager>();

    ② var user = new ApplicationUser()
        { UserName = Email.Text, Email = Email.Text };

    ③ IdentityResult result = manager.Create(user, Password.
        Text);
```

```

④ if (result.Succeeded)
{
    IdentityHelper.SignIn(manager, user,
isPersistent: false);

    IdentityHelper.RedirectToReturnUrl(
        Request.QueryString["ReturnUrl"], Response);
}

else
{
    ErrorMessage.Text = result.Errors.
FirstOrDefault();
}
}

```

- 1 – Obtém a instância de **UserManager** configurada no sistema. Nesse caso, a classe **ApplicationUserManager**;
- 2 – Cria uma instância de **ApplicationUser**, que é o usuário a ser adicionado;
- 3 – Chama o método **Create**. O resultado é do tipo **IdentityResult**;
- 4 – Se o usuário foi criado com sucesso, ele é autenticado. A classe **IdentityHelper** facilita este processo, por meio do método **SignIn**:

```

public static class IdentityHelper
{
    public static void SignIn(ApplicationUserManager manager,
        ApplicationUser user,
        bool isPersistent)
    {
        ① IAuthenticationManager authenticationManager = HttpContext
            .Current
            .GetOwinContext()
            .Authentication;

        ② authenticationManager.SignOut(DefaultAuthenticationTypes
            .ExternalCookie);
    }
}

```

```
③ var identity = manager.CreateIdentity(user,
                                         DefaultAuthenticationTypes
                                         .ApplicationCookie);

④ authenticationManager.SignIn(new AuthenticationProperties()
    { IsPersistent = isPersistent }, identity);
}
```

- 1 – Obtém uma instância do gerenciador de autenticação;
- 2 – Faz o log off, caso esteja logado;
- 3 – Cria uma instância da classe **ClaimsIdentity**, que identifica o usuário no sistema;
- 4 – Por meio do gerenciador de autenticação, faz o login no sistema, criando um cookie de autenticação.

Repare que, propositalmente, o sistema de segurança é composto de blocos independentes. É possível trocar qualquer parte do sistema substituindo os gerenciadores. Esse modelo de componentes com pouca dependência uns dos outros é uma característica encontrada em todos os novos modelos da Microsoft. Nesse processo, foram usados os gerenciadores que implementam as interfaces **IUserManager** e **IAuthenticationManager**.

5.4.3. Web Forms e autenticação externa

É possível economizar muito tempo e esforço se for utilizado um agente de autenticação externo no lugar de criar um sistema próprio. Um sistema de segurança deve, no mínimo, implementar as seguintes funcionalidades:

- Login;
- Verificação de usuário autenticado;
- Recuperação de senha;

- Criptografia de dados;
- Alteração de senha;
- Gerenciamento de dados extras do usuário;
- Bloqueio/liberação de usuário por um administrador.

Cada funcionalidade exige a determinação de como armazenar os dados, criar as telas de interação com o usuário, validar a entrada de dados e criptografar informações sigilosas como senhas, tarefas estas muito bem implementadas pelas grandes redes sociais. Usando um autenticador externo como Facebook, Google ou Twitter, todo esse processo é "terceirizado" para essas empresas e o usuário ainda tem o benefício de não precisar decorar mais um nome de usuário e senha para outro sistema.

A Microsoft usa esse conceito há muito tempo com o Windows Live. O mesmo login era usado para todos os sites. A primeira versão do ASP.NET possuía até um tipo de autenticação chamada **passport** que podia (e ainda pode) ser utilizada para ter acesso à rede do Windows Live por meio das credenciais do Hotmail.

Para fazer a ligação entre a aplicação atual e a aplicação externa, é necessário entrar no provedor e criar um aplicativo com uma palavra-chave compartilhada. O nome do aplicativo e a palavra-chave são usados para criar a autenticação externa.

5.4.3.1. OAuth

Para criar a ligação entre o aplicativo .NET e um aplicativo externo, os componentes OWIN utilizam um padrão aberto para autorização chamado **OAuth (Open Standard for Authorization)**. Esse padrão trabalha com o conceito de **Token**, que é uma chave conseguida por meio da troca de mensagens entre dois sistemas e que é usada para ter acesso temporário a recursos de um aplicativo.

Existem quatro definições envolvidas em um processo típico de autenticação:

1. **Resource Owner** (proprietário do recurso): A entidade que tem o controle sobre uma informação protegida. Essa entidade pode ser uma pessoa, um aplicativo, um dispositivo ou um serviço. O exemplo mais comum é uma pessoa que tem acesso a algum sistema por meio do seu nome de usuário e senha. Nesse caso, o **Resource Owner** é chamado de **End User** (usuário final);
2. **Client** (cliente): A aplicação que solicita informações em nome do usuário. Geralmente é um browser como Internet Explorer ou Chrome. Também é chamado de **User Agent**;
3. **Resource Server** (servidor de recursos): O servidor que hospeda os dados do usuário. Alguns documentos se referem a **Web Application** como sendo o servidor de recursos. Não está errado, porque as definições aqui apresentadas não se referem a nenhuma implementação física real;
4. **Authorization Server** (servidor de autorização): O servidor que contém o sistema de controle de usuários disponibilizado na Web segundo os padrões OAuth, por exemplo, Google, Facebook, Twitter.

O fluxo de informações, tomando como exemplo um usuário navegando em uma aplicação Web, é o seguinte:

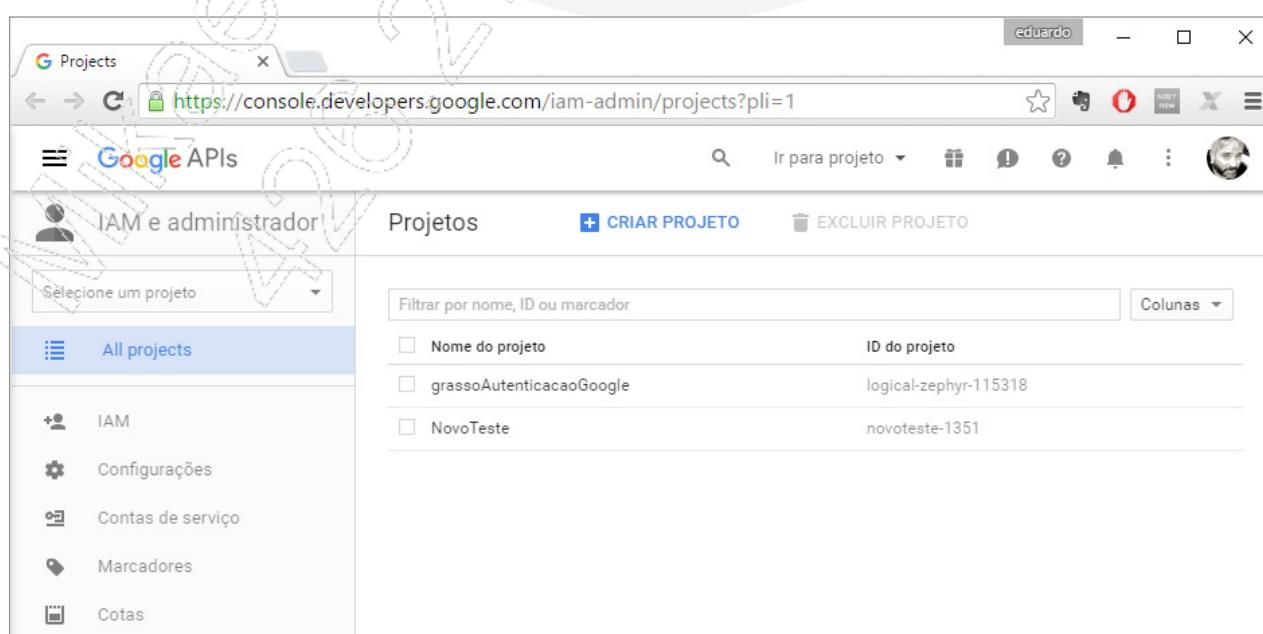
1. Um usuário, utilizando um navegador ou dispositivo, navega até a aplicação Web;
2. A aplicação Web redireciona o navegador para o servidor de autenticação, por exemplo, o Google. Importante: A aplicação Web não transfere o controle para o servidor de autenticação; ela manda o navegador chamá-lo por meio de uma ordem de redirecionamento;
3. O usuário então vê a tela do servidor de autenticação, preenche os dados e clica no botão **Enviar**. O navegador do usuário envia as credenciais que ele digitou para o servidor de autenticação;
4. O servidor de autenticação retorna uma autorização que só pode ser usada uma vez para o navegador e redireciona para a aplicação Web;

5. O navegador envia essa autorização para a aplicação Web. A aplicação Web deve ter uma página específica para receber esses dados;
6. A aplicação Web passa a autorização que recebeu do navegador e um código para o servidor de autenticação. A aplicação conhece esse código porque ela deve tê-lo recebido anteriormente do servidor de autenticação;
7. O servidor de autenticação recebe a autorização, o Id da aplicação e a chave, e, se tudo estiver certo, retorna o token para a aplicação Web;
8. A aplicação Web usa o token para obter informações sobre o usuário no servidor de autenticação. Esse token é um tipo de informação que apenas o servidor de autenticação sabe como foi feito e, portanto, pode validá-lo;
9. O servidor de autenticação responde para a aplicação com informações sobre o usuário e suas permissões.

5.4.3.2. Exemplo de autenticação com o Google

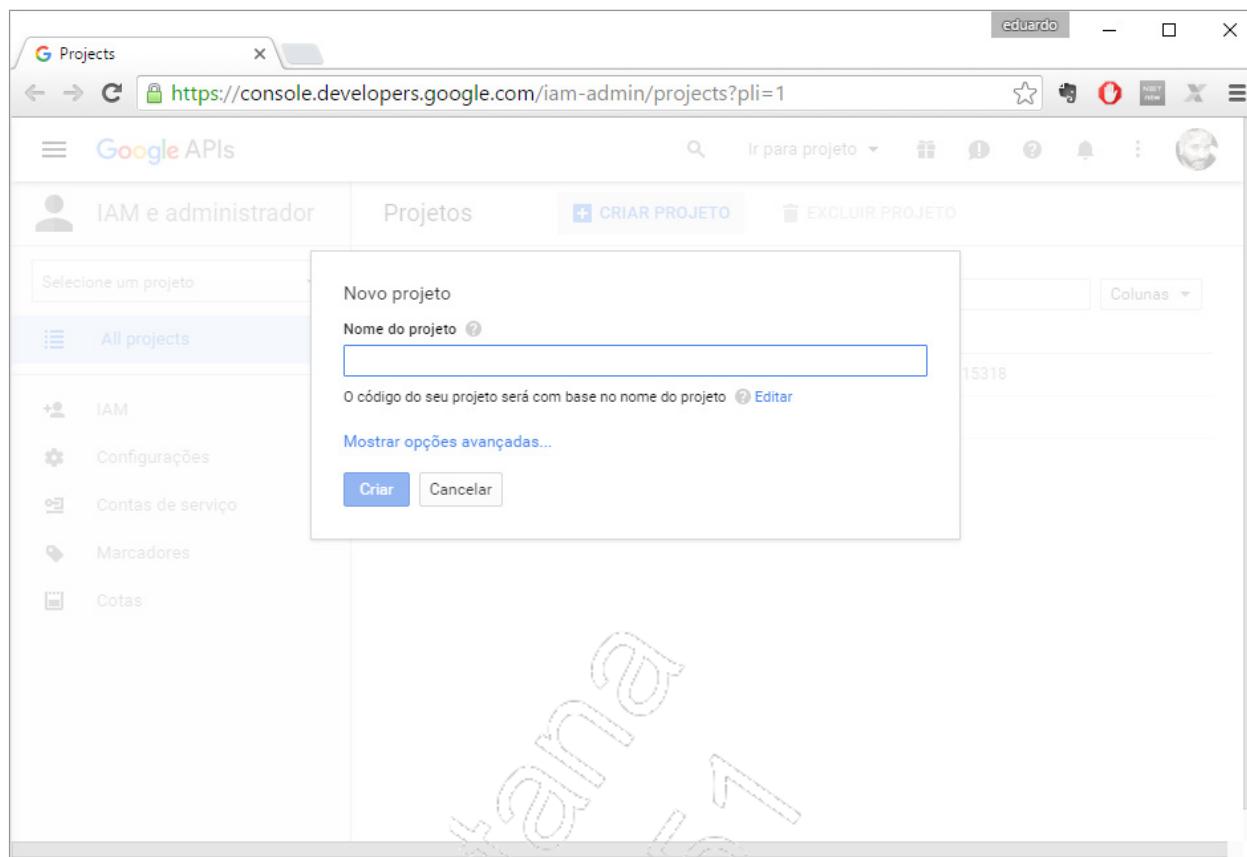
Vejamos, a seguir, um exemplo de como realizar uma autenticação com o Google:

1. Faça login no Google;
2. Navegue para <https://console.developers.google.com/project>;

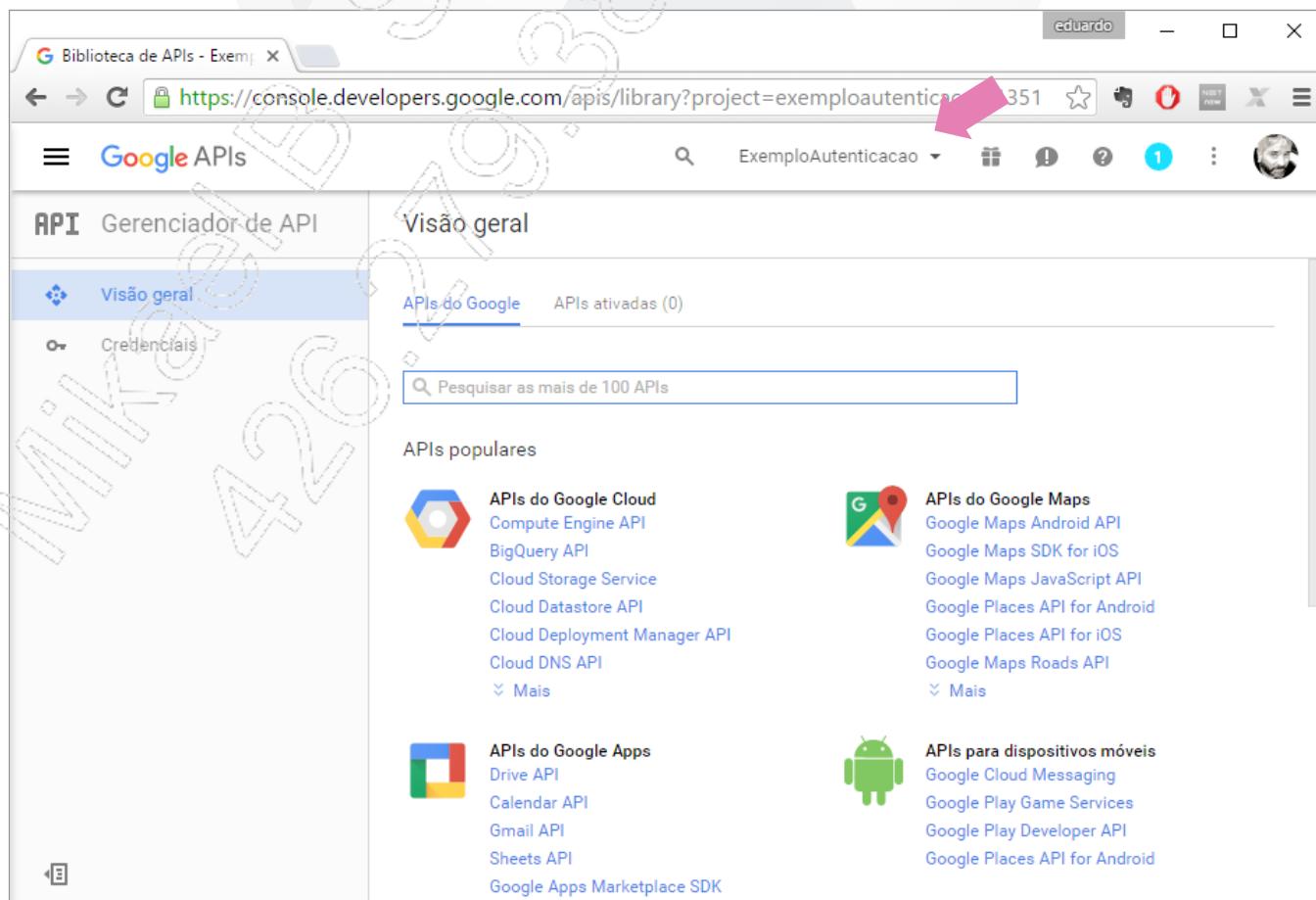


Visual Studio 2015 - ASP.NET com C# Recursos Avançados

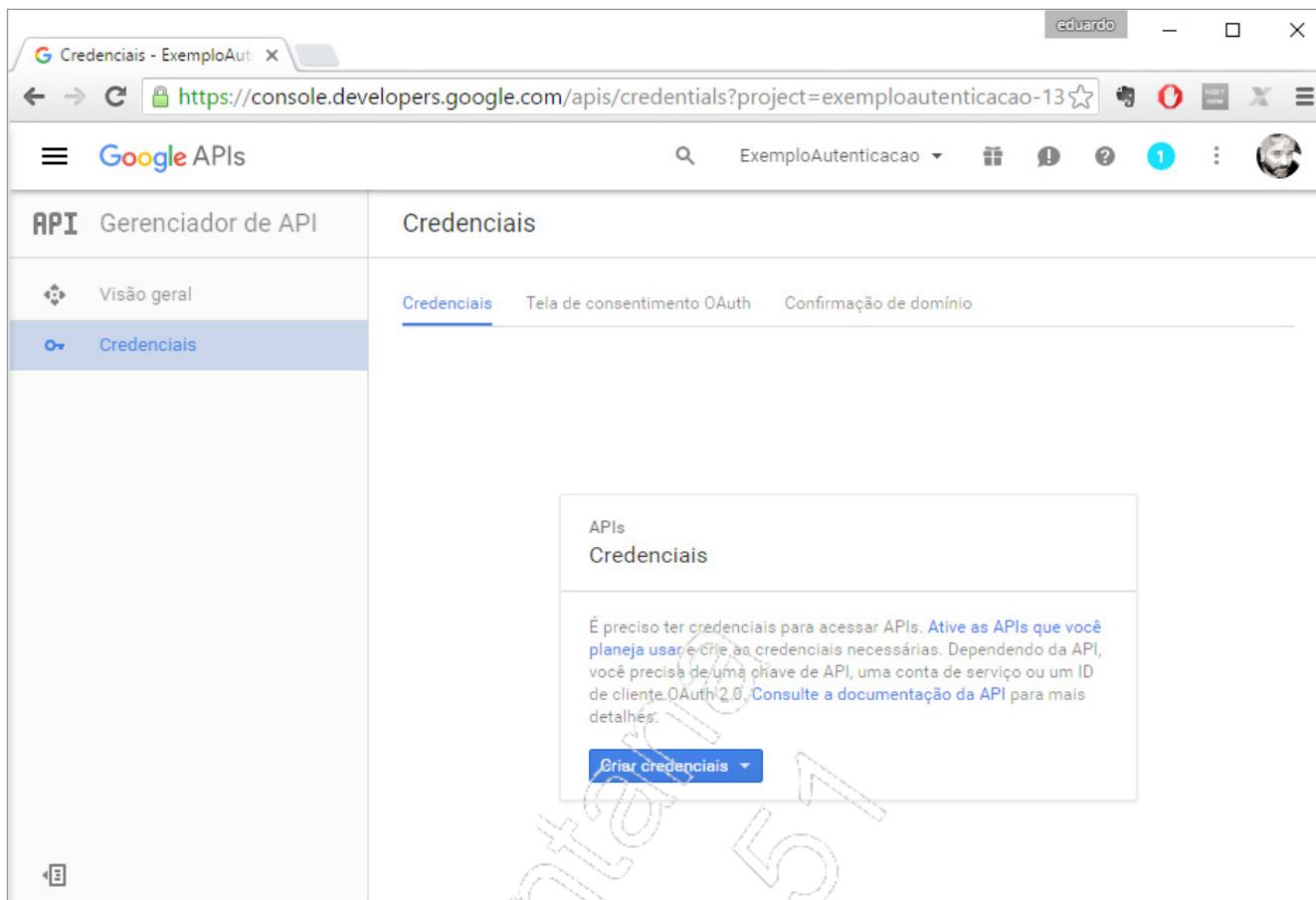
3. Clique em **CRIAR PROJETO**, informe um nome e clique em **Criar**;



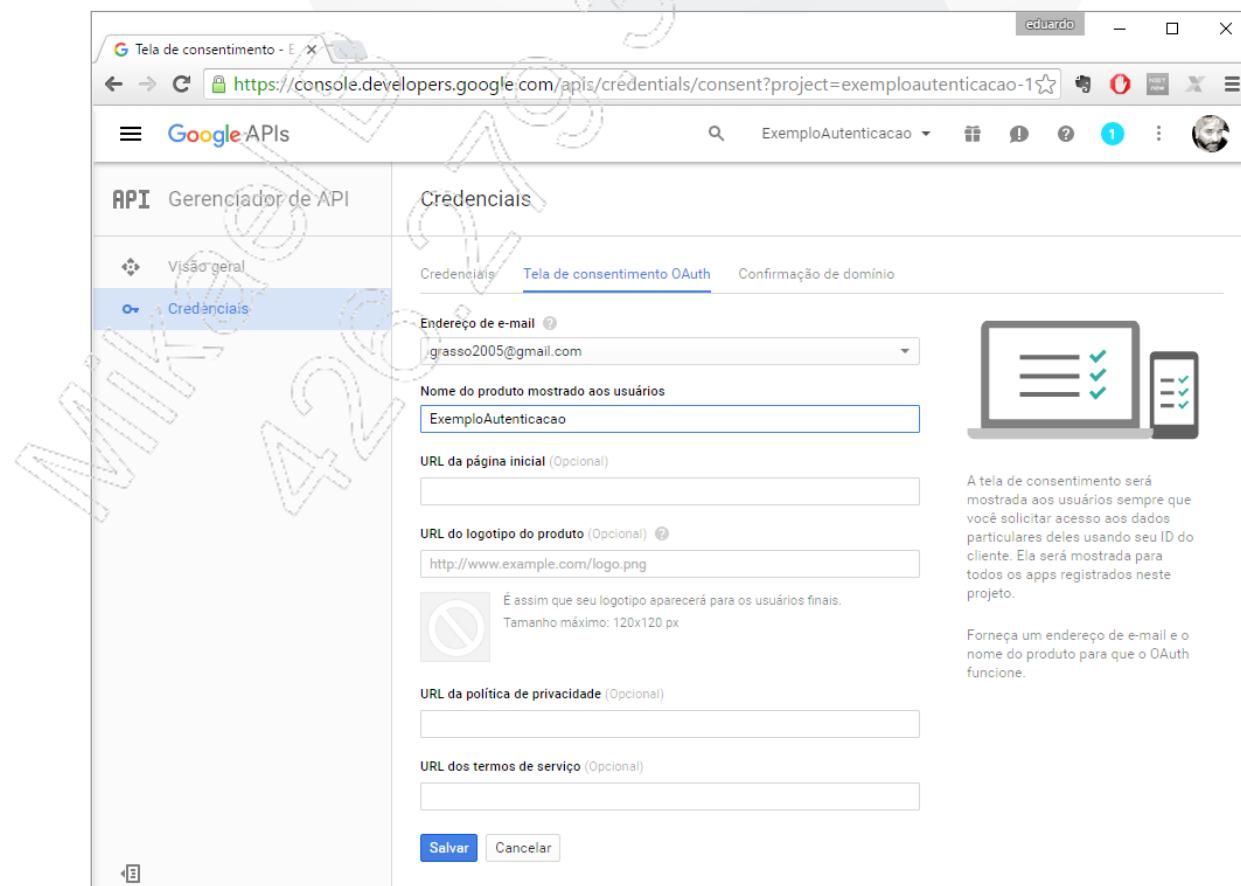
4. Espere até chegar na tela inicial de um projeto e veja se o nome do seu projeto aparece na tela;



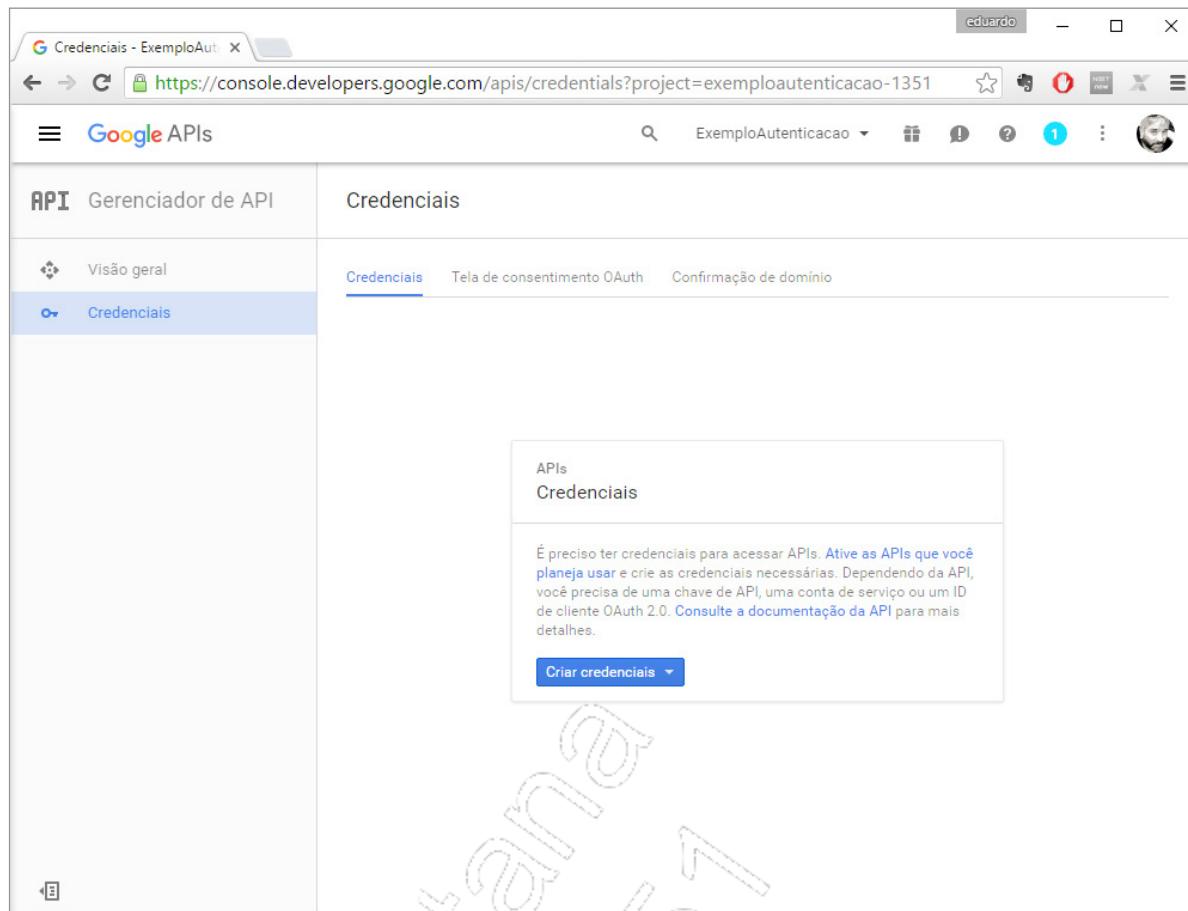
5. Clique em Credenciais;



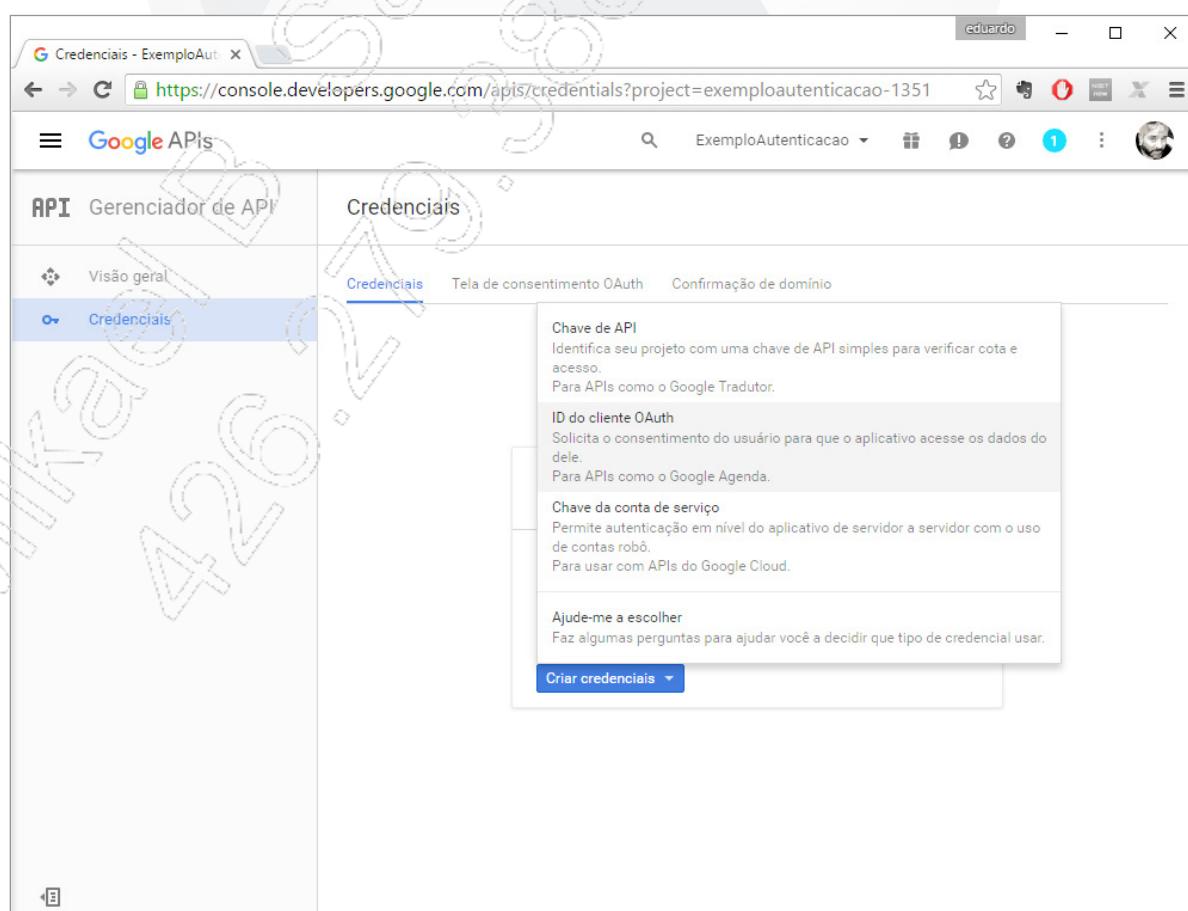
6. Clique em Tela de consentimento OAuth. Defina um nome do produto e clique em Salvar. Isso vai fazer voltar para a tela de credenciais;



Visual Studio 2015 - ASP.NET com C# Recursos Avançados



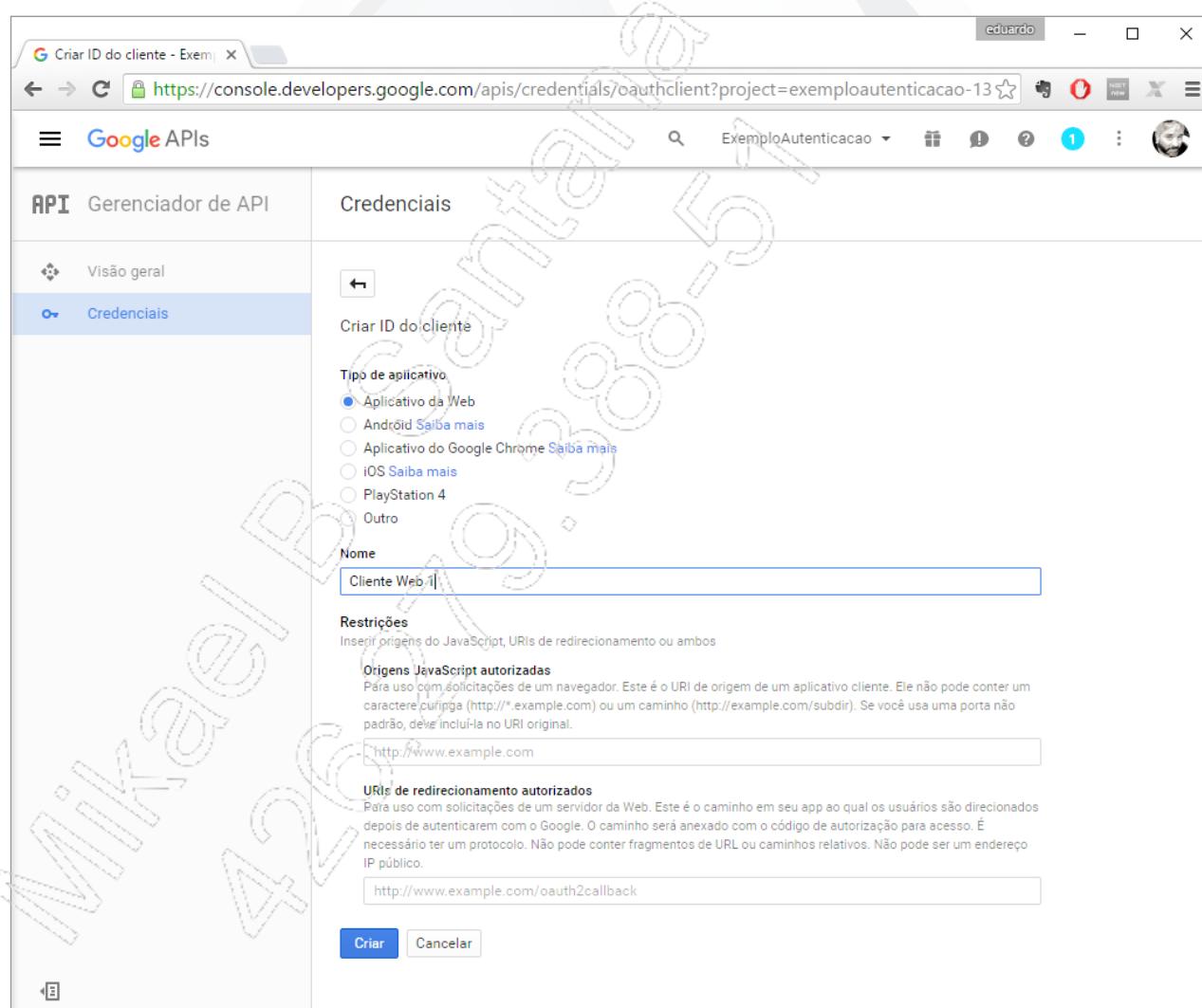
7. Clique em **Criar credenciais** e escolha **ID do cliente OAuth**;



8. O modo de criação de um aplicativo varia de acordo com o provedor. O Google o separa em diversos tipos: **Aplicativos da Web, Android, Aplicativo do Google, iOs, Playstation 4 e Outro**:

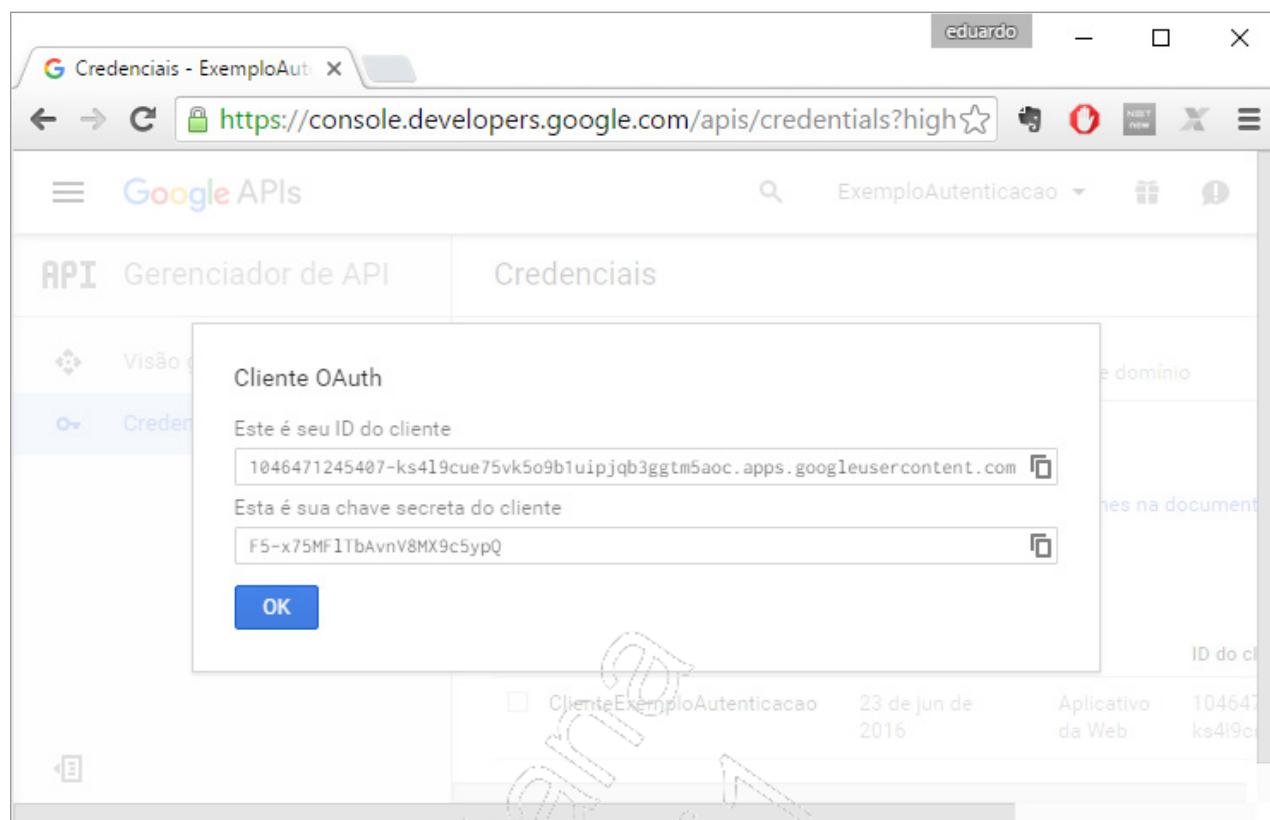
Nesse caso, deve ser selecionada a opção **Aplicativo Web**. A origem JavaScript autorizada é o endereço do servidor. O Google só permitirá que programas JavaScript vindo desse endereço tenham acesso a esse aplicativo. A URL de redirecionamento autorizada é o local de onde virá a solicitação de autenticação. Apenas páginas redirecionadas desse endereço para acesso a esse aplicativo serão aceitas.

Escolha um nome da credencial do cliente, defina as URLs de permissão e clique em **Criar**:

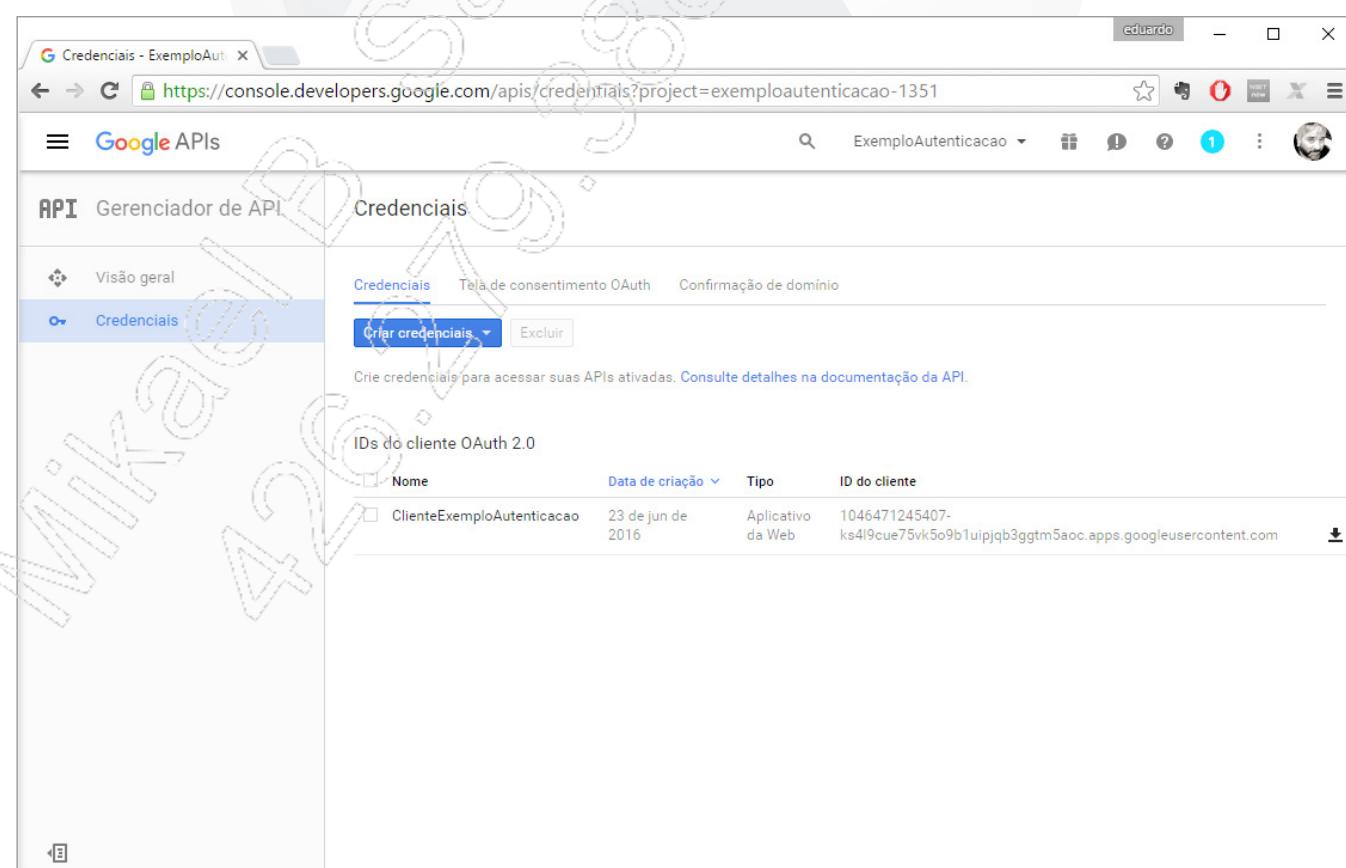


Visual Studio 2015 - ASP.NET com C# Recursos Avançados

9. Será criada uma ID de cliente e uma chave secreta;



10. Clique em OK e o sistema volta à tela de credenciais:



11. O aplicativo foi criado. O ID e chave secreta de cliente são informações importantes que devem ser passadas ao sistema do ASP.NET Identity quando for feita uma autenticação;

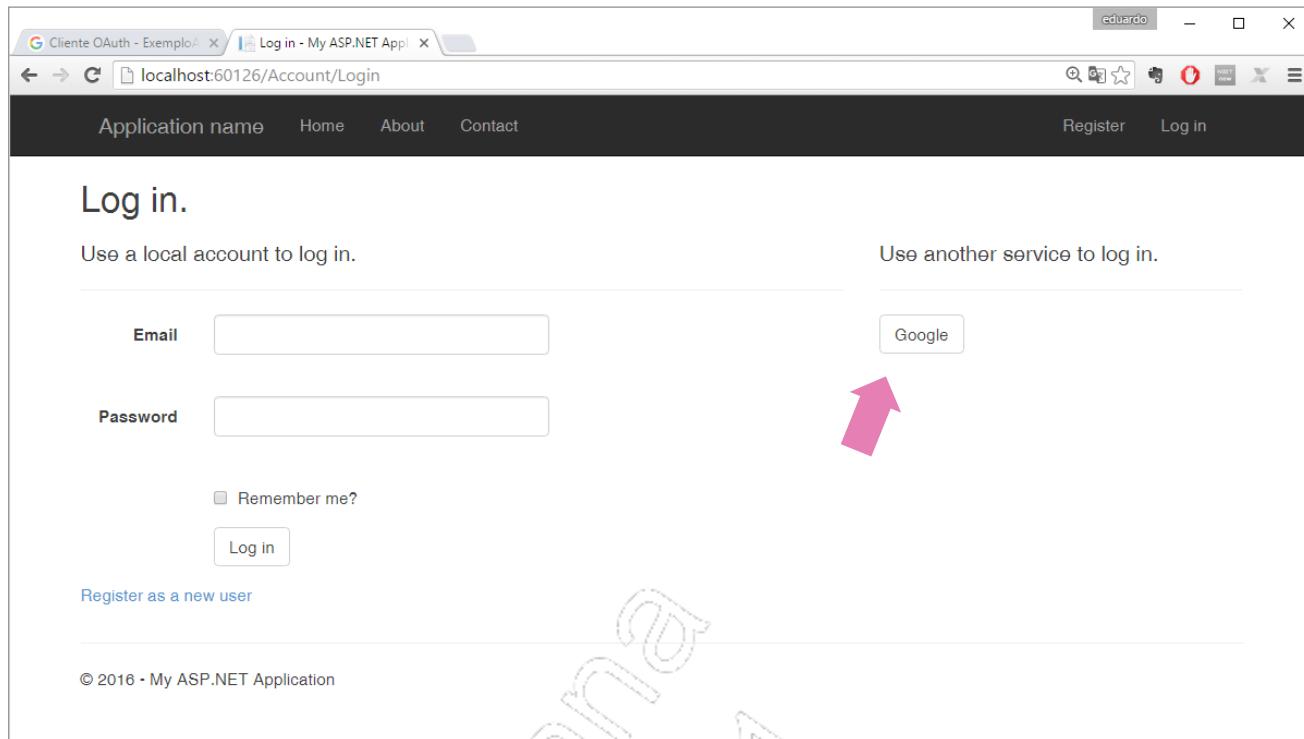
The screenshot shows the Google API Console interface. On the left, there's a sidebar with 'API' and 'Gerenciador de API' sections, and 'Visão geral' and 'Credenciais' items. A pink arrow points from the text above to the 'Credenciais' section. The main area is titled 'Credenciais' and shows a single client credential entry. It includes fields for 'ID do cliente' (1046471245407-ks4l9cue75vk5o9b1uipjqb3ggm5aoc.apps.googleusercontent.com), 'Chave secreta do cliente' (F5-x75MF1TbAvnV8MX9c5ypQ), and 'Data de criação' (23 de jun de 2016 06:55:30). Below these, there's a 'Nome' field containing 'ClienteExemploAutenticacao'. Under 'Restrições', it says 'Inserir origens do JavaScript, URIs de redirecionamento ou ambos' and lists 'Origens JavaScript autorizadas' as 'http://localhost'.

12. No arquivo **Startup.Auth.cs**, no método **UseGoogleAuthentication**, informe os dados obtidos no processo de criação dessa credencial (essa parte do código está comentada por padrão):

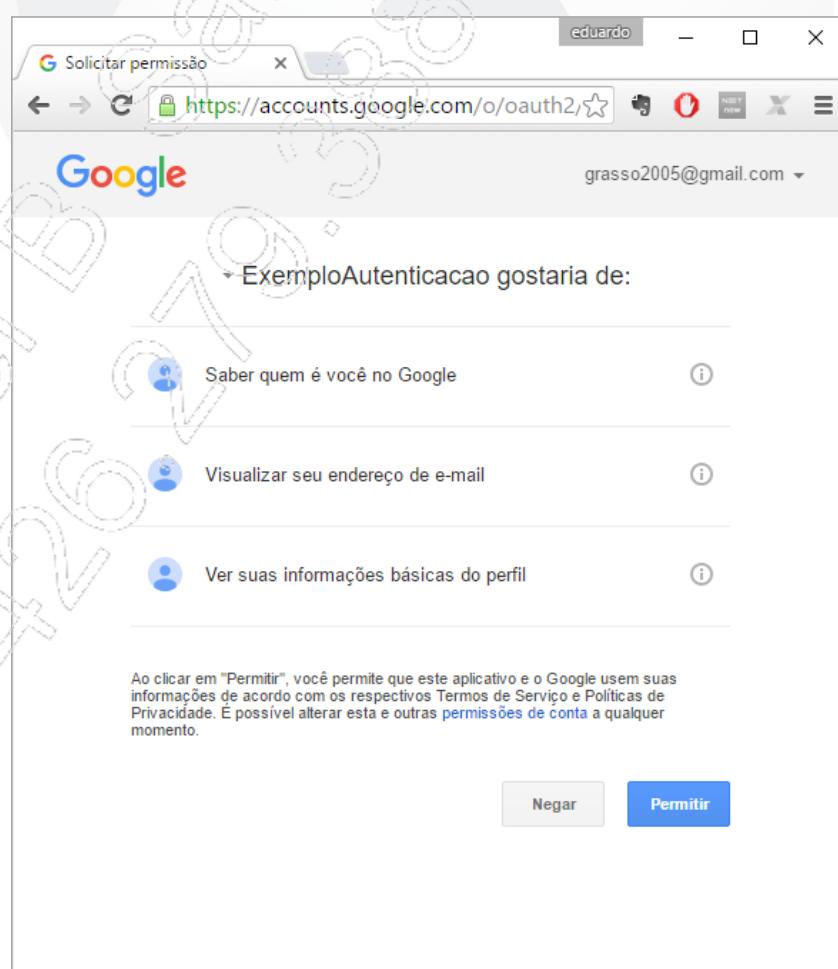
```
app.UseGoogleAuthentication(new
    GoogleOAuth2AuthenticationOptions()
{
    ClientId = "1046471245407-7...c.apps.googleusercontent.com",
    ClientSecret = "F5-x75MF1TbAvnV8MX9c5ypQ"
});
```

O método **UseGoogleAuthentication** é um método de extensão incluído com a biblioteca **Microsoft.Owin.Security.Google.Dll**, que é instalada via NuGet com o modelo padrão.

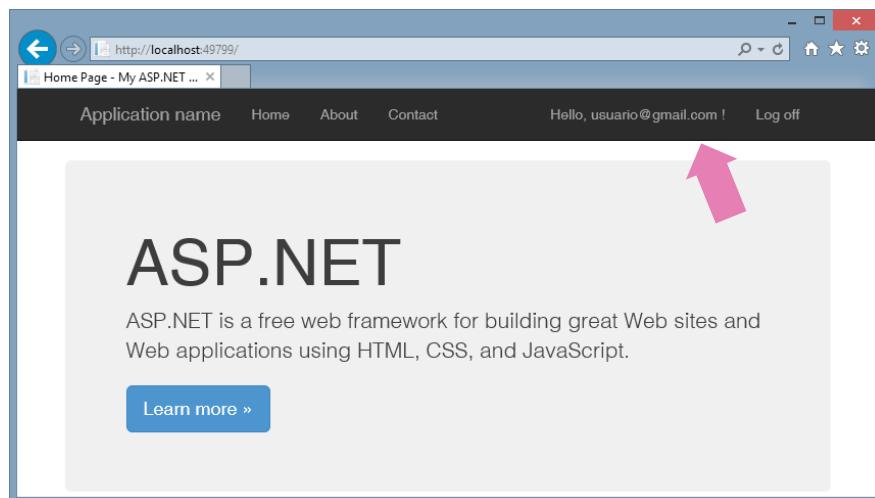
13. O processo está pronto. Ao executá-lo, a tela de login oferecerá a possibilidade de autenticação com o Google:



14. Ao clicar no botão do Google, a tela de login é exibida:



15. Depois de digitados usuário e senha, a autenticação é efetuada:



- **Como funciona (Web Forms)**

Na tela de login, aparece a autenticação possível por meio do Google, porque um User Control faz o papel de listar os autenticadores disponíveis:

- **Login.aspx**

```
<%@ Page Title="Log in" Language="C#" ...%>
<%@ Register Src="~/Account/OpenAuthProviders.ascx" ... %>

<asp:Content runat="server" ID="BodyContent">
    <h2><%: Title %></h2>
    ....
    <uc:OpenAuthProviders runat="server" ID="OpenAuthLogin" />
</asp:Content>
```

Abrindo esse User Control, existe um Web Control **ListView** que chama o método **GetProviderNames** para preencher os itens internos (a listagem foi simplificada para maior clareza).

- **OpenAuthProviders.ascx**

```
<%@ Control %>

<div id="socialLoginList">

    <h4>Use another service to log in.</h4>

    <asp:ListView
        runat="server"
        ID="providerDetails"
        ItemType="System.String"
        SelectMethod="GetProviderNames"
        ViewStateMode="Disabled">
        <ItemTemplate>

            <button
                type="submit"
                class="btn btn-default"
                name="provider"
                value="<%#: Item %>">
                </button>
            </p>
        </ItemTemplate>

        <EmptyDataTemplate>
            There are no external
            authentication services configured.
        </EmptyDataTemplate>
    </asp:ListView>
</div>
```

Esse método está definido na classe de code-behind e chama os tipos de autenticação definida usando o método **GetExternalAuthenticationTypes()** da classe **AuthenticationManager**, que retorna um array com objetos do tipo **Microsoft.Owin.Security.AuthenticationDescription**.

Essa classe (**Authentication Description**) contém uma propriedade chamada **AuthenticationType**, que é uma string com a descrição da autenticação. Usando **Linq** e o método **Select**, é retornado um **IEnumerable<string>** com a lista dos tipos de autenticação registrados.

```
public class OpenAuthProviders ....  
{  
    protected void Page_Load ....  
  
    public IEnumerable<string> GetProviderNames()  
    {  
        return Context.GetOwinContext().Authentication  
            .GetExternalAuthenticationTypes()  
            .Select(t => t.AuthenticationType);  
  
    }  
}
```

Essa página em si cria uma lista de botões do tipo **submit** com a lista dos autenticadores registrados. Cada botão faz um **post-back**, passando o nome do autenticador.

Caso tenha sido feito um **post-back**, o código do método **Page_Load** é acionado. A primeira parte verifica se foi passado o nome de um provider; se não foi passado, abandona a procedure.

```
if (IsPostBack)  
{  
    var provider = Request.Form["provider"];  
    if (provider == null)  
    {  
        return;  
    }  
  
    ....  
}
```

O processamento será redirecionado para o servidor externo. A variável **redirectUrl** armazena para onde vai a resposta do servidor de autenticação.

```
string redirectUrl = ResolveUrl(String.Format(
    CultureInfo.InvariantCulture,
    "~/Account/RegisterExternalLogin?{0}={1}&returnUrl={2}",
    IdentityHelper.ProviderNameKey,
    provider,
    ReturnUrl));
```

Existe o cuidado do ASP.NET de evitar um ataque chamado **CSRF (Cross-Site Request Forgery)**, em que um usuário autenticado, ao entrar em um site malicioso, pode enviar um formulário para um servidor. Para garantir que todos os posts venham do mesmo domínio do aplicativo Web, o ASP.NET implementa um recurso anti-CSRF, que consistem em introduzir um campo oculto no formulário e um cookie. Essa parte do código usa o ID do usuário como chave para validar a origem da requisição e evitar, com isso, um ataque CSRF.

```
var properties = new AuthenticationProperties()
{ RedirectUri = redirectUrl };

if (Context.User.Identity.IsAuthenticated)
{
    properties.Dictionary[IdentityHelper.XsrfKey] =
    Context.User.Identity.GetUserId();
}
```

Chegamos à parte principal do código, em que a página chama os componentes OWIN para que invoquem o autenticador externo.

```
Context.GetOwinContext().Authentication.
Challenge(properties, provider);
```

Neste momento, a página está no status "não autorizada" e deverá ter sido redirecionada pela resposta do servidor de autenticação.

```
Response.StatusCode = 401;
Response.End();
```

O servidor de autenticação recebe a solicitação para autenticar, autentica o usuário e redireciona para a página informada nos dicionários dos componentes OWIN.

O arquivo **RegisterExternalLogin.aspx** realiza o resto do processamento. O evento **Page_Load** executa as funções adiante.

Se não foi passado o nome de um provedor de autenticação por meio da URL, sai do método **Page_Load**:

```
ProviderName = IdentityHelper.  
GetProviderNameFromRequest(Request);  
if (String.IsNullOrEmpty(ProviderName))  
{  
    RedirectOnFail();  
    return;  
}
```

Se não for um **post-back**, ou seja, se a página estiver vindo redirecionada de um servidor de autenticação, o seguinte código é executado:

```
// Obtém o IUserManager  
var manager = Context  
    .GetOwinContext()  
    .GetUserManager<ApplicationUserManager>();  
  
// Obtém o ExternalLoginInfo  
var loginInfo = Context  
    .GetOwinContext()  
    .Authentication.GetExternalLoginInfo();  
  
// Se não obteve informações sobre o login, sai do evento  
if (loginInfo == null)  
{  
    RedirectOnFail();  
    return;  
}
```

```
// Procura o usuário atual na coleção de usuários do site
var user = manager.Find(loginInfo.Login);

// Se encontrou, faz o login e redireciona para a url padrão
if (user != null)
{
    IdentityHelper.SignIn(manager, user, isPersistent: false);
    IdentityHelper
        .RedirectToReturnUrl(Request.QueryString["ReturnUrl"],
Response);
}

// Se não encontrou e existe um usuário do site autenticado
else if (User.Identity.IsAuthenticated)
{
    [parte do código do usuário autenticado]
}

// Se não encontrou e não existe um usuário do site autenticado,
// o e-mail do
// usuário externo vai ser usado para cadastrar o usuário do site
else
{
    email.Text = loginInfo.Email;
}
```

Se existe um usuário autenticado no site e não foi encontrado externamente na lista de usuários, esse usuário deve ser cadastrado. O código a seguir está na parte do trecho anterior marcada como **[parte do código do usuário autenticado]**. Neste ponto é que entra a proteção contra ataques **Cross-Site Request Forgery**.

```
// Obtém dados do usuário autenticado pelo autenticador externo
var verifiedloginInfo = Context
    .GetOwinContext()
    .Authentication
    .GetExternalLoginInfo(
        IdentityHelper.XsrfKey,
        User.Identity.GetUserId())
    ;

//Se não é um usuário válido, sai.
if (verifiedloginInfo == null)
{
    RedirectOnFail();
    return;
}

//Este é o ponto principal. O usuário (verificado) é adicionado.
var result = manager.AddLogin(User.Identity.GetUserId(),
    verifiedloginInfo.Login);

//Se a inclusão foi um sucesso, vai para a url padrão
if (result.Succeeded)
{
    IdentityHelper.RedirectToReturnUrl(
        Request.QueryString["ReturnUrl"], Response);
}
else
{
    AddErrors(result);
}
```

A página **RegisterExternalLogin.aspx** contém algumas propriedades utilizadas no processo. São elas: **ProviderName** e **ProviderAccountKey**.

```
protected string ProviderName
{
    get { return (string)ViewState["ProviderName"] ?? String.
Empty; }
    private set { ViewState["ProviderName"] = value; }

}

protected string ProviderAccountKey
{
    get { return (string)ViewState["ProviderAccountKey"] ?? 
String.Empty; }
    private set { ViewState["ProviderAccountKey"] = value; }
}
```

A propriedade **ProviderName** é utilizada na mensagem da página: "Você foi autenticado por [Nome do Provedor]. Insira seu e-mail e clique no botão Login":

```
<p class="text-info">
    You've authenticated with <strong><%: ProviderName %></
    strong>.
    Please enter an email below for the current site
    and click the Log in button.
</p>
```

Ao clicar no botão **Login**, o usuário é criado. Note que o código é bastante simples:

```
//Obtém o gerenciador de usuário
var manager= Context.GetOwinContext()
    .GetUserManager<ApplicationUserManager>();

//Cria uma instância de IUser
var user = new ApplicationUser() { UserName = email.Text,
    Email = email.Text };

//Chama o método Create para criar um usuário
IdentityResult result = manager.Create(user);
```

Se o usuário foi criado com sucesso, este login é adicionado aos logins desse usuário. No conceito do ASP.NET Identity, um usuário pode ter vários logins.

```
// Se houve sucesso
if (result.Succeeded)
{

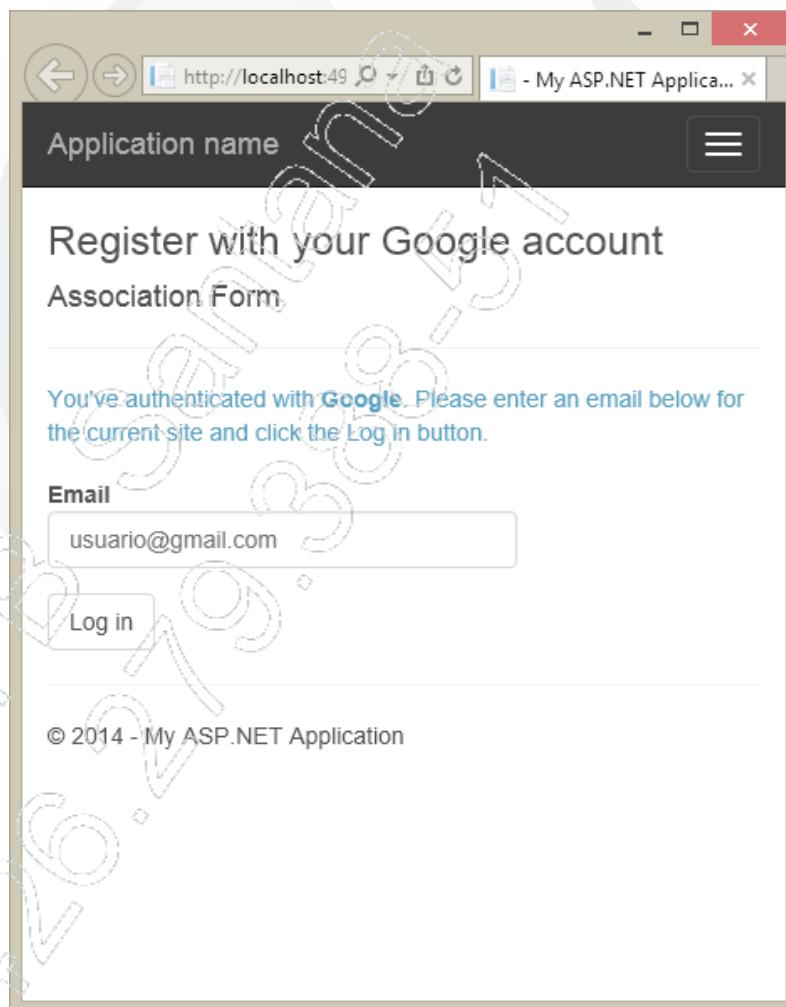
    // Obtém informações do Login
    var loginInfo = Context.GetOwinContext().Authentication
        .GetExternalLoginInfo();
    // Se não obteve as informações de login, sai
    if (loginInfo == null)
    {
        RedirectOnFail();
        return;
    }

    // Adiciona este login externo a este usuário
    result = manager.AddLogin(user.Id, loginInfo.Login);

    // Se este processo funcionou, faz o login e redireciona
    if (result.Succeeded)
    {
        IdentityHelper.SignIn(manager, user, isPersistent:
false);
        IdentityHelper.RedirectToReturnUrl(
            Request.QueryString["ReturnUrl"], Response);
        return;
    }
}
// Se chegou até aqui, algo deu errado
AddErrors(result);
```

O método **AddErrors** adiciona os erros vindos da coleção **Errors** do **IdentityResult** para a coleção de erros da página.

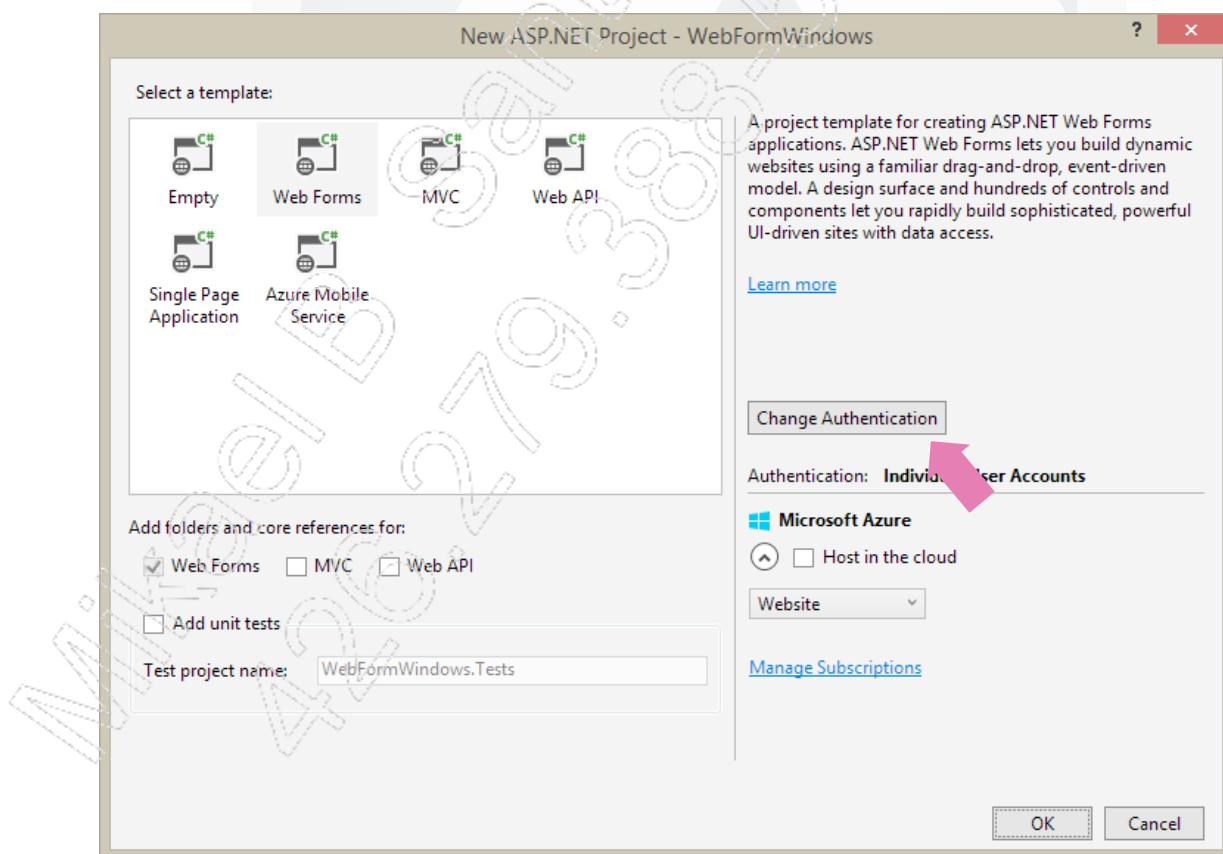
```
private void AddErrors(IdentityResult result)
{
    foreach (var error in result.Errors)
    {
        ModelState.AddModelError("", error);
    }
}
```



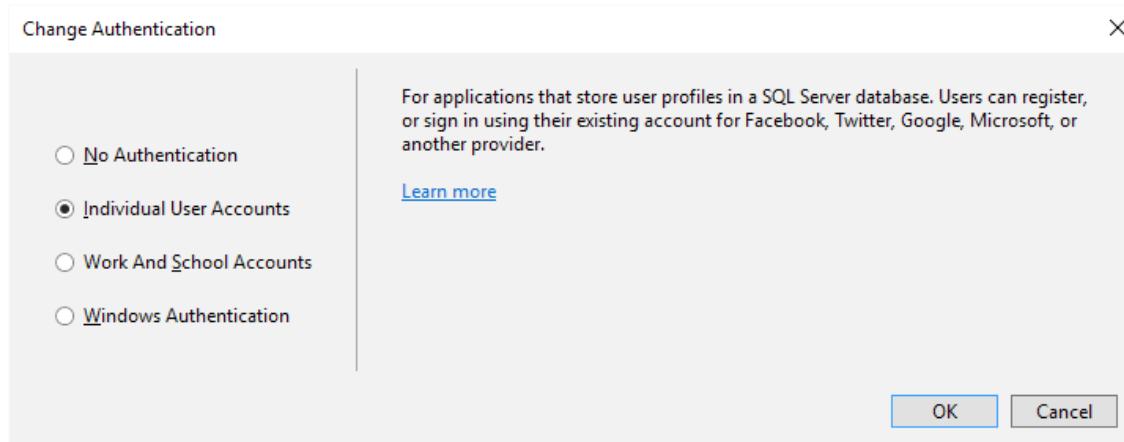
5.4.4. Web Forms e Windows

Se a aplicação Web estiver sendo utilizada apenas internamente e todos os usuários forem usuários autenticados pelo Windows, não é necessário criar um sistema de login. O usuário que fez o login no computador é que estará usando o aplicativo. Repare que essa opção limita totalmente o acesso externo de usuários de outros sistemas operacionais. Apenas computadores na mesma rede (ou acesso via VPN) onde está sendo executada a aplicação terão acesso. Em alguns casos, isso pode fornecer uma boa dose de segurança, principalmente para aplicações gerenciais.

A autenticação via Windows foi a primeira a ser utilizada pelo ASP.NET e precisa de muito pouco para funcionar. Ao iniciar um novo projeto Web, escolha a opção **Change Authentication**:



E, em seguida, escolha a opção **Windows Authentication**:



No arquivo de configuração, **web.config**, é definida a autenticação para o modo Windows:

```
<authentication mode="Windows" />
```

Ao mesmo tempo, é bloqueado o acesso de usuários anônimos a qualquer parte da aplicação Web:

```
<authorization>
  <deny users="?" />
</authorization>
```

O símbolo de interrogação (?) significa "não autenticados". Para criar um site onde todos os usuários, autenticados ou não, sejam bloqueados, usa-se o símbolo do asterisco (*):

```
<authorization>
  <deny users="*" />
</authorization>
```

Embora possa parecer sem sentido um site assim, se usarmos a combinação certa de permissões e bloqueios, é possível criar áreas restritas. Por exemplo, considere o comando a seguir:

```
<authorization>
    <allow users="admin" />
    <deny users="*" />
</authorization>
```

Nesse exemplo, um usuário (autenticado) chamado **admin** é liberado a usar o aplicativo e todos os outros, autenticados ou não, são proibidos. O web.config funciona dentro de pastas, portanto esta limitação não precisa ser no site todo, apenas em uma área administrativa.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- OWIN é uma especificação que define como os componentes de uma aplicação Web interagem com componentes .NET;
- OAuth é um protocolo que define formas de autenticação de usuários entre aplicações e servidores;
- IUser é a interface mínima para criação de um usuário;
- IUserManager é a interface mínima para criação de um repositório de usuários;
- Existem três possíveis autenticações: Windows, contas individuais ou contas organizacionais;
- Por meio de componentes OWIN e o protocolo OAuth, é possível autenticar usando um servidor de autenticação externo como Facebook ou Google.

5

Segurança (NET 4.6)

Teste seus conhecimentos

Mikael B
426.279.57



IMPACTA
EDITORA

1. Qual é a interface utilizada para definir um usuário no ASP.NET Identity?

- a) UserManager
- b) OWIN
- c) ApplicationUser
- d) IUser
- e) User.Identity

2. Qual interface deve ser implantada na classe que armazena dados de um usuário no ASP.NET Identity?

- a) IUser
- b) IStore
- c) IUserDb
- d) UserManager
- e) UserStore

3. A classe que gerencia as informações de um usuário deve implementar qual interface?

- a) IUser
- b) IUserStore
- c) IUserManager
- d) IUserIdentity
- e) OWIN

4. Que tipo é retornado pelo método Create da classe que gerencia os dados de um usuário?

- a) IdentityResult
- b) User
- c) Principal
- d) Role
- e) ApplicationUser

5. Como se chama a especificação que utiliza o conceito de Token e é utilizada para criar logins com servidores externos?

- a) ASP.NET Identity
- b) Nodes
- c) OData
- d) OAuth
- e) Facebook API

5

Segurança (.NET 4.6)

Mãos à obra!

Mikael B
426.279.57



IMPACTA
EDITORA

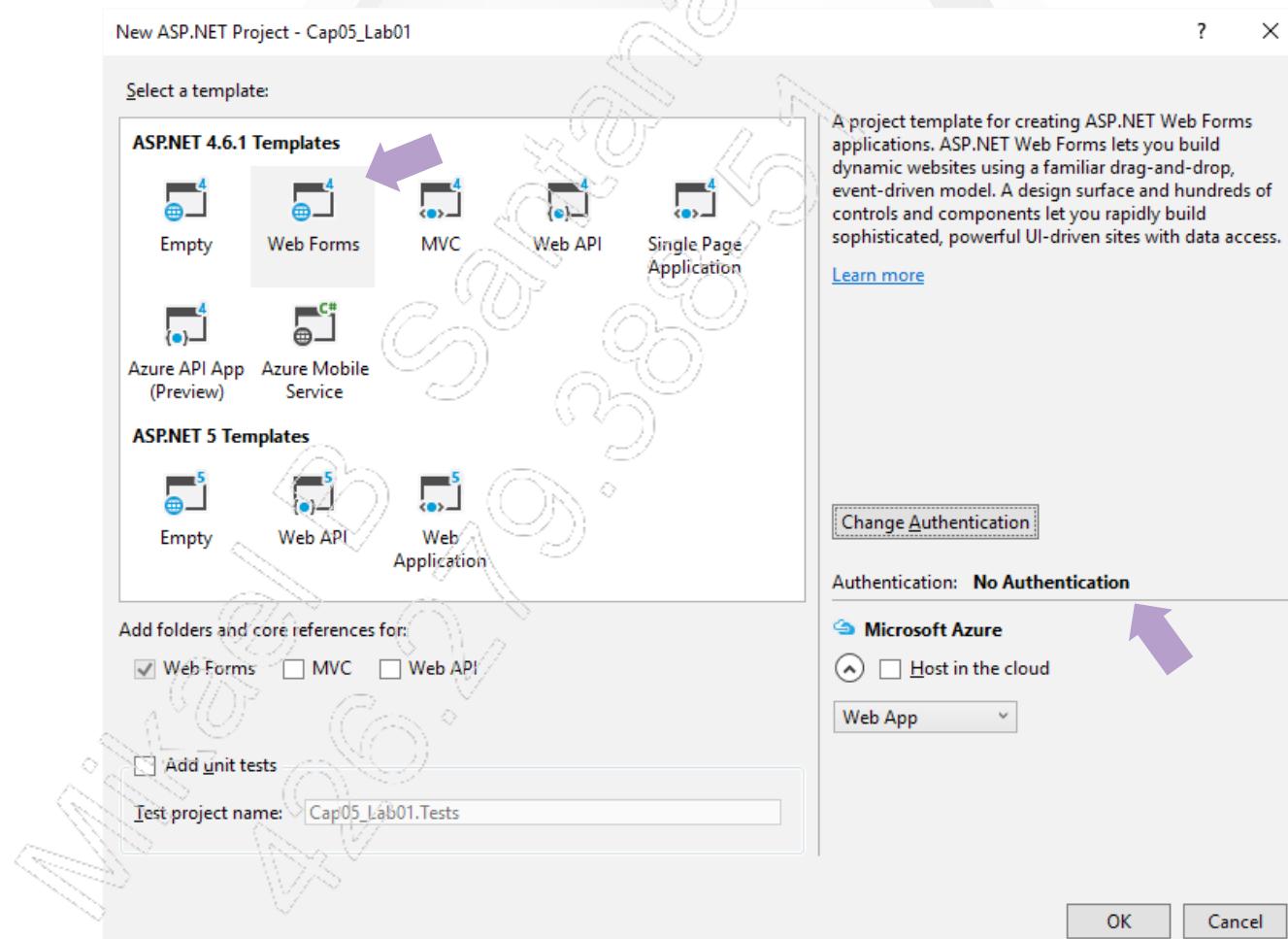
Laboratório 1

A – Criando um login simples com o ASP.NET Identity

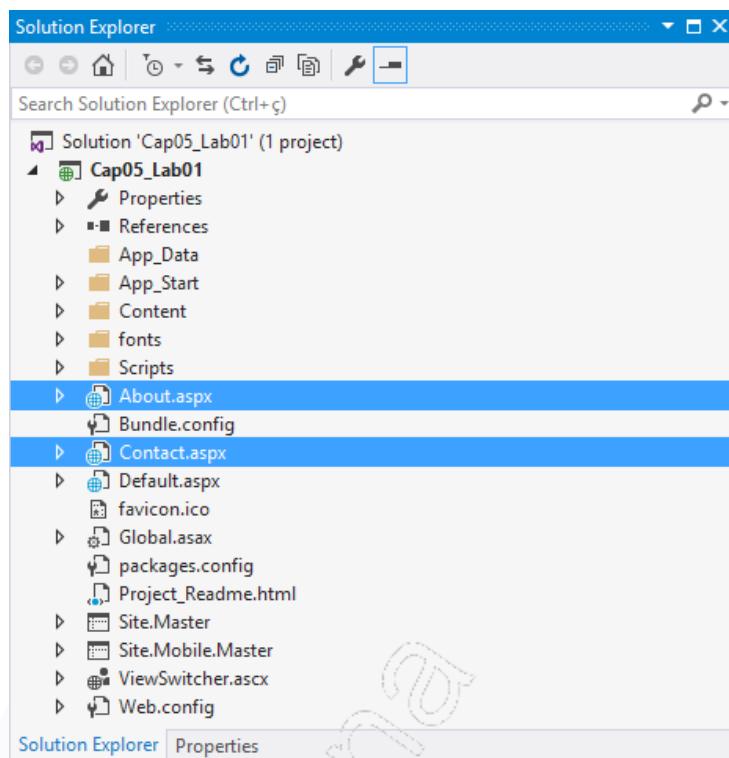
Neste laboratório, você criará um login simples usando ASP.NET Identity. Para isso, utilizará o modelo Web Forms para layout, OWIN para autenticação e User Entity Framework para persistência dos dados.

Siga os passos adiante:

1. Crie um novo projeto do tipo **ASP.NET Web Forms App**, sem autenticação (**No Authentication**), chamado **Cap05_Lab01**;



2. Exclua as páginas **About.aspx** e **Contact.aspx**:



3. Na Master Page, altere o título do aplicativo:

```
<title><%: Page.Title %> - WebApp</title>
```

4. Ainda na Master Page, altere os links do menu para as páginas **Início**, **Área Livre** e **Área Restrita**. Como estamos usando **FriendlyUrls**, não é preciso informar a extensão **.aspx**:

```
<ul class="nav navbar-nav">
    <li><a runat="server" href="/">Início</a></li>
    <li><a runat="server" href="/Livre">Área Livre</a></li>
    <li><a runat="server" href="/Restrita">Área Restrita</a></li>
</ul>
```

5. Altere o rodapé:

```
<footer>
    <p>&copy; <%: DateTime.Now.Year %> -
        Desenvolvido para o curso de ASP.NET
    </p>
</footer>
```

6. Na página **default.aspx**, apague todo o conteúdo:

```
<%@ Page Title="Home Page" Language="C#" ... %>

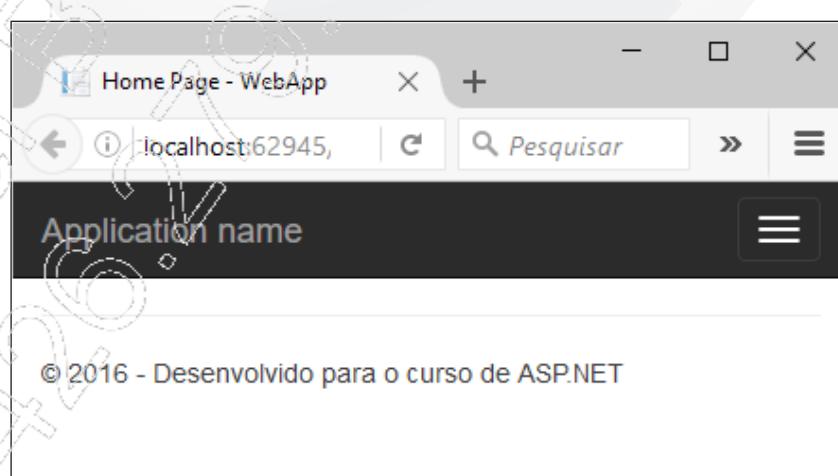
<asp:Content ID="BodyContent" ContentPlaceHolderID="Ma ...>

</asp:Content>
```

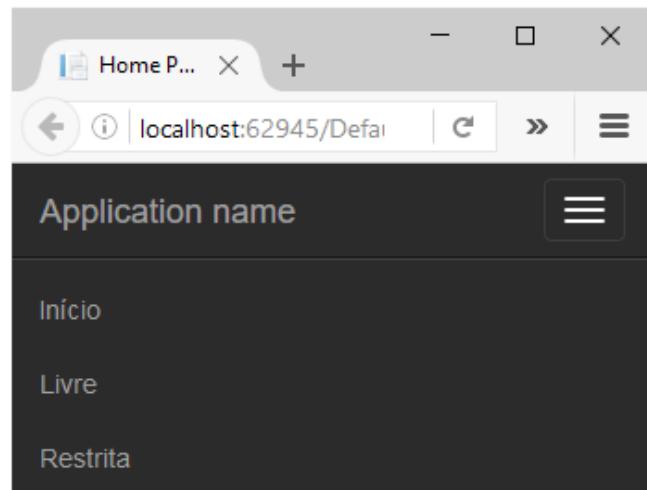
7. Teste o layout. Inicie em tela cheia:



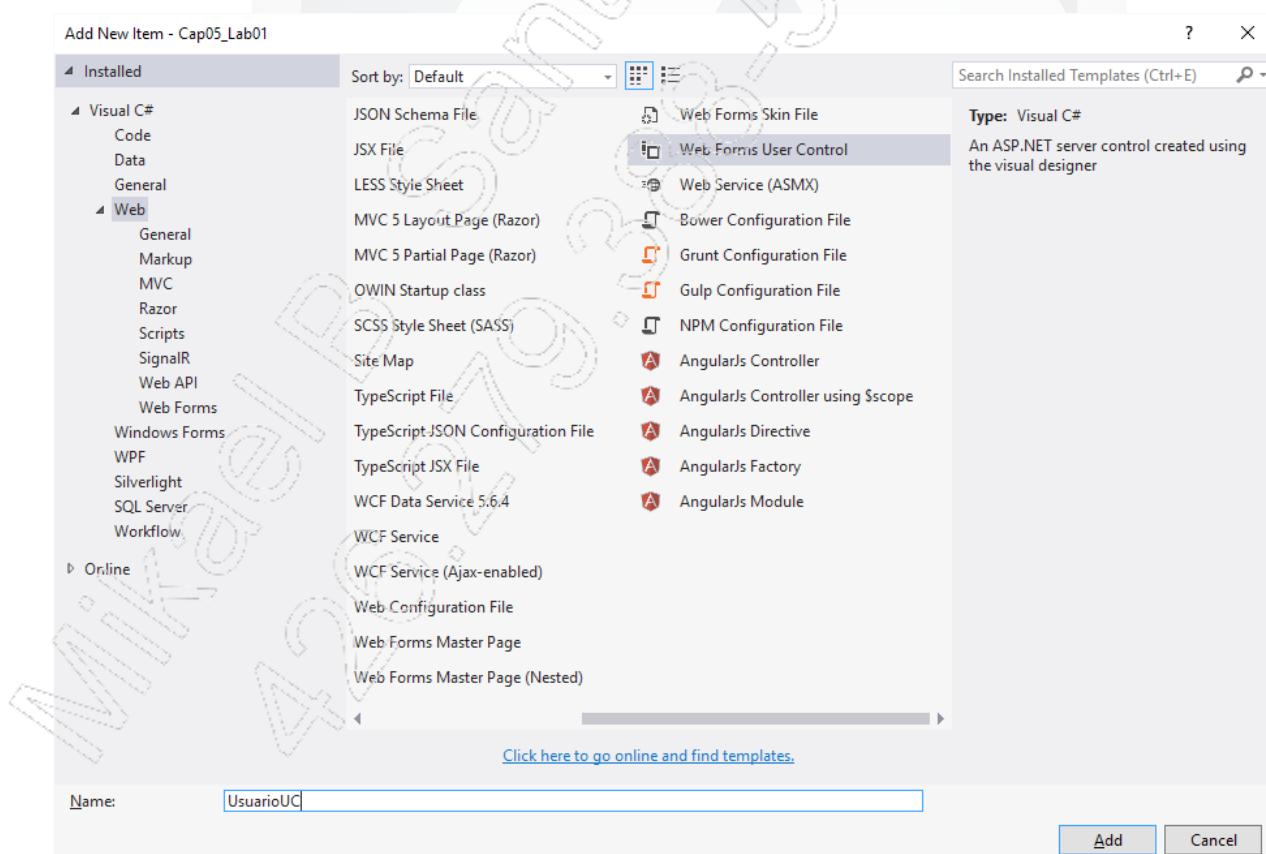
8. Em uma resolução menor, verifique se o menu se adapta:



9. Verifique se o menu se expande ao clicar no botão:



10. Adicione um User Control chamado **UsuarioUC.ascx**. Ele exibirá o nome do usuário e o botão **Login**. O Web Control **LoginView** fornece dois templates para serem exibidos quando o usuário está autenticado e quando não está:



Visual Studio 2015 - ASP.NET com C# Recursos Avançados

```
<%@ Control Language="C#" ... %>

<%-- WebControl LoginView --%>
<asp:LoginView runat="server" ViewStateMode="Disabled">

    <%-- Modelo para usuário não autenticado --%>
    <AnonymousTemplate>
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a runat="server" href("~/Usuario/Criar">
                    Cadastrar-se</a></li>

            <li>
                <a runat="server" href "~/Usuario/Login">
                    Log in</a></li>
        </ul>
    </AnonymousTemplate>

    <%-- Modelo para usuário autenticado --%>
    <LoggedInTemplate>
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a href="#"><%: Page.User.Identity.Name %></a>
            </li>

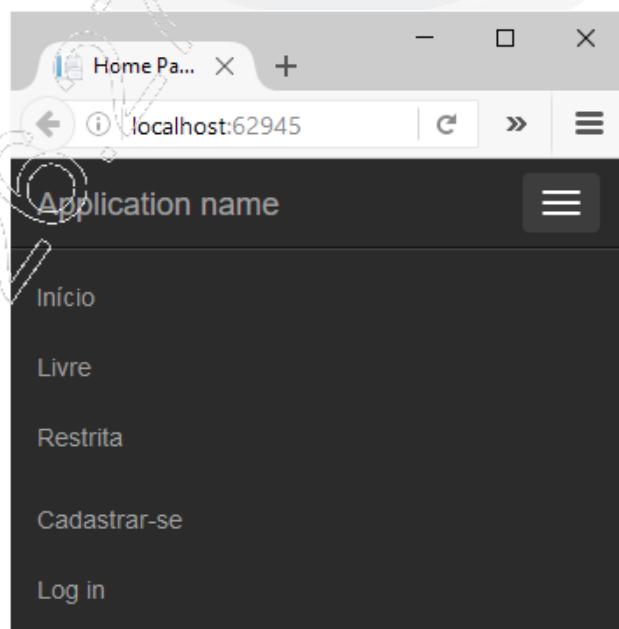
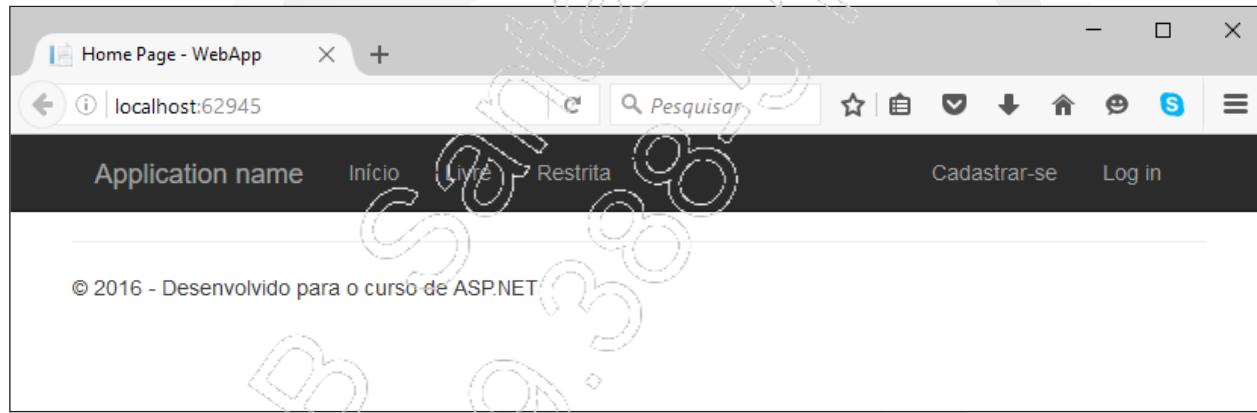
            <li>
                <asp:LoginStatus
                    runat="server"
                    LogoutAction="Redirect"
                    LogoutText="Log off"
                    LogoutPageUrl="~/usuario/logout.aspx" />
            </li>
        </ul>
    </LoggedInTemplate>
</asp:LoginView>
```

11. Insira esse User Control logo abaixo do menu na Master Page. Lembre-se de arrastá-lo da janela Solution Explorer para o seu código para que o Visual Studio crie o registro da tag:

```
<!-- Menu -->
<ul class="nav navbar-nav">
<li><a runat="server" href("~/")>Início</a></li>
<li><a runat="server" href "~/Livre">Área Livre</a></li>
<li><a runat="server" href "~/Restrita">Área Restrita</a></li>
</ul>
```

```
<!-- Usuário User Control -->
<uc1:UsuarioUC runat="server" ID="UsuarioUC" />
```

12. Teste o layout em várias resoluções de tela:



13. Crie uma pasta chamada **Usuario**. Dentro dela, crie dois Web Forms baseados na Master Page, denominados **Login.aspx** e **Criar.aspx**. Coloque apenas o título em cada um:

```
<%@ Page Title="Login" Language="C#" ... %>
<asp:Content ID="Content1" ...>

<h2>Login</h2>

</asp:Content>
```

```
<%@ Page Title="Criar" Language="C#" ... %>
<asp:Content ID="Content1" ...>

<h2>Registro de Usuário</h2>

</asp:Content>
```

14. Na raiz do site, crie as páginas **Restrita.aspx** e **Livre.aspx** e coloque os títulos:

```
<%@ Page Title="Restrita" Language="C#" ... %>
<asp:Content ID="Content1" ...>

<h2>Área Restrita</h2>

</asp:Content>
```

```
<%@ Page Title="Livre" ... %>
<asp:Content ID="Content1" ...>

<h2>Livre</h2>

</asp:Content>
```

15. Teste o layout e a navegação do site:

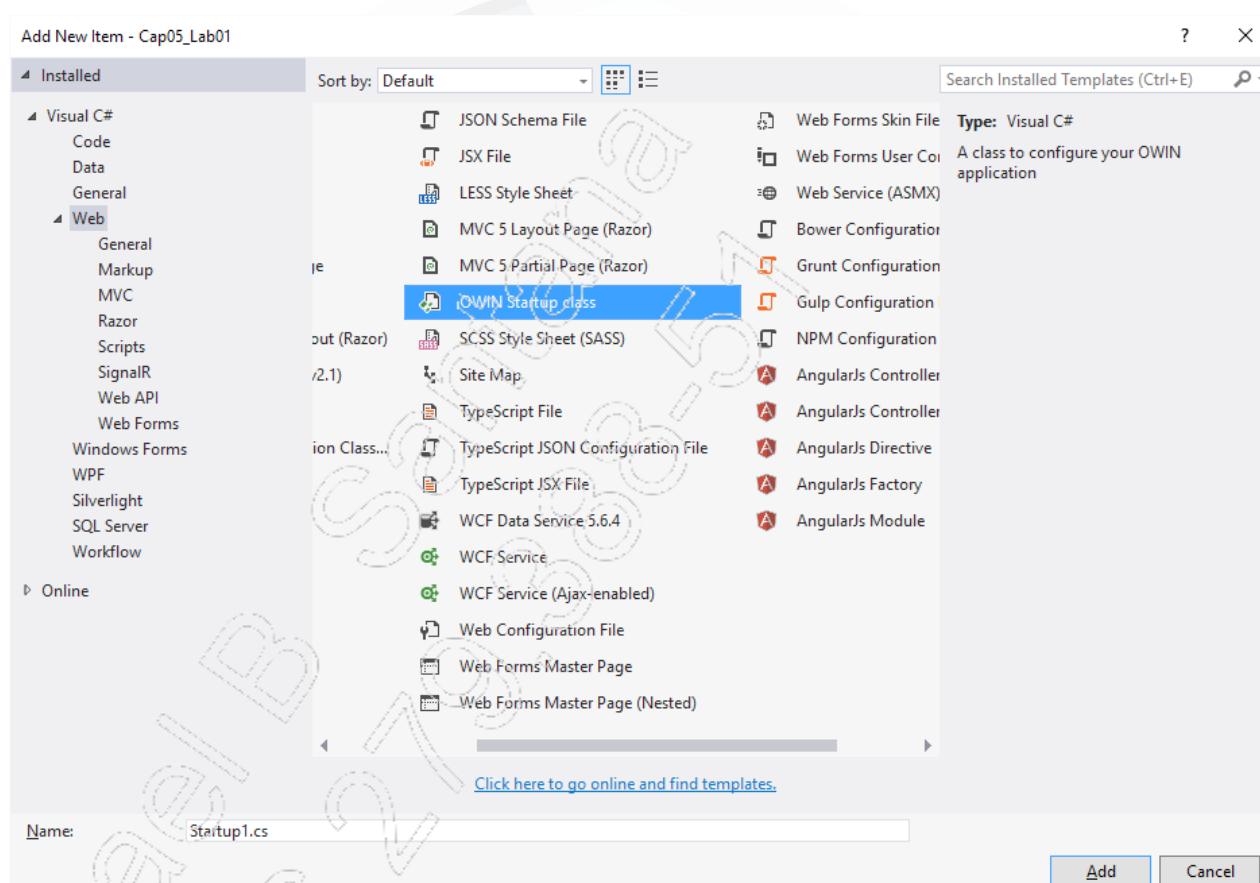
The image displays three separate browser windows, each showing a different page of a web application. All three windows have a similar layout with a dark header bar containing the application name 'WebApp' and navigation links for 'Início', 'Área Livre', 'Área Restrita', 'Cadastrar-se', and 'Log in'. The footer of each page includes the text '© 2014 - Desenvolvido para o curso de ASP.NET'.

- Livre**: The first window shows the 'Livre' page. The title 'Livre' is centered above a horizontal line. Below the line, there is some very small, illegible text.
- Usuário - Criar - WebApp**: The second window shows the 'Registro de Usuário' (User Registration) page. The title 'Registro de Usuário' is centered above a horizontal line. Below the line, there is some very small, illegible text.
- Login - WebApp**: The third window shows the 'Login' page. The title 'Login' is centered above a horizontal line. Below the line, there is some very small, illegible text.

16. Nesta segunda parte, vamos implementar o registro de usuário usando ASP.NET Identity e a autenticação com componentes OWIN. Adicione os seguintes pacotes usando NuGet:

- **Microsoft ASPNET Identity EntityFramework;**
- **Microsoft ASPNET Identity OWIN;**
- **Microsoft OWIN Host SystemWeb.**

17. Na raiz, insira o arquivo de configuração OWIN, denominado **Startup.cs**:



18. No arquivo **Startup.cs**, defina a autenticação via cookies e o caminho da página de login:

```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;
using Microsoft.Owin.Security.Cookies;
using Microsoft.AspNet.Identity;
```

```
[assembly: OwinStartup(typeof(ExemploIdentity.Startup))]

namespace ExemploIdentity
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {

            app.UseCookieAuthentication(
                new CookieAuthenticationOptions
                {
                    AuthenticationType =
                    DefaultAuthenticationTypes.ApplicationCookie,
                    LoginPath = new PathString("/Usuario/Login.aspx")
                }
            );
        }
    }
}
```

19. No arquivo **Web.config**, configure o modo de autenticação para nenhum:

```
<system.web>
    <authentication mode="None">
    </authentication>
</system.web>
```

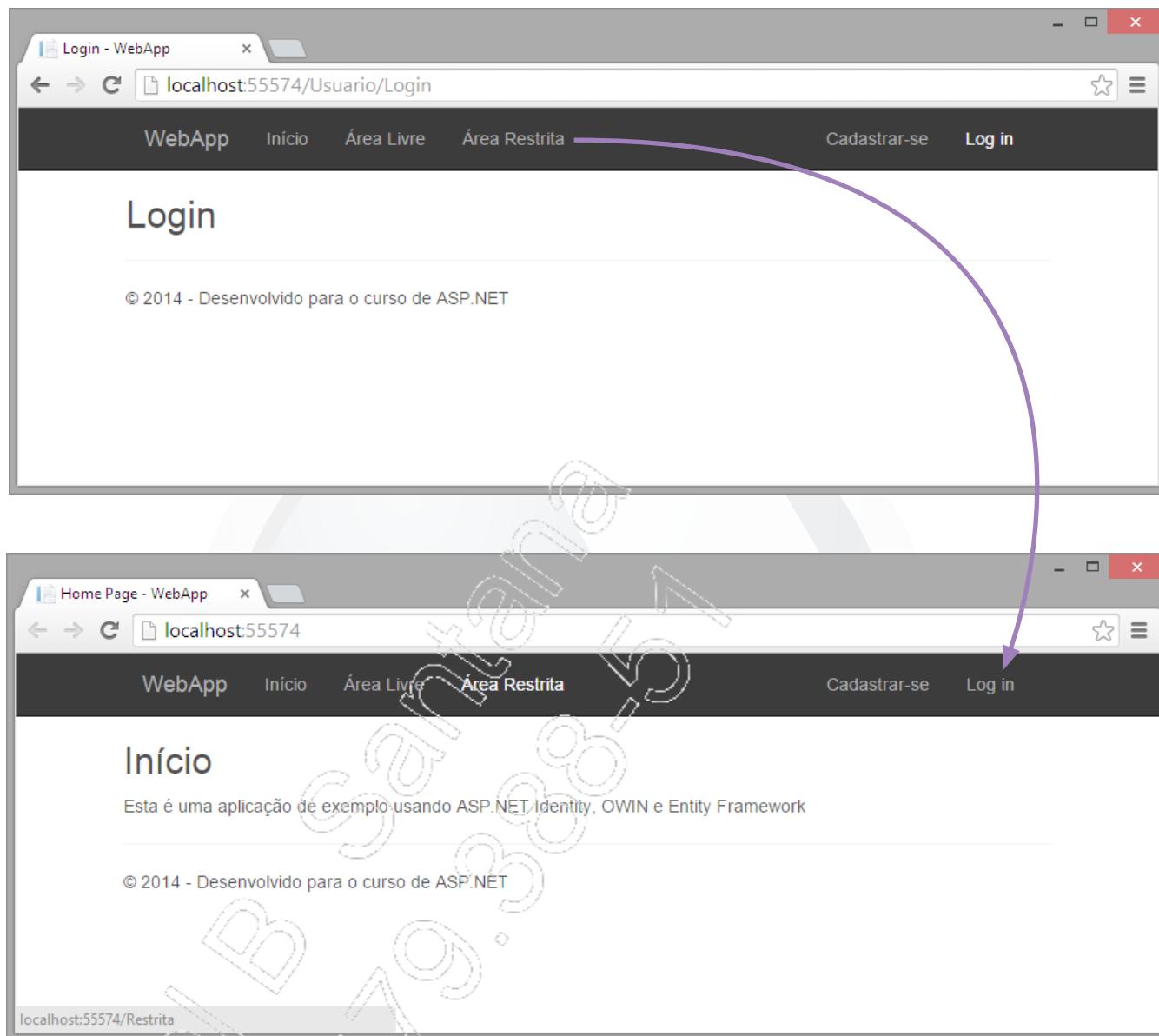
20. Ainda no arquivo **Web.config**, bloqueie a página **AreaRestrita.aspx**:

```
<location path="Restrita.aspx">
    <system.web>
        <authorization>
            <deny users="?" />
        </authorization>
    </system.web>
</location>

</configuration>
```

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

21. Com isso, é possível testar as restrições de acesso. Teste e veja que, ao tentar entrar na página **Restrita.aspx**, o usuário deverá ser redirecionado para a página de login:



22. No Web.config, adicione uma string de conexão padrão para o Entity Framework criar o banco de dados na pasta **App_Data**. Crie essa pasta na raiz do site, se necessário. O atributo **connectionString** foi dividido em várias linhas para facilitar a leitura, mas deve ser inserido no Web.config sem quebra de linha:

```
<connectionStrings>
  <add      name="DefaultConnection"
    connectionString=
      "Data Source=(LocalDb)\v11.0;
       AttachDbFilename=|DataDirectory|\ASPNET03CAP02.mdf;
       Initial Catalog=ASPNET03CAP02;
       Integrated Security=True"

    providerName="System.Data.SqlClient" />
</connectionStrings>

<system.web> ...
```

23. Na página **Usuario/Criar.aspx**, crie o formulário de cadastro com os campos **Nome** e **Senha**, começando com o **Título**:

```
<%-- Título --%>
<h2>Novo Registro de Usuário</h2>
<hr />
```

24. Crie uma área para mensagem, abaixo do **Título**:

```
<%-- Mensagem --%>
<p class="text-danger">
  <asp:Literal runat="server" ID="mensagemErro" />
</p>
```

25. Crie a **div** para marcar o formulário, usando as classes do Bootstrap:

```
<%-- Formulário --%>
<div class="form-horizontal">
</div>
```

26. Dentro da div do formulário, insira quatro componentes: **Nome**, **Senha**, **ConfirmarSenha** e um botão **Submit**:

```
<%-- Formulário --%>
<div class="form-horizontal">

    <%-- Nome --%>

    <%-- senha --%>

    <%-- Confirmar --%>

    <%-- Botão Gravar --%>

</div>
```

27. Crie a TextBox **Nome**, com a legenda e o validator, usando as classes do Bootstrap:

```
<%-- Nome --%>
<div class="form-group">

    <asp:Label runat="server"
        AssociatedControlID="Nome"
        CssClass="col-md-2 control-label">
        Nome</asp:Label>

    <div class="col-md-10">
        <asp:TextBox runat="server"
            ID="Nome"
            CssClass="form-control" />

        <asp:RequiredFieldValidator
            runat="server"
            ControlToValidate="Nome"
            CssClass="text-danger"
            ErrorMessage="Nome é obrigatório." />
    </div>
</div>
```

28. Agora, crie a TextBox senha:

```
<%-- senha --%>

<div class="form-group">

    <asp:Label runat="server"
        AssociatedControlID="Senha"
        CssClass="col-md-2 control-label">
        Senha
    </asp:Label>

    <div class="col-md-10">

        <asp:TextBox runat="server"
            ID="Senha"
            TextMode="Password"
            CssClass="form-control" />

        <asp:RequiredFieldValidator
            runat="server" ControlToValidate="Senha"
            CssClass="text-danger"
            ErrorMessage=
                "Senha é obrigatório."/>

    </div>
</div>
```

29. Crie, também, a TextBox **Confirmar Senha**:

```
<%-- Confirmar Senha --%>
<div class="form-group">

    <asp:Label runat="server"
        AssociatedControlID="ConfirmarSenha"
        CssClass="col-md-2 control-label">
        Confirmar Senha
    </asp:Label>

    <div class="col-md-10">

        <asp:TextBox runat="server"
            ID="ConfirmarSenha" TextMode="Password"
            CssClass="form-control" />

        <asp:RequiredFieldValidator
            runat="server" ControlToValidate="ConfirmarSenha"
            CssClass="text-danger"
            Display="Dynamic"
            ErrorMessage=
                "Confirmação de senha é obrigatório." />

        <asp:CompareValidator
            runat="server"
            ControlToCompare="Senha"
            ControlToValidate="ConfirmarSenha"
            CssClass="text-danger"
            Display="Dynamic"
            ErrorMessage=
                "A senha e a confirmação estão diferentes." />

    </div>
</div>
```

30. Configure o botão Confirmar:

```
<%-- Confirmar --%>
<div class="form-group">

    <div class="col-md-offset-2 col-md-10">

        <asp:Button runat="server"
            ID="confirmarButton"
            OnClick="confirmarButton_Click"
            Text="Confirmar"
            CssClass="btn btn-default" />

    </div>

</div>
```

Neste ponto, já é possível conferir o layout:



31. Escreva o código do evento **Click** do botão **Confirmar**:

```
//Obter os Valores do TextBox
string nome = Nome.Text;
string senha = Senha.Text;

//Obter a UserStore, UserManager
var usuarioStore = new UserStore<IdentityUser>();
var usuarioGerenciador =
    new UserManager<IdentityUser>(usuarioStore);

//Criar uma identidade
var usuarioInfo = new IdentityUser() { UserName = Nome.Text };
}

//Gravar
IdentityResult resultado =
    usuarioGerenciador.Create(usuarioInfo, Senha.Text);

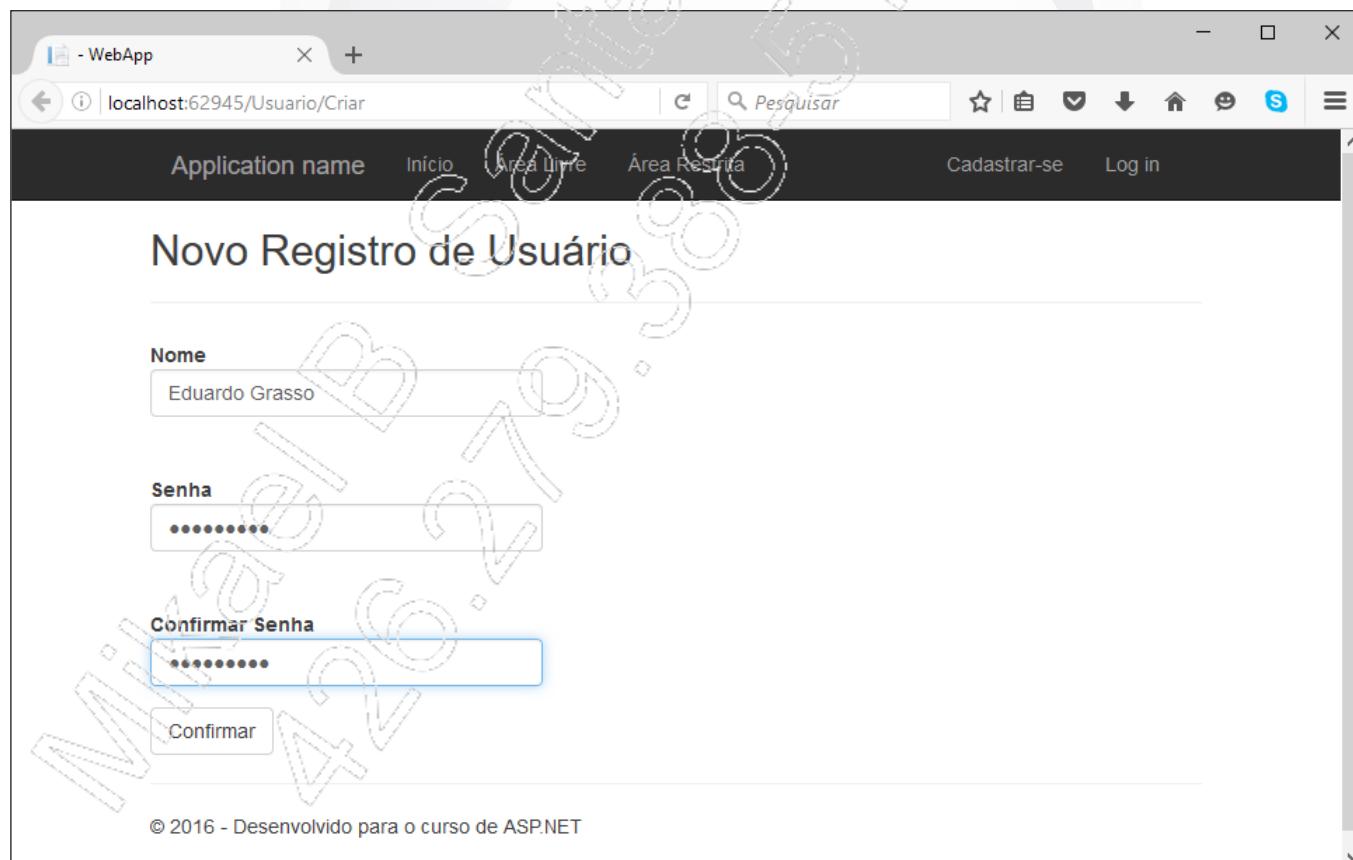
//Se ok,
if (resultado.Succeeded)
{
    //Autentica e volta para a página inicial
    var gerenciadorDeAutenticacao =
        HttpContext.Current.GetOwinContext().Authentication;

    var identidadeUsuario = usuarioGerenciador.CreateIdentity(
        usuarioInfo,
        DefaultAuthenticationTypes.ApplicationCookie);

    gerenciadorDeAutenticacao.SignIn(
        new AuthenticationProperties() { },
        identidadeUsuario);
}
```

```
Response.Redirect("~/default.aspx");  
  
}  
else //Erro, exibe o erro  
{  
  
    string erro=resultado.Errors.FirstOrDefault();  
    mensagemErro.Text = erro;  
  
}
```

32. Já é possível testar. Na página inicial, clique em **Cadastrar-se**, preencha o cadastro e confirme. O usuário deve ser redirecionado para a página **default.aspx**, na qual o nome aparece no canto superior direito, conforme a funcionalidade fornecida pelo controle **UsuarioUC.ascx**:



33. Configure a tela de login, que segue o mesmo layout da tela de cadastro:

```
<%@ Page Title="Login" Language="C#" ... %>
<asp:Content ID="Content1" ...>

    <%-- Título --%>
    <h2>Login</h2>
    <hr />

    <%-- Formulário --%>
    <div class="form-horizontal">

        <%-- Mensagem --%>
        <p class="text-danger">
            <asp:Literal runat="server" ID="mensagemErro" />
        </p>

        <%-- Nome --%>
        <div class="form-group">

            <asp:Label
                runat="server"
                AssociatedControlID="Nome"
                CssClass="col-md-2 control-label">
                Nome</asp:Label>

            <div class="col-md-10">
                <asp:TextBox
                    runat="server"
                    ID="Nome"
                    CssClass="form-control" />

                <asp:RequiredFieldValidator
                    runat="server"
                    ControlToValidate="Nome"
                    CssClass="text-danger"
                    ErrorMessage="Nome é obrigatório." />
            </div>
        </div>
    </div>
```

```
</div>

<%-- senha --%>
<div class="form-group">

    <asp:Label
        runat="server"
        AssociatedControlID="Senha"
        CssClass="col-md-2 control-label">
        Senha</asp:Label>

    <div class="col-md-10">
        <asp:TextBox runat="server"
            ID="Senha"
            TextMode="Password"
            CssClass="form-control" />

        <asp:RequiredFieldValidator
            runat="server"
            ControlToValidate="Senha"
            CssClass="text-danger"
            ErrorMessage="Senha é obrigatório." />
    </div>
</div>

<%-- Gravar --%>
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <asp:Button
            runat="server"
            ID="confirmarButton"
            Text="Cadastrar"
            CssClass="btn btn-default" />
    </div>
</div>

</div>
</asp:Content>
```

34. O código da tela de login usa a parte de autenticação igual à do cadastro, a única diferença é que o usuário é localizado e não inserido no banco. Escreva a lista do código do evento **Click** do botão **Confirmar**:

```
string nome = Nome.Text;
string senha = Senha.Text;

var usuarioStore = new UserStore<IdentityUser>();
var usuarioGerenciador =
    new UserManager<IdentityUser>(usuarioStore);

var usuario = usuarioGerenciador.Find(nome, senha);

if (usuario != null)
{
    var gerenciadorDeAutenticacao = HttpContext.Current
        .GetOwinContext().Authentication;

    var identidade = usuarioGerenciador.CreateIdentity(
        usuario,
        DefaultAuthenticationTypes.ApplicationCookie);

    gerenciadorDeAutenticacao.SignIn(
        new AuthenticationProperties()
            { IsPersistent = false },
        identidade);

    Response.Redirect("~/default.aspx");
}

else
{
    mensagemErro.Text = "Usuario ou senha invalida.";
}
```

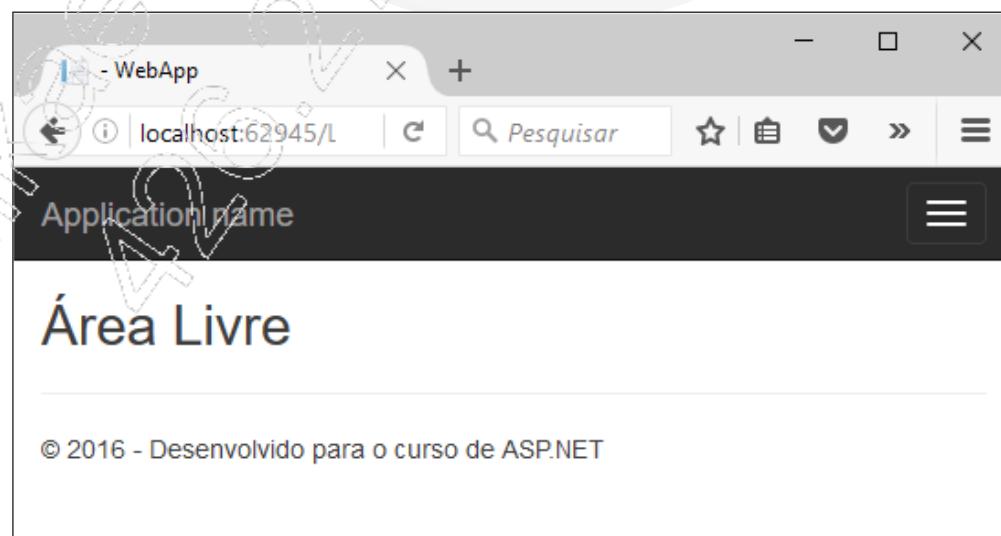
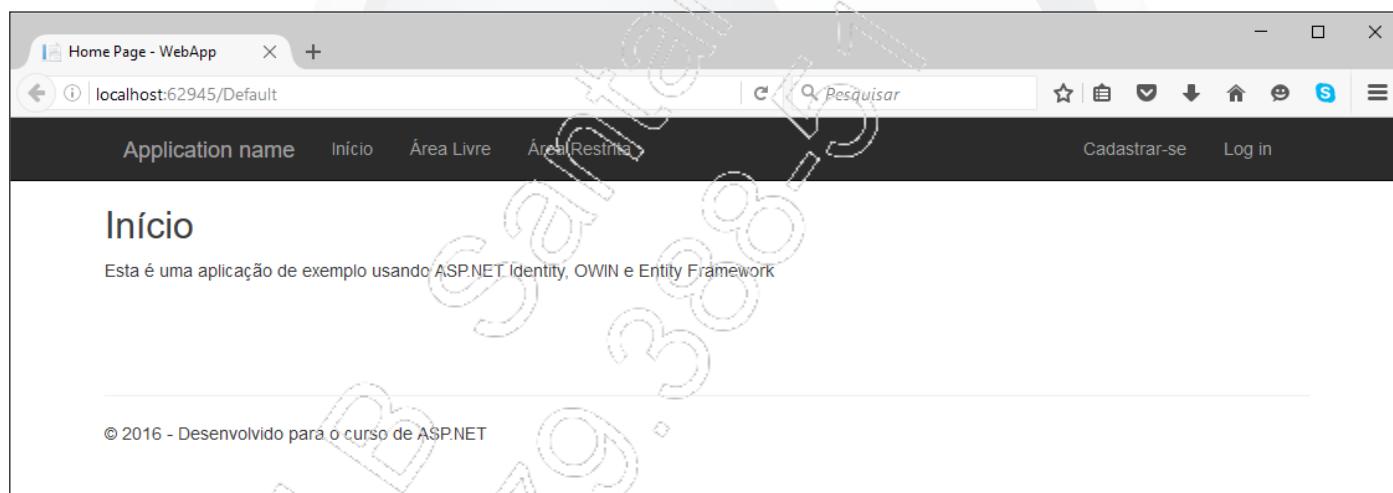
35. Finalmente, configure a página **Logout.aspx**, que desconecta um usuário. Escreva a lista do evento **Load**:

```
protected void Page_Load(object sender, EventArgs e)
{
    var gerenciadorDeAutenticacao =
        HttpContext.Current.GetOwinContext().Authentication;

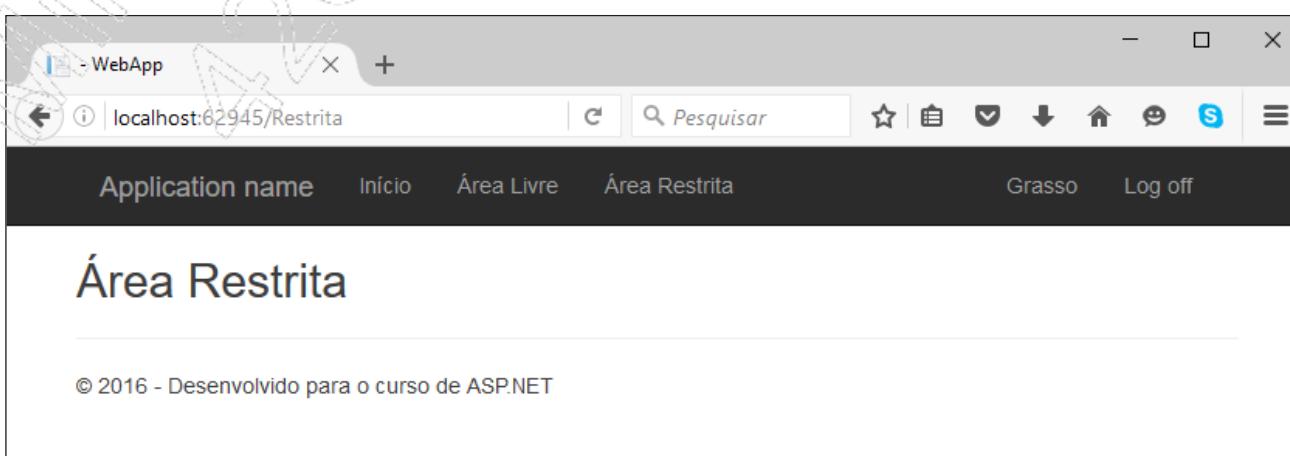
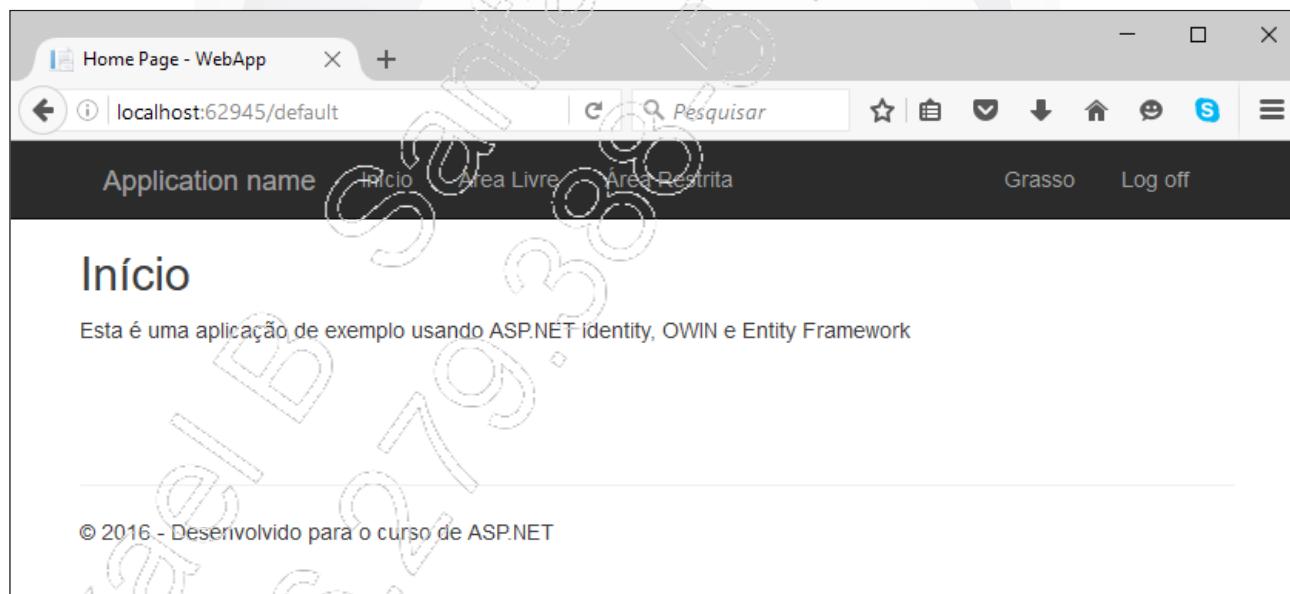
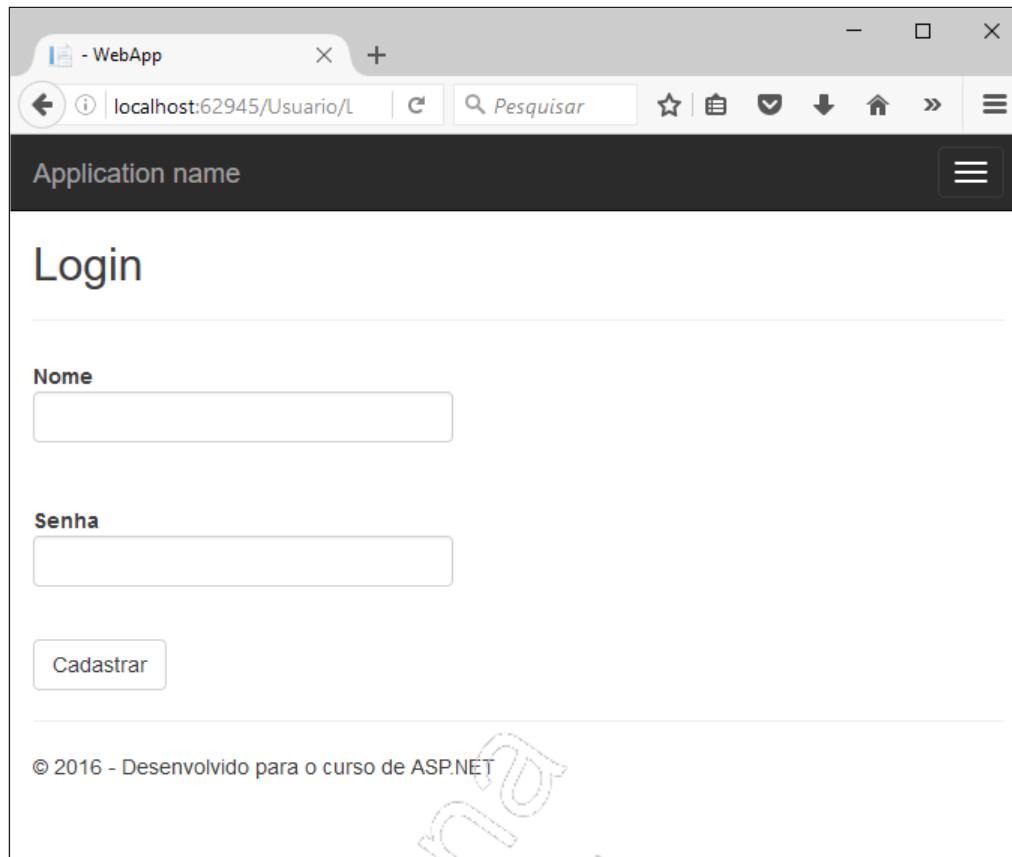
    gerenciadorDeAutenticacao.SignOut();
    Response.Redirect("~/default.aspx");

}
```

36. Teste a aplicação completa: tela inicial, **Log in**, **Log off**, **Área livre**, **Área Restrita** e **Cadastrar-se**.



Visual Studio 2015 - ASP.NET com C# Recursos Avançados



Resources, Localization e Globalization

6

- ✓ Culture;
- ✓ Resources;
- ✓ Resources via código;
- ✓ Alteração da cultura via código.

6.1. Introdução

Criar uma aplicação que possa ser lida em diversos idiomas é uma característica que ganha mais importância a cada dia, principalmente pela tendência crescente da distribuição globalizada oferecida por lojas de aplicativos.

A plataforma .NET utiliza, em todos os tipos de interfaces de usuários, tanto desktop quanto mobiles ou Web, o conceito de arquivo de recursos (**Resource Files**) e de cultura de interface (**UI Culture**) para gerenciar essa parte da estrutura de uma aplicação.

6.2. Culture

O namespace **System.Globalization** fornece uma classe chamada **CultureInfo**, que é o ponto central de todo o processo de adaptação do software à cultura e ao local onde o usuário está, processo este chamado de localização (**Localization**). A sintaxe para criar uma instância da classe **CultureInfo** é a seguinte:

```
var xxx = new CultureInfo("string");
```

O parâmetro **string** é um texto de quatro letras contendo a linguagem e o país. Por exemplo, **pt-br** significa **Português-Brasil**.

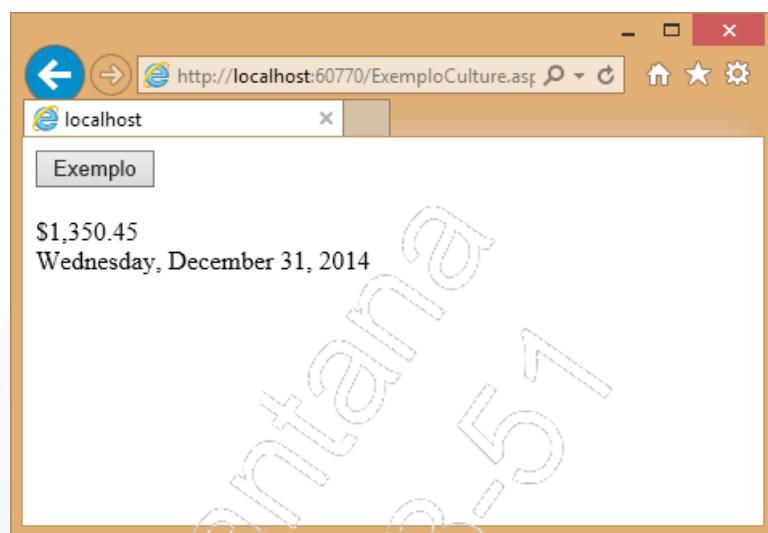
Veja este exemplo em uma Web Form, em que um valor monetário e uma data são exibidos:

```
//Cria uma Instância da Classe que
//define os formatos de um local
// en-us significa Inglês-Estados Unidos (English-United States)
var ci = new CultureInfo("en-us");

//Obtém um decimal e uma data
decimal valor = Convert.ToDecimal(1350.45);
DateTime data = new DateTime(2014, 12, 31);
```

Resources, Localization e Globalization

```
//Formata como Moeda e Data por extenso  
//usando a cultura criada  
string valorFormatado = string.Format(ci,"{0:c}", valor);  
string dataExtenso = string.Format(ci,"{0:D}", data);  
  
//Exibe nos controles Labels  
moedaLabel.Text = valorFormatado;  
dataLabel.Text = dataExtenso;
```



Para que não se tenha que informar a cultura toda vez que for exibir um número e uma data, é possível informar a cultura do site no Web.config, por meio do atributo **Globalization**:

```
<?xml version="1.0" encoding="utf-8"?>  
<configuration>  
  <system.web>  
    <globalization culture="en-us"/>  
    <compilation debug="true" targetFramework="4.5.1" />  
    <httpRuntime targetFramework="4.5.1" />  
  </system.web>  
</configuration>
```

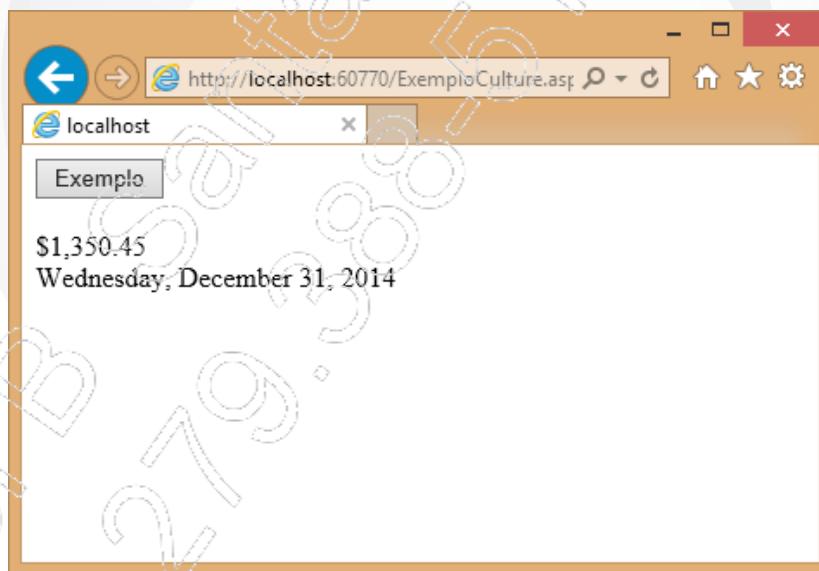
Dessa forma, o mesmo resultado poderia ser obtido com o código mais enxuto, sem informar a localização da cultura da interface:

```
//Obtém um decimal e uma data
decimal valor = Convert.ToDecimal(1350.45);
DateTime data = new DateTime(2014, 12, 31);

//Formata como Moeda e Data por extenso
string valorFormatado = string.Format("{0:c}", valor);
string dataExtenso = string.Format("{0:D}", data);

//Exibe nos controles Labels
moedaLabel.Text = valorFormatado;
dataLabel.Text = dataExtenso;
```

O resultado é exatamente o mesmo:



No modelo ASP.NET Core, a cultura pode ser definida na classe **Startup**, no método **Configure**. O método de extensão **UseRequestLocalization** está na classe **ApplicationBuilderExtensions** e no seguinte namespace: **Microsoft.AspNetCore.Localization**:

```
...
using Microsoft.AspNetCore.Localization;
....
public class Startup
{ ...
```

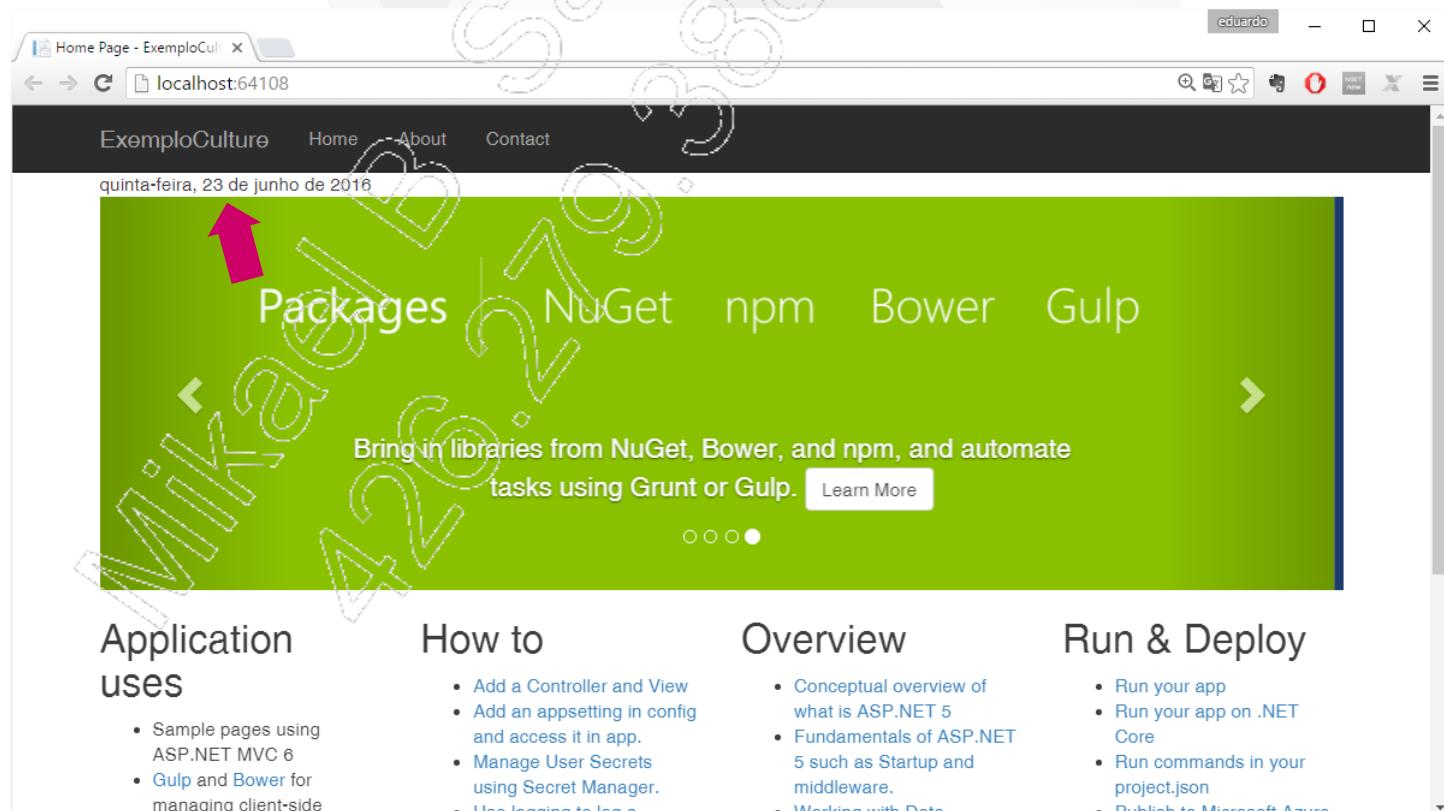
Resources, Localization e Globalization

```
public void Configure(IAApplicationBuilder app,
IHostingEnvironment env,
ILoggerFactory loggerFactory)
{
    app.UseRequestLocalization(new RequestCulture("pt-br"));
    ...
}
```

Na View **Home / Index**, um pequeno código após a declaração inicial do HTML, uma div contendo a data por extenso:

```
@{
    ViewData["Title"] = "Home Page";
}

<div>
    @DateTime.Now.ToString()
</div>
```



6.3. Resources

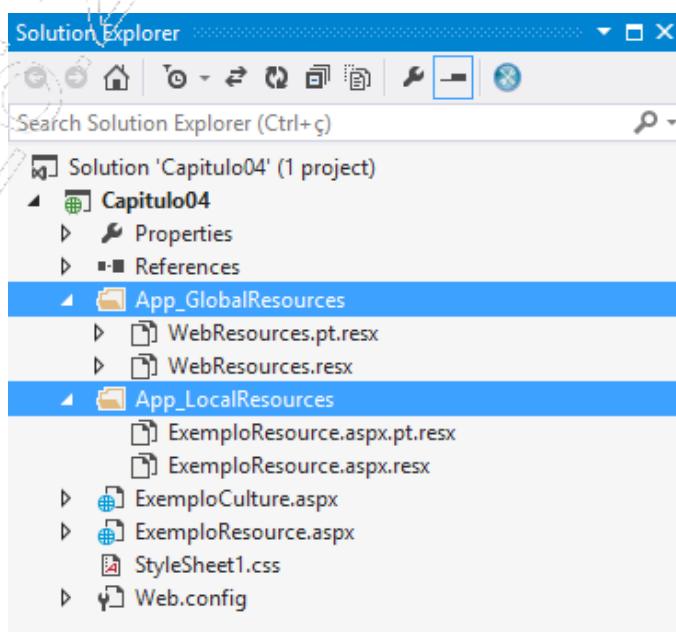
O que pode ser conseguido com a classe **CultureInfo** e com o parâmetro **Globalization** abrange apenas a formatação, e não o conteúdo. Se houver a necessidade de traduzir o aplicativo para vários idiomas, será necessário utilizar um arquivo de recursos (**Resource File**).

Um **Resource File** é um arquivo XML que contém definições usando o conceito de chave e valor, semelhante a um dicionário. Por exemplo:

Chave	Valor
BotaoOK	Confirmar.
BotaoCancel	Voltar.
MsgOK	Processamento concluído com sucesso.
MsgErro	Houve erro no processamento.

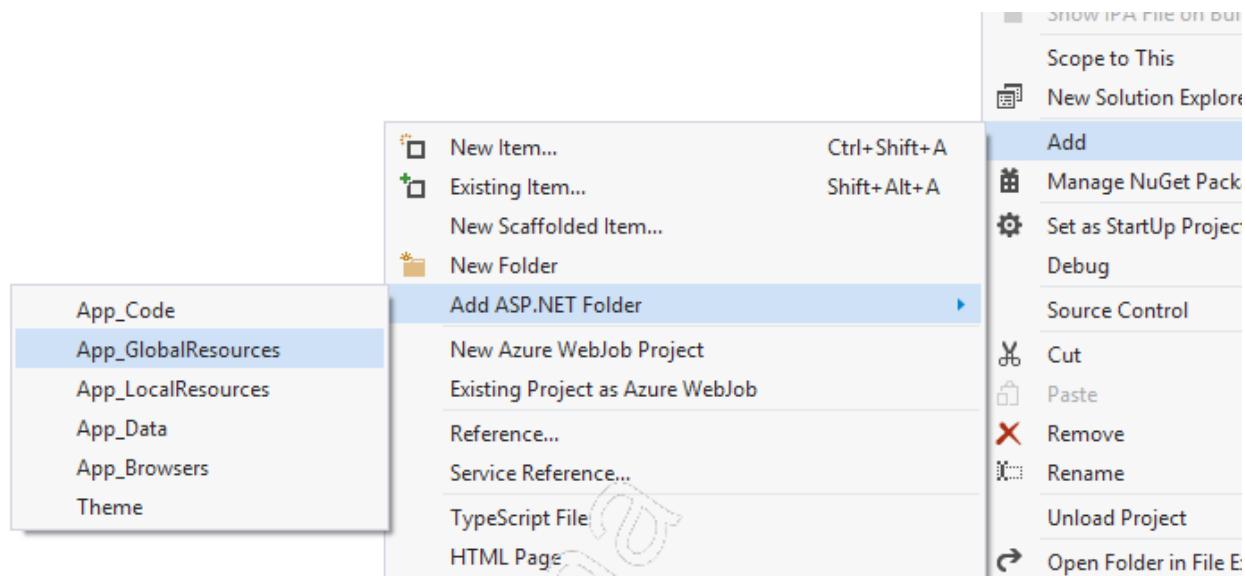
Criando arquivos diferentes para cada linguagem e tomando o cuidado de nunca usar um texto diretamente no aplicativo, e sim a chave, é possível usar o arquivo relacionado à **CultureInfo** do usuário.

No ASP.NET, os arquivos de recursos podem ser **globais**, ou seja, podem ser lidos por qualquer arquivo do aplicativo, ou podem ser **locais**, sendo exclusivos de uma página. Os arquivos globais devem ficar em uma pasta especial chamada **App_GlobalResources** que deve ficar na raiz da aplicação. Os arquivos locais devem ficar em uma pasta chamada **App_LocalResources**.



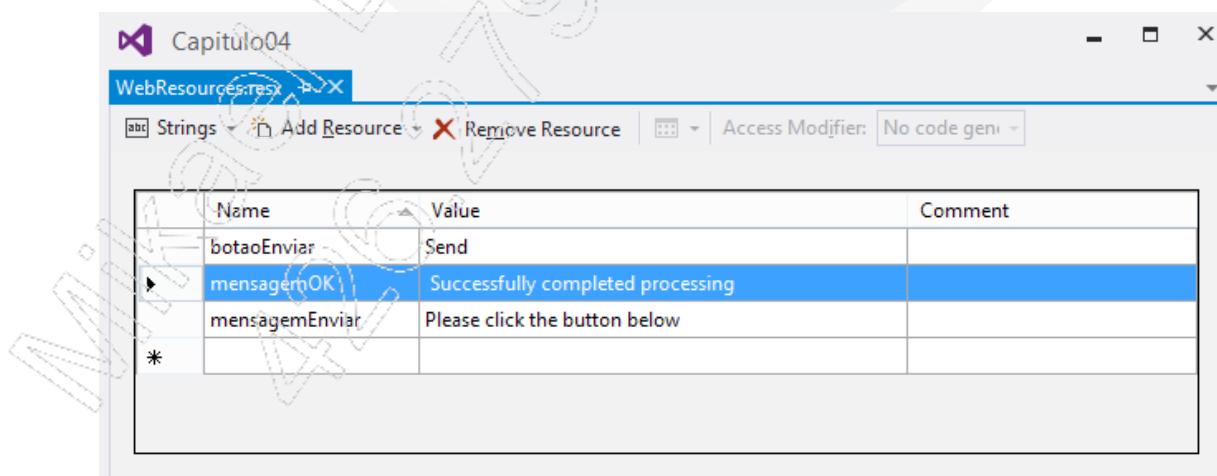
6.3.1. Global Resources

Para adicionar a pasta **Global**, escolha, no menu de contexto, **Add ASP.NET Folder** e, em seguida, **App_GlobalResources**:

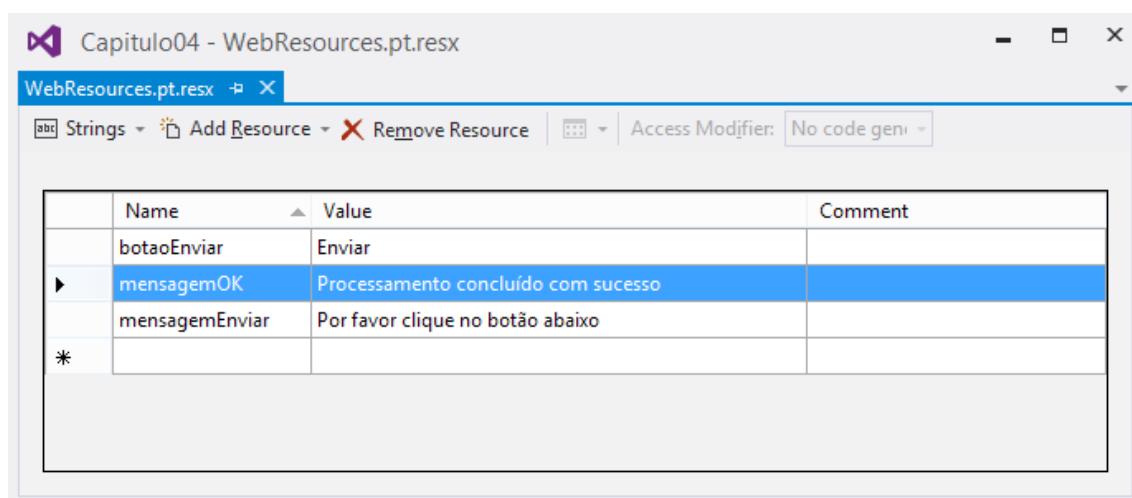


O nome do arquivo global deve ser **WebResources.resx** para a cultura padrão. Para outra cultura, devemos acrescentar as letras da linguagem e/ou país. Por exemplo, para o arquivo em inglês, o nome deve ser **WebResources.en.resx**; para português, o nome deve ser **WebResources.pt.resx**.

Dentro do arquivo devem ser definidos pares de valores em português e em inglês:



Visual Studio 2015 - ASP.NET com C# Recursos Avançados



Nas páginas HTML, serão usadas sempre as chaves para definir o texto que será exibido. O texto será exibido na linguagem da cultura atual. Para definir a chave em HTML é necessário usar a seguinte sintaxe:

```
<asp:Button  
    runat="server"  
    Text="<%$ Resources:WebResources, botaoEnviar %>"  
    ID="botaoEnviar"  
    OnClick="botaoEnviar_Click" />
```

O símbolo `<%$` é uma tag especial do tipo **Expression Builder** e consiste em um prefixo (no caso, **Resources**), dois-pontos e uma lista de itens que é resolvida em tempo de execução. É particularmente útil em situações em que existem coleções de chaves e valores, exatamente como é o caso aqui.

! **Resources** é o tipo. **WebResources** é a coleção.
botaoEnviar é a chave.

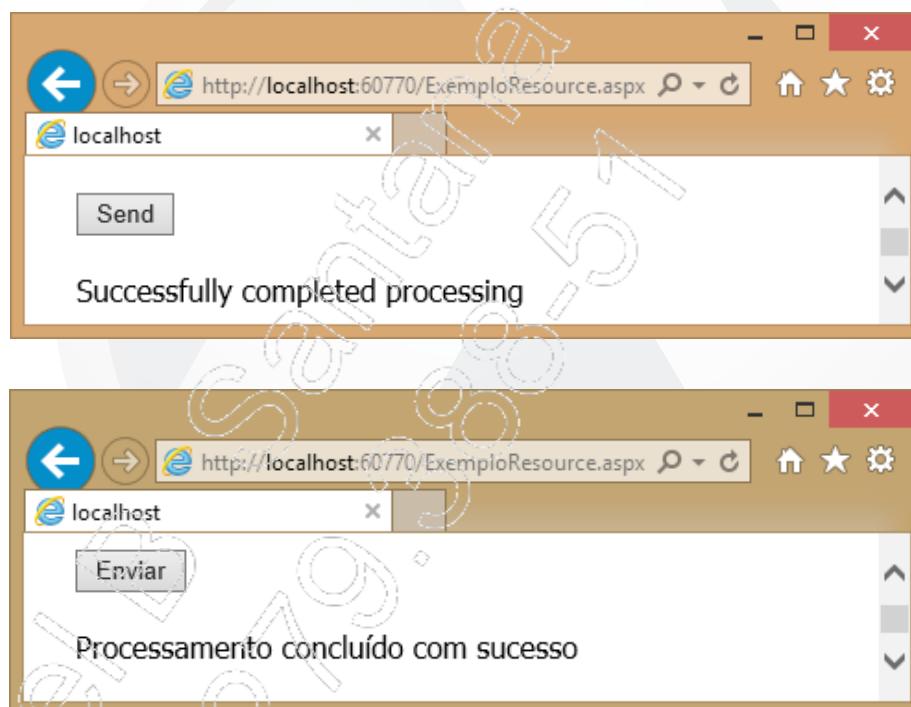
Vejamos um exemplo em um contexto completamente diferente. Considere a seguinte estrutura no Web.config:

```
<appSettings>  
    <add key="empresa" value="Empresa XYZ"/>  
</appSettings>
```

Nesse caso, é apenas o recurso (**AppSettings**) e a chave (**empresa**). É possível incluir a informação da empresa em um label usando a seguinte expressão:

```
<asp:Label runat="server"
    Text="<%$ appSettings: empresa %>">
</asp:Label>
```

Voltando ao botão **Enviar** e ao arquivo de **Resources**, quando a página estiver em inglês, ou seja, a cultura atual estiver definida para o inglês, será exibida a mensagem em inglês e, quando estiver em português, a mensagem será obtida por meio do arquivo em português. Essa alteração de cultura pode ser feita via programação ou por meio do Web.config.



6.3.2. Local Resources

Os **arquivos locais** são arquivos de recursos relacionados exclusivamente a uma única página e ficam em uma pasta chamada **App_LocalResources**. O que liga o arquivo à página é o nome. Se a página se chama **Produtos.aspx**, o arquivo de recursos deve se chamar **Produtos.aspx.resx**. Para a cultura padrão e para outra cultura, devemos inserir as letras antes da extensão. Por exemplo, para a cultura **Português**, o nome deve ser **Produtos.aspx.pt.resx**, e para a cultura **Espanhol-Espanha**, o nome deve ser **Produtos.aspx.es-es.resx**.

O arquivo de recurso local deve mapear o nome da propriedade explicitamente. Por exemplo:

Button1.Text	Clique aqui
Label1.Text	Registro Incluído com sucesso
xxx.Text	Confirma a exclusão?

O nome em si (**Button1**, **Label1**, **xxx**) não é importante, porque deve ser informado na hora de vincular, mas a propriedade deve ser exatamente aquela que se deseja alterar. Para fazer a referência, utiliza-se o atributo **meta**:

```
<asp:Label runat="server"  
    ID="label1"  
    meta:resourceKey="xxx" ></asp:Label>
```

No exemplo anterior, a propriedade **Text** do label será definida para a frase **Confirma a exclusão?**, porque essa propriedade está definida na chave do arquivo de recurso (**xxx.Text**).

É costume usar os termos **implícito**, para arquivos locais, e **explícito**, para arquivos globais, pelo modo diferente como cada um trata os recursos de cultura e globalização.

6.4. Resources via código

Muitas vezes será necessário definir o valor de um item via código, em tempo de execução. Quando é inserida uma classe de **Resources** global, é possível ter acesso aos recursos diretamente por meio da classe **WebResources**. Cada chave aparece como uma propriedade da classe.

```
mensagemLabel.Text = Resources.WebResources.mensagemOK;
```

Além disso, o método herdado de **TemplateControl**, chamado **GetGlobalResourceObject()**, permite obter dados dos recursos globais, e o método **GetLocalResourceObject()** permite obter dados dos recursos locais.

Para obter a mensagem anterior, o comando equivalente seria o seguinte:

```
mensagemLabel.Text = GetGlobalResourceObject("WebResources",  
    "mensagemOK").ToString();
```

É necessário colocar **ToString()** no final porque esse método retorna um **object**. A propriedade criada em tempo de execução pela classe **WebResources** é superior porque dá um acesso fortemente tipado aos elementos declarados.

A sintaxe para obter um recurso local é mais simples porque sempre se refere à página onde está sendo executado o código:

```
var frase = GetLocalResourceObject("mensagem");
```

6.5. Alterando a cultura via código

É possível usar o objeto **Request** para obter a localização do usuário e, então, retornar a página com a cultura apropriada para o usuário. Muitas vezes o usuário está usando um browser ou um computador em uma cultura que não é a sua. É sempre bom fornecer ao usuário uma opção de escolher a linguagem desejada. Este tópico mostrará passo a passo como fazer isso.

Definir a cultura em tempo de execução se resume em definir a propriedade **CurrentCulture** do processo atual para a cultura desejada. Isso deve ser feito dentro de um evento da página chamado **InitializeCulture**. Esse método deve ser sobreposto, conforme mostrado a seguir:

```
using System.Globalization;  
using System.Threading;  
...  
protected override void InitializeCulture()  
{  
    Thread.CurrentCulture.CurrentCulture =  
        CultureInfo.CreateSpecificCulture("pt-br");  
}
```

Isso deve ser definido antes de qualquer coisa, antes inclusive do **Page_Load**. Como é possível criar um botão para o usuário escolher um idioma e, então, definir esse idioma, se ele deve ser definido antes de processar o botão? A maneira de contornar isso é armazenar a cultura desejada em uma variável de sessão, cookie, ou **QueryString** e redirecionar a página para, então, capturar isso no início. Para exemplificar, é necessário criar um processo completo:

1. Para iniciar, crie dois botões para escolher a cultura:

```
<asp:Button runat="server"
    Text="English"
    id="englishButton"
    OnClick="englishButton_Click" />

<asp:Button runat="server"
    Text="Portuguese"
    ID="portugueseButton"
    OnClick="portugueseButton_Click" />
<hr />
```

2. Crie um label com a mensagem para clicar no botão:

```
<asp:Label runat="server"
    Text="<%$ Resources:WebResources, mensagemEnviar %>">
</asp:Label>

<br />
<br />
```

3. Crie um botão para o usuário clicar:

```
<asp:Button runat="server"
    Text="<%$ Resources:WebResources, botaoEnviar %>">
    ID="botaoEnviar"
    OnClick="botaoEnviar_Click" />
<br />
<br />
```

Resources, Localization e Globalization

4. Crie um label para exibir uma mensagem:

```
<br />
<br />

<asp:Label runat="server" ID="mensagemLabel"></asp:Label>
```

5. Crie um rodapé obtendo dados do recurso local:

```
<br />
<br />

<asp:Label runat="server"
           ID="rodapeLabel"
           meta:resourceKey="rodapeLabel" >
</asp:Label>
```

6. Crie a listagem completa:

```
<asp:Button runat="server" Text="English"
           id="englishButton"
           OnClick="englishButton_Click" />

<asp:Button runat="server" Text="Portuguese"
           ID="portuguesButton"
           OnClick="portuguesButton_Click" />

<hr />

<asp:Label runat="server"
           Text="<%$ Resources:WebResources, mensagemEnviar %>">
</asp:Label>

<br />
<br />
```

```
<asp:Button  
    runat="server"  
    Text="<%$ Resources:WebResources,botaoEnviar %>"  
    ID="botaoEnviar" OnClick="botaoEnviar_Click" />  
  
<br />  
<br />  
<br />  
<br />  
  
<asp:Label runat="server" ID="mensagemLabel"></asp:Label>  
  
<asp:Label runat="server" ID="rodapeLabel"  
    meta:resourceKey="rodapeLabel" >  
</asp:Label>
```

7. Adicione dois arquivos de recursos locais:

App_LocalResources/NomeDaPagina.aspx.resx
rodapeLabel.Text Designed for ASP.NET course © 2014

App_LocalResources/NomeDaPagina.aspx.pt.resx
rodapeLabel.Text Desenvolvido para o curso de ASP.NET © 2014

8. Adicione dois arquivos de recursos globais:

App_GlobalResources/WebResources.pt.resx
botaoEnviar Enviar
mensagemEnviar Por favor, clique no botão abaixo
mensagemOK Processamento concluído com sucesso

App_GlobalResources/WebResources.resx
botaoEnviar Send
mensagemEnviar Please click the button below
mensagemOK Sucessfully completed processing

Resources, Localization e Globalization

9. Ao clicar em português ou inglês, a sessão é gravada e a página redirecionada. A propriedade **RawUrl** retorna a URL atual que o navegador informou, portanto, redirecionará exatamente para a mesma página:

```
protected void portuguesButton_Click(object sender,
EventArgs e)
{
    Session["culture"] = "pt-br";
    Response.Redirect(Request.RawUrl);
}
```

```
protected void englishButton_Click(object sender, EventArgs
e)
{
    Session["culture"] = "en-us";
    Response.Redirect(Request.RawUrl);
}
```

10. Aqui, vemos um método para definir a cultura no processo atual, cujo retorno é realizado pela propriedade **CurrentThread**. Essa propriedade é fornecida pela classe **Thread**. A propriedade **CurrentCulture** do processo atual define a cultura em que a página será processada:

```
private void DefinirCulture(string idiomaPais)
{
    UICulture = idiomaPais;
    Culture = idiomaPais;

    Thread.CurrentThread.CurrentCulture =
        new CultureInfo(idiomaPais);

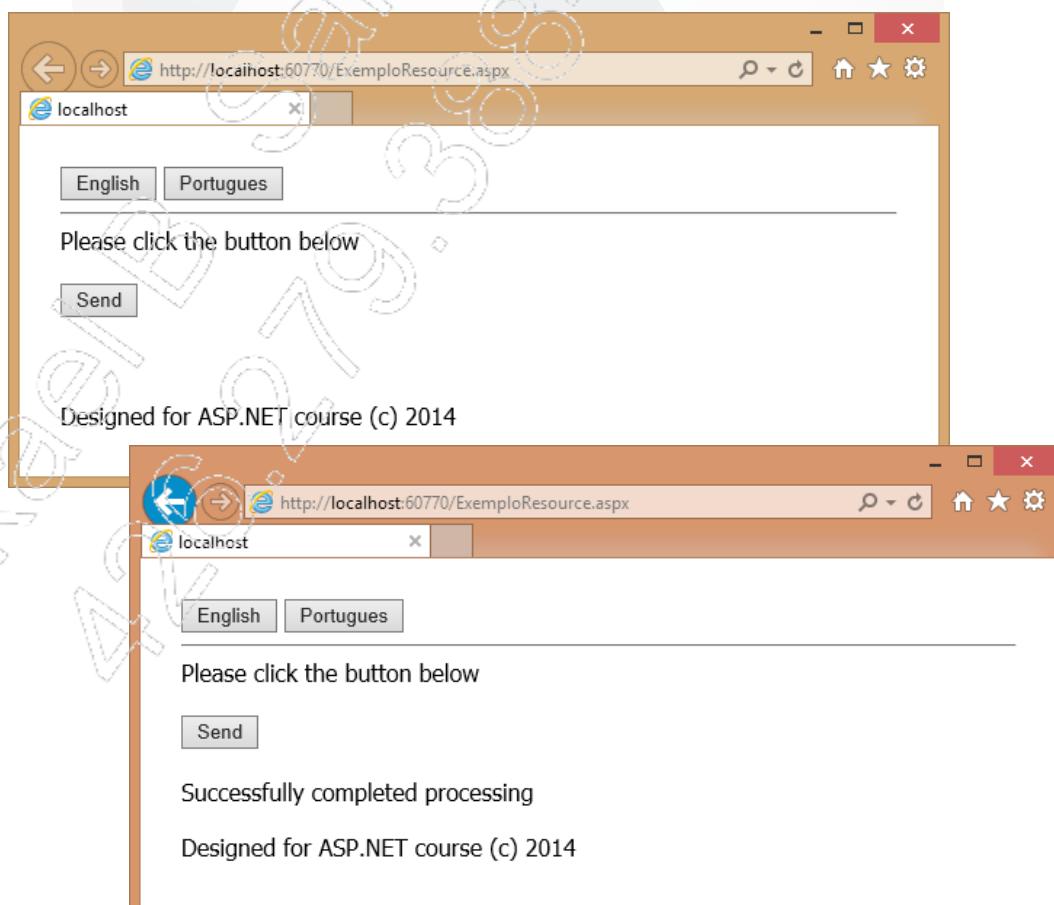
    Thread.CurrentThread.CurrentUICulture = new
        CultureInfo(idiomaPais);
}
```

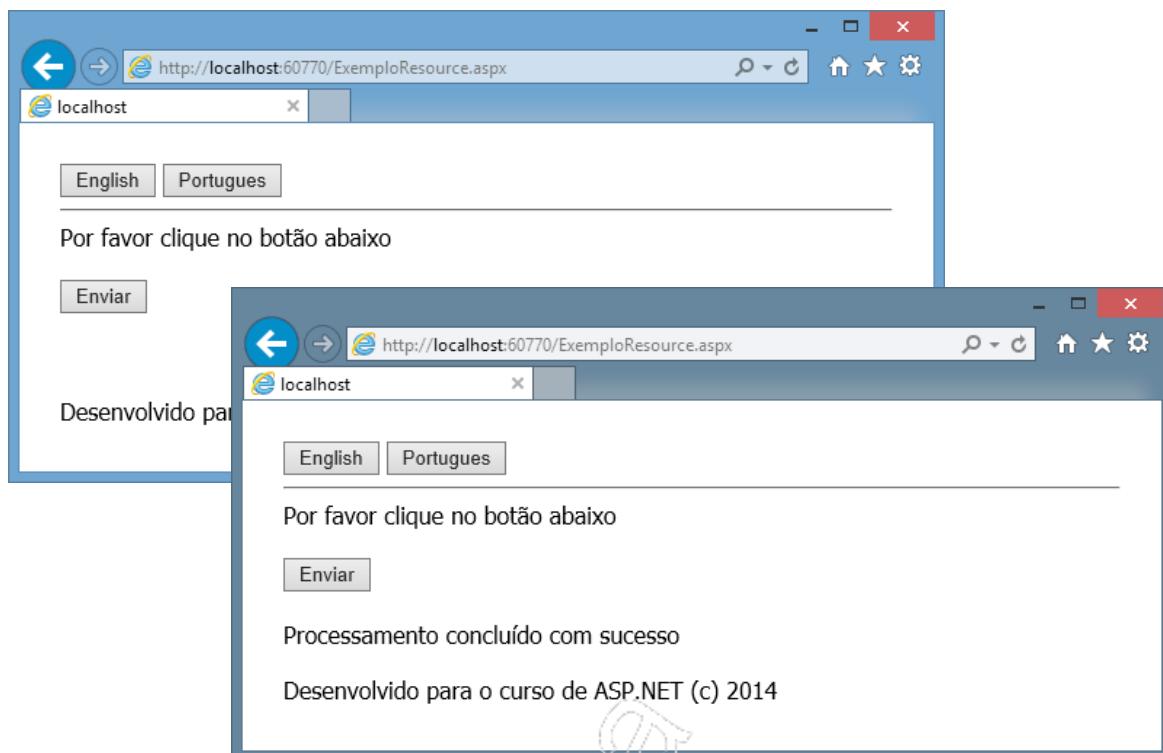
11. E, finalmente, sobreponha o método **InitializeCulture**. Se existir uma variável de sessão (porque foi redirecionada do evento do botão), use essa variável para definir a cultura:

```
protected override void InitializeCulture()
{
    if (Session["culture"] != null)
    {
        DefinirCulture(Session["culture"].ToString());
    }

    base.InitializeCulture();
}
```

Ao executar o programa, a cultura atualmente definida no Web.config será usada. Ao clicar no botão de idioma (inglês, português), a sessão é definida, a página redirecionada e a CultureInfo do processo atual é alterada, fazendo que o ASP.NET busque as mensagens nos arquivos de recursos correspondentes.





Mikael B Santana
426.270.388-57

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- **System.Globalization.CultureInfo** é a principal classe relacionada à localização da aplicação;
- Os arquivos de recursos (**Resource Files**) fornecem um meio de criar um aplicativo em vários idiomas;
- Os arquivos de recursos globais são armazenados na pasta **App_GlobalResources**;
- Os arquivos de recursos locais são relativos a uma página e são armazenados na pasta **App_LocalResources**;
- Vários arquivos de recursos podem existir alterando apenas o nome, colocando informações do país e idioma no nome do arquivo;
- O método **InitializeCulture()** deve ser sobreposto para definir a cultura da página em tempo de execução;
- O método **GetLocalResourceObject** é utilizado para obter informações sobre um arquivo de recursos locais;
- O método **GetGlobalResourceObject** é utilizado para obter informações sobre um arquivo de recursos globais;
- O ASP.NET cria diversas propriedades na classe **WebResources** para fornecer acesso aos recursos definidos;
- A string que define a cultura é composta de duas partes: o idioma e o país. **pt-br** significa **Português-Brasil**.

6

Resources, Localization e Globalization

Teste seus conhecimentos

Mikael
B
C
A
426.279
2



IMPACTA
EDITORA

1. Em qual namespace se encontra a classe CultureInfo?

- a) System
- b) System.Resources
- c) System.Globalization
- d) System.Web.Globalization
- e) System.Web

2. No ASP.NET 4.6, o arquivo Global.asax é utilizado para definir o CultureInfo. Qual arquivo é utilizado no ASP.NET Core?

- a) Setup.ini
- b) Startup.cs
- c) project.json
- d) global.json
- e) setup.web

3. Usando Resources, em qual pasta devem ficar os arquivos relativos a uma página específica?

- a) App_LocalResources
- b) App_GlobalResources
- c) Na mesma pasta da página.
- d) App_Start
- e) Bin

4. Qual a sintaxe correta para exibir o texto de um botão obtendo-o de um arquivo de recursos globais?

- a) <asp:Button Text="<%\$Resources: WebResources, botaoOK%>"
- b) <asp:Button meta:resourceKey='botaoOK'>
- c) <asp:Button meta:resourceKey='WebResources, botaoOK'>
- d) <asp:Button Text="<%\$ Resources: WebResources, botaoOK%">
- e) <asp:Button Text=<!-- Resources: WebResources, botaoOK-->

5. Qual método deve ser sobreposto para criar páginas em linguagem escolhida pelo usuário?

- a) page_load
- b) pre_init
- c) application_start
- d) InitializeCulture
- e) session_start

6

Resources, Localization e Globalization

Mãos à obra!

Mikael Brantina
426.2793-0000



IMPACTA
EDITORA

Laboratório 1

A – Utilizando a classe CultureInfo

Neste laboratório, você utilizará a classe **CultureInfo** para obter informações sobre as configurações regionais disponíveis no .NET Framework e exibirá formatos comuns de data e número em diversas culturas.

As telas serão as seguintes:

- O usuário fornece um número e uma data válida para visualizar como esses dados são formatados nas diversas culturas.



- Ao preencher um número e uma data, aparece uma lista das culturas disponíveis com os dados exibidos em cada formato.

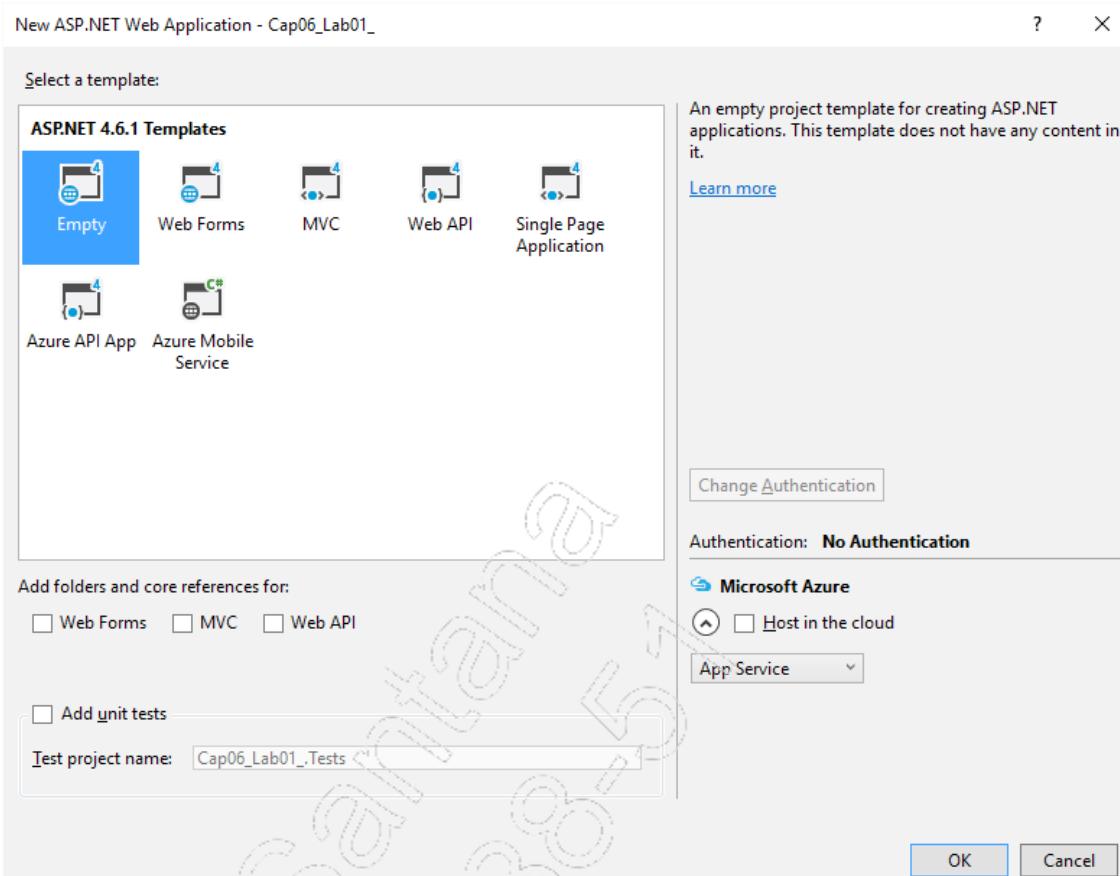
The screenshot shows a web browser window with the URL <http://localhost:59299/Default.aspx>. The page title is "CultureInfo". Below the title, there is a section titled "Exibir dados em diversas culturas". It contains two input fields: "Número:" with the value "1230" and "Data:" with the value "01/05/1970". A "Confirmar" button is present. Below these inputs is a table displaying data for various cultures. The columns are: Nome (Name), Código (Code), Data (Date), Data por Extenso (Extended Date), Número (Number), and Formato Monetário (Monetary Format). The table lists numerous cultures, including Afrikaans, English (United Kingdom), Portuguese (Brazil), Spanish (Spain), French (France), German (Germany), Italian (Italy), Dutch (Netherlands), and many Arabic-speaking countries like Saudi Arabia, Iraq, and Libya, among others. Each row shows the culture name, its code, a date (01/05/1970), its extended date (e.g., "Friday, 01 May 1970"), the number "1,230.00", and its monetary format (e.g., "R 1 230,00").

Nome	Código	Data	Data por Extenso	Número	Formato Monetário
Idioma Invariável (País Invariável)		05/01/1970	Friday, 01 May 1970	1,230.00	R 1,230.00
Africâner	af	1970/05/01	01 Mei 1970	1 230,00	R 1 230,00
Africâner (África do Sul)	af-ZA	1970/05/01	01 Mei 1970	1 230,00	R 1 230,00
Amárico	am	1/5/1970	፩፻፲፭ ቀን ፳፻፲፭	1,230.00	ETB1,230.00
Amárico (Etiópia)	am-ET	1/5/1970	፩፻፲፭ ቀን ፳፻፲፭	1,230.00	ETB1,230.00
Árabe	ar	25/02/90	٢٥/٣٩٠	1,230.00	1,230.00
Árabe (E.A.U.)	ar-AE	01/05/1970	٠١ ١٩٧٠	1,230.00	1,230.00
Árabe (Bahrein)	ar-BH	01/05/1970	٠١ ١٩٧٠	1,230.000	1,230.000
Árabe (Argélia)	ar-DZ	01-05-1970	٠١ ١٩٧٠	1,230.00	1,230.00
Árabe (Egito)	ar-EG	01/05/1970	٠١ ١٩٧٠	1,230.000	1,230.00
Árabe (Iraque)	ar-IQ	01/05/1970	٠١ ١٩٧٠	1,230.00	1,230.00
Árabe (Jordânia)	ar-JO	01/05/1970	٠١ ١٩٧٠	1,230.000	1,230.000
Árabe (Kuwait)	ar-KW	01/05/1970	٠١ ١٩٧٠	1,230.000	1,230.000
Árabe (Líbano)	ar-LB	01/05/1970	٠١ ١٩٧٠	1,230.00	1,230.00
Árabe (Líbia)	ar-LY	01/05/1970	٠١ ١٩٧٠	1,230.000	1,230.000

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

Para isso, siga o passo a passo adiante:

1. Crie um novo projeto do tipo **ASP.NET (4.6) Empty** chamado **Cap06_Lab01**;



2. Adicione uma folha de estilo (**Estilos.css**) e um Web Form (**default.aspx**);

3. Associe a folha de estilo à página:

```
<head runat="server">  
    <title></title>  
    <link href="Estilos.css" rel="stylesheet" />  
</head>
```

4. Defina, dentro do formulário, as áreas do site: cabeçalho, conteúdo e rodapé:

```
<form id="form1" runat="server">

    <header>

    </header>

    <section class="conteudo">

    </section>

    <footer>

    </footer>

</form>
```

5. Defina, na folha de estilo, a formatação das áreas:

```
body {
    font-family:'Segoe UI', Tahoma, Geneva, Verdana;
    margin:0px;
}

header {
    padding:10px;
    background-color:#ff6a00;
    color:#ffffff;
}

section.conteudo {

    padding:20px;
    min-height:300px;
}
```

```
section.conteudo h2 {  
    color:#e55f00;  
}  
  
footer{  
    padding:5px;  
    margin-top:200px;  
}
```

6. Defina o header. É apenas um título:

```
<header>  
    <h1>CultureInfo</h1>  
</header>
```

7. Defina o rodapé. É um parágrafo:

```
<footer>  
    <p>Desenvolvido para o curso de ASP.NET(c) 2014</p>  
</footer>
```

8. Visualize a página:



9. Crie o formulário dentro de <section>:

```
<section class="conteudo" >

    <!-- Título -->
    <h2>Exibir dados em diversas culturas</h2>

    <!-- Mensagem -->
    <asp:Label runat="server"
        ID="mensagem"
        CssClass="mensagem" Visible="false">
    </asp:Label>

    <!-- Formulário -->

    <!-- Número -->
    <div class="form-grupo">

        <asp:Label runat="server"
            AssociatedControlID="numero">Número:</asp:Label>

        <asp:TextBox runat="server" ID="numero"></asp:TextBox>

    </div>

    <!--Data -->
    <div class="form-grupo">
        <asp:Label runat="server"
            AssociatedControlID="data">Data:</asp:Label>

        <asp:TextBox runat="server" ID="data"></asp:TextBox>
    </div>

    <!--Botões -->
    <div class="form-botoes">

        <asp:Button CssClass="botao" runat="server"
            ID="confirmar" Text="Confirmar" />

    </div>

</section>
```

10. Ainda dentro de <section>, insira, por último, a grid que vai exibir os dados:

```
<section class="conteudo" >

    <!-- Título -->
    <h2>Exibir dados em diversas culturas</h2>

    <!-- Mensagem -->
    <asp:Label . .

    <!-- Número -->
    <div class="form-grupo" ...>

        <!--Data -->
        <div class="form-grupo" ...>

            <!--Botões -->
            <div class="form-botoes" ...></div>

    <!-- Tabela -->
    <div class="tabela">
        <asp:GridView runat="server" ID="gv"
            AutoGenerateColumns="false">
            <Columns>
                <asp:BoundField DataField="Nome"
                    HeaderText="Nome" />

                <asp:BoundField DataField="Codigo"
                    HeaderText="Código" />

                <asp:BoundField DataField="DataSimples"
                    HeaderText="Data" />

                <asp:BoundField DataField="DataExtenso"
                    HeaderText="Data por Extenso" />

                <asp:BoundField DataField="Numero"
                    HeaderText="Número" />

                <asp:BoundField DataField="Moeda"
                    HeaderText="Formato Monetário" />
            </Columns>
        </asp:GridView>
    </div>
</div>
```

```
</Columns>
</asp:GridView>
</div>
</session>
```

11. Configure alguns estilos a mais que também são necessários:

```
.form-grupo {
    display:block;
    margin-bottom:15px;
}

.form-grupo label {
    display:block;
}

.form-botoes {
    margin-top:10px;
}

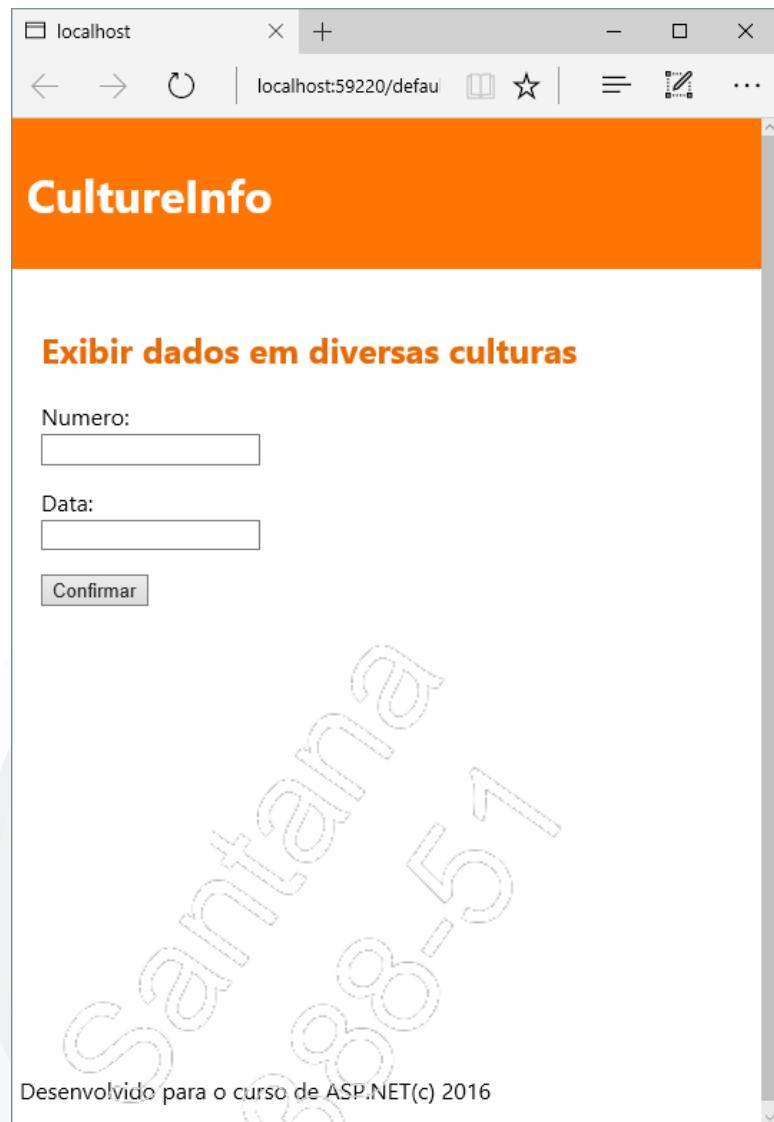
.tabela {
    margin-top:20px;
    font-size:80%;
}

.tabela td{
    padding:6px;
    white-space:nowrap;
}

.tabela th{
    padding:6px;
}

.mensagem {
    display:block;
    font-size:105%;
    font-weight:bold;
    margin-top:10px;
    margin-bottom:10px;
    color:red;
}
```

12. Visualize a página:



13. No code-behind, crie uma classe para exibir valores formatados:

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }

    // Classe para retornar valores
    // formatados de acordo com a cultura
    // de um país e idioma
    private class CulturaValores
    {
```

```
    public string Nome { get; set; }
    public stringCodigo { get; set; }
    public string DataSimples { get; set; }
    public string DataExtenso { get; set; }
    public string Numero { get; set; }
    public string Moeda { get; set; }
}

}
```

14. Crie um método para preencher uma instância dessa classe. É necessária uma declaração **using** para o namespace **System.Globalization**:

```
//
// Retorna uma instância da classe CulturaValores
// preenchida com dados de uma cultura
//
private static CulturaValores ObterCulturaValores(
    int n, DateTime d, CultureInfo cultureInfo)
{
    return new CulturaValores()
    {
        Nome = cultureInfo.DisplayName,
        Código = cultureInfo.Name,
        DataSimples = d.ToString("d", cultureInfo),
        DataExtenso = d.ToString("D", cultureInfo),
        Numero = n.ToString("n", cultureInfo),
        Moeda = n.ToString("c", cultureInfo)
    };
}
```

15. Crie um método para obter a lista de todas as culturas disponíveis:

```
//
// Retorna uma lista de objetos do tipo
// CulturaValores preenchido com todas
// as culturas disponíveis no .NET Framework
//
```

```
private List<CulturaValores> ObterInfoCultura(
    int n, DateTime d)
{
    var lista = new List<CulturaValores>();

    var cultureInfoLista =
        CultureInfo.GetCultures(CultureTypes.AllCultures);

    foreach (var cultureInfo in cultureInfoLista)
    {
        lista.Add(ObterCulturaValores(n, d, cultureInfo));
    }

    return lista;
}
```

16. Configure o evento **Click** do botão para que ele chame o método **ObterInfoCultura** e vincule a grid:

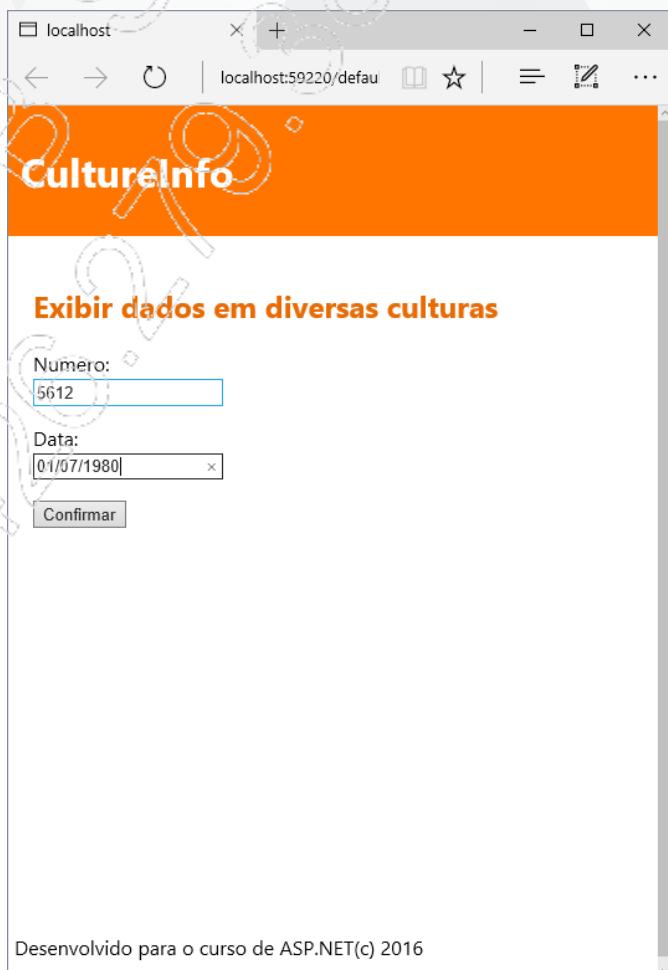
```
protected void confirmar_Click(object sender, EventArgs e)
{
    //Apaga a mensagem e limpa a Grid
    mensagem.Visible=false;
    gv.DataSource = null;

    //Tenta obter dados de número e data
    int n;
    DateTime d;

    if(!int.TryParse(numero.Text, out n))
    {
        mensagem.Text="Número inválido";
        mensagem.Visible=true;
        return;
    }
```

```
if (!DateTime.TryParse(data.Text, out d))  
{  
    mensagem.Text = "Data inválida";  
    mensagem.Visible = true;  
    return;  
}  
  
//Obtém a lista de valores  
//de data e número preenchido  
//nas diversas culturas disponíveis  
var lista = ObterInfoCultura(n, d);  
  
//Vincula a Grid  
gv.DataSource = lista;  
gv.DataBind();  
}  
}
```

17. Teste o aplicativo completo:



Visual Studio 2015 - ASP.NET com C# Recursos Avançados

CultureInfo

Exibir dados em diversas culturas

Numero:

Data:

Nome	Código	Data	Data por Extenso	Número	Formato Monetário
Idioma Invariável (País Invariável)		07/01/1980	Tuesday, 01 July 1980	5,612.00	R\$ 5,612.00
Afar	aa	01/07/1980	Talaata, Qado Dirri 01, 1980	5,612.00	Br5,612.00
Afar (Djibouti)	aa-DJ	01/07/1980	Talaata, Qado Dirri 01, 1980	5,612.00	Fdj5,612
Afar (Eritreia)	aa-ER	01/07/1980	Talaata, Qado Dirri 01, 1980	5,612.00	Nfk5,612.00
Afar (Etiópia)	aa-ET	01/07/1980	Talaata, Qado Dirri 01, 1980	5,612.00	Br5,612.00
Africâner	af	1980-07-01	Dinsdag, 01 Julie 1980	5 612,00	R5 612,00
Africanês (Namíbia)	af-NA	1980-07-01	Dinsdag 1 Julie 1980	5 612,00	\$5 612,00
Africâner (África do Sul)	af-ZA	1980-07-01	Dinsdag, 01 Julie 1980	5 612,00	R5 612,00
Aghem	agq	1/7/1980	tsu?ughœ 1 ndzɔŋjɔdùmlo 1980	5 612,00	5 612FCFA
Aghem (Camarões)	agq-CM	1/7/1980	tsu?ughœ 1 ndzɔŋjɔdùmlo 1980	5 612,00	5 612FCFA
Akan	ak	1980/07/01	Benada, 1980 Ayewoho-Kitawonsa 01	5,612.00	GH₵5,612.00
Akan (Gana)	ak-GH	1980/07/01	Benada, 1980 Ayewoho-Kitawonsa 01	5,612.00	GH₵5,612.00
Amárico	am	01/07/1980	መከኑት, 1 ዓይነት 1980	5,612.00	ብር5,612.00
Amárico (Etiópia)	am-ET	01/07/1980	መከኑት, 1 ዓይነት 1980	5,612.00	ብር5,612.00
Árabe	ar	19/08/00	١٩/١٤٠٠/١٩٨٠	5,612.00	ر.س. 5,612.00
Árabe (Mundo)	ar-001	1/7/1980	١٩٨٠، ١ يوليو	5,612.00	XDR 5,612.00
Árabe (E.A.U.)	ar-AE	01/07/1980	١٩٨٠، ١ يوليه	5,612.00	د.إ. 5,612.00
Árabe (Bahrein)	ar-BH	01/07/1980	١٩٨٠، ١ يوليه	5,612.00	ب.ب. 5,612.00
Árabe (Djibouti)	ar-DJ	1/7/1980	١٩٨٠، ١ يوليو	5,612.00	Fdj 5,612

7

MVC - Validação e filtros

- ✓ Validação;
- ✓ Data Annotation;
- ✓ ModelState;
- ✓ Display;
- ✓ DisplayFormat;
- ✓ EditorForModel;
- ✓ Geração automática de código.

7.1. Introdução

Validar dados é uma tarefa que exige uma atenção especial em qualquer aplicação. O processo de verificar e manter a integridade dos dados engloba diversas áreas, como verificação de tipos de dados, limites, credenciais, criptografia, certificados, meios de transporte de informação e formatos. Uma parte desse processo é a validação dos dados que o usuário digita.

7.2. Validação

O processo de validação de campos consiste em verificar os seguintes critérios:

- **Tipo de dados:** Verificar se o campo é do tipo **Data**, **Inteiro**, **Booleano**, **Texto** ou **Moeda** e se o conteúdo do campo é compatível, de acordo com a **CultureInfo** da aplicação;
- **Faixa:** Verificar se o valor está dentro de uma faixa aceitável. Por exemplo, o dia do mês deve ser um número entre 1 e 31, a data final de um relatório deve ser maior ou igual à data inicial, e assim por diante;
- **Formato:** Alguns itens seguem convenções de formato. Por exemplo, a sigla do estado deve ser duas letras maiúsculas.

A validação deve ser feita no cliente, se possível, e sempre no servidor. A validação no cliente geralmente utiliza JavaScript, que é facilmente interceptado e manipulado, sendo, portanto, muito pouco seguro. Uma vez que os dados chegaram no servidor, devem ser verificados quanto à sua integridade e aplicadas as regras de negócio necessárias.

7.3. Data Annotation

O MVC utiliza uma tecnologia chamada **Data Annotation**, que se resume em declarar as regras que devem ser aplicadas ao Model. O resto do processo acontece, em grande parte, automaticamente. Considere a seguinte classe de modelo (ela será usada para validação):

```
public class Produto
{
    public string Nome { get; set; }
    public decimal Preco { get; set; }
}
```

Nos subtópicos seguintes, veremos os principais validadores.

7.3.1. Required

Para tornar os campos obrigatórios, é necessário incluir o atributo **Required** neles. O namespace em que residem as classes de anotações usadas pelo MVC é o **System.ComponentModel.DataAnnotations**.

```
public class Produto
{
    [Required]
    public string Nome { get; set; }

    [Required]
    public decimal Preco { get; set; }

}
```

Vejamos a criação de um Controller simples:

```
public class ProdutoController : Controller
{
    public ActionResult Novo()
    {
        return View();
    }
}
```

Vejamos, também, a criação de uma View usando o **Produto** como modelo. O helper HTML fornece o método **ValidationSummary**, que interage com o modelo declarado na View e cria uma validação usando, a princípio, a biblioteca jQuery.

```
@model Capitulo07.Models.Produto

@{
    Layout=null;
}

<script src="/Scripts/jquery-1.10.2.js"></script>
<script src="/Scripts/jquery.validate.js"></script>
<script src="/Scripts/jquery.validate.unobtrusive.js"></script>

<h2>Novo Produto</h2>

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

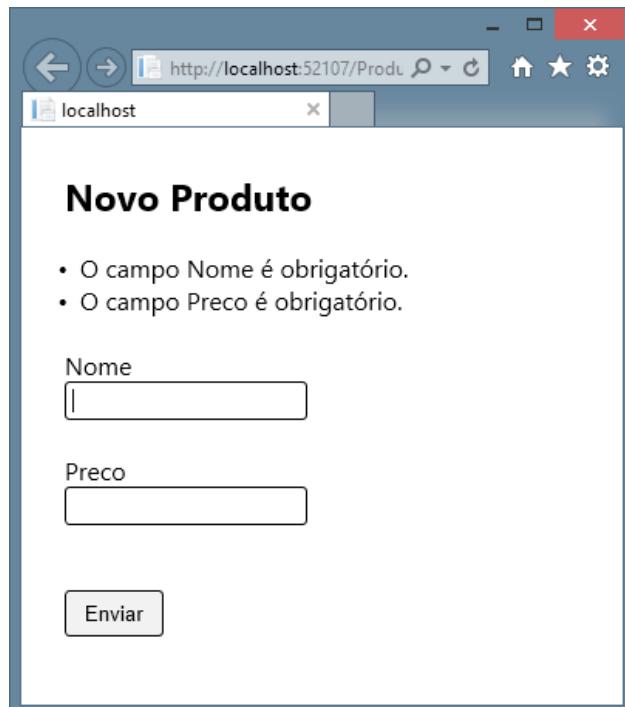
    @Html.LabelFor(model => model.Nome)

    @Html.TextBoxFor(model => model.Nome)

    @Html.LabelFor(model => model.Preco)

    @Html.TextBoxFor(model => model.Preco)

    <input type="submit" value="Gravar" />
}
```

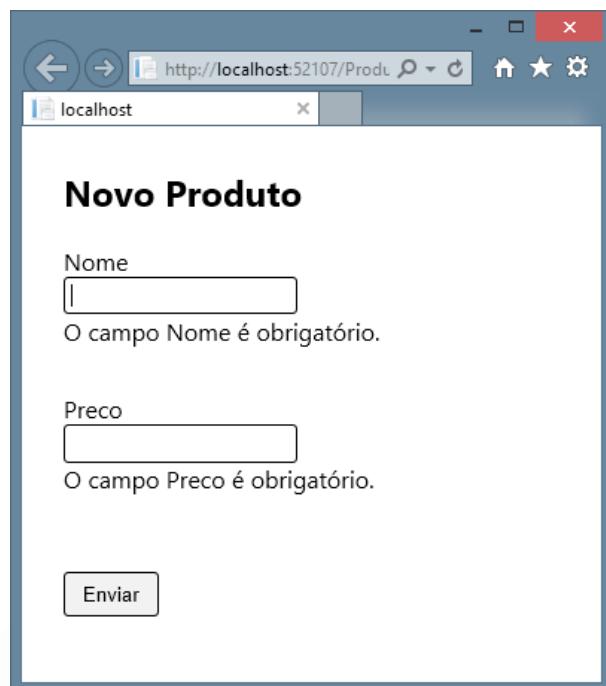


É possível mostrar a mensagem de validação junto do TextBox usando o método **ValidationMessageFor**, disponibilizado pelo helper HTML.

```
@using (Html.BeginForm())
{
    @Html.LabelFor(model => model.Nome)
    @Html.TextBoxFor(model => model.Nome)
    @Html.ValidationMessageFor(model => model.Nome)

    @Html.LabelFor(model => model.Preco)
    @Html.TextBoxFor(model => model.Preco)
    @Html.ValidationMessageFor(model => model.Preco)

    <input type="submit" value="Enviar" />
}
```



7.3.2. StringLength

O atributo **StringLength** define o tamanho total permitido para um texto. É muito útil para não truncar informações ao gravar no banco de dados.

```
public class Produto
{
    [Required]
    [StringLength(30)]
    public string Nome { get; set; }

    [Required]
    public decimal Preco { get; set; }
}
```

É possível alterar a mensagem padrão, definindo o atributo **ErrorMessage**:

```
[StringLength(30, ErrorMessage="Tamanho máximo: 30  
caracteres")]
```



7.3.3. Regular Expression

Usar expressões regulares para validar formatos de entrada é uma das maneiras mais versáteis que existe. Pode ser usada para qualquer informação que tenha um formato definido: e-mail, CPF, CNPJ, IP, placa de carro, sigla de estado etc. No exemplo adiante, a classe **Veículo** aceita no campo **Placa** apenas os formatos corretos:

```
public class Veiculo  
{  
    [RegularExpression("[A-Z]{3}[0-9]{4}")]  
    [Required]  
    public string Placa { get; set; }  
  
    [Required]  
    public int Ano { get; set; }  
  
    [Required]  
    public string Modelo{ get; set; }  
}
```

7.3.4. Range

O atributo **Range** define os limites de campos numéricos. Os parâmetros a serem passados são o valor mínimo e o máximo. A faixa válida inclui esses valores. O exemplo a seguir exibe um uso típico: o valor mínimo e máximo para a entrada de um dia do mês.

```
public class Pagamento
{
    [Range(1, 31)]
    public int Mes { get; set; }

    public int Ano { get; set; }

    public decimal Valor { get; set; }
}
```

7.3.5. Remote

O atributo **Remote** permite realizar validações personalizadas no servidor. Por exemplo, em um programa de vendas, talvez seja necessário consultar o estoque ao informar uma quantidade. Isso só é possível fazendo uma chamada ao servidor.

```
[Remote("VerificarEstoque", ErrorMessage="EstoqueInsuficiente")]
public int Quantidade { get; set; }
```

O atributo **Remote** espera passar o nome de um método que retorne um valor booleano, encapsulado em um objeto JSON.

```
public JsonResult VerificarEstoque(int quantidade)
{
    bool = ObterInfoEstoque(quantidade, produtoId);
    return Json(false, JsonRequestBehavior.AllowGet);
}
```

7.3.6. Compare

O atributo **Compare** verifica se duas propriedades de um modelo são iguais. É usado tipicamente para confirmação de dados:

```
[Compare("Senha")]
public string RepetirSenha { get; set; }
```

Observe que, em uma das versões no MVC, existe um conflito entre **Remote** e **Compare**. O atributo **Compare** estava definido nos namespaces **System.Web.Mvc** e **System.ComponentModel.DataAnnotations**. Não é um problema sério, pois, se isso acontecer, basta referenciar uma das classes pelo nome completo:

```
[System.Web.Mvc.Compare("Senha")]
public string RepetirSenha { get; set; }
```

7.4. ModelState

Ao clicar no botão **Enviar**, os dados do formulário são enviados para a URL informada. O método que receberá os dados do formulário deverá ter as seguintes características:

1. Deve ser marcado com o atributo **HttpPost**;
2. Deve receber como parâmetro uma instância do modelo, parâmetros iguais aos campos do formulário ou uma coleção do tipo **FormCollection**. A melhor estratégia é receber uma instância do modelo;
3. Deve ser verificado se o modelo é válido por meio da propriedade **IsValid** da classe **ModelStateDictionary**, disponibilizada por meio da propriedade **ModelState** da classe **Controller**.

O exemplo seguinte mostra o método que recebe o formulário de produtos:

```
[HttpPost]
public ActionResult Novo(Produto p)
{
    if (ModelState.IsValid)
    {
        //Código para Gravar o Produto
        //Geralmente redireciona para uma página
        //com mensagem de sucesso
    }

    //Se não redirecionou, volta para a View
    return View(p);
}
```

A classe **ModelStateDictionary** (disponibilizada por meio da propriedade **ModelState**) contém todos os dados de validação de um formulário, incluindo cada campo, se é válido ou não e, se não for, a mensagem de erro correspondente.

7.5. Display

Alguns atributos interagem com o método **LabelFor** e **TextBoxFor** do helper HTML. O atributo **Display** determina qual texto será exibido no lugar do nome da propriedade:

- **Model**

```
public class Produto
{
    [Display(Name="Nome do Produto")]
    public string Nome { get; set; }

    [Display(Name="Preço Unitário")]
    public decimal Preco { get; set; }
}
```

- **View**

```
<h2>Novo Produto</h2>
@Html.LabelFor(model => model.Nome)
@Html.TextBoxFor(model => model.Nome)

@Html.LabelFor(model => model.Preco)
@Html.TextBoxFor(model => model.Preco)
```

- **Resultado**



A classe de atributo **Display** define as seguintes propriedades:

- **AutoGeneratedField**

Indica se o campo deve ser gerado automaticamente. Isso faz com que o campo fique somente leitura.

- **Description**

É a descrição do campo. Esta propriedade não interage automaticamente com os helpers **TextBoxFor** ou **LabelFor**, mas é fácil construir uma extensão do helper HTML para exibir a descrição. É uma das vantagens de se construir um sistema aberto, que pode ser expandido:

```
using System.Linq.Expressions;
using System.Web;
using System.Web.Mvc;

namespace System.Web.Mvc.Html
{
    public static class HtmlExtensions
    {
        public static IHtmlString DescriptionFor<TModel,
TValue>(
            this HtmlHelper<TModel> html,
            Expression<Func<TModel, TValue>> expression)
        {
            var metadata = ModelMetadata
                .FromLambdaExpression(expression, html.ViewData);
            var description = metadata.Description;
            return new HtmlString(description);
        }
    }
}
```

Essa classe cria o método de extensão **DescriptionFor**, que pode ser usado para exibir uma descrição do campo, por exemplo, usando a propriedade **Title**. A sintaxe completa do método **TextBoxFor** é a seguinte:

```
TextBoxFor( expressao que identifica o campo, objeto com
propriedades html)
```

Por exemplo, para definir um controle **input** para um campo **Email** (portanto, Id e nome serão **Email**) com as propriedades **Style** e **MaxLength** definidas, o comando seria o seguinte:

```
TextBoxFor(m=>m.Email, new { style="color:red", maxLength="40" })
```

No caso do helper para exibir a descrição como um **toolTip**, pode ser usado este modelo:

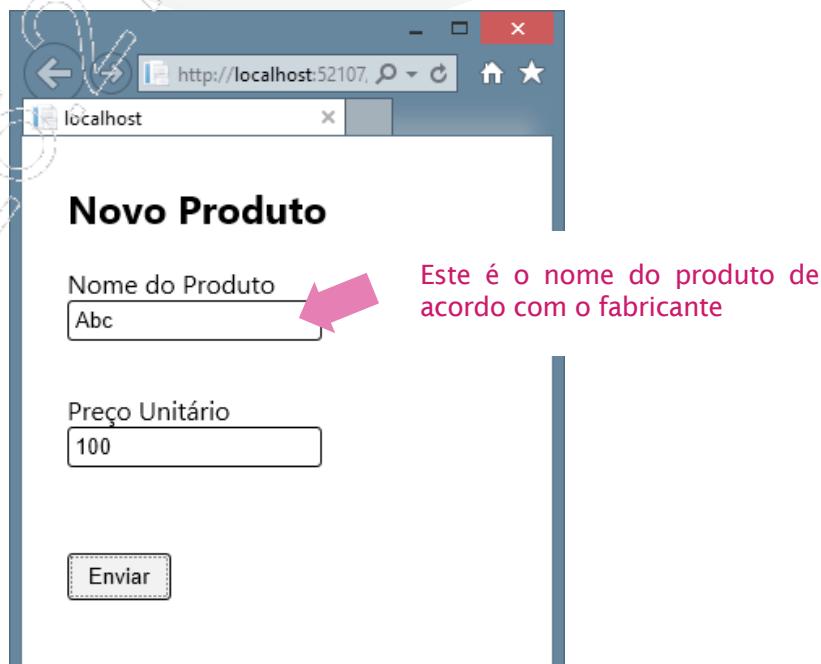
```
@Html.TextBoxFor(model => model.Nome,
new { title = @Html.DescriptionFor(model => model.Nome) })
```

No modelo, se a descrição estiver definida, será exibida como **toolTip** (propriedade **Title** do HTML), conforme mostrado adiante:

```
public class Produto
{
    [Display(
        Name="Nome do Produto",
        Description="Este é o nome do produto de acordo com o
        fabricante")]
    public string Nome { get; set; }

    [Display(Name="Preço Unitário")]
    public decimal Preco { get; set; }

}
```



- **GroupName**

Uma string usada para agrupar campos na interface.

- **Name**

O nome que aparece na legenda do campo. Se não for informado, o nome da propriedade é exibido.

- **Order**

A ordem em que esse campo aparece. Não precisa ser numerado exatamente. Cada número atribuído tem um peso. Esta propriedade é utilizada quando o formulário de um Model é criado automaticamente.

- **Prompt**

Texto usado como marca d'água nas caixas de texto. Quando o usuário começa a digitar, esse texto some. Alguns componentes de interface implementam essa funcionalidade.

- **ResourceType**

Retorna o tipo (**Type**) do modelo usado nas propriedades **Name**, **Description** e **Display**.

- **ShortName**

O nome usado quando a propriedade é exibida em uma tabela, nos títulos das colunas. Por padrão, o nome da propriedade é exibido ou o texto definido no parâmetro **Name** do atributo **Display**.

- **ReadOnly**

O atributo **ReadOnly** bloqueia o campo para edição nos métodos **EditorFor** e **TextBoxFor** do helper HTML.

7.6. DisplayFormat

Este atributo define o formato que será usado para exibir os dados de um campo.

```
[Display(Name="Preço Unitário")]
[DisplayFormat(DataFormatString="{0:c}")]
public decimal Preco { get; set; }
```

Em uma View, ao utilizar um método para exibir dados do produto, o formato é aplicado automaticamente.

```
<h4>Produto</h4>
<hr />
<dl>
    <dt> @Html.DisplayNameFor(model => model.Nome) </dt>
    <dd> @Html.DisplayFor(model => model.Nome) </dd>
    <dt> @Html.DisplayNameFor(model => model.Preco) </dt>
    <dd> @Html.DisplayFor(model => model.Preco) </dd>
</dl>
```

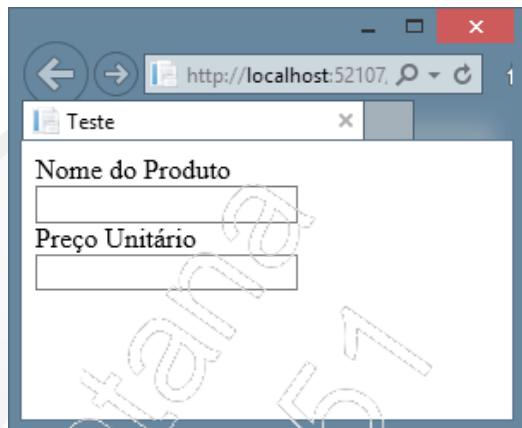


7.7. EditorForModel

O método **EditorForModel** do helper HTML utiliza o modelo para criar automaticamente os labels e as caixas de texto necessárias para a edição de um modelo simples. A declaração é simples:

```
@Html.EditorForModel()
```

Esse comando gera a seguinte página:



Que é o resultado do seguinte código HTML:

```
<div class="editor-label"><label for="Nome">Nome do Produto</label></div><div class="editor-field"><input class="text-box single-line" id="Nome" name="Nome" type="text" value="" /><span class="field-validation-valid" data-valmsg-for="Nome" data-valmsg-replace="true"></span></div><div class="editor-label"><label for="Preco">Preço Unitário</label></div><div class="editor-field"><input class="text-box single-line" data-val="true" data-val-number="The field Preço Unitário must be a number." data-val-required="O campo Preço Unitário é obrigatório." id="Preco" name="Preco" type="text" value="" /><span class="field-validation-valid" data-valmsg-for="Preco" data-valmsg-replace="true"></span></div>
```

Embora seja complicado entender o porquê de não existir espaço e nem separação dos elementos, arrumar e colocar alguns comentários no código HTML gerado pelo método **EditorFor** torna mais claro o seu entendimento. Ele é sempre composto de uma legenda (**label**), um campo (**input**) e um validador (**span**), estando cada grupo dentro de uma Div.

```
<!-- Nome do Produto -->
<div class="editor-label">
    <label for="Nome">Nome do Produto</label>
</div>

<div class="editor-field">
    <input class="text-box single-line"
        id="Nome" name="Nome" type="text" value="" />
    <span class="field-validation-valid"
        data-valmsg-for="Nome"
        data-valmsg-replace="true"></span>
</div>

<!--Preço do Produto -->
<div class="editor-label">
    <label for="Preco">Pre&#231;o Unit&#225;rio</label>
</div>

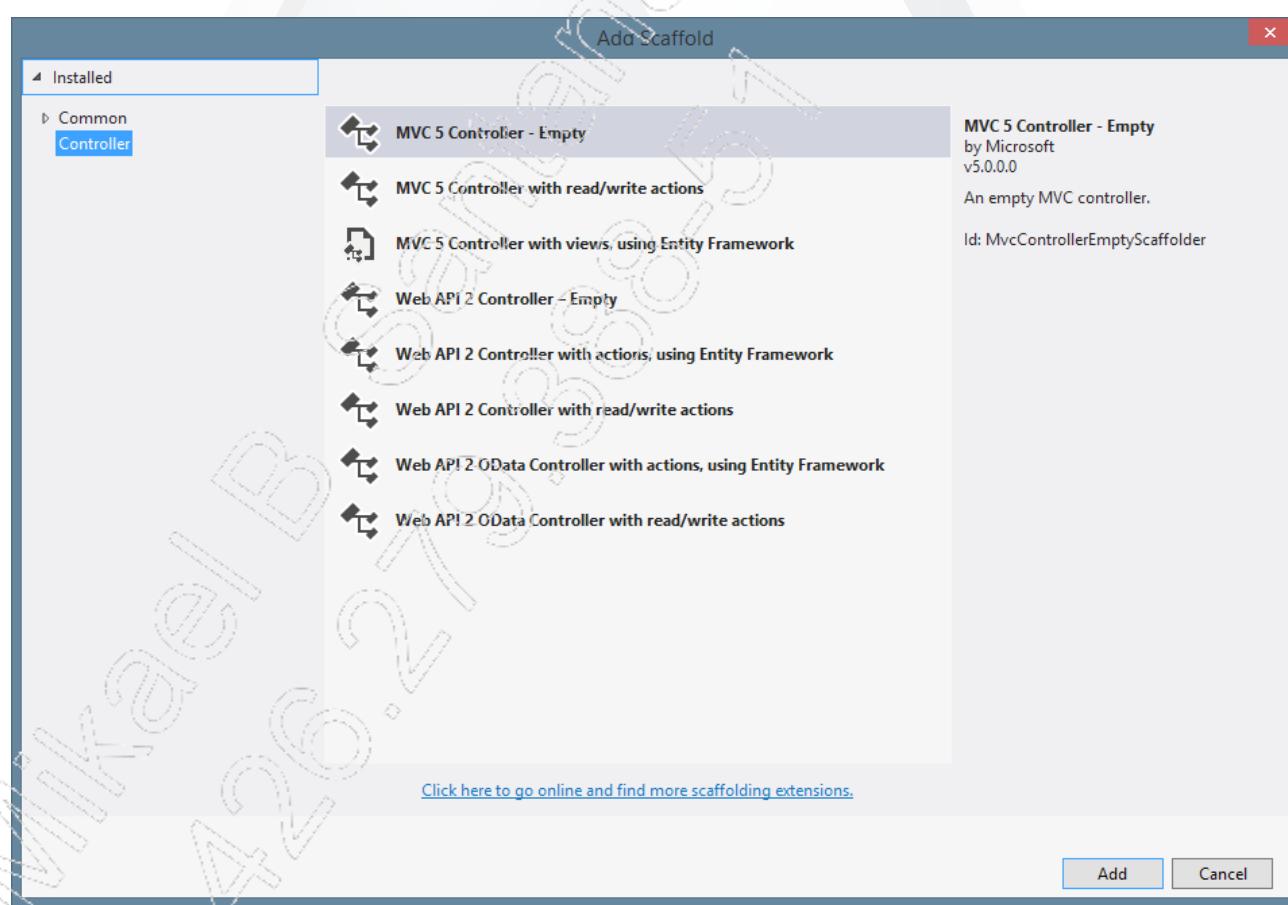
<div class="editor-field">
    <input class="text-box single-line" data-val="true"
        data-val-number=" must be a number."
        data-val-required="Preço Unitário é obrigatório."
        id="Preco"
        name="Preco"
        type="text"
        value="" />
    <span class="field-validation-valid"
        data-valmsg-for="Preco"
        data-valmsg-replace="true"></span>
</div>
```

7.8. Geração automática de código

Uma vez criado o modelo de dados, o Visual Studio contém recursos para criar modelos para os Controllers e as Views nas operações mais comuns de banco de dados, como listagem de dados, inclusão, alteração, exclusão e exibição de um registro.

7.8.1. Criando um Controller

Para criar uma classe Controller, escolha, na pasta **Controller**, a opção **Add Controller**. As opções de modelos são as seguintes:



- **MVC 5 Controller – Empty:** Uma classe Controller apenas com o método **Index**;
- **MVC 5 Controller with read/write actions:** Uma classe Controller com os métodos básicos para criar itens (**Create**), visualizar (**Index** e **Details**), alterar (**Edit**) e excluir (**Delete**);
- **MVC 5 Controller with views, using Entity Framework:** Este modelo exige mais configuração. É necessária uma classe Model, uma classe de contexto (**DbContext**) do Entity Framework. O Visual Studio cria não só a classe Controller, mas também as Views e todas as operações CRUD (Create, Read, Update e Delete);
- **Web API 2 Controller – Empty:** Um modelo usando a Web API sem nenhum método. Mesmo assim, inclui todas as bibliotecas necessárias para rodar o framework Web API e configura o registro inicial no **Global.asax** para ficar disponível na aplicação;
- **Web API 2 Controller with actions, using Entity Framework:** Cria os métodos (não cria as Views) com as operações CRUD usando uma classe de contexto do Entity Framework, usando o modelo da Web API, com os métodos começando com GET, PUT ou POST ou DEL;
- **Web API 2 Controller with read/write actions:** Criar os métodos para operações CRUD padrão da Web API;
- **Web API 2 OData Controller with actions, using Entity Framework:** Cria os métodos (não cria as Views) com as operações CRUD usando uma classe de contexto do Entity Framework, usando o modelo OData;
- **Web API 2 OData Controller with read/write actions:** Cria os métodos para operações CRUD padrão OData.

O exemplo a seguir mostra, de maneira simplificada e resumida, o código gerado usando a opção **MVC 5 Controller with read/write actions**:

```
public class ExemploReadWriteController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    public ActionResult Details(int id)
    {
        return View();
    }

    public ActionResult Create()
    {
        return View();
    }

    [HttpPost]
    public ActionResult Create(FormCollection collection)
    {
        return View();
    }

    public ActionResult Edit(int id)
    {
        return View();
    }

    [HttpPost]
    public ActionResult Edit(int id, FormCollection collection)
    {
        return View();
    }
}
```

```
public ActionResult Delete(int id)
{
    return View();
}

[HttpPost]
public ActionResult Delete(int id, FormCollection
collection)
{
    return View();
}

}
```

Os métodos criados são modelos para executar as seguintes ações:

- **Index**

Index é o método padrão e retorna a View padrão. Normalmente, essa View apresenta uma lista de itens que podem ser editados e um botão **Novo**.

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

- **Details**

```
public ActionResult Details(int id)
{
    return View();
}
```

O método **Details** recebe um Id para localizar o registro. Não está implementado, mas o código interno deste método é algo como o código a seguir:

```
public ActionResult Details(int id)
{
    var item=Db.ObtemItem(id);

    return View(item);
}
```

- **Create**

O método **Create** exibe uma View com um formulário vazio. Ao fazer um POST, esse formulário é enviado e a sobrecarga deste método é executada. Uma possível implementação está listada a seguir:

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

[HttpPost]
public ActionResult Create(FormCollection collection)
{
    Db.Incluir(collection["Nome"], collection["Email"]);
    return View();
}
```

- **Edit**

O método **Edit** localiza o registro, mostra um formulário para edição e, então, faz um POST para gravar os dados alterados.

```
public ActionResult Edit(int id)
{
    var p=Db.LocalizarItem(id)
    return View(p);
}
```

```
[HttpPost]
public ActionResult Edit(int id, FormCollection collection)
{
    bool ok=Db.Alterar(id, collection["Nome"],
collection["Email"]);

    return View(ok);
}
```

- **Delete**

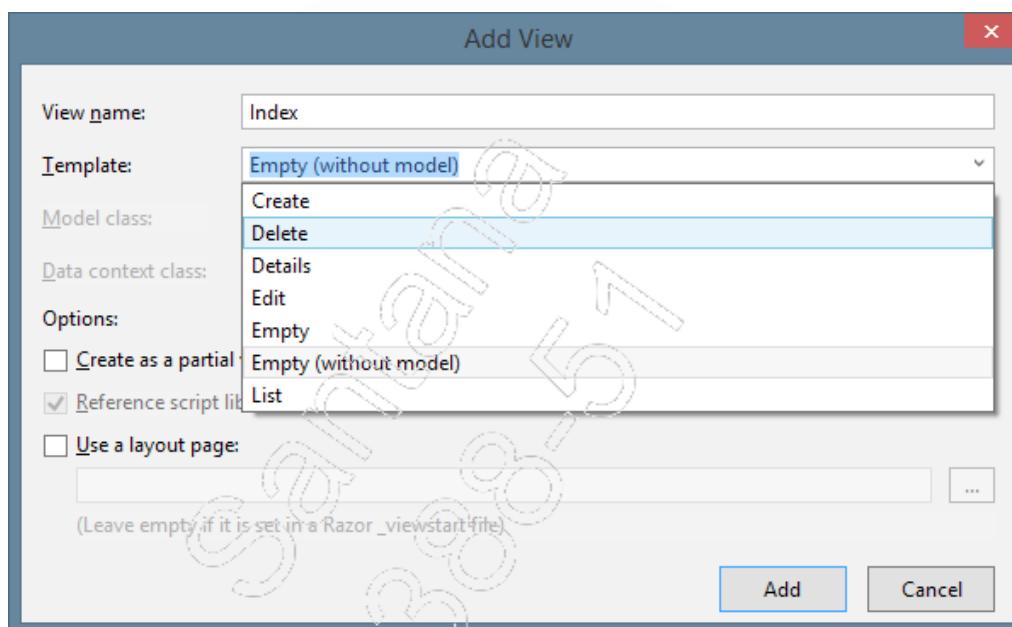
O método **Delete** é igual ao método **Edit**, excluindo os dados do banco.

```
public ActionResult Delete(int id)
{
    var p=Db.LocalizarItem(id)
    return View(p);
}

[HttpPost]
public ActionResult Delete(int id, FormCollection
collection)
{
    bool ok=Db.Excluir(id)
    return View(ok);
}
```

7.8.2. Criando uma View

As Views podem ser geradas automaticamente, adicionadas manualmente e geradas uma a uma, para cada método da classe Controller. Para criar uma View a partir de um método, é necessário usar o menu de contexto do método action de um Controller e escolher a opção **Add View**. Uma janela ficará disponível na qual será possível escolher o nome da View e um modelo (**Template**). O formulário, visualizador ou listagem é criado dessa forma automaticamente, facilitando muito o trabalho inicial.



Uma View criada pelo Visual Studio utiliza todos os recursos de validação, folhas de estilo e helpers para a construção de formulários, bundles e minifications.

A seguir, uma View criada pelo Visual Studio e os recursos utilizados:

View fortemente tipada

```
@model Capitulo07.Models.Cliente
```

Layout de Web Pages

```
@{  
    Layout = null;  
}
```

HTML5

```
<!DOCTYPE html>

<html>
<head>
```

Bootstrap

```
<meta name="viewport" content="width=device-width" />
<title>Create</title>
</head>
<body>
```

Bundles

```
@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/jqueryval")
```

Helpers

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

Security

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

Classes do Bootstrap

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

Validation

```
@Html.ValidationSummary(true, "", new { @class =
"text-danger" })
<div class="form-group">
```

Helpers para criação de itens de formulário e validação

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

Bootstrap - Colunas

```
@Html.ValidationSummary(true, "", new { @class =
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Create"
class="btn btn-default" />
    </div>
</div>
</div>
</div>
```

Helpers: Links

```
@Html.ActionLink("Back to List", "Index")
</div>
</body>
</html>
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Os principais métodos disponibilizados pelo helper HTML para criação de elementos de formulário HTML são: **LabelFor**, **TextBoxFor**, **ValidationMessage**, **BeginForm**, **CheckBoxFor**, **DropDownListFor**, **TextAreaFor**, **PasswordFor**, **RadioButtonFor** e **ValidationSummary**;
- **Data Annotation** é um dos recursos utilizados para validar a entrada de dados de usuários em formulários no ASP.NET MVC;
- Para marcar um campo como obrigatório, é necessário usar o atributo **Required** no modelo de dados;
- Para definir a legenda de uma propriedade de uma classe Model, é necessário usar o atributo **Display**;
- Para definir o formato de exibição de dados, usa-se o atributo **DisplayFormat**;
- Os principais validadores são: **Required**, **StringLength**, **Compare**, **Remote**, **Range** e **RegularExpression**;
- A classe **ModelState** é usada para saber se o modelo é válido ou não, por meio de sua propriedade **IsValid**;
- O método **EditorForModel()**, do helper HTML, é utilizado para criar um formulário de edição completo, com legendas e campos, baseado em um modelo de dados;
- É possível criar classes Controllers automaticamente por meio dos modelos do Visual Studio. Existem modelos que não usam nenhuma tecnologia específica, outros usam o Entity Framework, a Web API ou OData;
- É possível criar Views por meio dos assistentes do Visual Studio, diretamente a partir dos métodos dos Controllers.

7

MVC - Validação e filtros

Teste seus conhecimentos

Mikael
B
279.
426.



IMPACTA
EDITORA

1. Qual atributo deve ser utilizado para um campo obrigatório, usando Data Annotation?

- a) RequiredFieldValidator
- b) RequiredField
- c) Required
- d) NotOptional
- e) MustBeHere

2. Quais métodos do helper HTML são utilizados para gerar legendas e caixas de texto?

- a) @Html.LegendFor e @Html.FieldFor
- b) @Html.LegendFor e @Html.TextFor
- c) @Html.LegendFor e @Html.TextBoxFor
- d) @Html.LabelFor e @Html.TextBoxFor
- e) @Html.LabelFor e @Html.TextFor

3. Qual classe e método devem ser usados para verificar se um formulário passou por uma validação?

- a) ModelState.IsValid
- b) Model.IsValid
- c) ModelState.NoErrors
- d) Model.Errors.Count==0
- e) ModelState.Errors==null

4. Qual atributo pode ser usado em um campo para validar um formato de dados, como CEP ou e-mail?

- a) [Required]
- b) [Format]
- c) [Mask]
- d) [RequiredFormat]
- e) [RegularExpression]

5. Em um relatório, um valor de uma compra foi impresso como “18721.12” e o correto seria “R\$ 18.721,12”. Qual atributo pode resolver isso?

- a) [Format]
- b) [Display]
- c) [Money]
- d) [DisplayFormat]
- e) [FormatDisplay]

7

MVC - validação e filtros

Mãos à obra!



IMPACTA
EDITORA

Laboratório 1

Neste laboratório, você irá criar um site com imagens por meio do framework MVC, utilizando autenticação OWIN, Razor, métodos do helper HTML, EntityFramework Code First para persistir dados e classes do Bootstrap, bem como configurar o bundle para incluir arquivos CSS, utilizar o objeto **Application** para armazenar informações e o **ModelState** para transferir mensagens de erro para a página.

Além disso, você irá criar formulários para enviar imagens, realizar validações utilizando Data Annotations, validações do lado do servidor, verificando se existem arquivos anexados em formulários, e manipular imagens.

O aplicativo a ser elaborado é o site de uma empresa de viagens com pacotes para o mundo todo. A tela inicial mostra o nome da empresa e dois links: um para os destinos de viagem e outro para uma área administrativa.

O processo será dividido em quatro partes:

- A – Criando a estrutura da aplicação Web;**
- B – Criando a área administrativa;**
- C – Implementando o acesso a dados;**
- D – Implementando a autenticação com OWIN.**

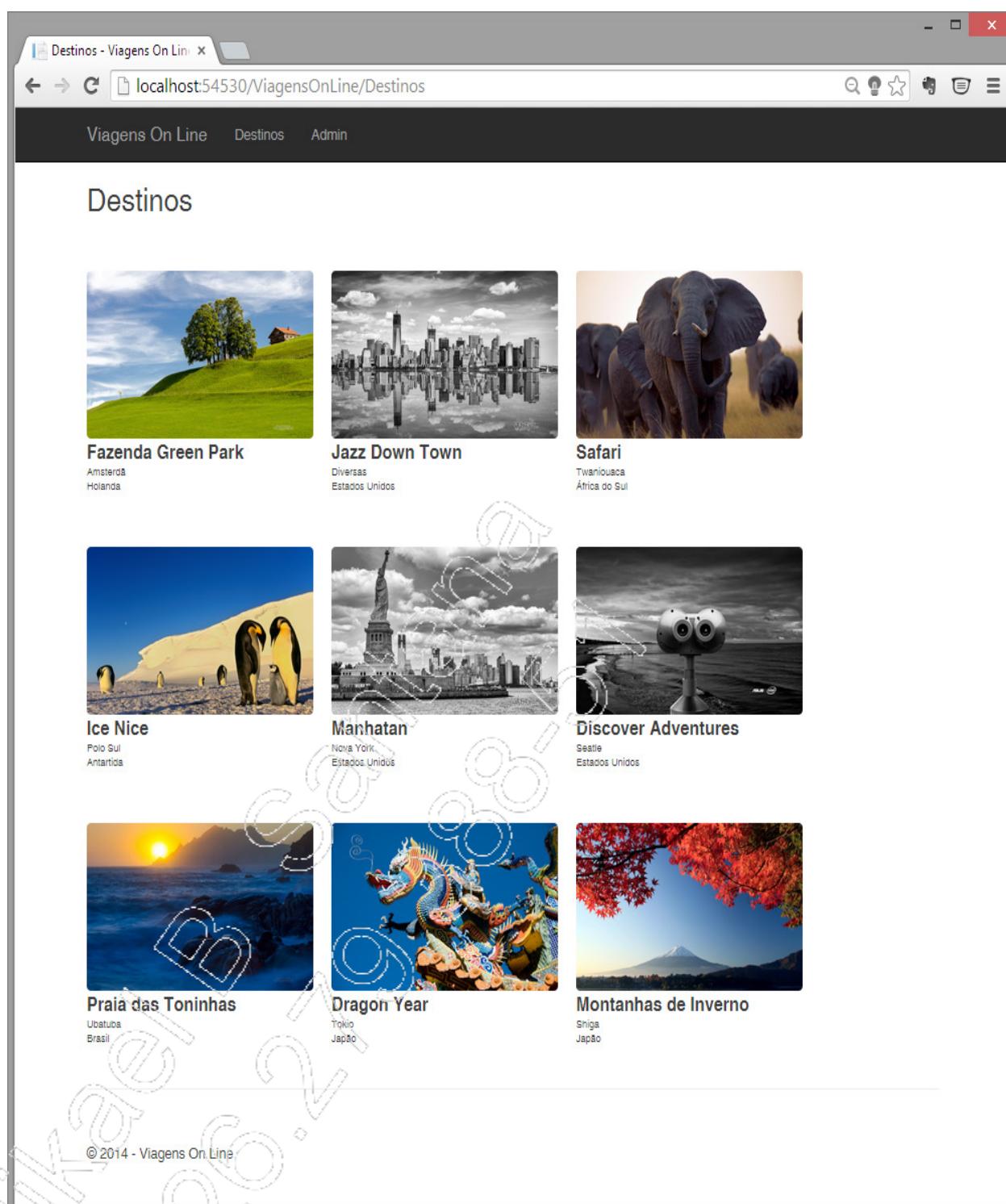
A interface de usuário terá as seguintes telas:

- Uma tela de entrada:



Visual Studio 2015 - ASP.NET com C# Recursos Avançados

- Clicando em **Destinos**, o usuário verá uma lista de viagens disponíveis:



- Clicando em **Admin**, o usuário terá acesso a uma área administrativa com login de acesso:

Login

Nome:

Senha:

© 2014 - Viagens On Line

- A área de administração permitirá incluir, alterar e excluir itens da lista. A primeira tela visível é a listagem:

	Nome	País	Cidade	
	Fazenda Green Park	Holanda	Amsterdã	Alterar Excluir
	Jazz Down Town	Estados Unidos	Diversas	Alterar Excluir
	Safari	África do Sul	Twaniouaca	Alterar Excluir
	Ice Nice	Antartida	Polo Sul	Alterar Excluir
	Manhattan	Estados Unidos	Nova York	Alterar Excluir
	Discover Adventures	Estados Unidos	Seattle	Alterar Excluir
	Praia das Toninhas	Brasil	Ubatuba	Alterar Excluir
	Dragon Year	Japão	Tokio	Alterar Excluir
	Montanhas de Inverno	Japão	Shiga	Alterar Excluir

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

- O administrador poderá clicar em **Novo Destino** para incluir um novo registro no banco de dados, com upload de fotos:

DestinoNovo - Viagens On Line

localhost:54530/Admin/DestinoNovo

Viagens On Line Destinos Admin

Novo Destino

Nome

País

Cidade

Foto
 Escolher arquivo Nenhum arquivo selecionado

Gravar

Voltar

© 2014 - Viagens On Line

- Os dados deverão ser validados:

DestinoNovo - Viagens On Line

localhost:54530/Admin/DestinoNovo

Viagens On Line Destinos Admin

Novo Destino

Nome
 O campo Nome é obrigatório.

País
 O campo País é obrigatório.

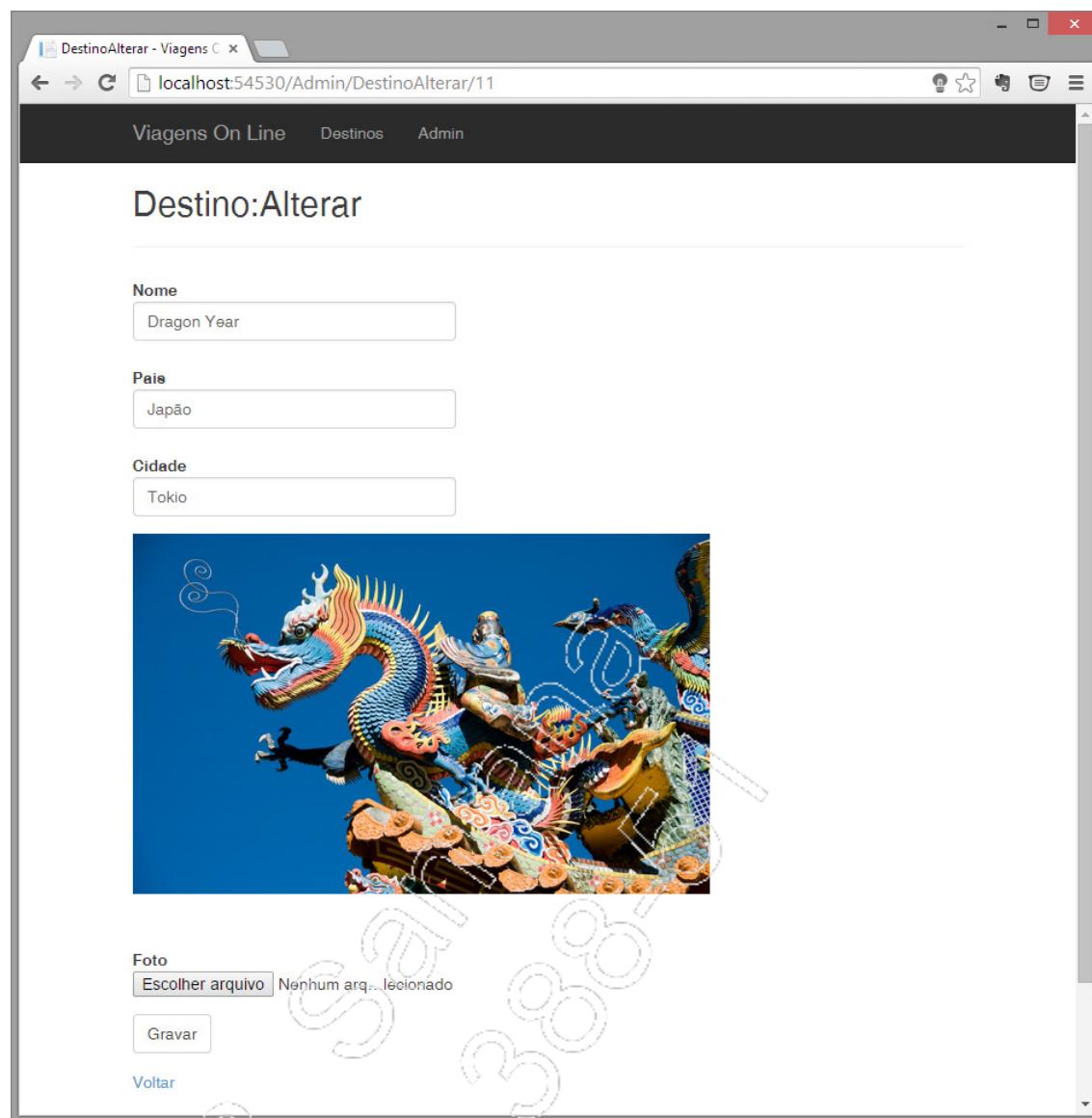
Cidade
 O campo Cidade é obrigatório.

Foto
 Escolher arquivo Nenhum arquivo selecionado

Gravar

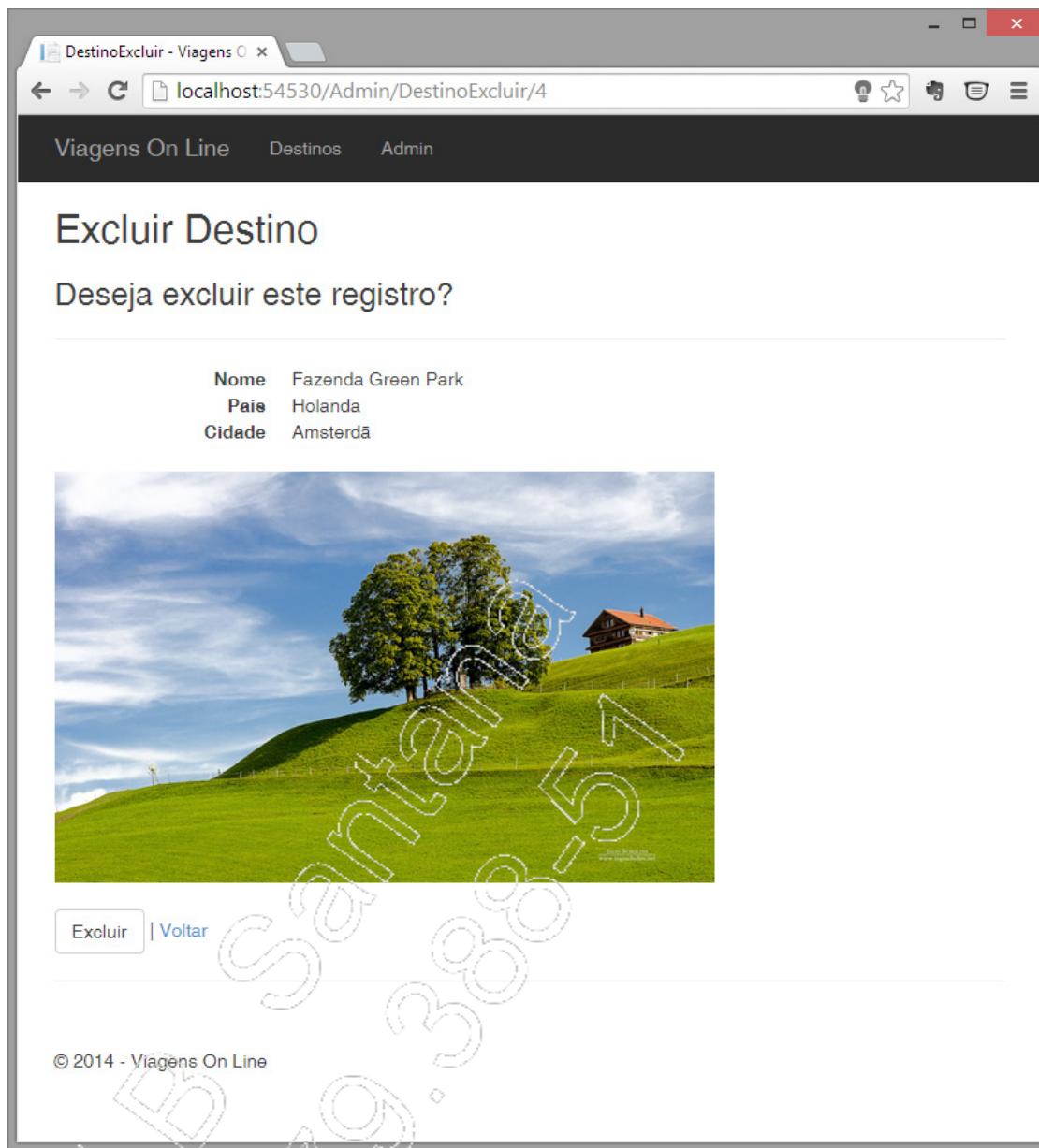
Voltar

- Clicando em Alterar, o administrador poderá alterar qualquer registro:



Mikael B
426.270.
300.

- Clicando em **Excluir**, o administrador poderá excluir qualquer item:



Veremos, agora, o passo a passo de todo o projeto, dividido em quatro blocos: Estrutura, Área Administrativa, Acesso a Dados e Autenticação.

A - Criando a estrutura da aplicação Web

1. Crie um novo projeto MVC sem autenticação chamado **Cap07_Lab01**;
2. Adicione os seguintes pacotes por meio do NuGet: **Entity Framework**, **Microsoft.Owin.Security.Cookies** e **Microsoft.Owin.Host.System.Web**;

3. Altere o arquivo **Global.asax** para conter uma variável global chamada **appName**, que será usada para exibir o nome do aplicativo;

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        Application["appName"] = "Viagens On Line";

        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(...);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

4. Altere o arquivo **\Views\Shared_Layout.cshtml** para criar os menus corretos e para exibir o nome da aplicação por meio da variável **Context.Application["appName"]**:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name=...>

    <title>@ViewBag.Title - @Context.Application["appName"]</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" ...>
```

```
<span class="icon-bar"></span>
<span class="icon-bar"></span>
<span class="icon-bar"></span>
</button>
@Html.ActionLink((string)Context.
Application["appNome"],
"Inicio", "ViagensOnLine",
new { area = "" }, new { @class = "navbar-brand" })
</div>
<div class="navbar-collapse collapse">
<ul class="nav navbar-nav">
<li>@Html.ActionLink("Destinos",
"Destinos", "ViagensOnLine")</li>
<li>@Html.ActionLink("Admin",
"DestinoListagem", "Admin")</li>
</ul>
</div>
</div>
</div>
<div class="container body-content">
@RenderBody() <hr />
<footer>
<p>&copy; @DateTime.Now.Year - @Context.
Application["appNome"]</p>
</footer>
</div>
@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/bootstrap")
@RenderSection("scripts", required: false)
</body></html>
```

5. Altere a classe App_Start / RouteConfig.cs para registrar as URLs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
```

```
namespace CAP07_LAB01
{
    public class RouteConfig
    {
        public static void RegisterRoutes(
            RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "ViagensOnLine",
                action = "Inicio",
                id = UrlParameter.Optional }

);
        }
    }
}
```

As seguintes URL serão usadas:

- **ViagensOnLine/Inicio;**
- **ViagensOnLine/Destinos;**
- **ViagensOnLine/Admin/Login;**
- **ViagensOnLine/Admin/DestinosListagem;**
- **ViagensOnLine/Admin/DestinosIncluir;**
- **ViagensOnLine/Admin/DestinosAlterar;**
- **ViagensOnLine/Admin/DestinosAlterar/id;**
- **ViagensOnLine/Admin/DestinosExcluir.**

6. Adicione um arquivo CSS na pasta **/Content** chamado **Estilos.css**:

```
.imagemInicial {  
  
    width:100%;  
    border-radius:5px;  
}  
  
.alerta {  
  
    padding-top:5px;  
    padding-bottom:5px;  
    color:red;  
    font-weight:bold;  
}  
  
.imagemForm {  
  
    margin-bottom:20px;  
    width:500px;  
}  
  
h2 {  
  
    margin-bottom:20px;  
}  
  
footer{  
  
    margin-top:50px;  
}
```

7. No arquivo de configuração /App_Start/BundleConfig, inclua este arquivo:

```
using System.Web;
using System.Web.Optimization;

namespace CAP07_LAB01
{
    public class BundleConfig
    {
        public static void RegisterBundles(...)
        {
            bundles.Add(new ScriptBundle("~/bundles/jquery")
                .Include(
                    "~/Scripts/jquery-{version}.js"));

            bundles.Add(new ScriptBundle("~/bundles/jqueryval")
                .Include(
                    "~/Scripts/jquery.validate*"));

            bundles.Add(new ScriptBundle("~/bundles/modernizr")
                .Include("~/Scripts/modernizr-*"));

            bundles.Add(new ScriptBundle("~/bundles/bootstrap")
                .Include("~/Scripts/bootstrap.js",
                    "~/Scripts/respond.js"));

            bundles.Add(new StyleBundle("~/Content/css").Include(
                "~/Content/bootstrap.css",
                "~/Content/site.css",
                "~/Content/estilos.css"));

            BundleTable.EnableOptimizations = true;
        }
    }
}
```

8. Apague as Views **Index**, **About** e **Contact**;
9. Apague a classe **HomeController**;
10. Adicione uma classe Controller chamada **ViagensOnLineController**:

```
using CAP07_LAB01.Db;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace CAP07_LAB01.Controllers
{
    public class ViagensOnLineController : Controller
    {
        //
        // Início
        //
        public ActionResult Inicio()
        {
            return View();
        }
    }
}
```

11. Insira uma imagem (pode ser qualquer imagem grande) na pasta **Content** chamada **fundo.jpg**;

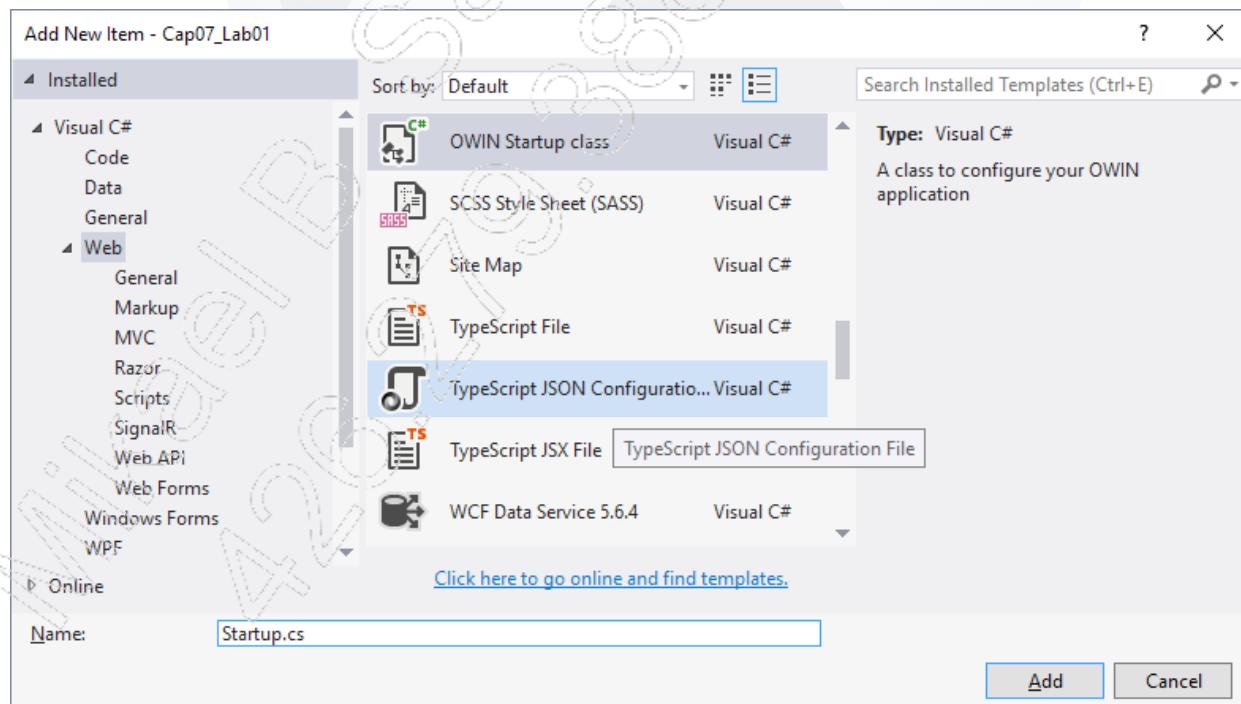
12. Adicione uma view chamada Início:

/Views/ViagensOnLine/Inicio.cshtml

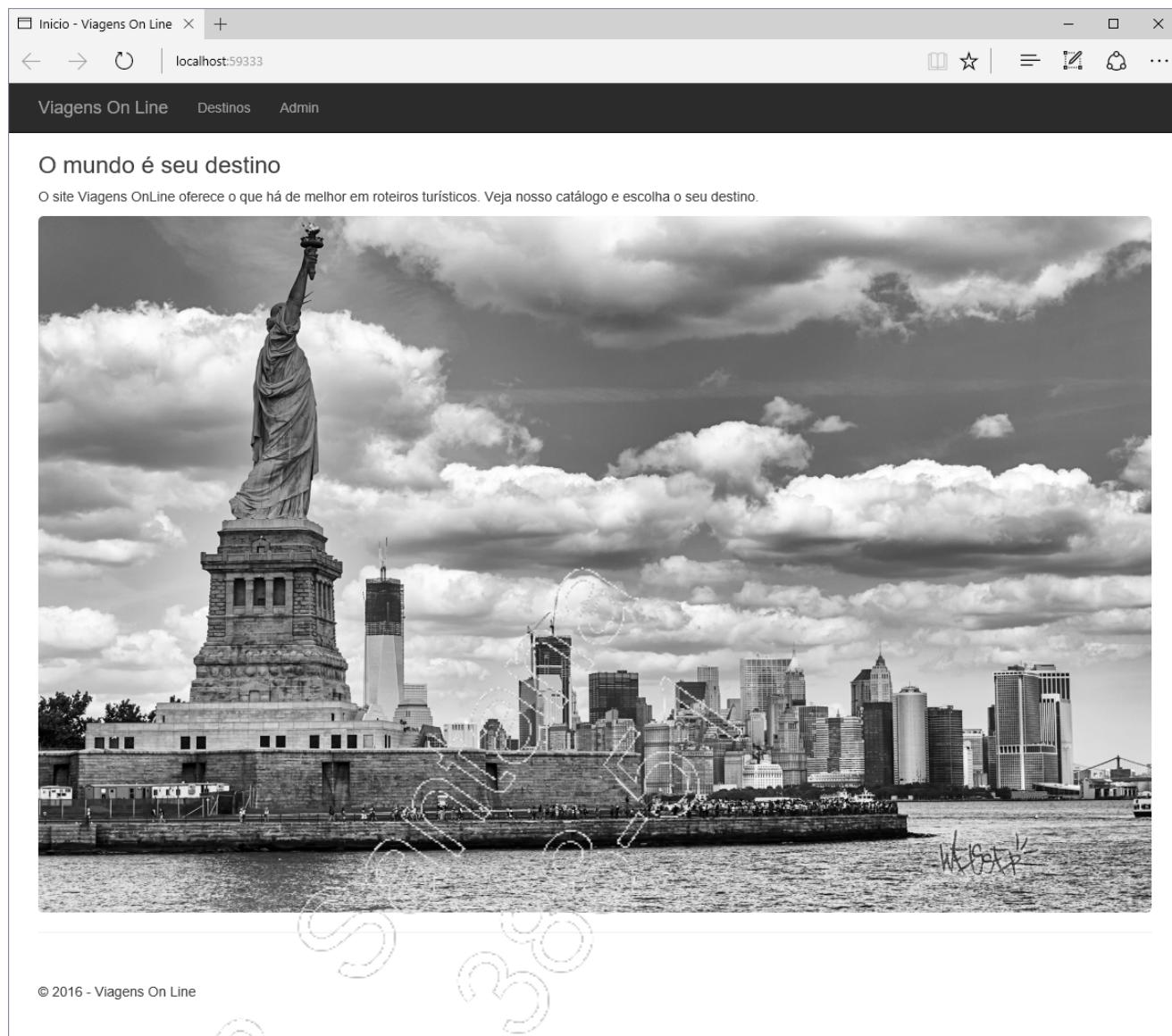
```
@{  
    ViewBag.Title = "Inicio";  
}  
  
<h3>O mundo é seu destino</h3>  
  
<p>O site Viagens OnLine oferece o que há de melhor em roteiros turísticos. Veja nosso catálogo e escolha o seu destino. </p>  
  

```

13. Na raiz do site, insira um arquivo de configuração OWIN, clicando em **Add New Item** e selecionando **OWIN Startup class**. Esse arquivo deve ficar na raiz do site e seu nome deve ser **Startup.cs**:



14. Visualize a página inicial, na URL **ViagensOnLine/Inicio**:



B – Criando a área administrativa

1. Adicione um Controller chamado Admin:

```
public class AdminController : Controller  
{  
}
```

Esse Controller é responsável por incluir, alterar e excluir os destinos de viagem do aplicativo. É uma área que só pode ser usada por um administrador. A parte de autenticação, porém, será feita por último para facilitar os testes de banco de dados e telas. Caso contrário, seria necessário digitar o usuário e senha a cada teste.

2. Adicione um método chamado **DestinoNovo**:

```
public class AdminController : Controller
{
    //
    // Incluir Destino
    //
    [HttpGet]
    public ActionResult DestinoNovo()
    {
        return View();
    }
}
```

3. Os destinos são modelos de dados que contêm uma foto, um nome, o país e a cidade. Defina esse modelo na pasta **Models**:

```
public class Destino
{
    public int DestinoId { get; set; }

    public string Nome { get; set; }

    public string Pais { get; set; }

    public string Cidade { get; set; }

    public string Foto { get; set; }
}
```

4. Adicione uma View para o método **DestinoNovo**. Essa View deve usar o modelo **Destino**. Para isso, dentro do método **DestinoNovo**, escolha, com o botão secundário, **Add View**:

A janela do assistente aparecerá. Ela deverá ser configurada da seguinte forma:

- View name: **DestinoNovo**

É o nome da view. O arquivo será **/Views/Admin/DestinoNovo.cshtml**.

- Template: **Create**

O Visual Studio criará um modelo de formulário vazio. Algumas modificações serão necessárias porque uma imagem será enviada.

- Model class: **Destino**

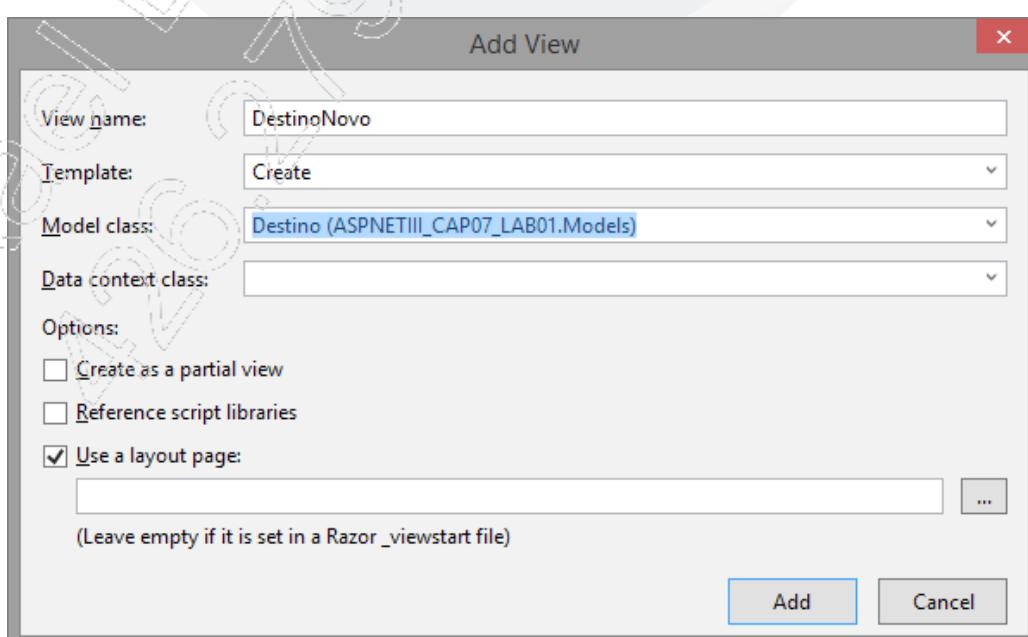
É o modelo usado na View. As propriedades dessa classe serão transformadas em legendas e caixas de texto.

- Data context class: **vazio**

Não usaremos o assistente para o banco de dados. Este campo ficará em branco.

- **Use a layout page**

Deverá ficar marcado e o nome do arquivo em branco. O arquivo de layout será obtido do arquivo **_ViewStart.cshtml**.



A página gerada pelo assistente cria grande parte do código HTML e Razor necessária para um formulário de cadastro.

No caso específico deste formulário, algumas alterações são necessárias: um arquivo de imagem deve ser transferido (**Upload**) para o servidor, o texto do botão deve ser alterado de **Create** para **Gravar**, e o link de retorno deve exibir o texto **Voltar**.

5. Modifique o código gerado em alguns pontos: O formulário deverá enviar uma foto, portanto o form gerado deverá ter o atributo **enctype** definido para **multipart/form-data**;

/Views/Admin/DestinoNovo.cshtml

```
@model CAP07_LAB01.Models.Destino

@{
    ViewBag.Title = "DestinoNovo";
}



## Novo Destino

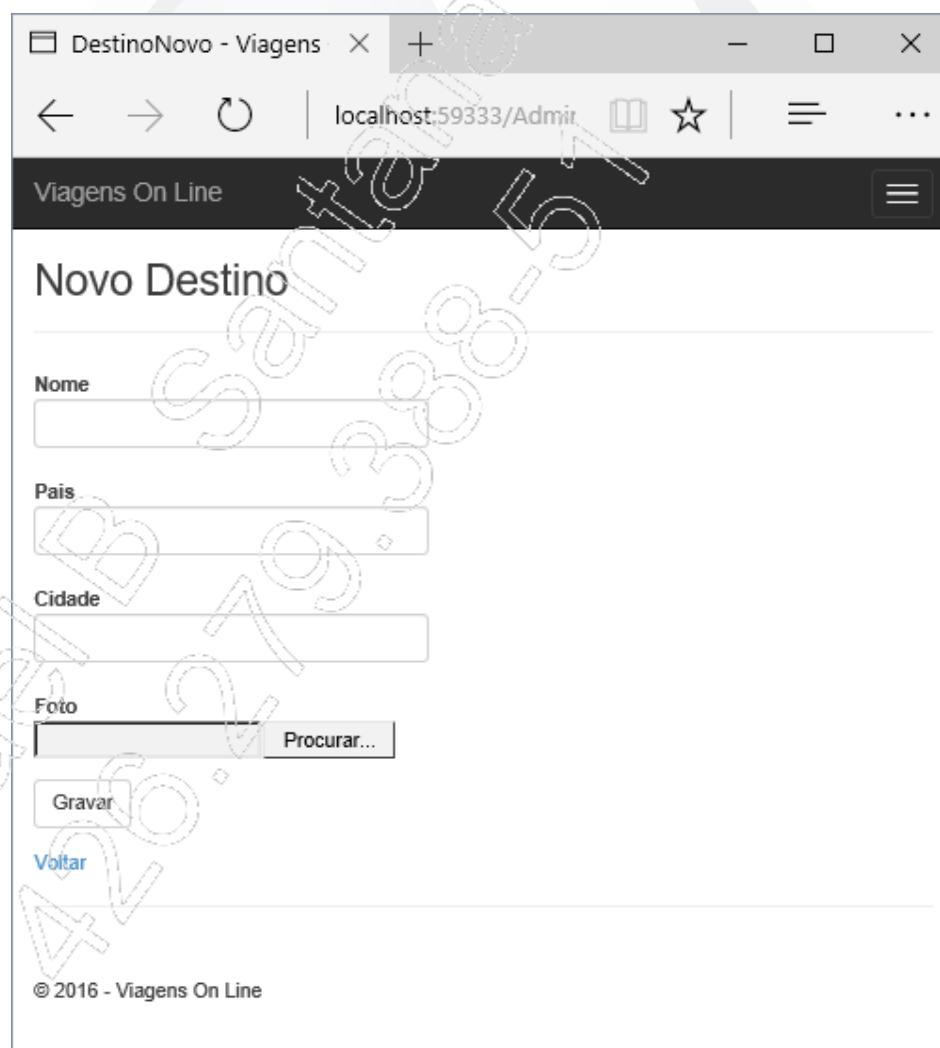


@using (Html.BeginForm("DestinoNovo", "Admin", FormMethod.Post, new { enctype = "multipart/form-data" }))
{
    <div class="form-horizontal">
        ... Controles Nome, País e Cidade não precisa alterar ....
        ... A foto deve ser um controle upload (input type=file) ....
        <div class="form-group">
            @Html.LabelFor(model => model.Foto, ...
            <div class="col-md-10">
                <input type="file" name="Foto" />
            </div>
        </div>
}
```

```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Gravar" .../>
    </div>
</div>
}

<div>
    @Html.ActionLink("Voltar", "DestinoListagem")
</div>
```

6. Visualize a página:



Ao enviar o formulário, um POST e os controles de tela serão enviados. Antes de criar o método que vai receber o formulário, é necessário criar o banco de dados.

Usando o Entity Framework e o modo **Code First**, a estrutura do banco será criada automaticamente.

C – Implementando o acesso a dados

1. Antes de criar o banco de dados, marque as propriedades do modelo como campos obrigatórios. Isso faz que o banco de dados seja criado corretamente, com valores nulos não permitidos. O atributo **Required** está no namespace **System.ComponentModel.DataAnnotations**;

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Web;

namespace CAP07_LAB01.Models
{
    public class Destino
    {
        public int DestinoId { get; set; }

        [Required]
        public string Nome { get; set; }

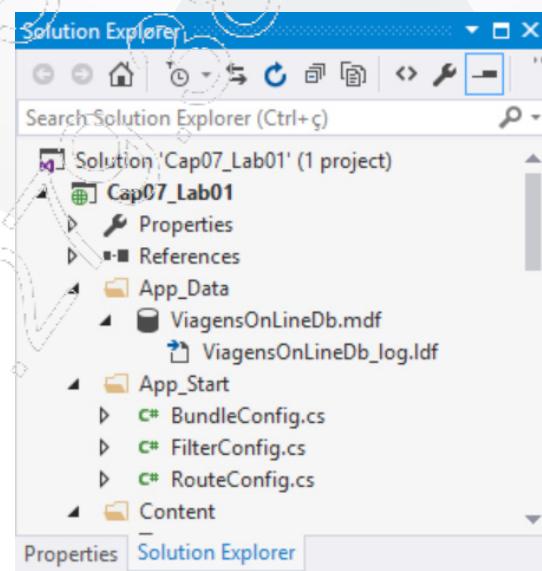
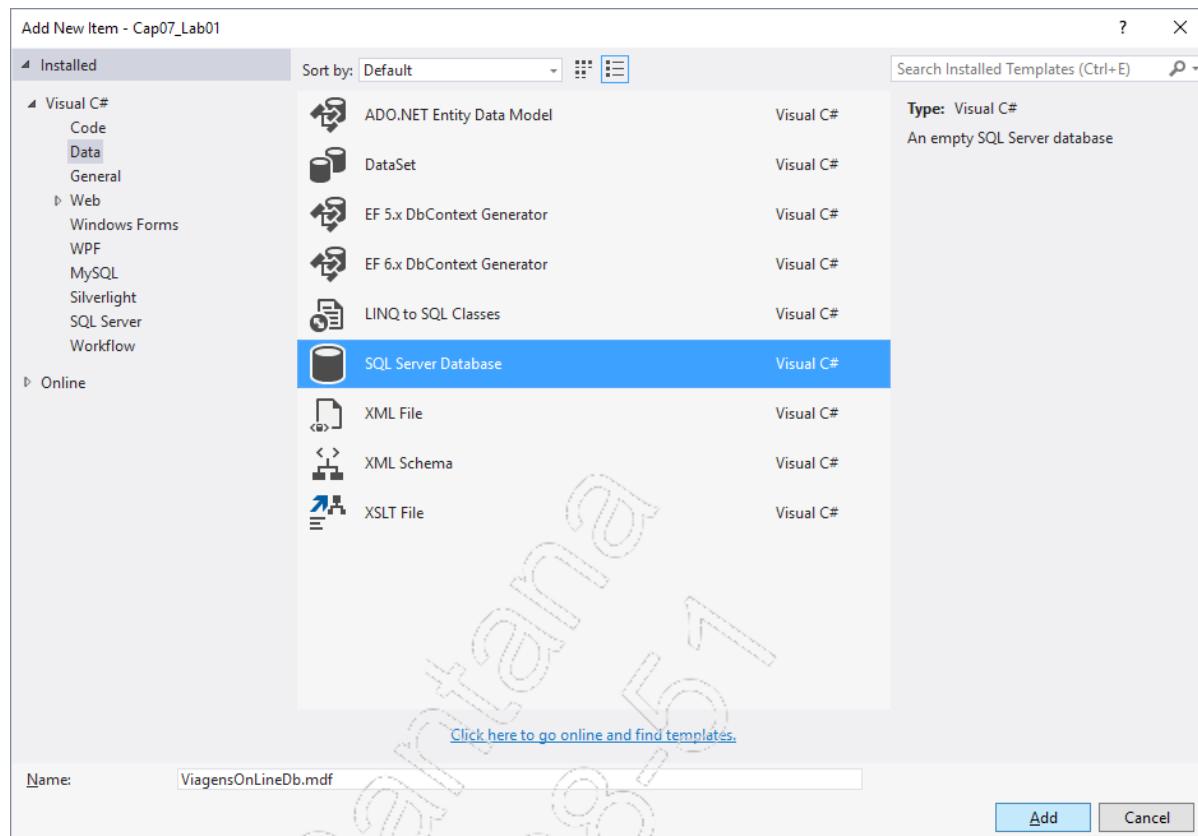
        [Required]
        public string Pais { get; set; }

        [Required]
        public string Cidade { get; set; }

        [Required]
        public string Foto { get; set; }
    }
}
```

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

2. Adicione um arquivo de banco de dados do SQL Server do projeto. Isso é feito por meio de **Add New Item**, escolhendo a opção **Data** e **SQL Server Database**. Defina o nome como **ViagensOnLineDb**. O arquivo será colocado na pasta **App_Data**;



3. Dê um duplo-clique no arquivo **App_Data/ViagensOnLine.mdf**. Isso abrirá o arquivo no Server Explorer. Vá até as propriedades do arquivo e copie a string de conexão. Ela será útil para o próximo passo;

4. A string de conexão deve ser algo como o mostrado a seguir:

```
Data Source=(LocalDB)\MSSQLLocalDb;  
AttachDbFilename=C:\caminho\App_Data\ViagensOnLineDb.mdf;  
Integrated Security=True;
```

A string pode mudar dependendo da versão do SQL Server, do local onde está o seu projeto e de como o SQL Server está instalado.

O importante no momento é que foi o Visual Studio que a criou quando fez a conexão com o banco de dados na sua máquina, portanto, é garantido que estará funcionando.

Existem várias alternativas para armazenar a string de conexão. Uma estratégia comum é armazená-la no Web.config. Dessa forma fica fácil alterar a string quando o aplicativo for publicado em ambiente de produção.

Para simplificar este projeto, a string será armazenada na própria classe de acesso a dados.

5. Crie uma pasta chamada **/Db**. Dentro dela, crie uma classe chamada **ViagensOnLineDb**:

/Db/ViagensOnLineDb.cs

```
using CAP07_LAB01.Models;  
using System;  
using System.Collections.Generic;  
using System.Data.Entity;  
using System.Linq;  
using System.Web;  
  
namespace CAP07_LAB01.Db  
{  
    public class ViagensOnLineDb : DbContext  
    {
```

```
private const string conexao =
    @"Data Source=(LocalDB)\v11.0;
AttachDbFilename=
    c:\caminho\App_Data\ViagensOnLineDb.mdf;
Integrated Security=True";

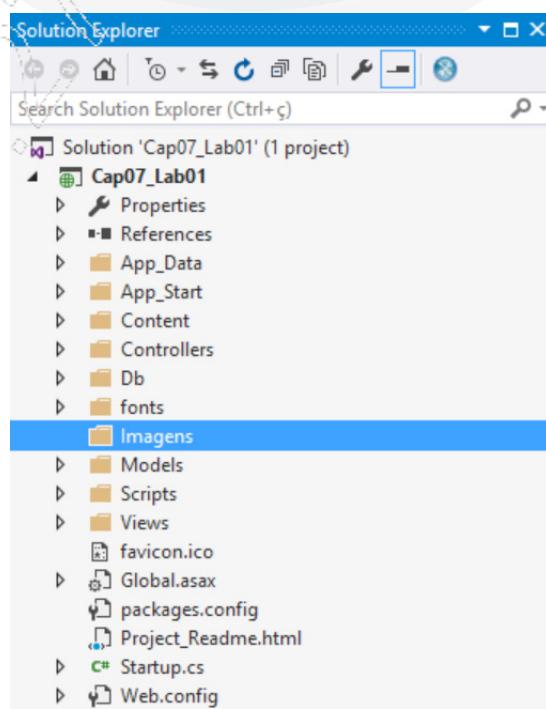
public ViagensOnLineDb()
    : base(conexao)
{ }

public DbSet<Destino> Destinos { get; set; }

}
```

A classe **ViagensOnLineDb** é uma típica classe do Entity Framework. A tabela **Destinos** é uma coleção de objetos do tipo **Destino**. O construtor passa a string de conexão para a classe **DbContext**. Lembre-se de que a string de conexão definida em **private const string** deve ser aquela copiada do Server Explorer. Com tudo preparado, já é possível fazer uma inclusão no banco de dados. A tabela será criada na primeira tentativa de uso.

6. Crie uma pasta chamada **/imagens**. As imagens que forem sendo enviadas para o servidor ficarão nessa pasta;



7. Na classe **AdminController**, crie o método que gravará a imagem enviada. O método **SaveAs** faz todo o trabalho. É necessário informar o caminho. A classe **Path** está no namespace **System.IO**. Faça o **using**, se necessário;

```
//  
// Gravar Foto  
  
private string GravarFoto(HttpRequestBase Request)  
{  
    string nome = Path.GetFileName(Request.Files[0].FileName);  
  
    string pastaVirtual = "~/imagens";  
  
    string pathVirtual = pastaVirtual + "/" + nome;  
  
    string pathFisico = Request.MapPath(pathVirtual);  
  
    Request.Files[0].SaveAs(pathFisico);  
  
    return nome;  
}
```

8. Crie um campo para armazenar o nome da View **DestinoListagem**. Em diversos momentos, a tela de listagem será chamada. Isso é feito por meio do método **RedirectToAction(string)**, portanto é melhor armazenar o valor dele em uma variável, para não ficar passando a string diversas vezes;

```
public class AdminController : Controller  
{  
    private const string ActionDestinoListagem =  
"DestinoListagem";  
    ...  
}
```

9. Crie outro método que retorne uma instância de **ViagensOnLineDb**. Nesse caso também, todos os métodos que acessam dados usam essa classe. Então o melhor é um método que retorne uma instância. Não é uma boa ideia criar uma instância de **dbContext** e deixá-la ativa na memória, principalmente porque o MVC trabalha muito com múltiplas threads e pode tornar-se difícil controlar o que está na memória. A melhor abordagem é sempre criar uma instância dentro de uma cláusula **using**:

```
//  
// Retorna uma Instância de DbContext  
  
private ViagensOnLineDb ObterDbContext()  
{  
    return new ViagensOnLineDb();  
}
```

10. Na classe **AdminController**, crie o método que grava um destino. É o mesmo método **NovoDestino** quando é recebido com um POST:

```
//  
// Gravar Novo Destino  
  
[HttpPost]  
public ActionResult DestinoNovo(Destino destino)  
{  
    //Se alguma validação falhou...  
    if (!ModelState.IsValid)  
    {  
        return View(destino);  
    }  
  
    // Foto é obrigatória  
    if (Request.Files.Count == 0 ||  
        Request.Files[0].ContentLength==0)  
    {
```

```
ModelState.AddModelError("",  
    "É necessário enviar uma Foto");  
return View(destino);  
}  
  
//Grava  
try  
{  
    //Grava a foto e retorna o nome  
destino.Foto = GravarFoto(Request);  
  
    using (var db = ObterDbContext())  
    {  
        db.Destinos.Add(destino);  
        db.SaveChanges();  
        return RedirectToAction("DestinoListagem");  
    }  
}  
catch (Exception ex)  
{  
    ModelState.AddModelError("", ex.Message);  
    return View(destino);  
}  
}
```

11. Antes de testar, crie a **View DestinoListagem**, apenas com o título para testar o método **DestinoNovo**:

```
/Views/Admin/DestinoListagem.cshtml  
  
<h2>Destino:Listagem</h2>
```

12. Teste a inclusão:

DestinoNovo - Viagens X + - □ ×

← → ⏪ | http://localhost:59333/ | ☰ ☆ | ⏹ ...

Viagens On Line ☰

Novo Destino

Nome

País

Cidade

Foto

[Voltar](#)

© 2016 - Viagens On Line

DestinoListagem - Viagens X + - □ ×

← → ⏪ | min/DestinoListagem | ☰ ☆ | ⏹ ...

Viagens On Line ☰

Destino: Listagem

© 2016 - Viagens On Line

13. Verifique se foi gravado no banco de dados e se a imagem foi gravada na pasta **imagens**:

The screenshot shows two windows side-by-side. The top window is 'dbo.Destinos [Data]' in SSMS, displaying a table with columns: Destinoid, Nome, País, Cidade, and Foto. One row is visible: Destinoid 1, Nome Andrômeda, País Brasil, Cidade Rio de Janeiro, and Foto showing a file path. The bottom window is the 'Solution Explorer' in Visual Studio, showing the project 'Cap07_Lab01' with its files: Properties, References, App_Data, App_Start, bin, Content, Controllers, Db, fonts, and Imagens. Inside the Imagens folder, three files are listed: 2_chadweisser_nynohpanoramic_bnw.jpg, 2_mayurkotlika_iniangrey/hornbill.jpg, and 2_nasaspacescapes_0006_7.jpg.

14. Crie o método **DestinoListagem** da classe **AdminController**:

```
//  
// Lista dos Destinos  
//  
public ActionResult DestinoListagem()  
{  
    List<Destino> lista = null;  
    using (var db = ObterDbContext())  
    {  
        lista = db.Destinos.ToList();  
    }  
  
    return View(lista);  
}
```

15. Crie a View DestinoListagem (pode apagar a anterior ou completá-la):

```
/Views/Admin/DestinoListagem.cshtml
@model IEnumerable<CAP07_LAB01.Models.Destino>

{@{ ViewBag.Title = "DestinoListagem"; }

<h2>Destino:Listagem</h2>

<p>@Html.ActionLink("Novo Destino", "DestinoNovo")</p>

<table class="table">
    <tr>
        <th></th>
        <th>@Html.DisplayNameFor(model => model.Nome)</th>
        <th>@Html.DisplayNameFor(model => model.Pais)</th>
        <th>@Html.DisplayNameFor(model => model.Cidade)</th>
    <th></th>
    </tr>

    @foreach (var item in Model) {
        <tr>
            <td></td>
            <td>@Html.DisplayFor(modelItem => item.Nome)</td>
            <td>@Html.DisplayFor(modelItem => item.Pais)</td>
            <td>@Html.DisplayFor(modelItem => item.Cidade)</td>
            <td>
```

```
    @Html.ActionLink("Alterar", "DestinoAlterar",
        new { id=item.DestinoId }) |
    @Html.ActionLink("Excluir", "DestinoExcluir",
        new { id=item.DestinoId })
</td>
</tr>
}
</table>
```

Visualize a listagem pela URL **Admin/DestinoListagem** (seria interessante incluir alguns itens):

Nome	País	Cidade	Ações
Fazenda Green Park	Holanda	Amsterdã	Alterar Excluir
Jazz Down Town	Estados Unidos	Diversas	Alterar Excluir
Safari	Africa do Sul	Twaniouaca	Alterar Excluir
Ice Nice	Antartida	Polo Sul	Alterar Excluir
Manhattan	Estados Unidos	Nova York	Alterar Excluir
Discover Adventures	Estados Unidos	Seattle	Alterar Excluir
Praia das Toninhas	Brasil	Ubatuba	Alterar Excluir
Dragon Year	Japão	Tokio	Alterar Excluir
Montanhas de Inverno	Japão	Shiga	Alterar Excluir
Praia	Brasil	Salvador	Alterar Excluir

© 2016 - Viagens On Line

16. Na classe AdminController, crie o método DestinoAlterar:

```
[HttpGet]  
public ActionResult DestinoAlterar(int id)  
{  
    using (var db = ObterDbContext())  
    {  
        var destino = db.Destinos.Find(id);  
        if (destino != null) { return View(destino); }  
    }  
    return RedirectToAction(ActionDestinoListagem);  
}
```

17. Crie o método DestinoAlterar com POST:

```
[HttpPost]  
public ActionResult DestinoAlterar(Destino destino)  
{  
    //Se o modelo é válido..  
    if (ModelState.IsValid)  
    {  
        using (var db = ObterDbContext())  
        {  
            //Obtém o original  
            var destinoOriginal = db.Destinos.Find(destino.  
                DestinoId);  
  
            //Se encontrou, altera o original  
            if (destinoOriginal != null) {  
                destinoOriginal.Nome = destino.Nome;  
                destinoOriginal.Cidade = destino.Cidade;  
                destinoOriginal.Pais = destino.Pais;  
  
                //Altera a imagem apenas se enviou outra  
                if (Request.Files.Count > 0 &&  
                    Request.Files[0].ContentLength > 0) {  
                    destinoOriginal.Foto = GravarFoto(Request);  
                }  
            }  
        }  
    }  
    return RedirectToAction(ActionDestinoListagem);  
}
```

```

    //Grava
    db.SaveChanges();
    return RedirectToAction(ActionDestinoListagem);
}
}

}

//Se chegou aqui e não foi redirecionado, é porque
// houve algum problema
return View(destino);
}

```

18. Crie a View **DestinoAlterar**. É praticamente igual à tela de inclusão:

```

@model Cap07_Lab01.Models.Destino

@{ ViewBag.Title = "DestinoAlterar"; }

<h2>Destino:Alterar</h2>

@using (Html.BeginForm("DestinoAlterar", "Admin", FormMethod.Post, new { enctype = "multipart/form-data" }))
{
    <div class="form-horizontal">
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        @Html.HiddenFor(model => model.DestinoId)
        <div class="form-group">@Html.LabelFor(model => model.Nome, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-10">@Html.EditorFor(model => model.Nome, new { htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Nome, "", new { @class = "text-danger" })</div></div>
        <div class="form-group">@Html.LabelFor(model => model.Pais, htmlAttributes: new { @class = "control-label col-md-2" })

```

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

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

<input type="hidden" name="Foto" value="@Model.Foto" />
<div class="imagemForm"><img src("~/imagens/@Model.Foto" class="imagemForm" /></div>

<div class="form-group">@Html.LabelFor(model => model.Foto, htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10"><input type="file" name="Foto" /></div>
    </div>
    <div class="form-group"><div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Gravar" class="btn btn-default" /></div></div></div>
}

<div>@Html.ActionLink("Voltar", "DestinoListagem")</div>
```

19. Teste a alteração: na listagem, clique em Alterar, que deve trazer os dados. Não é obrigatório enviar uma imagem. Se não enviá-la, os dados da imagem não são alterados;



20. O método de exclusão exibe os dados e pergunta se deseja excluir:

```
// Confirmar antes de excluir
//
[HttpGet]
public ActionResult DestinoExcluir(int id)
{
    using (var db = ObterDbContext())
    {
```

```
    var destino = db.Destinos.Find(id);
    if (destino != null)
    {
        return View(destino);
    }
}
return RedirectToAction(ActionDestinoListagem);
}

// Excluir
[HttpPost]
public ActionResult DestinoExcluir(int id,
                                    FormCollection form)
{
    using (var db = ObterDbContext())
    {
        var destino = db.Destinos.Find(id);
        if (destino != null)
        {
            db.Destinos.Remove(destino);
            db.SaveChanges();
        }
    }
    return RedirectToAction(ActionDestinoListagem);
}
```

21. Crie a View de Exclusão:

```
@model Cap07_Lab01.Models.Destino

@{
    ViewBag.Title = "DestinoExcluir";
}

<h2>Excluir Destino</h2>
```

```
<h3>Deseja excluir este registro?</h3>
<div>
    <hr />
    <dl class="dl-horizontal">
        <dt>@Html.DisplayNameFor(model => model.Nome) </dt>
        <dd>@Html.DisplayFor(model => model.Nome) </dd>

        <dt>@Html.DisplayNameFor(model => model.Pais) </dt>
        <dd>@Html.DisplayFor(model => model.Pais) </dd>

        <dt>@Html.DisplayNameFor(model => model.Cidade) </dt>
        <dd>@Html.DisplayFor(model => model.Cidade) </dd>
    </dl>
    <div class="imagemForm">
        
    </div>
    @using (Html.BeginForm())
    {
        <div class="form-actions no-color">
            <input type="submit" value="Excluir"
                class="btn btn-default" /> |
            @Html.ActionLink("Voltar", "DestinoListagem")
        </div>
    }
</div>
```

22. Teste a exclusão:



23. Agora na classe **ViagensOnLineController**, crie o método **Destinos** (que é uma listagem):

```
//  
// Destinos  
//  
public ActionResult Destinos()  
{  
    using(var db=new ViagensOnLineDb())  
    {  
        return View(db.Destinos.ToArray());  
    }  
}
```

24. A view **Destinos** utiliza alguns estilos. Inclua esses estilos na folha de estilo:

```
.destinos-box {  
    float: left;  
    margin-right: 20px;  
    margin-bottom: 20px;  
    margin-top: 30px;  
}  
.destinos-box img{  
    width:250px;  
    border-radius:5px  
}  
  
.destinosNome {  
    font-size:18px;  
    font-weight:bold  
}  
  
.destinosCidadePaís {  
    font-size:10px  
}
```

25. Crie a View Destinos:

```
@model IEnumerable<CAP07_LAB01.Models.Destino>

{@{
    ViewBag.Title = "Destinos";
}

<h2>Destinos</h2>

@foreach (var item in Model)
{
    <div class="destinos-box" >

        <div>
            
        </div>

        <div class="destinosNome">
            @Html.DisplayFor(modelItem => item.Nome)
        </div>

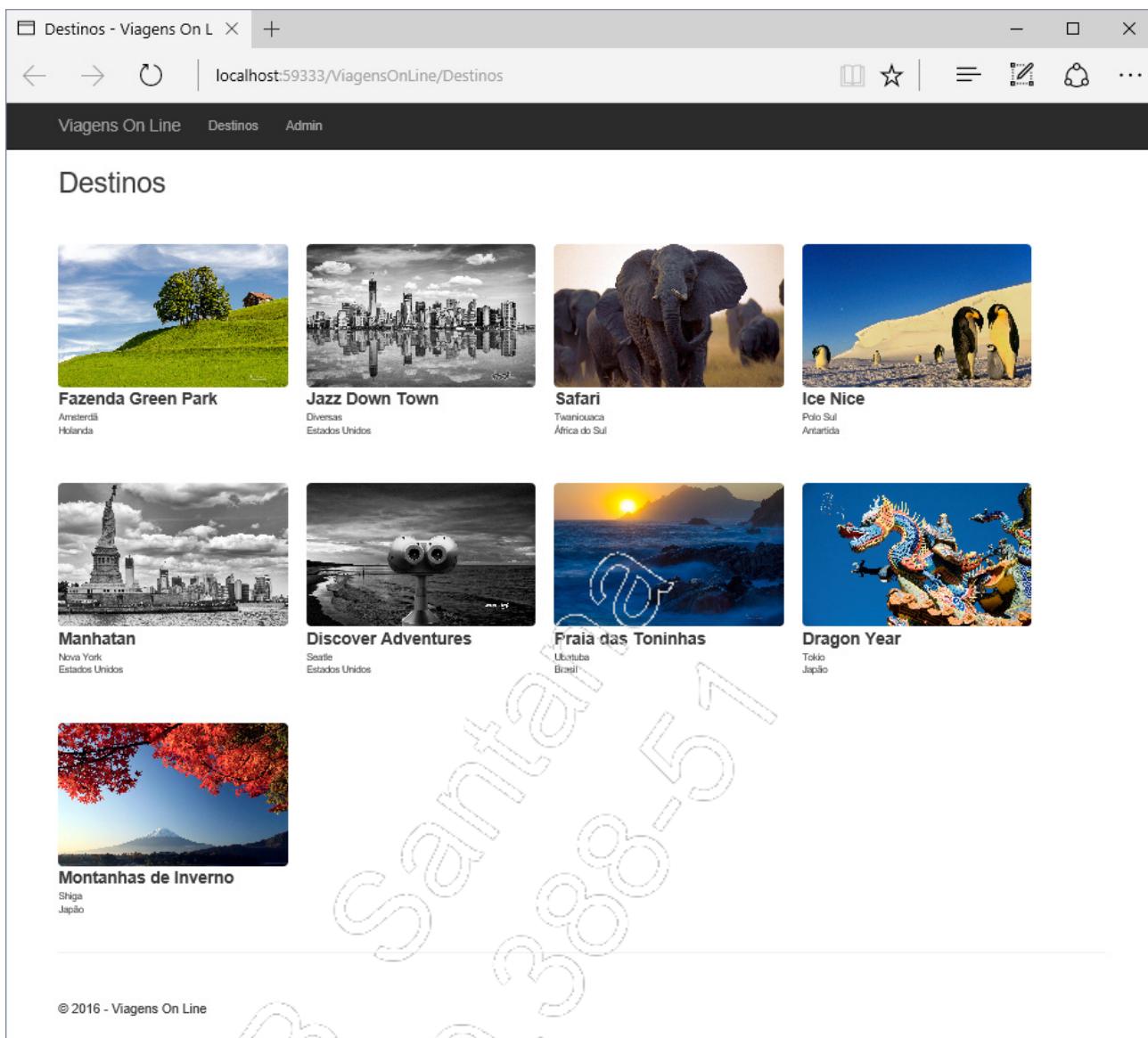
        <div class="destinosCidadePais">
            @Html.DisplayFor(modelItem => item.Cidade)
        </div>

        <div class="destinosCidadePais">
            @Html.DisplayFor(modelItem => item.Pais)
        </div>

    </div>
}

<div style="clear:both"></div>
```

26. Teste a View Destinos, por meio da URL ViagensOnLine/Destinos:



D – Implementando a autenticação com OWIN

O usuário administrador é o único que pode manipular o banco de dados. A seguir, implementaremos a autenticação OWIN:

1. Neste arquivo, será definida a autenticação por cookies. É definido um nome para essa autenticação e um local de redirecionamento:

```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;
using Microsoft.Owin.Security.Cookies;
```

```
[assembly: OwinStartup(typeof(Cap07_Lab01.Startup))]

namespace Cap07_Lab01
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.UseCookieAuthentication(
                new CookieAuthenticationOptions()
                {
                    AuthenticationType = "AppViagensOnLineCookie",
                    LoginPath = new PathString("/Admin/Login")
                });
        }
    }
}
```

2. Crie o método **Login** na classe **AdminController**. Repare no atributo **AllowAnonymous**, que declara que usuários não autenticados podem utilizar esse método;

```
// 
// Login
//
[AllowAnonymous]
public ActionResult Login()
{
    return View();
}
```

3. Crie a View Login:

```
/Views/Admin/Login.cshtml
{@{
    ViewBag.Title = "Login";
}

<h2>Login</h2>
<br/><br />

@if (ViewBag.Mensagem != null)
{   <div class="alerta"> @ViewBag.Mensagem</div>   }

@using (Html.BeginForm("Login", "Admin", "Post"))
{
    <div class="form-group">
        @Html.Label("Nome:") <br /> @Html.TextBox("Nome")
    </div>

    <div class="form-group">
        @Html.Label("Senha:") <br />
        @Html.TextBox("Senha")
    </div>

    <div class="form-group">
        <input type="submit" value="Enviar" />
    </div>
}
```

4. Crie o método que recebe os dados de login e autentica o usuário. Nesse caso, o usuário deve se chamar **admin** e a senha **admin**. Em uma situação real, o correto é obter o usuário de um repositório de dados para validar;

```
[AllowAnonymous]
[HttpPost]
public ActionResult Login(string nome, string senha)
{
    if (string.IsNullOrEmpty(nome) )
    {
        ViewBag.Mensagem="Digite o nome";
        return View();
    }

    if (string.IsNullOrEmpty(senha))
    {
        ViewBag.Mensagem = "Digite o senha";
        return View();
    }

    if (nome != "admin" && senha != "admin")
    {
        ViewBag.Mensagem = "Usuário ou senha inválida";
        return View();
    }

    //Um Array de claims. Claim é uma declaração que
    //o usuário faz. Nesse caso, são duas: Ele se chama
    // Administrador e pertence ao grupo admin
    Claim[] claims= new Claim[2];
    claims[0] = new Claim(ClaimTypes.Name, "Administrador");
    claims[1] = new Claim(ClaimTypes.Role, "admin");

    //Nome para identificar
    string nomeAutenticacao = "AppViagensOnLineCookie";
```

```
//Identidade
ClaimsIdentity identity =
    new ClaimsIdentity(claims, nomeAutenticacao);

//Autentica
Request.GetOwinContext().Authentication.SignIn(identity);

//Redireciona para a pasta destinos
return RedirectToAction(ActionDestinoListagem);
}
```

5. A última parte é declarar que os métodos da classe **AdminController** só podem ser utilizados por um usuário autenticado. Isso é feito pelo atributo **Autorize** colocado na classe:

```
[Authorize]
public class AdminController : Controller
{
    ....
}
```

6. Teste essa última parte: Quando o usuário tenta entrar em **admin/DestinoListagem**, o componente OWIN redireciona para a tela de login, até que seja autenticado.



Login - Viagens On Line X + ← → ⌂ localhost:59333/Adminr ☰ ☆ ⌚ ...

Viagens On Line ☰

Login

Nome:

Senha:

© 2016 - Viagens On Line

DestinoListagem - Viage X + ← → ⌂ localhost:59333/Admin/DestinoListagem ☰ ☆ ⌚ ...

Viagens On Line Destinos Admin

Destino:Listagem

[Novo Destino](#)

Nome	País	Cidade	Ações
Fazenda Green Park	Holanda	Amsterdã	Alterar Excluir
Jazz Down Town	Estados Unidos	Diversas	Alterar Excluir
Safari	África do Sul	Twaniouaca	Alterar Excluir
Ice Nice	Antartida	Polo Sul	Alterar Excluir
Manhattan	Estados Unidos	Nova York	Alterar Excluir
Discover Adventures	Estados Unidos	Seattle	Alterar Excluir
Praia das Toninhas	Brasil	Ubatuba	Alterar Excluir
Dragon Year	Japão	Tokio	Alterar Excluir
Montanhas de Inverno	Japão	Shiga	Alterar Excluir

© 2016 - Viagens On Line

8

Single Page App

- ✓ Introdução a Knockout, MVVM e Observer;
- ✓ O modelo SPA;
- ✓ Executando o modelo SPA.

8.1. Introdução

Single Page Application (SPA), ou aplicação em uma única página, é uma expressão usada para descrever um tipo de aplicação Web que carrega uma página HTML uma única vez e utiliza AJAX ou outra tecnologia para atualizar o conteúdo, sem (aparentemente) realizar outra requisição para o servidor.

Esse tipo de aplicação não é novidade. Desde o início do desenvolvimento de aplicações para Web, os programadores encontravam formas de atualizar dinamicamente as páginas HTML de um aplicativo. A explosão do AJAX contribuiu para o aparecimento de bibliotecas JavaScript e frameworks que implementavam padrões de codificação pouco comuns para os programadores da época, principalmente para aqueles que vieram do universo Windows e tinham pouco ou nenhum contato com design patterns como Observer, Prototype ou Repository.

O Visual Studio disponibiliza um modelo para criação de aplicações SPA que utiliza os seguintes componentes:

- Knockout.js;
- Web API.

Knockout é uma biblioteca open source JavaScript para criar interfaces. Unindo a biblioteca Knockout com os serviços da Web API, o Visual Studio fornece uma robusta infraestrutura para criar aplicações SPA, que utilizam padrões conhecidos do mercado e ferramentas maduras, conhecidas e consolidadas.

O template Knockout vem, por padrão, com o Visual Studio, mas outros templates estão disponíveis para outras bibliotecas e frameworks. Alguns muito utilizados são os seguintes:



BACKBONEJS



Single Page Apps Done Right



ANGULARJS
by Google



Se o programador já tem experiência com alguma biblioteca dessas, pode fazer o download do template no site www.asp.net e utilizar a biblioteca com que tem mais familiaridade.

Com a versão ASP.NET Core, um grande número de componentes open source passou a fazer parte dos modelos de projetos. Entre eles, os seguintes:



Para entender o template Single Page Application do Visual Studio, é necessário conhecer o básico do framework Knockout, a arquitetura MVVM (Model-View-ViewModel), o conceito de design patterns e o padrão Observer.

8.2. Introdução a Knockout, MVVM e Observer

Knockout é uma biblioteca que simplifica a criação de interfaces usando JavaScript. É open source e está constantemente sendo atualizada pela comunidade de suporte.

O princípio da biblioteca Knockout é criar declarações nas tags HTML definindo uma fonte de dados. Quando a fonte de dados muda, o controle na interface muda automaticamente, e o contrário também é verdadeiro: quando o controle na interface é atualizado, o dado relacionado é atualizado automaticamente.

Esse modelo de programação em que os objetos são monitorados é chamado, apropriadamente, de **Observer**. E essa arquitetura em que existe um modelo de dados (Model), um visualizador (View) e um programa que coordena o tráfego de informações (ViewModel) é chamada **MVVM (Model-View-ViewModel)**.

8.2.1. MVVM (Model-View-ViewModel)

Model-View-ViewModel é um modelo usado para criar a interface de usuário e consiste em dividir a construção da interface em três partes:

- **Model**

Model é todo o conjunto de objetos que define e armazena os dados do aplicativo, independente da interface. Os objetos que manipulam o Model geralmente ficam em um servidor e são manipulados por componentes instalados nesse servidor.

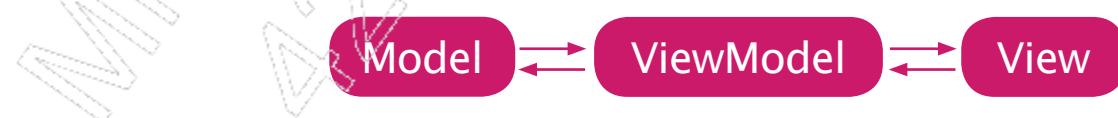
- **ViewModel**

É uma representação do Model ou de parte dele na interface. Essa representação não tem nenhuma ligação com relacionamentos, chaves ou bancos de dados. São apenas classes que armazenam e manipulam os dados na tela, da maneira como serão vistos, e não da maneira como são gravados. O **ViewModel** também não tem nenhuma informação da tela (HTML, XAML), de modo que a tecnologia de visualização pode ser substituída sem afetar o sistema.

- **View**

View é a parte visível da aplicação que exibe os dados da ViewModel. É responsável por atualizar a tela, enviar e receber informações da ViewModel. Não tem nenhum contato com o **Model** diretamente.

Do ponto de vista da biblioteca Knockout, uma ViewModel é uma classe em JavaScript que representa um modelo (vindo de Model) e uma View é uma página HTML.



8.2.2. Biblioteca Knockout

Para iniciar o uso do Knockout, é necessário fazer o download da biblioteca e colocar uma referência na página HTML:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <script src="knockout-3.2.0.js"></script>
</head>
<body>

</body>
</html>
```

O ViewModel é uma classe em JavaScript. No exemplo adiante, foi criado um objeto chamado **exemploViewModel** com dois campos: **Nome** e **Idade**. O JavaScript tem essa facilidade embutida, que é a de criar objetos de classes anônimas (tipo **Object**) simplesmente declarando seus campos e valores com a mesma sintaxe de um objeto JSON:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <script src="knockout-3.2.0.js"></script>
    <script type="text/javascript">
        var exemploViewModel={ Nome:'Maria', Idade:20 };
    </script>
</head> ...
```

O próximo passo é vincular na View (HTML) os dados da ViewModel. Para isso, o Knockout utiliza o recurso **declarative bindings**, que consiste em definir atributos em tags HTML, identificando os campos.

O atributo para vincular uma propriedade de um objeto da View ao Model se chama **data-bind** e consiste em duas partes, separadas por dois-pontos: o nome da propriedade e o nome do campo que será vinculado.

No exemplo a seguir, a propriedade **text** do objeto **span** será preenchida pelo valor contido no campo **Nome** do objeto declarado no **ViewModel**:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <script src="knockout-3.2.0.js"></script>
    <script type="text/javascript">
        var exemploViewModel={ Nome:'Maria', Idade:20 };
    </script>
</head>
<body>
    Nome: <span data-bind="text: Nome"></span> <br/>
    Idade: <span data-bind="text: Idade"></span><br/>
</body>
</html>
```

O último passo é ativar o vínculo entre o objeto **exemploViewModel** e a tag **span** que contém o atributo **data-bind**. Isso deve ser feito depois que as tags forem carregadas na memória, portanto, serão colocadas depois das declarações:

```
<html>
<head>
    <script src="knockout-3.2.0.js"></script>
    <script type="text/javascript">
        var exemploViewModel={ Nome:'Maria', Idade:20 };
    </script>
</head>
<body>
```

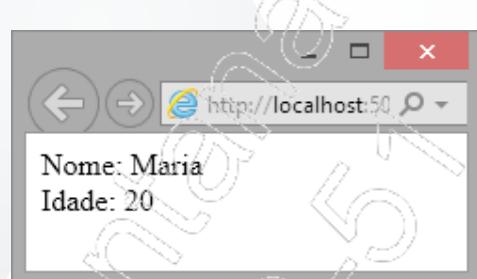
Nome:

Idade:


```
<script type="text/javascript">
    ko.applyBindings(exemploViewModel);
</script>
```

```
</body>
</html>
```

O objeto **ko** é criado pela biblioteca **Knockout**. O método **applyBindings** carrega os valores definidos no objeto passado como parâmetro nos lugares declarados na página:



8.2.3. Observable

O exemplo anterior ainda não mostra o objetivo principal da biblioteca, que é criar objetos que automaticamente atualizem os dados. Do jeito que foi feito, será necessário chamar o método **applyBindings** toda vez que os dados forem alterados. Para que a atualização seja automática, a solução é criar um **observable**.

O exemplo adiante cria o tipo de campo **observable**, um tipo especial do JavaScript que notifica quando há mudanças. Automaticamente, todos os objetos vinculados são alterados:

```
var exemploViewModel = {
    Produto: ko.observable('Notebook'),
    Preco: ko.observable(2000)
};
```

Para visualizar, é interessante vincular uma variável a um controle **input** para digitar uma alteração e visualizar a mudança:

```
<input data-bind="value: Produto" />
```

No caso do controle **input**, a propriedade a ser vinculada é a propriedade **value**. Veja o exemplo completo a seguir:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <script src="knockout-3.2.0.js"></script>
</head>
<body>

    Nome:<br/>
    <input data-bind="value: Produto" />
    <br/><br/>

    Preco:<br/>
    <input data-bind="value: Preco" />
    <br/><br/>

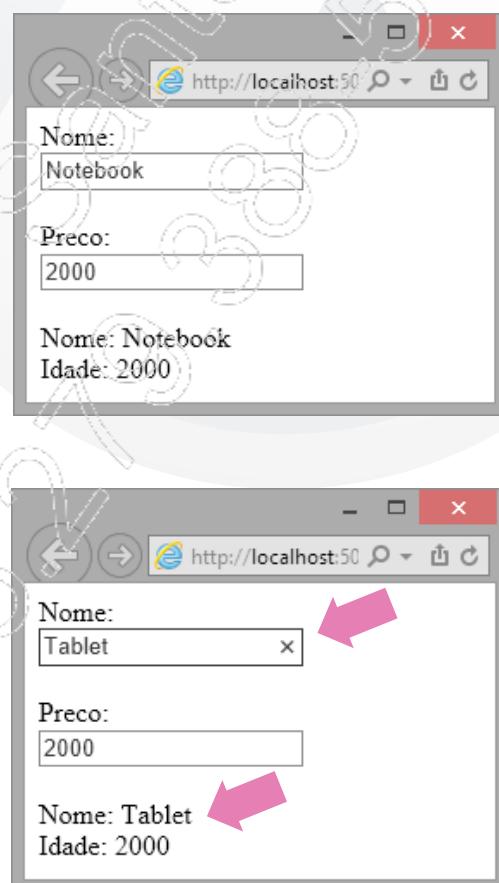
    Nome:
    <span data-bind="text: Produto"></span>
    <br/>

    Idade:
    <span data-bind="text: Preco"></span>
    <br/>

    <script type="text/javascript">
```

```
var exemploViewModel = {  
    Produto: ko.observable('Notebook'),  
    Preco: ko.observable(2000)  
};  
  
ko.applyBindings(exemploViewModel);  
  
</script>  
  
</body>  
  
</html>
```

Ao alterar o campo **Nome**, no TextBox, o **span** também se altera, porque o campo **Nome** está sendo "observado" pelo programa JavaScript, que notifica quando há mudanças no modelo:



8.2.4. Observable Array

Em uma interface, existem basicamente dois tipos de objetos: itens e coleções de itens. As coleções geralmente apresentam alguns desafios, porque necessitam de alguns "cuidados especiais" para funcionar corretamente, como controlar o item atual, notificar itens excluídos, exibir dados paginados, formatar colunas e linhas etc. Usando os recursos de vinculação do Knockout, essas tarefas são simplificadas, porque as responsabilidades de exibir e manter os dados são separadas.

Para exibir dados de uma coleção, é usado o vínculo **foreach**:

```
<ul data-bind="foreach:meuArray">
```

Cada elemento do array, se tiver propriedades, pode ser acessado diretamente:

```
<span data-bind="text: Nome"></span>
<span data-bind="text: Telefone"></span>
```

Se for um tipo primitivo como string, pode ser obtido por meio do operador **\$** e da variável **data**. A informação contida na variável **\$data** é uma propriedade de um objeto chamado **Binding Context**:

```
<span data-bind="text: $data"></span>
```

Os modelos podem ser aninhados. É possível obter uma referência ao **context** imediatamente anterior, como a propriedade **\$parent**. O exemplo adiante exibe como fica a sintaxe quando a referência é feita a um item atual e a um item anterior:

Produto:

Categoria:

A seguir, veja um exemplo simples de vínculo, usando um array apenas:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <script src="knockout-3.2.0.js"></script>
</head>
<body>

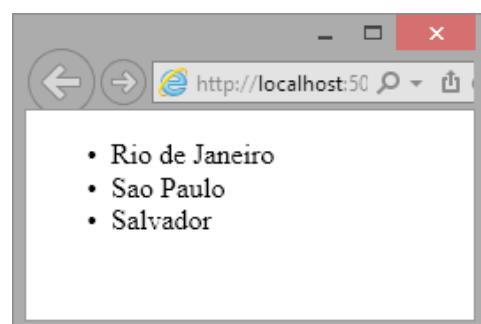
    <ul data-bind="foreach: exemploObservableArray">
        <li>
            <span data-bind="text: $data"></span>
        </li>
    </ul>

    <script type="text/javascript">

        var exemploObservableArray = ko.observableArray();
        exemploObservableArray.push("Rio de Janeiro");
        exemploObservableArray.push("Sao Paulo");
        exemploObservableArray.push("Salvador");

        ko.applyBindings(exemploObservableArray);

    </script>
</body>
</html>
```

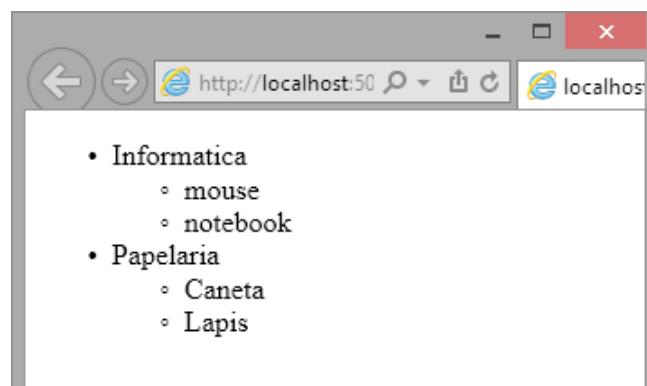


Agora, veja outro exemplo usando dados aninhados:

```
<html>
<head>
    <title></title>
    <script src="knockout-3.2.0.js"></script>
</head>
<body>
    <ul data-bind="foreach:estoque">
        <li>
            <span data-bind="text: categoria"></span>
            <ul data-bind="foreach: produtos">
                <li>
                    <span data-bind="text: $data"></span>
                </li>
            </ul>
        </li>
    </ul>
    <script type="text/javascript">
var estoque = ko.observableArray();
estoque.push({
    categoria: 'Informatica',
    produtos: ['mouse', 'notebook']
});

estoque.push({
    categoria: 'Papelaria',
    produtos: ['Caneta', 'Lapis']
});

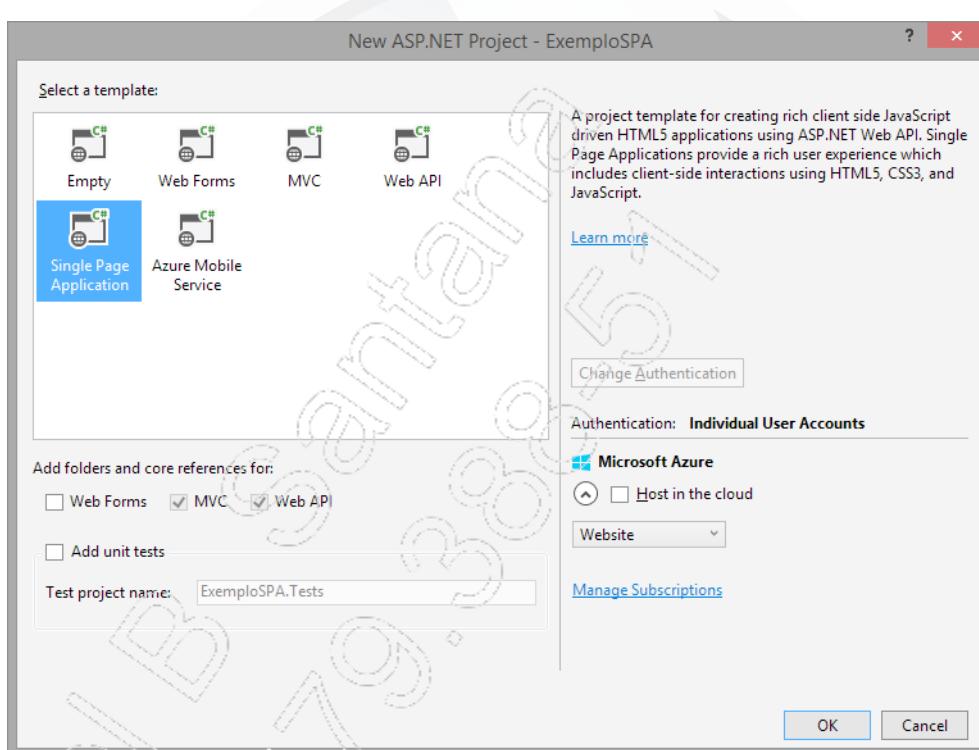
ko.applyBindings(estoque);
    </script>
</body>
</html>
```



8.3. O modelo SPA

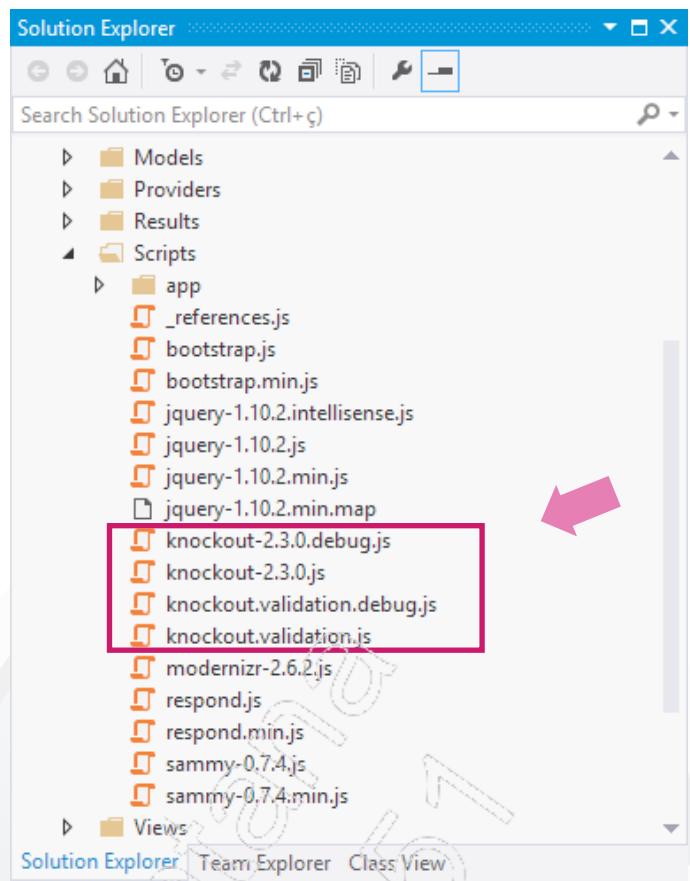
O modelo oferecido pelo Visual Studio como ponto de partida utiliza a biblioteca Knockout e o modelo MVVC como ponto central para manipulação de elementos de tela. Para atualizar o modelo, são utilizados a Web API e o Entity Framework.

Para iniciar um projeto Single Page Application, escolha **New Project / Web** e a opção **Single Page Application** na caixa de diálogo. Repare que não é possível desmarcar os vínculos a **MVC**, **Web API** e **Authentication User Accounts**. Esse modelo usa um campo adicional nos dados do usuário e exibe na tela inicial este valor:

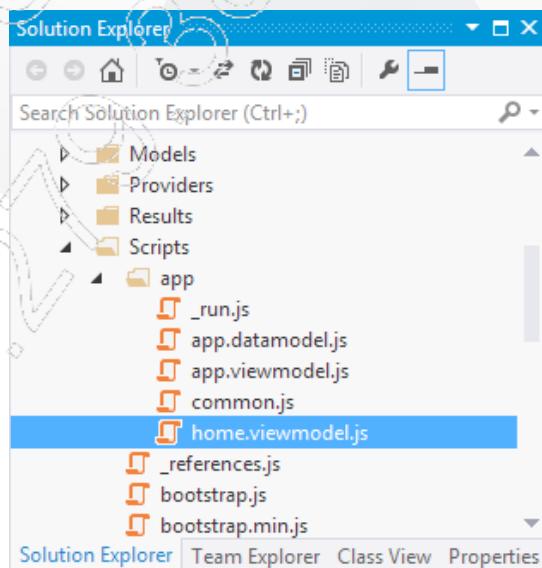


Visual Studio 2015 - ASP.NET com C# Recursos Avançados

A pasta **Scripts** faz referências à biblioteca Knockout:



Dentro da pasta **Scripts / app** se encontra o código JavaScript do aplicativo:



No arquivo **Scripts / app / home.viewmodel.js** está o código que cria o item **observable** usando a biblioteca Knockout:

```
function HomeViewModel(app, dataModel) {
    var self = this;
    self.myHometown = ko.observable("");
    Sammy(function () {
        this.get('#home', function () {
            $.ajax({
                method: 'get',
                url: app.dataModel.userInfoUrl,
                contentType: "application/json; charset=utf-8",
                headers: {
                    'Authorization': 'Bearer ' +
                        app.dataModel.getAccessToken() },
                success: function (data) {
                    self.myHometown('Your Hometown is : ' +
                        data.hometown); }
            });
        });
        this.get('/', function () {
            this.app
                .runRoute('get', '#home') });
    });
    return self;
}
app.addViewModel({
    name: "Home",
    bindingMemberName: "home",
    factory: HomeViewModel
});
```

Ainda dentro da pasta **Scripts / app**, o arquivo **_run.js** inicia o aplicativo:

```
$(function () {
    app.initialize();

    // Activate Knockout
    ko.validation.init({ grouping: { observable: false } });
    ko.applyBindings(app);
});
```

Na pasta **Views / home**, o arquivo Razor **_home.cshtml** contém o código HTML que vincula os dados (parte do HTML foi omitido):

```
<!-- ko with: home -->

<div class="jumbotron">
    <h1>ASP.NET</h1>
    <p class="lead">ASP.NET is a free web framework... </p>
    <p><a href="http://asp.net" ...</a></p>
</div>

<div class="row">
    <div class="col-md-4">
        <h2>Your information</h2>
        <p>This section shows how you...</p>
        <p data-bind="text: myHometown"></p>
        <p><a href="http://...">Learn more</a></p>
    </div>

    <div class="col-md-4">
        <h2>Getting started</h2>
        <p>ASP.NET Single Page Application (SPA)...
        </p>
        <p><a class="btn btn-default">Learn more &raquo;</a></p>
    </div>

    <div class="col-md-4">
        <h2>Web Hosting</h2>
        <p>You can easily find a</p>
        <p><a href="http://go">Learn more &raquo;</a></p>
    </div>

</div>

<!-- /ko -->
```

MyHometown é obtida após uma chamada AJAX ao controlador **MeController**:

```
success: function (data) {  
    self.myHometown('Your Hometown is : ' + data.hometown);  
}
```

Este é o método chamado pelo AJAX para obter o ViewModel com a propriedade **myHometown**. Esse método retorna o objeto do tipo **GetViewModel** e tem esse nome apropriado porque é uma chamada GET para obter a ViewModel:

```
// GET api/Me  
public GetViewModel Get()  
{  
    var user = UserManager.FindById(User.Identity.GetUserId());  
    return new GetViewModel() { Hometown = user.Hometown };  
}
```

A classe **GetViewModel** está definida em **Models** e contém apenas o campo retornado:

```
public class GetViewModel  
{  
    public string Hometown { get; set; }  
}
```

É importante lembrar que esse campo é obtido do usuário usando o ASP.NET Identity, pelo método **UserManager.FindById()** no método GET da Web API. A seguir, está a definição da classe de usuário:

```
public class ApplicationUser : IdentityUser  
{  
    public string Hometown { get; set; }  
  
    public async  
        Task<ClaimsIdentity>GenerateUserIdentityAsync(  
            UserManager<ApplicationUser> manager)  
    {
```

```
    var userIdentity =
        await manager.CreateIdentityAsync(
            this,
            DefaultAuthenticationTypes
                .ApplicationCookie);

    return userIdentity;
}
}
```

Esse campo também está presente quando o usuário se registra, por isso, está definido na classe utilizada para criar a tela de cadastro. Observe como essa classe está repleta de **Data Annotations** usada para criar o formulário:

```
public class RegisterViewModel
{
    [Required]
    [EmailAddress]
    [Display(Name = "Email")]
    public string Email { get; set; }

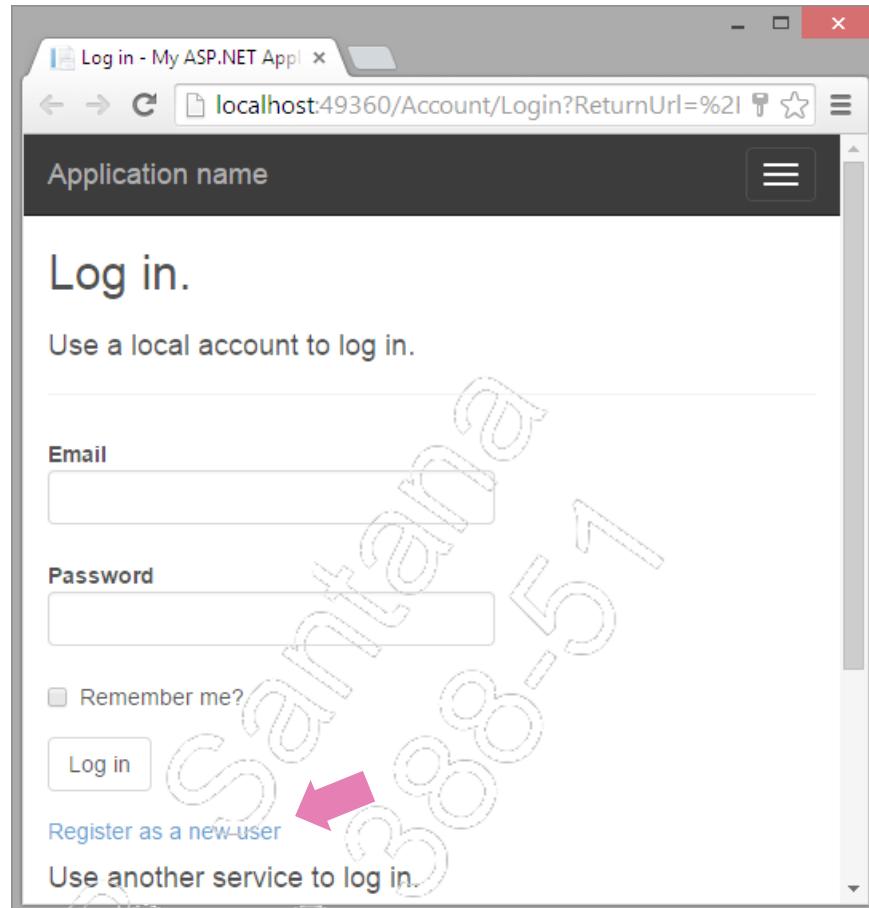
    [Required]
    [StringLength(100, ErrorMessage =
        "The {0} must be at least {2} characters long.",
        MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password",
        ErrorMessage = "The password and
        confirmation password do not match.")]
    public string ConfirmPassword { get; set; }

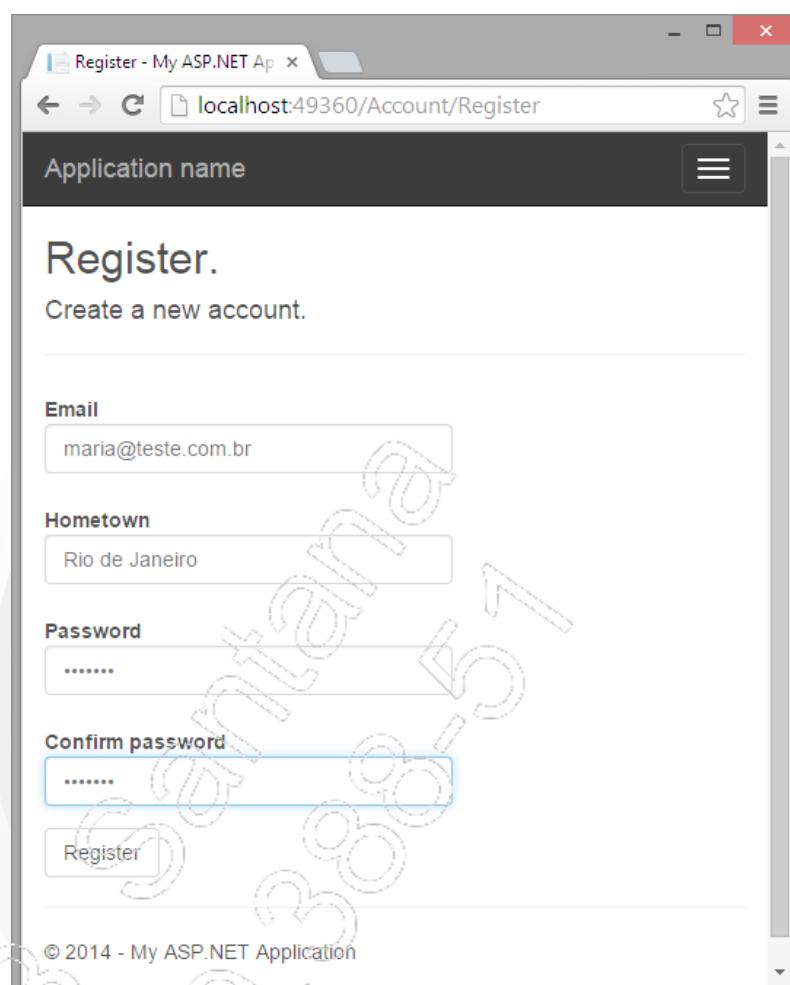
    [Display(Name = "Hometown")]
    public string Hometown { get; set; }
}
```

8.4. Executando o modelo SPA

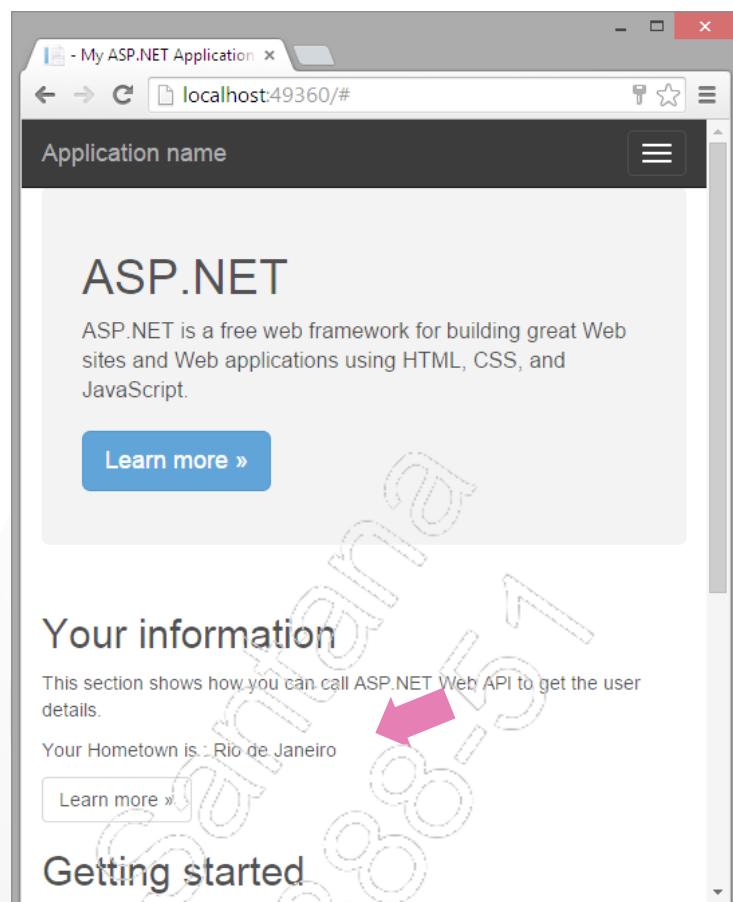
Para ver o SPA em funcionamento, o primeiro passo é registrar-se. O sistema não aceita usuários anônimos. Se nenhum login foi feito ainda, o sistema redireciona para a tela de login. É necessário clicar em **Register as a new user**:



O campo **Hometown** aparece para ser preenchido junto com as outras informações de usuário:



Os dados são gravados e a página é redirecionada para **Home**. Esta é a página onde o campo **Hometown** é exibido na tela inicial:



O modelo do Visual Studio é um ótimo ponto de partida com um robusto modelo para iniciar uma aplicação SPA utilizando serviços e componentes totalmente integrados.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- **SPA (Single Page Application)** é um tipo de programa que utiliza HTML, CSS e JavaScript para carregar uma única vez a página e, então, atualizar o conteúdo dinamicamente, sem realizar novas requisições para o servidor;
- **Knockout** é uma biblioteca JavaScript para facilitar a criação interfaces;
- **MVVC (Model-View-ViewModel)** é um modelo para criação de interfaces composto de três partes: os dados da aplicação (**Model**), a visualização (**View**) e um programa em JavaScript que cria um modelo para tela e controla as atualizações e comunicações com os servidores (**ViewModel**);
- **Observer** é um design pattern em que um objeto fornece mensagens quando o estado de uma informação muda, permitindo que os dados sejam atualizados automaticamente;
- A biblioteca Knockout utiliza o padrão Observer e o modelo MVVC;
- O modelo SPA disponibilizado pelo Visual Studio utiliza as bibliotecas JavaScript Knockout e jQuery, o framework Web API e AJAX para obter dados do servidor, MVC como infraestrutura geral do site, Bootstrap como gerenciador de estilos, ASP.NET Identity para gerenciamento de usuário e Entity Framework para gravar informações no SQL Server.

8

Single Page App

Teste seus conhecimentos

Mikael B
Santana
57
426.279.0000



IMPACTA
EDITORA

1. Qual é o papel do Model no modelo MVVM?

- a) Exibir os dados.
- b) Controlar os eventos da View.
- c) Armazenar os dados da aplicação.
- d) Armazenar os dados que vão aparecer na tela.
- e) Validar os dados.

2. Qual atributo HTML é utilizado para vincular dados na biblioteca Knockout?

- a) bind
- b) data
- c) databind
- d) data-bind
- e) bind-data

3. Qual é o nome do objeto principal da biblioteca Knockout?

- a) knockoutApp
- b) ko
- c) this
- d) self
- e) that

4. Qual método deve ser chamado para iniciar o processo de vínculo?

- a) initApp
- b) ApplInit
- c) applyBindings
- d) initBindings
- e) init

5. Qual o nome do modelo utilizado pela biblioteca Knockout para atualizar automaticamente os elementos de tela?

- a) MVC
- b) Bind
- c) Observable
- d) Handle
- e) View

8

Single Page App

Mãos à obra!

Mikael B
Santana
426.279.57



IMPACTA
EDITORA

Laboratório 1

A – Criando uma Single Page Application que registre uma lista de produtos

Neste laboratório, você irá criar uma **Single Page Application** com a função de registrar uma lista de produtos, bem como ler e gravar os dados do produto em XML. A comunicação com o servidor é feita por meio da Web API, já a comunicação com a Web API, por meio de jQuery e AJAX. A tela usará um **ViewModel** e os design patterns **Observer** e **MVVM**, utilizando a biblioteca **knockout.js**. Note que a página de interação com o usuário é totalmente em HTML.

As telas são as seguintes:

Ao carregar a tela inicial, uma lista de produtos aparece:

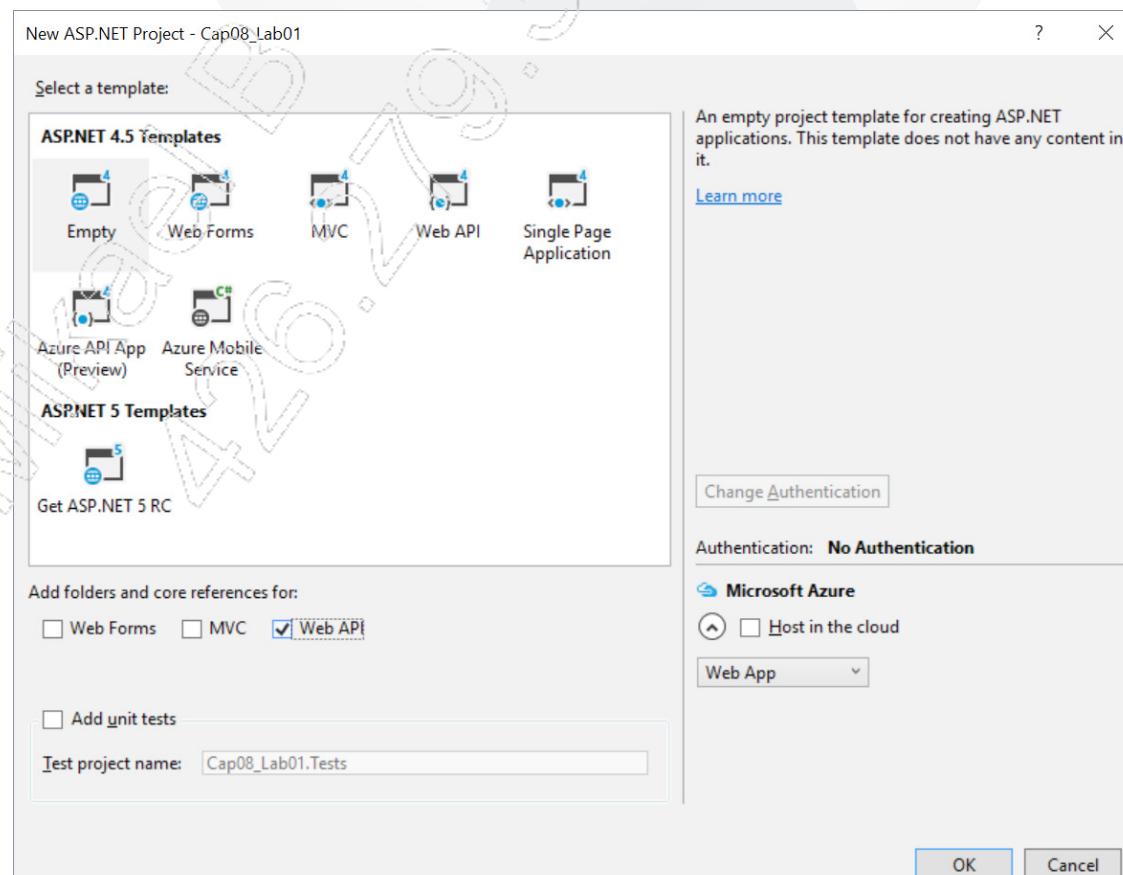


O usuário pode adicionar novos produtos e excluí-los da lista. Essa é a única tela do aplicativo. Ao clicar em **Gravar**, os dados são enviados para o servidor e gravados em um arquivo XML.

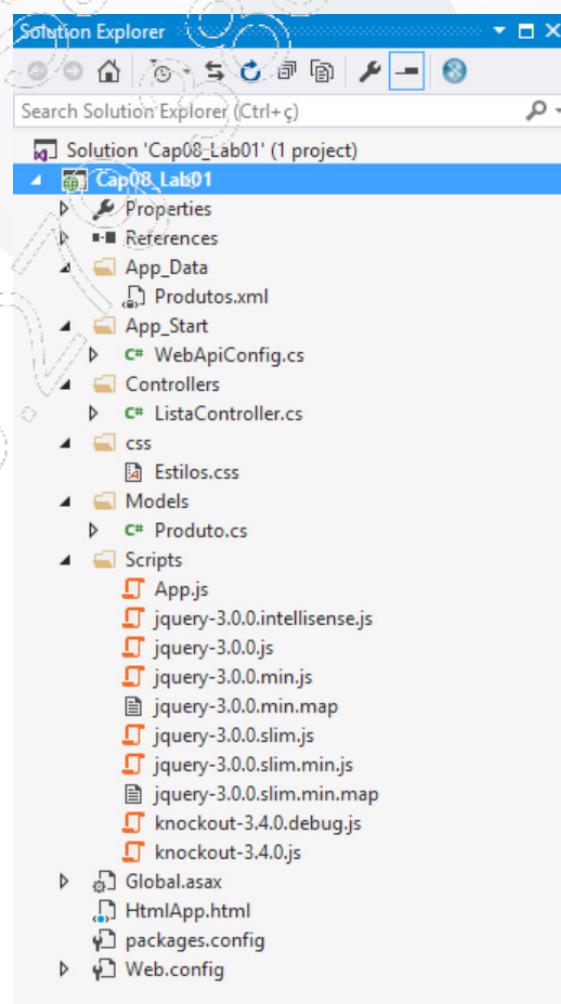
As tecnologias e os conceitos utilizados são os seguintes:

- JavaScript;
- Knockout;
- jQuery;
- Web API;
- SPA;
- HTML;
- Serialização/desserialização;
- JSON;
- XML;
- Protocolo HTTP.

1. Crie um novo projeto ASP.NET vazio e marque a opção **Web API** como tecnologia a ser incluída:



2. Adicione as seguintes pastas: **App_Data**, **Models**, **Scripts** e **css**;
3. Na raiz, adicione um arquivo HTML chamado **HtmlApp.html**;
4. Adicione, via NuGet, a biblioteca **jQuery**;
5. Adicione, na pasta **Controllers**, um arquivo do tipo Web API Controller Class chamado **ListaController.cs**;
6. Na pasta **css**, adicione uma folha de estilo chamada **Estilos.css**;
7. Na pasta **Models**, adicione uma classe chamada **Produto**;
8. Na pasta **App_Data**, adicione um arquivo XML chamado **Produtos.xml**;
9. Na pasta **Scripts**, adicione um arquivo JavaScript chamado **App.js**;
10. Adicione, via NuGet, a biblioteca **knockout.js**;
11. Confira a estrutura do site:



12. A folha de estilos define alguns estilos básicos:

```
body { margin:0px; font-family: 'Century Gothic' }

header{ background-color:#ff6a00;
        color:white; margin-bottom:20px; padding:10px;
}

section{ padding:20px; }

#lista{ min-height:100px; margin-top:30px; }

#formItem { margin-top:10px; }

.cmd { font-size:10px; color:#ff0000; }

.inputValor { width:60px; text-align:right; }

.table-lista td{padding:5px; }

.table-lista tr {padding:5px; }

.table-lista input {padding:5px; }

.table-lista { margin-bottom:20px; }

.botao { padding:5px; }

footer {
    border-top:1px solid #ccc;
    position:fixed;
    bottom:0px; left:0px; right:0px; background-color:#ccc;
}
    footer p {
        padding: 0px 0px 0px 20px;
    }

.campoValor { width:80px; margin-right:5px; }
```

13. Na página **HtmlApp.html**, associe as folhas de estilo, os arquivos de script e as seções básicas:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>

    <link href="css/Estilos.css" rel="stylesheet" />
    <script src="Scripts/jquery-2.1.1.min.js"></script>
    <script src="Scripts/knockout-3.2.0.debug.js"></script>
    <script src="Scripts/App.js"></script>

</head>
<body>

    <header>
        <h1>SinglePageApp - Lista de Compras</h1>
    </header>

    <section>
    </section>

    <footer>
        <p>Desenvolvido para o curso de ASP.NET (c) 2014</p>
    </footer>

</body>
</html>
```

14. Já é possível testar o layout inicial:



15. Antes de vincular os dados, é necessário obter as informações. Os dados dos produtos estão em um arquivo XML, localizado na pasta **App_Data**, chamado **Produtos.xml**. Crie um ou dois produtos de teste;

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfProduto>
```

```
<Produto>
    <Descricao>Caneta</Descricao>
    <Valor>10</Valor>
</Produto>
```

```
<Produto>
    <Descricao>Caderno</Descricao>
    <Valor>24</Valor>
</Produto>
```

```
</ArrayOfProduto>
```

16. Na pasta **Models**, insira o modelo de dados de um produto:

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

namespace Cap08_Lab01.Models
{
    public class Produto
    {
        public string Descricao { get; set; }
        public decimal Valor { get; set; }

    }
}
```

17. Na pasta **Controllers**, crie o método que lerá os produtos na classe **ListaControllers**. É necessário incluir o namespace **System.IO** para a classe **StreamWriter**:

```
namespace Cap08_Lab01.Controllers
{
    public class ListaController : ApiController
    {

        // 
        // Obter Produto
        //
        private List<Produto> ObterProdutos()
        {
            string arquivo=HttpContext.Current
                .Server
                .MapPath("~/App_Data/produtos.xml");

            XmlSerializer serializer = new
                XmlSerializer(typeof(List<Produto>));

            List<Produto> lista = null;
```

```
        using (var stream = new StreamReader(arquivo))
        {
            lista =
                (List<Produto>) serializer.Deserialize(stream);
        }
        return lista;
    }

}
```

18. Ainda na pasta **Controllers**, crie o método que gravará os produtos no arquivo XML:

```
//
// Gravar Produto
//
private void GravarProdutos(List<Produto> lista)
{
    string arquivo = HttpContext.Current
        .Server
        .MapPath("~/App_Data/produtos.xml");

    XmlSerializer serializer = new
        XmlSerializer(typeof(List<Produto>));

    using (var stream = new StreamWriter(arquivo))
    {
        serializer.Serialize(stream, lista);
    }
}
```

19. Esses dois métodos serão utilizados quando o programa em JavaScript interagir com a Web API, por meio dos protocolos GET e POST. A seguir, o método GET, ainda da classe **ListaController**:

```
// GET api/<controller>
public List<Produto> Get()
{
    try
    {
        var lista = ObterProdutos();
        return lista;
    }
    catch
    {
        return new List<Produto>();
    }
}
```

20. Por último, o método POST da classe **ListaController**:

```
// POST api/<controller>

public string Post([FromBody] List<Produto> lista)
{
    try
    {
        GravarProdutos(lista);
        return string.Empty;
    }
    catch (Exception ex)
    {
        return ex.Message;
    }
}
```

21. Esta é a classe principal, a classe **ListaController**, de forma resumida:

```
public class ListaController : ApiController
{
    private List<Produto> ObterProdutos()
    { ... }

    private void GravarProdutos(List<Produto> lista)
    { ... }

    public List<Produto> Get()
    { ... }

    public string Post([FromBody] List<Produto> lista)
    { ... }
}
```

22. Voltando à página HTML, a parte <section> é o local onde a lista de produtos aparecerá. É neste ponto que entram os atributos HTML que vinculam os dados;

```
<section>
    <table class="table-lista">
        <tr>
            <th style="text-align:left">Descricao</th>
            <th>Valor</th>
            <th></th>
        </tr>
        <tbody data-bind="foreach: produtos">
            <tr>
                <td><input data-bind='value: Descricao' /></td>
                <td>
                    <input data-bind='value: Valor' class="inputValor" />
                </td>
            </tr>
        </tbody>
    </table>
</section>
```

```
<td><a href="#" data-bind='click: $root.excluirProduto'
      class="cmd" >excluir </a>
</td>
</tr>
</tbody>

</table>
<hr/>
<p>

<button class="botao" data-bind='click:
gravar,enable: produtos().length>0'>
Gravar</button>

<button class="botao" data-bind='click: addProduto'>
Adicionar um Produto</button>

</p>

</section>
```

23. Agora chegamos à parte principal. O jQuery e o Knockout fazem todo o trabalho. O jQuery precisa obter os dados do servidor, passar para o Knockout e gerenciar o ViewModel. O primeiro passo é criar uma função para ser executada (jQuery) quando a página estiver carregada. No arquivo **App.js**, inclua este código JavaScript:

```
$(function () {
});
```

24. Dentro dessa função, iremos declarar o ViewModel usado pelo Knockout. Repare que é uma função na qual é passado um parâmetro inicial. Esse parâmetro será a lista de produtos do servidor;

```
$ (function () {  
  
    //  
    // ViewModel  
    //  
    var produtosViewModel = function (produtos) {  
  
    }  
  
}) ;
```

25. A função ViewModel define vários métodos. O principal é o **ko.observableArray**, que define um array que notifica quando há mudança, fazendo a interface atualizar-se automaticamente. Outra função importante é **ko.toJS**, que retorna um objeto JavaScript, perfeito para passar via AJAX usando o jQuery;

```
//  
// ViewModel  
//  
var produtosViewModel = function (produtos) {  
  
    var self = this;  
  
    self.produtos = ko.observableArray(produtos);  
  
    self.addProduto = function () {  
        self.produtos.push({  
            Descricao: "",  
            Valor: "0,00"  
        });  
    };  
  
    self.excluirProduto = function (produto) {  
        self.produtos.remove(produto);  
    };
```

```
self.gravar = function () {  
  
    var jqxhr = $.post("api/Lista",  
    { ':': ko.toJS(self.produtos) },  
    function (data) {  
        if (data != '') {  
            alert(data);  
        }  
        else {  
            alert('dados gravados com sucesso.')  
        }  
    }) ;  
};  
  
});  
});
```

26. É importante perceber que a função **produtosViewModel** está apenas definida, nada está sendo executado. O próximo comando é que inicia o processo: ele faz uma chamada AJAX para o servidor, obtém os produtos e inicia a ViewModel:

```
$(function () {  
  
    // ViewModel  
    var produtosViewModel = function (produtos) {  
  
        ...  
    };  
  
    // Inicialização  
    //  
    $.ajax(  
    {  
        url: 'api/Lista',  
  
        type: 'get',  
    });  
});
```

```
        contentType: "application/json",  
  
        success: function (data) {  
  
            ko.applyBindings(  
                new produtosViewModel(data));  
        },  
  
        error: function () {  
            alert('ops');  
        }  
    });  
});
```

Note que agora tudo se conecta. A sequência é a seguinte:

1. O jQuery chama o servidor por meio de uma chamada AJAX;
2. A Web API recebe a chamada;
3. A Web API chama um método que desserializa a lista de produtos, que está em XML;
4. A Web API retorna a lista de produtos;
5. A função **callback** do AJAX chama a função **applyBindings** inicializando o framework Knockout;
6. No mesmo comando, é passado como parâmetro uma instância de **produtosViewModel**, com os dados retornados pela Web API como parâmetro;
7. A função **produtosViewModel** cria um **observableArray**;
8. Neste momento, todo objeto em HTML vinculado é atualizado. A partir daí, apenas as variáveis (**produtosViewModel**) devem ser alteradas.

O botão **Adicionar um Produto** chama o seguinte método do **produtosViewModel**:

```
self.addProduto = function () {
    self.produtos.push({
        Descricao: '',
        Valor: "0,00"
    });
};
```

Repare que a única coisa que esse método faz é adicionar um item vazio no array. A tela é atualizada automaticamente. O mesmo acontece para alterar ou excluir. Apenas a lista de dados é alterada. O comando **Gravar** envia a lista para o serviço que vai gerar um XML com esses dados.

Essa é a contribuição do modelo MVVM (Model-View-ViewModel) para a construção de aplicativos mais simples e eficientes.

27. Teste a aplicação completa.



Desenvolvido para o curso de ASP.NET (c) 2016

9

Testes unitários

- ✓ Tipos de teste;
- ✓ TDD – Test-Driven Development;
- ✓ Projeto de teste;
- ✓ Classe e método de teste;
- ✓ Teste de unidade nos modelos.



IMPACTA
EDITORA

9.1. Introdução

A possibilidade de algo dar errado em um software tem relação direta com a complexidade do código, o ambiente onde é executado, as integrações com sistemas externos, a entrada de dados por parte de usuários e por possíveis erros de concepção na hora de escolher a tecnologia adequada para as tarefas a serem executadas.

Em outras palavras, existe muita coisa que pode dar errado. E a probabilidade de erro aumenta conforme aumentam a complexidade e as necessidades do sistema.

Testar corretamente o código é tão importante quanto criá-lo. Essa tarefa pode ser mais complicada do que parece à primeira vista.

Outro ponto importante é a produtividade. Em um ambiente de desenvolvimento, é importante que o sistema esteja todo sempre pronto para ser compilado e publicado. Isso só é possível se as mudanças e novas implementações forem constantemente testadas e se o impacto que uma parte do código tem sobre outra puder ser percebido o mais rápido possível. Uma metodologia de desenvolvimento orientado a testes possibilita que o sistema esteja sempre em um estado funcional.

Considere uma entrada de dados simples na qual um usuário digita dados básicos, como nome, endereço, data de nascimento etc. Veja, então, algumas coisas que devem ser verificadas:

- Todos os dados estão no formato correto? Os dados estão em faixas de valores válidas?
- A informação que será gravada já existe? Não está sendo duplicada?
- O usuário atual tem permissão para realizar esse cadastro?
- Os dados estão vindo do local esperado? É do domínio correto?
- Os dados como CPF ou CEP são válidos? O e-mail é válido? Está no formato correto? Esse e-mail está sendo usado por outra pessoa?

- O total de caracteres enviados para cada campo está correto? Se forem passados mais caracteres do que o esperado, o sistema trata corretamente?
- Existe uma proteção contra ataques?
- O que acontece se centenas de requisições simultâneas forem executadas?

Essa pode ser uma lista interminável, porque existem muitos aspectos de um aplicativo que merecem atenção, e vão desde a lógica do código até o limite do ambiente onde será executado. Existem, por isso, diversas categorias de testes que podem ser executados.

9.2. Tipos de teste

Veja, a seguir, quais são os tipos de teste que podem ser executados:

- **Testes de unidade:** Este tipo de teste serve para verificar a lógica do código. Chama-se teste de unidade porque, normalmente, é feito em uma parte (ou unidade) do código, como uma biblioteca ou um projeto;
- **Testes de integração:** Testes para validar a integração de uma parte do sistema com outra. Pode ser para testar um serviço, uma biblioteca ou validar uma interface;
- **Testes de stress:** Este tipo é feito para verificar os limites de um aplicativo. Muito comum em Web sites, é utilizado para verificar quantos usuários simultâneos o sistema suporta até começar a esgotar os recursos de memória, rede ou armazenamento físico;
- **Testes de aceitação:** Teste para verificar como o sistema se comporta em ambiente real e como se integra com os outros sistemas que estão em produção. Geralmente, este teste é feito em uma pequena amostra do destino de usuários-final e fornece uma primeira impressão do sistema, antes de entrar em produção real.

A experiência do usuário é cada vez mais complexa: envolve interação no site, recebimento de e-mail, download de aplicativos, validação de credenciais e atualização de status em diversos dispositivos. Todo esse fluxo de dados deve ser constantemente validado.

9.3. TDD – Test-Driven Development

Testar programas é tão importante que existe uma metodologia de desenvolvimento baseada em testes, chamada **TDD (Test-Driven Development)**, ou desenvolvimento orientado a testes.

O ponto mais interessante (e polêmico) do TDD é que o teste é escrito antes do código. Ou seja, antes de escrever o código real, é escrito um teste, automático, pelo qual ele deve passar. À primeira vista, isso pode parecer estranho, mas faz todo o sentido: os programadores escrevem os códigos com atenção redobrada, porque já sabem o resultado que devem alcançar.

O segundo ponto é que os ciclos de testes e de escrita de códigos são muito curtos. Isso possibilita quebrar o sistema em muitos pequenos pedaços e identificar qualquer problema logo no início. Esse tipo de filosofia faz parte de um modelo de desenvolvimento chamado Agile Development, ou desenvolvimento ágil. O criador desse tipo de desenvolvimento, Kent Beck, chamou o resultado da implantação do TDD de *eXtreme Programming (XP)*, que acabou virando uma metodologia à parte.

Apesar do nome **Test**, o objetivo principal do TDD não é testar um sistema, mas sim especificar. O teste é uma maneira criativa de originar uma especificação, sem gastar páginas e páginas de explicação sobre o que o sistema deve fazer para ser interpretado, criado e, só então, testado. E, geralmente, nessa fase de teste é que se descobre que o sistema não atende aos requisitos. Não tem especificação melhor do que uma série de requisitos que o código deve vencer para estar pronto. Tendo o teste como ponto de partida, faz sentido. Não é à toa que o Visual Studio, em todo projeto, inclui a possibilidade de um projeto de teste ser adicionado automaticamente à solução.

9.4. Projeto de teste

Para entender o teste unitário, é necessário criar um projeto simples e validar o resultado que se espera de cada método de cada classe que se deseja testar.

Considere um projeto do tipo Class Library, chamado **Biblioteca**, com uma classe chamada **Calculos**, com quatro métodos: **Somar**, **Subtrair**, **Multiplicar** e **Dividir**. Não é uma classe muito interessante, mas é simples o suficiente para exemplificar o teste de unidade:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Biblioteca
{
    public class Calculos
    {
        public int Somar(int a, int b)
        {
            return a + b;
        }

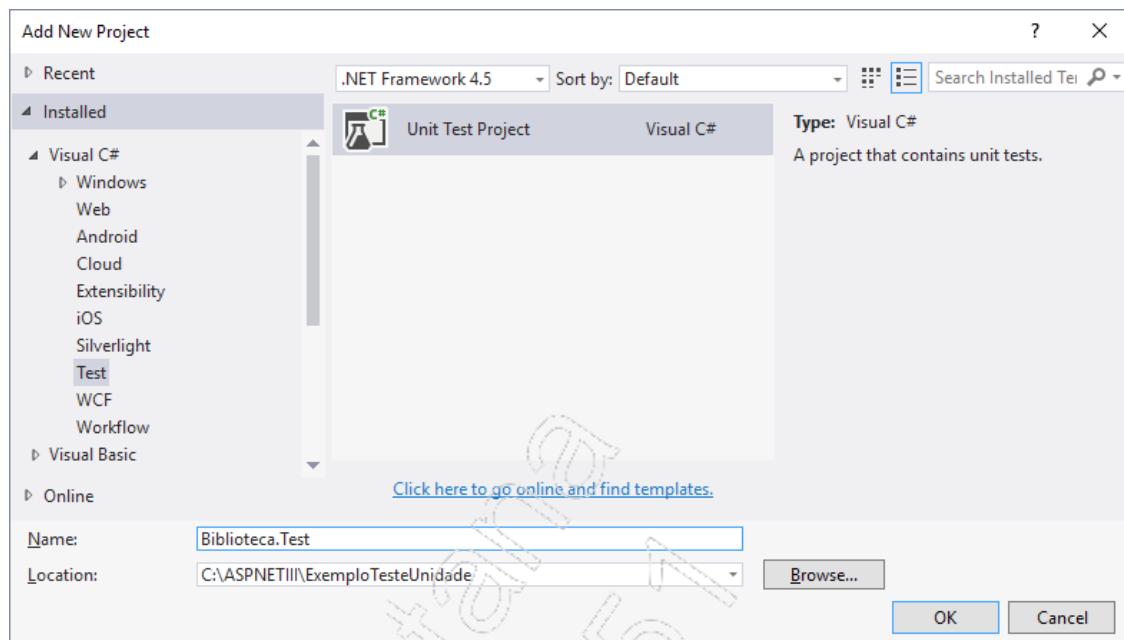
        public int Subtrair(int a, int b)
        {
            return a - b;
        }

        public int Multiplicar(int a, int b)
        {
            return a * b;
        }

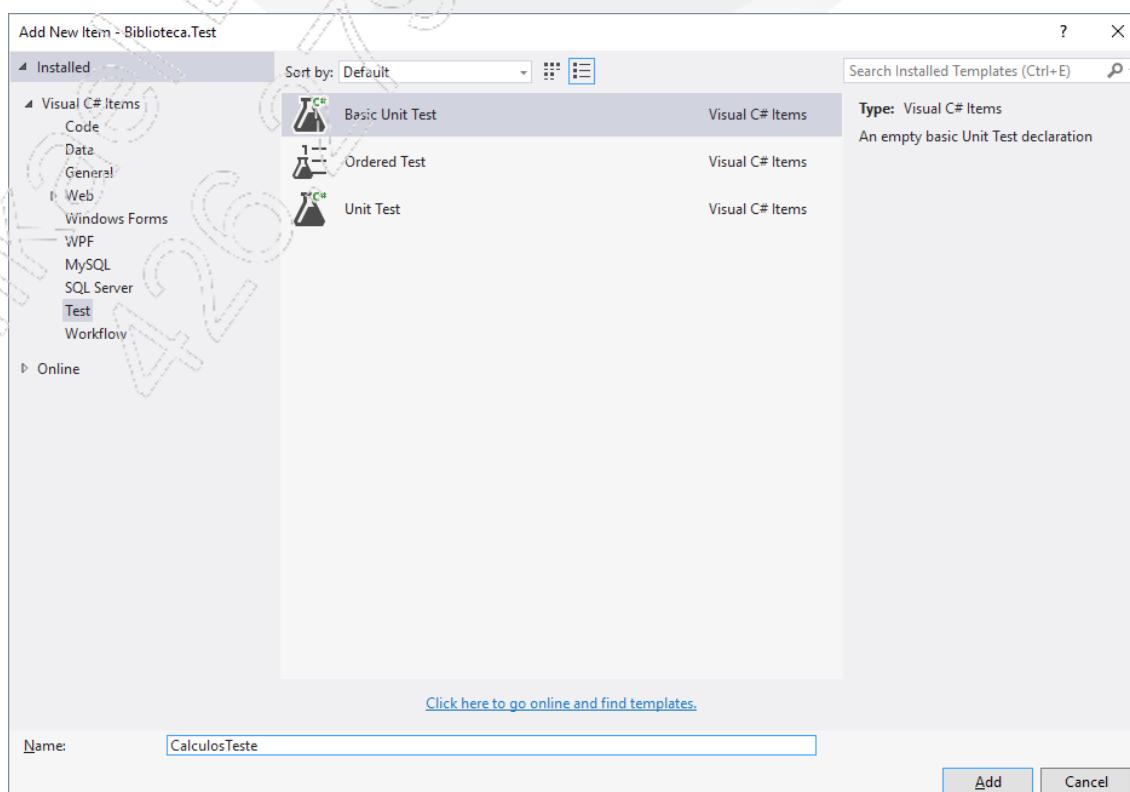
        public double Dividir(int a, int b)
        {
            return a / b;
        }
    }
}
```

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

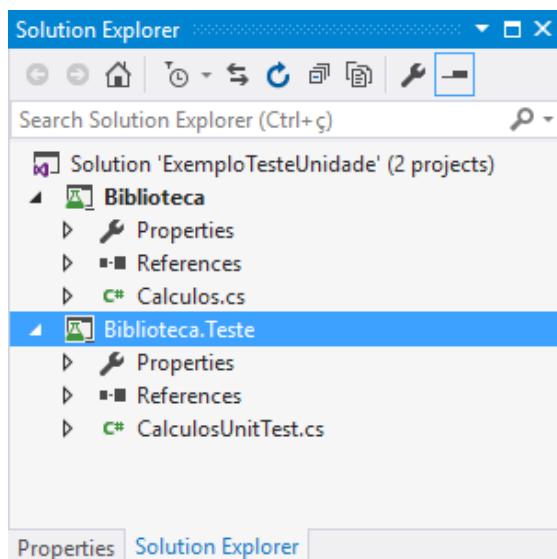
O primeiro passo é adicionar à solução atual um projeto de teste. No menu de contexto da solução, escolha **Add / New Project**. Na janela exibida, escolha a categoria **Test** e a opção **Unit Test Project**. Normalmente, o teste de unidade tem o mesmo nome da unidade que se pretende testar, mais a palavra **test** no final. Nesse caso, um bom nome seria **Biblioteca.Test**:



Agora, a solução tem dois projetos: o Class Library **Biblioteca** e o projeto de teste **Biblioteca.Test**. O ideal é excluir a classe gerada automaticamente e incluir uma classe de teste. O objetivo dessa classe é testar a classe **Calculos**, então, o nome apropriado será **CalculosTeste**. No menu do projeto, escolha **Add / Class** e a opção **Basic Unit Test**:



A solução fica conforme a imagem a seguir:



O namespace onde estão as classes para testes é **Microsoft.VisualStudio.TestTools.UnitTesting**.

A classe de atributo **TestClass** deve ser usada para decorar qualquer classe que terá métodos escritos para testar métodos de outra classe.

A classe de atributo **TestMethod** deve decorar qualquer método de teste. O projeto de teste deve fazer uma referência ao projeto onde está a classe que se deseja testar.

Nesse caso, o projeto **Biblioteca.Test** deve fazer uma referência ao projeto **Biblioteca**.

9.5. Classe e método de teste

Um método de teste é um método que realiza uma operação na classe que se deseja testar e compara se o resultado obtido é o resultado esperado. Isso é feito com a classe **Assert**, que disponibiliza métodos para comparar resultados. O método mais frequentemente usado é **AreEqual** (é igual), cuja sintaxe é a seguinte:

```
Assert.AreEqual( esperado, obtido, "mensagem de erro");
```

Em que:

- **esperado**: É o valor que se espera obter. Por exemplo, se eu estiver testando o método de somar passando os números 2 e 3, o resultado esperado é 5;
- **obtido**: É o valor obtido. Aqui, deve ser chamado o método com os parâmetros. É por isso que deve ser feita uma referência;
- **"mensagem de erro"**: É opcional. Esta mensagem é exibida no relatório, caso o teste falhe.

O exemplo adiante cria dois métodos de teste para validar o método **Somar** da classe **Calculos** da Class Library **Biblioteca**:

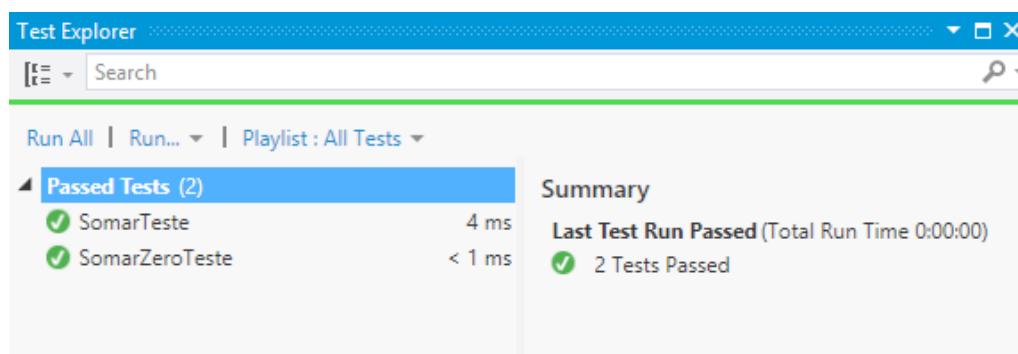
```
[TestClass]
public class CalculosTestes
{
    Biblioteca.Calculos calculos = new Calculos();

    [TestMethod]
    public void SomarTeste()
    {
        Assert.AreEqual(4, calculos.Somar(2, 2),
            "2 mais 2 deveria ser 4");
    }

    [TestMethod]
    public void SomarZeroTeste()
    {
        Assert.AreEqual(0, calculos.Somar(0, 0),
            "0 + 0 deveria ser 0");
    }
}
```

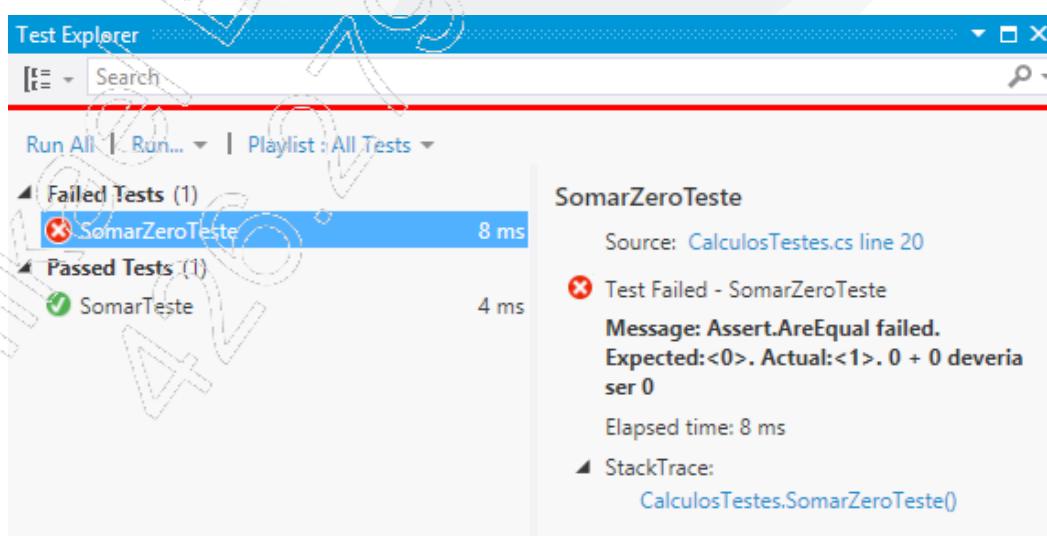
O próximo passo é rodar o teste. No menu **Tests**, escolha a opção **Run** e, em seguida, escolha **All Tests**.

Como era esperado, o método **Somar** passou nos dois testes:



Para visualizar um relatório de testes quando algo dá errado, faremos uma pequena alteração no método **Somar** para que o teste com 0 dê o resultado errado. É claro que isso é só para demonstrar um teste falhando, afinal, não tem o menor sentido mudar o valor da variável **a** para 1 quando for zero. Veja:

```
public int Somar(int a, int b)
{
    if (a == 0) a = 1;
    return a + b;
}
```



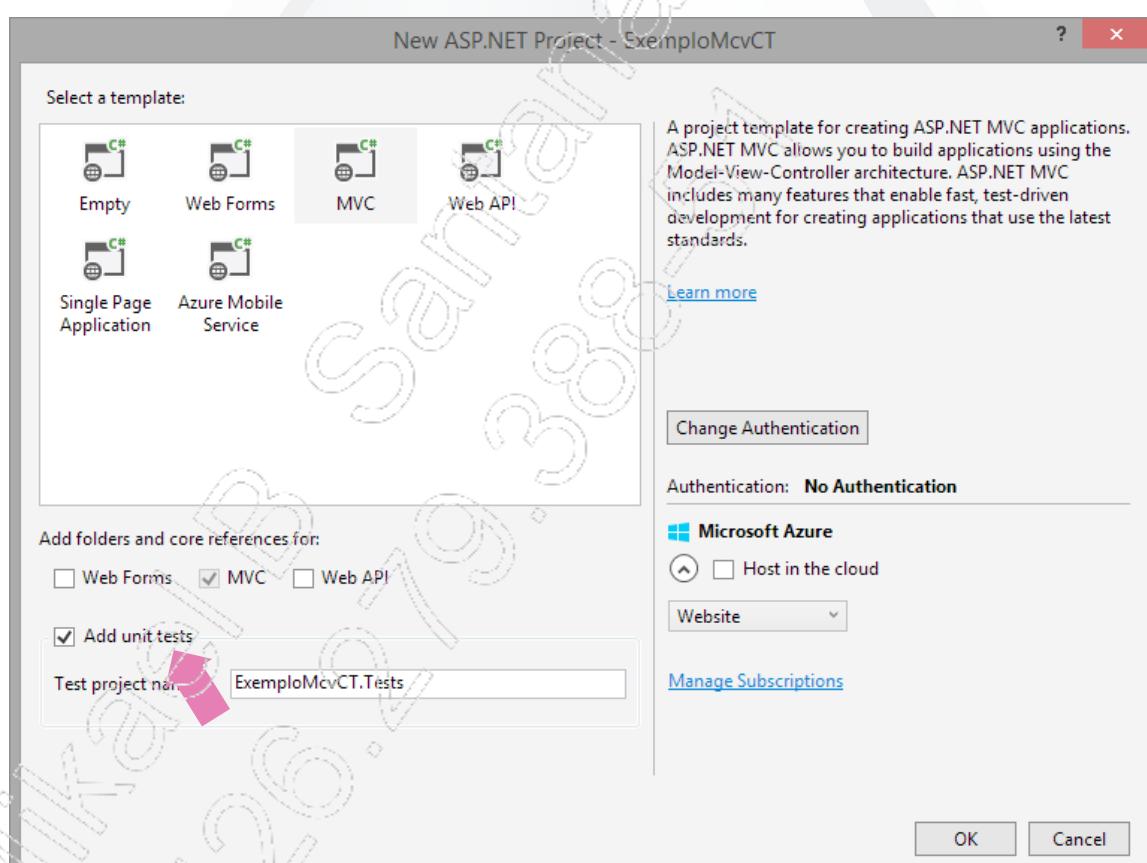
A classe **Assert** ainda conta com os seguintes métodos:

- **AreEqual**: Verifica se dois itens são iguais. Existem 18 sobrecargas. A assinatura mais comum é passar dois objetos e uma string. São comparados os dois objetos e a string é a mensagem de erro;
- **AreNotEqual**: Verifica se dois itens são diferentes;
- **AreNotSame**: Verifica se duas instâncias da mesma classe não estão apontando para o mesmo endereço, ou seja, se são a mesma instância;
- **AreSame**: Verifica se duas instâncias são a mesma, ou seja, apontam para o mesmo endereço na memória;
- **Equals**: Verifica se dois objetos são iguais (não se são a mesma instância, mas se têm o mesmo valor nas propriedades);
- **IsInstanceOfType**: Retorna se um objeto é uma instância de um tipo específico;
- **IsNotNull**: Retorna se um objeto não é nulo;
- **IsNull**: Retorna se um objeto é **null**;
- **IsTrue**: Retorna se um objeto é booleano **true**;
- **IsFalse**: Retorna se um objeto é booleano **false**.

9.6. Teste de unidade nos modelos

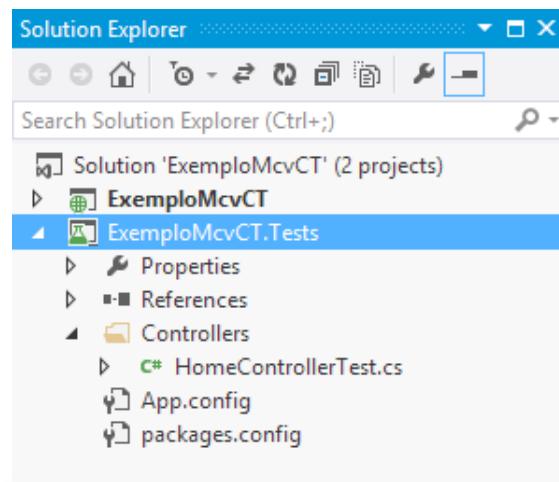
Todos os modelos do Visual Studio para Web oferecem a opção de incluir um projeto de testes. O MVC sempre foi conhecido por ser orientado a testes, desde antes do Visual Studio. O modelo dos Web Forms foi muito criticado no começo pela dificuldade de realizar testes, já que o code-behind era totalmente acoplado à página HTML. Com as versões novas, já é possível incluir testes de unidade em todos os tipos de projetos, de maneira automática.

Para incluir um teste de unidade, quando escolher um projeto na janela de tipos de projetos Web, marque a opção **Add unit tests**:



Visual Studio 2015 - ASP.NET com C# Recursos Avançados

O Visual Studio não só inclui um projeto de testes como também cria uma classe de teste para a classe **HomeController**:



```
[TestClass]
public class HomeControllerTest
{
    [TestMethod]
    public void Index()
    {
        HomeController controller = new HomeController();
        ViewResult result = controller.Index() as ViewResult;
        Assert.IsNotNull(result);
    }
}
```

```
[TestMethod]
public void About()
{
    HomeController controller = new HomeController();
    ViewResult result = controller.About() as ViewResult;
    Assert.AreEqual("Your application description page.",
                    result.ViewBag.Message);
}
```

```
[TestMethod]
public void Contact()
{
    HomeController controller = new HomeController();
    ViewResult result = controller.Contact() as ViewResult;
    Assert.IsNotNull(result);
}
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Testes de unidade são testes executados no código para verificar erros de lógica;
- Testes de integração são executados para testar as interfaces e integrações entre os componentes de uma aplicação;
- TDD é um modelo de desenvolvimento orientado a testes. A sigla significa Teste-Driven Development;
- A classe **TestClass** deve ser usada para decorar a classe que contém métodos de teste;
- A classe **TestMethod** deve ser usada para decorar os métodos de teste de uma classe;
- A classe **Assert** tem métodos estáticos que servem para declarar testes que podem ser executados e os resultados podem ser vistos em um relatório produzido pelo Visual Studio.

9

Testes unitários

Teste seus conhecimentos

Mikael B
Santana
426.279.57



IMPACTA
EDITORA

1. Um aplicativo funciona bem no ambiente de testes, mas, quando publicado em ambiente de produção, não funciona. Qual o tipo de teste que reduz esse tipo de risco?

- a) Teste de stress.
- b) Teste de lógica.
- c) Teste de mesa.
- d) Teste de integração.
- e) Teste de aceitação.

2. Qual classe deve ser usada para criar declarações que serão testadas em um programa?

- a) Debug
- b) Trace
- c) Page
- d) Object
- e) Assert

3. Qual atributo é utilizado para decorar um método de teste?

- a) Serializable
- b) Testable
- c) TestMethod
- d) TestProxy
- e) TDD

4. Qual o nome da metodologia em que os testes são desenvolvidos antes mesmo dos códigos?

- a) TDA
- b) XP
- c) TDD
- d) XAML
- e) XML

5. Qual método da classe Assert pode ser usado para verificar se o objeto é uma instância de um tipo?

- a) AreEqual
- b) typeOf
- c) GetType
- d) IsInstanceOfType
- e) IsTypeofInstance

9

Testes unitários Mãos à obra!

Mikael B. Sartana
426.279.357

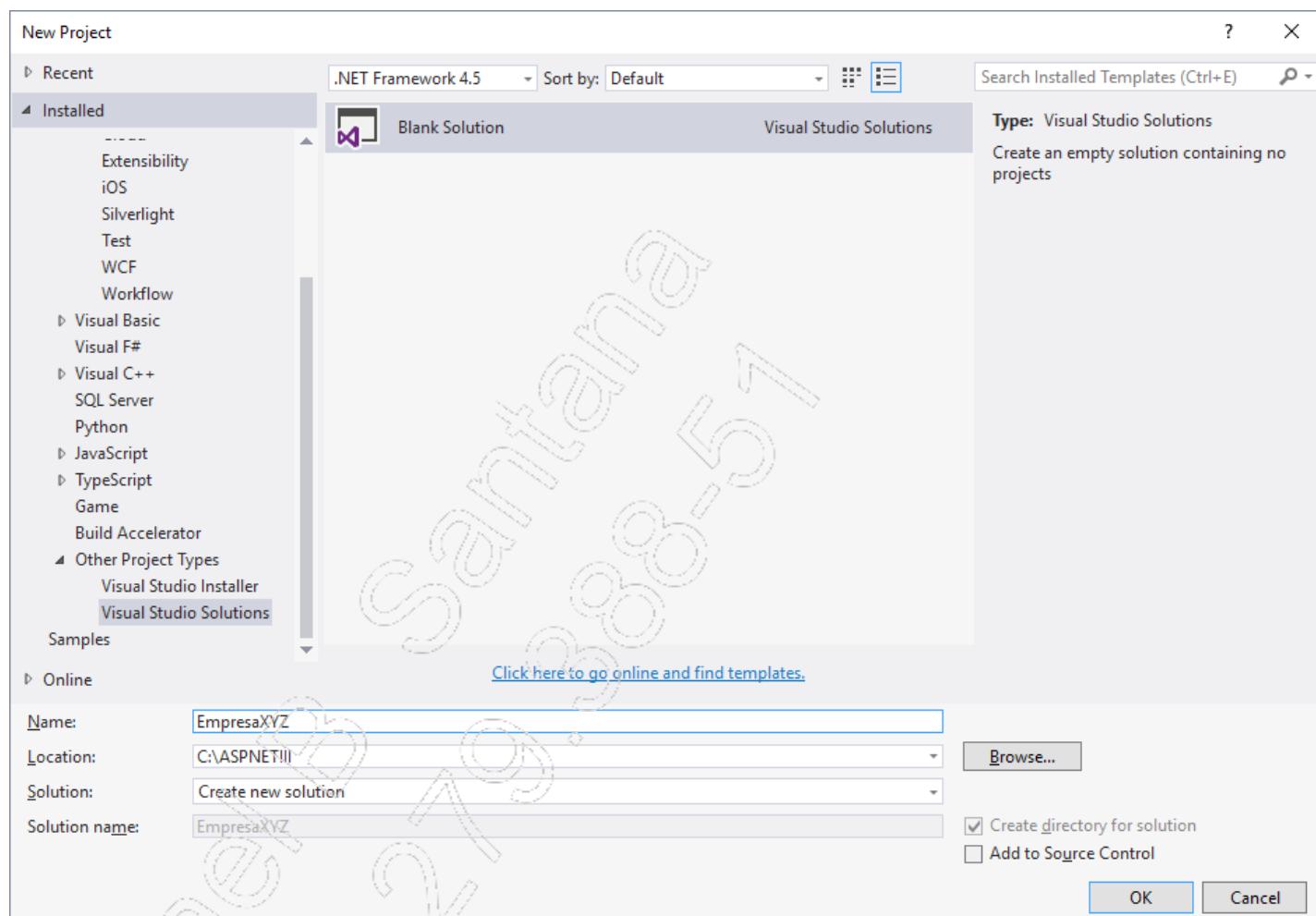


IMPACTA
EDITORA

Laboratório 1

A - Criando um aplicativo em três camadas e um teste de unidade para as bibliotecas de classe

1. Crie uma solução vazia (Other Project Types / Visual Studio Solutions / Blank Solution) chamada EmpresaXYZ:



2. Adicione à solução um projeto do tipo Class Library chamado **EmpresaXYZ.Models**;

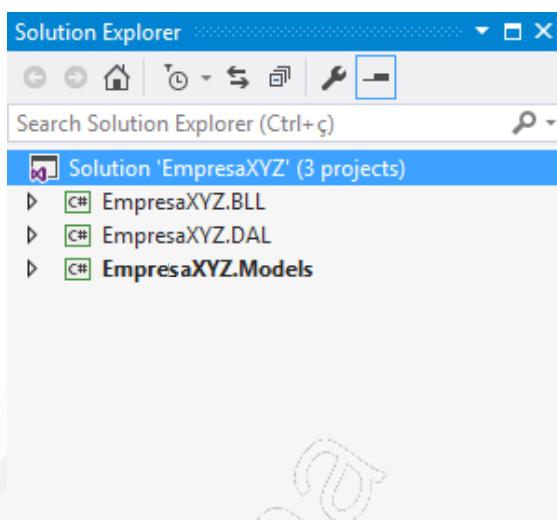
3. Adicione outro projeto do tipo Class Library chamado **EmpresaXYZ.DAL**;

4. No projeto **EmpresaXYZ.DAL**, adicione uma referência para o projeto **EmpresaXYZ.Models**;

5. Adicione um projeto chamado **EmpresaXYZ.BLL** à solução;

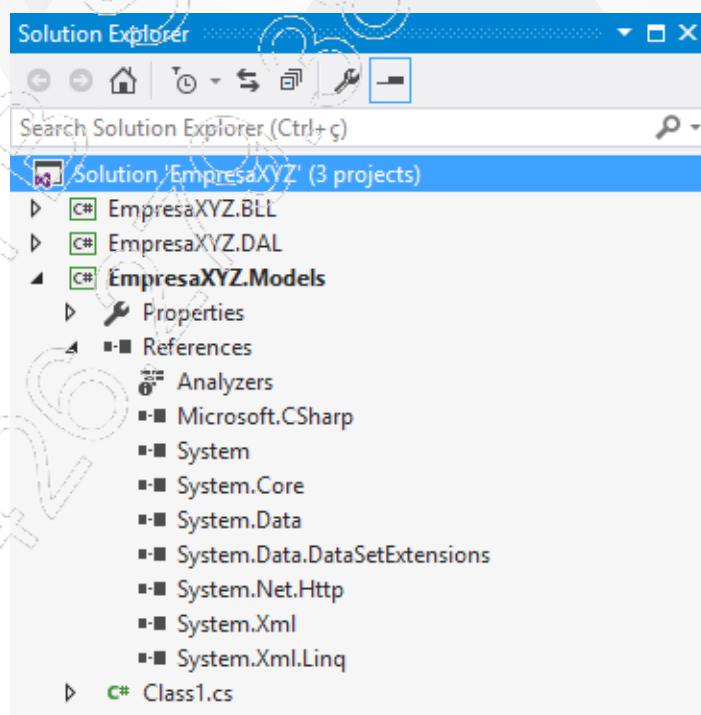
6. Neste projeto, adicione duas referências: ao projeto **EmpresaXYZ.Models** e ao projeto **EmpresaXYZ.DAL**;

O seu projeto deve estar desta forma:



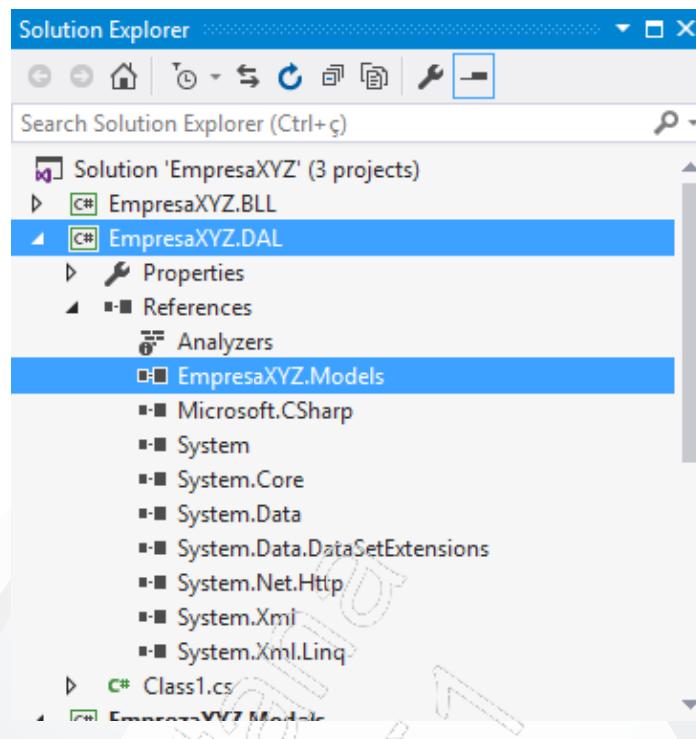
7. Confira as referências:

- O projeto **Models** não tem nenhuma referência, além daquelas fornecidas pelo Visual Studio;

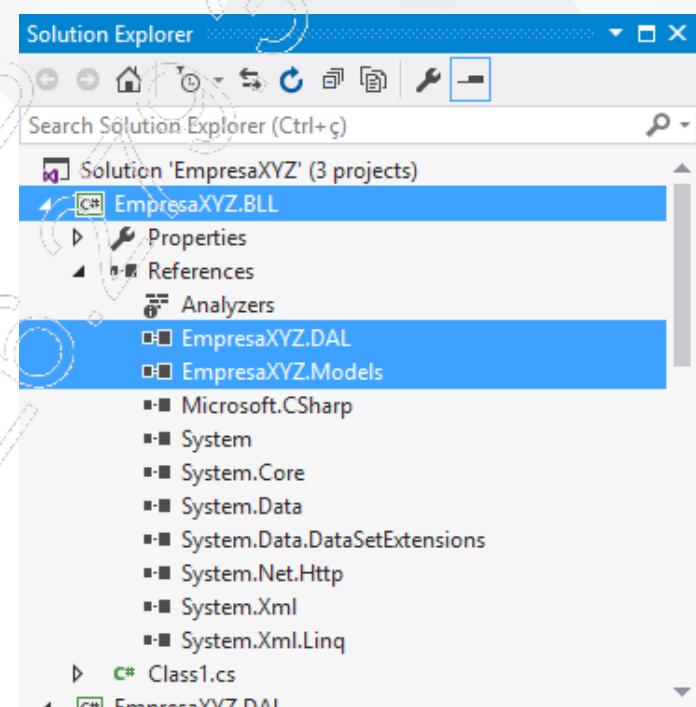


Visual Studio 2015 - ASP.NET com C# Recursos Avançados

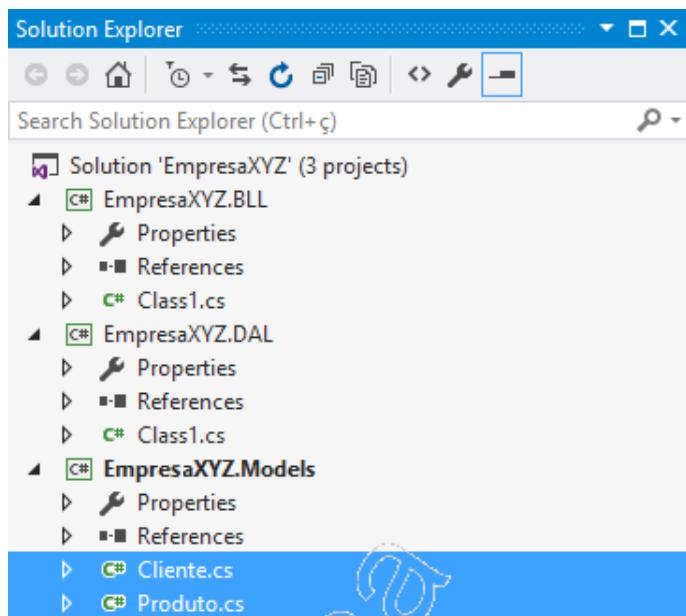
- O projeto **DAL** (Data Access Layer – camada de acesso a dados) tem uma referência ao projeto **Models**;



- O projeto **BLL** (Business Logic Layer – camada de regras de negócio) tem duas referências: a **Models** e a **DAL**.



8. Para começar a programar, crie duas classes no projeto **Models**, chamadas **Cliente** e **Produto**:



- **Cliente.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EmpresaXYZ.Models
{
    public class Cliente
    {
        public int ClienteId { get; set; }
        public string Nome { get; set; }
        public string Email { get; set; }

    }
}
```

- **Produto.cs**

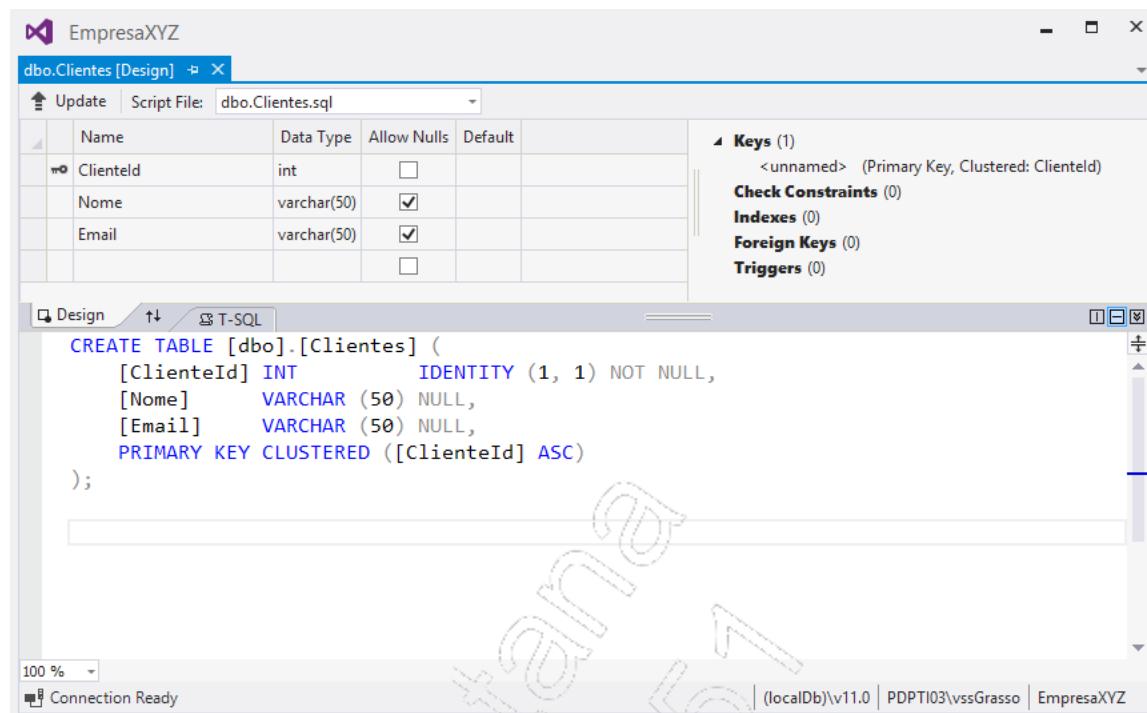
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EmpresaXYZ.Models
{
    public class Produto
    {
        public int ProdutoId { get; set; }
        public string Nome { get; set; }
        public decimal Preco { get; set; }
    }
}
```

 Essas são as classes de domínio. Como são classes muito simples, apenas com propriedades, não é necessário um projeto de testes.

B – Criando o banco de dados

1. Adicione, no Visual Studio, um banco de dados SQL Server chamado **EmpresaXYZ** e crie a tabela **Clientes**:



2. No projeto **DAL**, adicione uma classe chamada **DbHelper**. Utilize a diretiva **using** para fazer referência aos namespaces do SQL Server e das classes comuns do ADO.NET:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
using System.Data.SqlClient;  
using System.Data.Common;  
using System.Data;  
  
namespace EmpresaXYZ.DAL  
{  
    internal class DbHelper  
    {  
        }  
    }  
}
```

3. Crie uma string de conexão. Ela pode ser diferente desta porque depende da versão e do ambiente em que o SQL Server está instalado:

```
namespace EmpresaXYZ.DAL
{
    internal class DbHelper
    {

        //
        // String de Conexão
        //

        public static string Conexao =
            @"Data Source=(localDb)\v11.0;
Initial Catalog=EmpresaxYZ;
Integrated Security=True;";

    }
}
```

4. Esta classe terá alguns métodos de ajuda. Crie um método para abrir a conexão, se ainda não estiver aberta:

```
//
// Abre a conexão se necessário
//
private static void AbrirConexaoSeNecessario(DbCommand cmd)
{
    if (cmd.Connection.State !=
        System.Data.ConnectionState.Open)
        cmd.Connection.Open();
}
```

5. Crie um método que retorna uma instância de um **command()**:

```
//  
// Método que retorna um Command  
//  
public static SqlCommand ObterCommand(string sql)  
{  
    var cmd = new SqlCommand();  
    cmd.CommandText = sql;  
    cmd.Connection = new SqlConnection(Conexao);  
    return cmd;  
}
```

6. Crie um método que retorna o resultado de um **execute Scalar**, quando é um **inteiro**:

```
//  
// ExecuteScalar - Retorna Int  
//  
public static int ExecuteScalarInt(DbCommand cmd)  
{  
    AbrirConexaoSeNecessario(cmd);  
  
    int resultado = Convert.ToInt32(cmd.ExecuteScalar());  
    return resultado;  
}
```

7. Crie um método que retorna um **DataReader**:

```
//  
// Retorna um DataReader  
//  
public static DbDataReader ExecuteReader(DbCommand cmd)  
{  
    AbrirConexaoSeNecessario(cmd);  
    DbDataReader dr =  
        cmd.ExecuteReader(CommandBehavior.CloseConnection);  
    return dr;  
}
```

8. Crie o último método, que executa e retorna o número de registros:

```
//  
// Execute NonQuery  
//  
public static int ExecuteNonQuery(DbCommand cmd)  
{  
    AbrirConexaoSeNecessario(cmd);  
    int resultado = cmd.ExecuteNonQuery();  
    return resultado;  
}
```

9. Esses métodos facilitam bastante o próximo passo que é o arquivo de clientes. Insira uma classe chamada **ClienteDAL**:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using EmpresaXYZ.Models;  
  
namespace EmpresaXYZ.DAL  
{  
    public class ClienteDAL  
    {  
        }  
    }  
}
```

10. Crie o método **Incluir**. Repare como a classe **Helper** ajuda, deixando o código mais limpo:

```
public class ClienteDAL
{
    //Incluir
    public int Incluir(Cliente cli)
    {
        string sql = @"INSERT INTO CLIENTES(NOME, EMAIL)
                        VALUES (@NOME, @EMAIL);
                        SELECT @@IDENTITY";
        int resultado = 0;
        using (var cmd = DbHelper.ObterCommand(sql))
        {
            cmd.Parameters.AddWithValue("@NOME", cli.Nome);
            cmd.Parameters.AddWithValue("@EMAIL", cli.Email);
            resultado= DbHelper.ExecuteScalarInt(cmd);
        }

        return resultado;
    }
}
```

11. Crie o método Listagem:

```
public class ClienteDAL
{
    //Incluir
    public int Incluir(Cliente cli) ...

    //Listagem
    public List<Cliente> Listagem()
    {
        string sql = @"SELECT CLIENTEID, NOME, EMAIL
                      FROM CLIENTES";

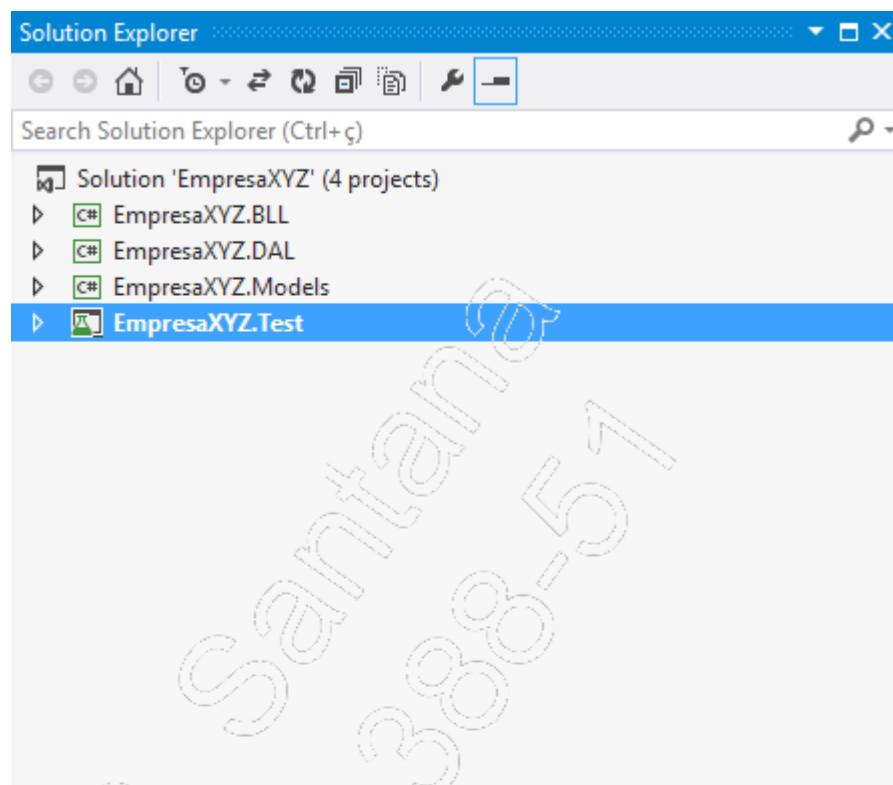
        var lista = new List<Cliente>();

        using (var cmd = DbHelper.ObterCommand(sql))
        {
            using (var dr = DbHelper.ExecuteReader(cmd))
            {
                while (dr.Read())
                {
                    lista.Add(new Cliente() {
                        Nome = dr["Nome"].ToString(),
                        Email = dr["Email"].ToString(),
                        ClienteId=Convert.ToInt32(dr["ClienteId"])
                    });
                }
            }
        }
        return lista;
    }
}
```

C – Testando a unidade

É neste ponto que entra o teste de unidade. A parte de banco de dados está completa, mas só vai ser testada quando a interface for construída. Vamos inserir um projeto de teste na solução e testar se essa classe está funcionando.

1. Insira um novo projeto de teste de unidade chamado **EmpresaXYZ.Test**:



2. Adicione referências aos projetos **Models**, **DAL** e **BLL** no projeto de teste;
3. Renomeie a classe padrão para **ClienteDALUnitTest**;

4. Teste a inclusão e a listagem:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EmpresaXYZ.Models;
using System.Collections.Generic;

namespace EmpresaXYZ.DAL.Test
{

    [TestClass]
    public class ClienteDALUnitTests
    {
        [TestMethod]
        public void Incluir()
        {
            var cliente = new Cliente();
            cliente.Nome = "Maria da Silva Teste01";
            cliente.Email = "Maria@teste.com.br";

            var dal = new ClienteDAL();
            int novoId = dal.Incluir(cliente);

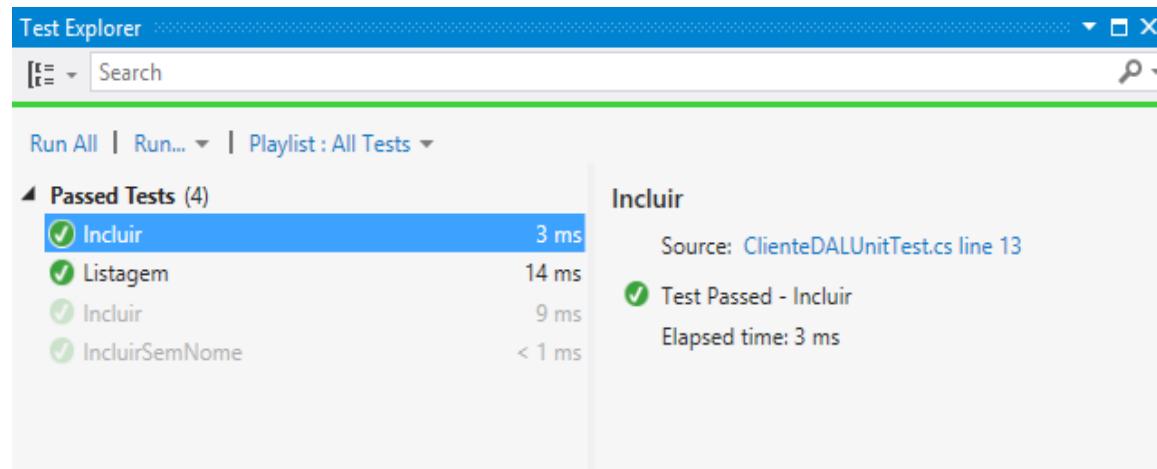
            Assert.AreEqual(true, novoId > 0,
                @"O ID do cliente deveria
                ser um número maior que zero!");
        }

        [TestMethod]
        public void Listagem()
        {
            var dal = new ClienteDAL();
            var lista = dal.Listagem();

            Assert.AreEqual(true, lista.Count > 0,
                "Deve haver 1 cliente na lista");

        }
    }
}
```

5. No menu **Tests**, escolha **Run / AllTests**:



6. Crie a classe **ClienteBLL**, no componente **BLL**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using EmpresaXYZ.BLL;
using EmpresaXYZ.Models;
using EmpresaXYZ.DAL;

namespace EmpresaXYZ.BLL
{
    public class ClienteBLL
    {
        public int Incluir(Cliente cli)
        {
            Validar(cli);
            var dal = new ClienteDAL();
            return dal.Incluir(cli);
        }
    }
}
```

```
public List<Cliente> Listagem()
{
    var dal = new ClienteDAL();
    return dal.Listagem();

}
```

```
private void Validar(Cliente cli)
{
    if (string.IsNullOrEmpty(cli.Nome))
    {
        throw new Exception(
            "O nome deve ser informado");
    }
}
```

7. É possível criar um arquivo de teste para cada projeto ou um arquivo só para todos. Neste caso, vamos incluir outra classe. Adicione uma classe no arquivo de teste chamada **ClienteBLLUnitTest.cs**:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EmpresaXYZ.Models;
using EmpresaXYZ.BLL;

namespace EmpresaXYZ.BLL.Test
{
    [TestClass]
    public class ClienteBLLUnitTest
    {

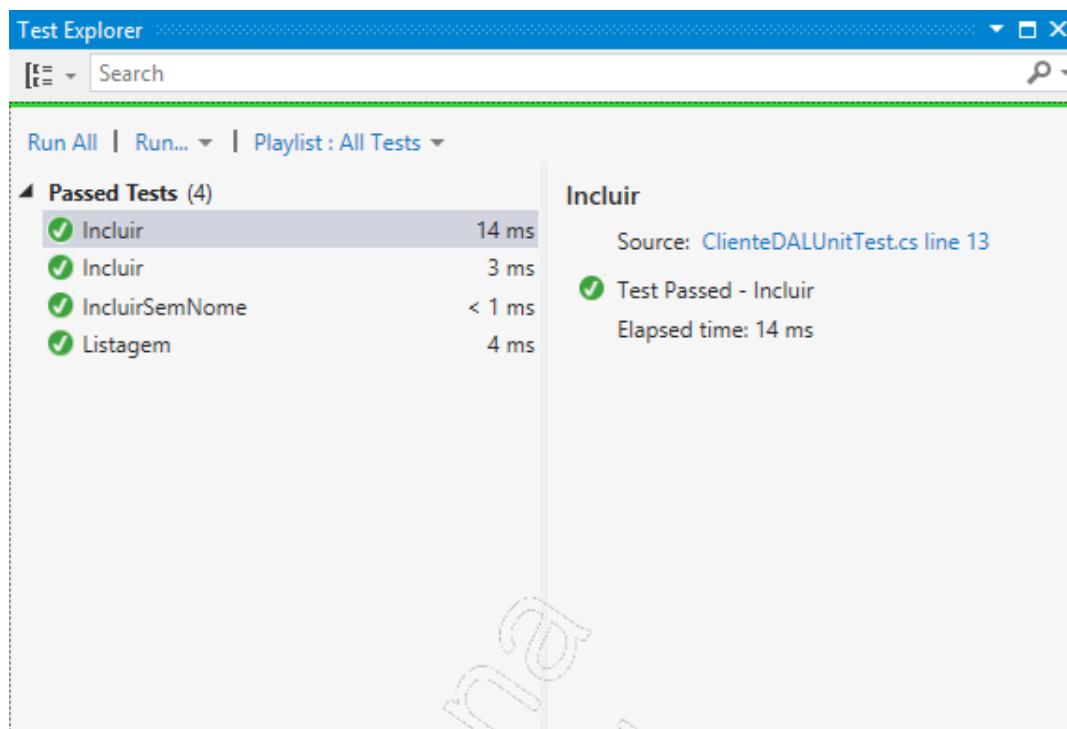
        [TestMethod]
        public void Incluir()
        {
            var cli = new Cliente()
            {
                Nome = "José",
                Email = "Jose@teste.com.br"
            };
            var bll = new ClienteBLL();
            int resultado = bll.Incluir(cli);

            Assert.AreEqual<bool>(true, resultado > 0);
        }
    }
}
```

```
[TestMethod]
public void IncluirSemNome()
{
    var cli = new Cliente()
    {
        Nome = null,
        Email = "Jose@teste.com.br"
    };
    try
    {
        var bll = new ClienteBLL();
        int resultado = bll.Incluir(cli);
        Assert.Fail(
            "Deveria ter dados erro. Passei o Nome Null");
    }
    catch (Exception ex)
    {
        Assert.AreEqual<string>(
            "O nome deve ser informado", ex.Message);
    }
}
}
```

 No caso desse último teste, é gerada uma exceção. Mas isso é correto, já que a exceção é a resposta certa se o nome não é passado. Por isso que o método `IncluirSemNome` captura o erro e define que está correto se a mensagem for "O nome deve ser informado".

8. Rode todos os testes:



Pronto! Agora a infraestrutura está pronta e foi testada. É bem pouco provável que aconteça algum erro na interface.

9. Para testar, adicione à solução um projeto do tipo **ASP.NET MVC** sem autenticação. Chame esse projeto de **EmpresaXYZ.UI.Web**;

10. Adicione duas referências: ao projeto **Models** e ao projeto **BLL**;

11. Altere o controller **Home**:

```
using EmpresaXYZ.BLL;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace EmpresaXYZ.UI.web.Controllers
{
    public class HomeController : Controller
    {
```

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

public ActionResult Clientes()
{
    var bll = new ClienteBLL();
    var lista = bll.Listagem();
    return View(lista);
}

}
```

12. Crie uma View para Clientes:

```
@model IEnumerable<EmpresaXYZ.Models.Cliente>

@{
    ViewBag.Title = "Clientes";
}



## Clientes

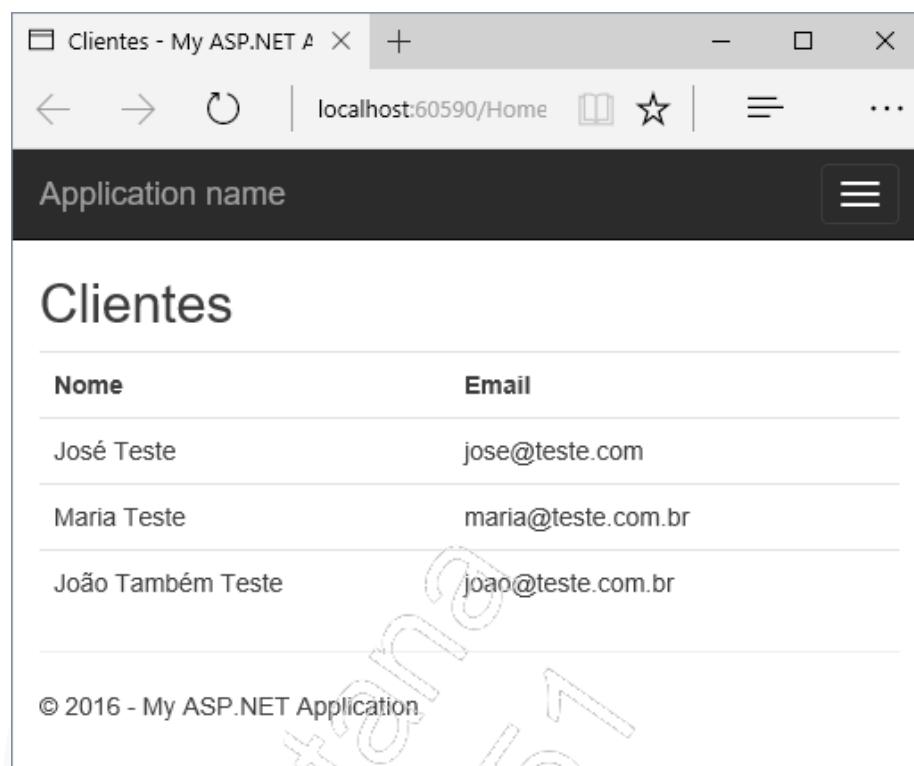


| @Html.DisplayNameFor(model => model.Nome) | @Html.DisplayNameFor(model => model.Email) |
|-------------------------------------------|--------------------------------------------|
| @Html.DisplayFor(modelItem => item.Nome)  | @Html.DisplayFor(modelItem => item.Email)  |


```

Visual Studio 2015 - ASP.NET com C# Recursos Avançados

13. Visualize. Os testes feitos com o arquivo de clientes geraram vários registros que podem ser vistos agora:



Este é o modelo de projeto em camadas. É uma arquitetura bastante robusta e que pode ser utilizada em qualquer tipo de projeto e serve como ponto de partida para outras arquiteturas como DDD (Domain-Driven Design) ou SOA (Service-Oriented Application).

