

Elys Sanntos

[**Programação**](https://cursos.alura.com.br/category/programacao)[**Cursos de .NET API**](https://cursos.alura.com.br/category/programacao#net-api)

SUMÁRIO

[Introdução 3](#_Toc63414113)

[**Curso de** ASP.NET Identity parte 1: Gerencie contas de usuários 3](#_Toc63414114)

[Faça esse curso de .NET API e: 3](#_Toc63414115)

[**Apresentação:** 4](#_Toc63414116)

[**Aulas:** 5](#_Toc63414117)

[Principais comandos 5](#_Toc63414118)

[Conflitos 6](#_Toc63414119)

[Conclusão 6](#_Toc63414120)



# Introdução

# **Curso de ASP.NET Identity parte 1: Gerencie contas de usuários**

## Faça esse curso de .NET API e:

* Entenda a arquitetura do AspNet Identity
* Valide senhas seguras
* Faça o gerenciamento de contas de usuário
* Envie emails de confirmação de conta
* Use o Owin para desacoplar sua aplicação do servidor

# **Apresentação:**

Olá! Tudo bem? Sou Guilherme Matheus e dou as boas vindas ao curso de ASP.NET Identity Parte 1. Nessa primeira parte do curso, utilizaremos um fórum do nosso cliente — o ByteBank — como projeto. Nele, usaremos o [ASP.NET Identity](https://www.asp.net/identity) para fazer o gerenciamento de identidade de usuárias e usuários. Trata-se de um framework ASP.NET utilizado para o gerenciamento de identidades. Ou seja, ele não é um framework de segurança.

Nessa primeira parte do curso, passaremos pelas principais classes e pela arquitetura do ASP.NET Identity. Vamos formar conhecimento para entender como funciona esse framework e porquê determinadas escolhas de design, em relação à interfaces e à sincronicidade, foram feitas pela Microsoft para esse código.

Passaremos pelos pacotes necessários para instalar o Identity e um projeto criado a partir do zero. Ou seja, não precisaremos de um projeto criado com ASP.NET Identity para fazer uso de suas funcionalidades. Também estudaremos o [Owin](http://owin.org/), entendendo porquê ele necessário em um projeto ASP.NET Identity e como utilizá-lo no nosso projeto.

Nesse curso, para utilizarmos esse framework, precisaremos armazenar nossas usuárias e usuários em algum lugar. Pensando nisso, utilizaremos o [Entity Framework](https://docs.microsoft.com/pt-br/ef/) para fazer a persistência desses dados. Mas, ter conhecimento sobre ele não será obrigatório para o que estudaremos. Qualquer conteúdo que explorarmos acerca dele, será discutido e resolvido no próprio vídeo da aula.

Então, passaremos pela arquitetura, pacotes, Owin e sincronicidade do ASP.NET Identity e, no final do curso, conseguiremos enviar um e-mail de confirmação de conta para usuárias e usuários. Trata-se daquele recurso de segurança que não permite a entrada em um sistema, sem ter confirmação por e-mail antes.

Esse curso de ASP.NET Identity foi dividido em quatro partes. Nessa primeira, formaremos um conhecimento para que nas próximas três partes, possamos implementar recursos avançados, como: autenticação de dois fatores e autenticação externa, além de restrições de acesso por meio de roles. Também entenderemos como é esse conceito de Clay. Legal?

Então, vamos começar essa jornada pelo ASP.NET Identity?

#### Propósito do AspNet Identity

Durante o curso, aprenderemos sobre a arquitetura e as principais classes do AspNet Identity! Mas, afinal, qual é o propósito do AspNet Identity?

O gerenciamento de identidades de usuários

O AspNet Identity é uma biblioteca de gerenciamento de identidades de usuários .

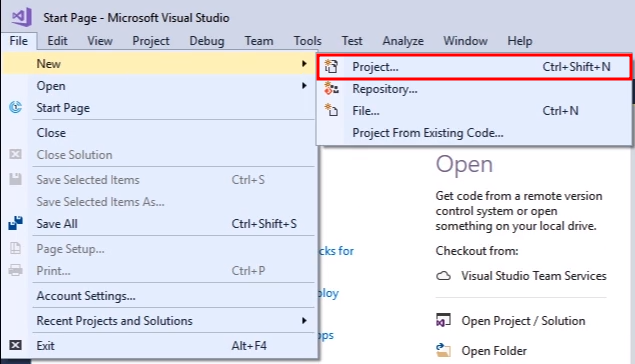
# **Aulas:**

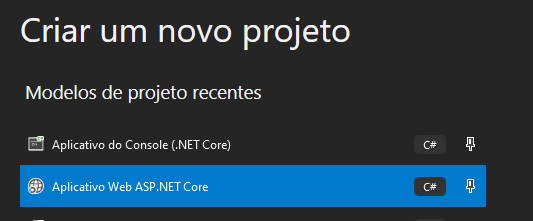
#### Aula 01: Começando com AspNet Identity

#### Instalando os pacotes necessários

Vamos começar o curso de ASP.NET Identity? No cenário que trabalharemos, o ByteBank é nosso cliente, e ele quer criar um novo meio de comunicação para estreitar relações com seus usuários. Esse novo meio de comunicação será um **fórum**, no qual as pessoas poderão discutir assuntos relacionados ao mundo financeiro, ou seja, sobre: investimentos, cartões e o próprio banco, ByteBank.

Como será uma aplicação ASP.NET, abriremos uma nova instância do Visual Studio e criaremos um projeto ASP.NET MVC, a partir do menu "File > New > Project".

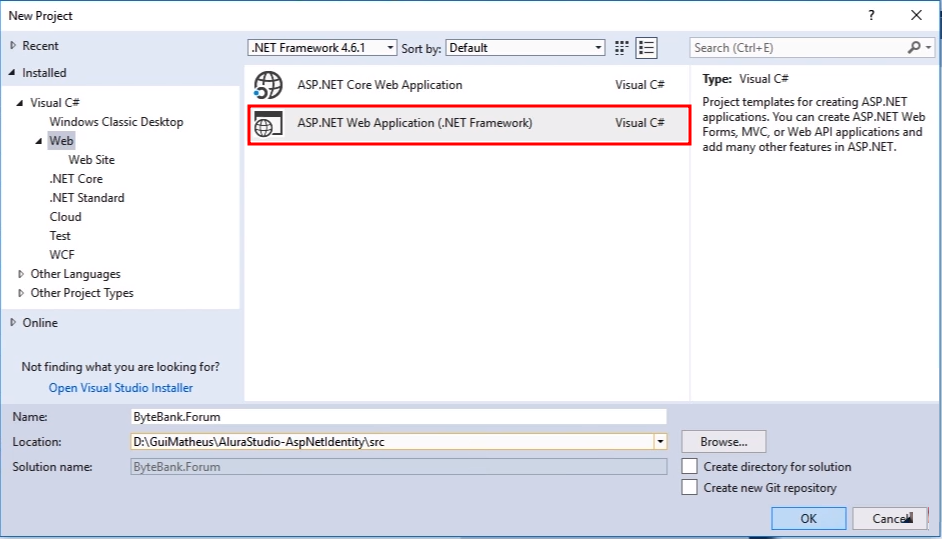


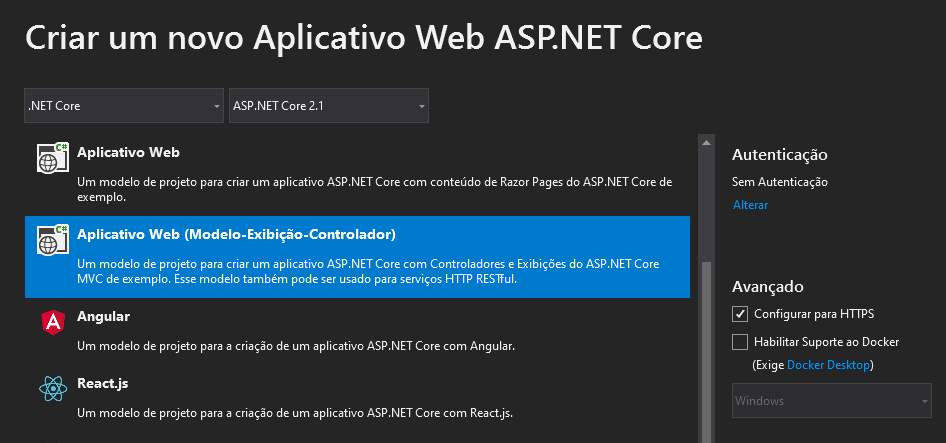


Para desenvolver esse curso, foi utilizado o [Visual Studio 2017, versão Community](https://visualstudio.microsoft.com/pt-br/vs/community/?rr=https%3A%2F%2Fwww.google.com%2F). Sendo assim, na janela que se abrirá para configuração do novo projeto, no menu esquerdo, navegaremos por "Visual C# > Web", no qual encontraremos a opção "ASP.NET Web Application (.NET Framework)".

Importante notar que trata-se de um .NET Framework. Esse é o texto apresentado na versão 2017 do Visual Studio. Talvez, a versão 2015 ou outra que você esteja utilizando apresente um texto diferente, mas o importante é ser uma aplicação ASP.NET, .NET Framework.

Com essa opção selecionada, nomearemos nosso projeto como ByteBank.Forum. Em "Location", é possível definir o local de instalação. Com tudo devidamente configurado, clicaremos em "OK", no canto inferior direito.

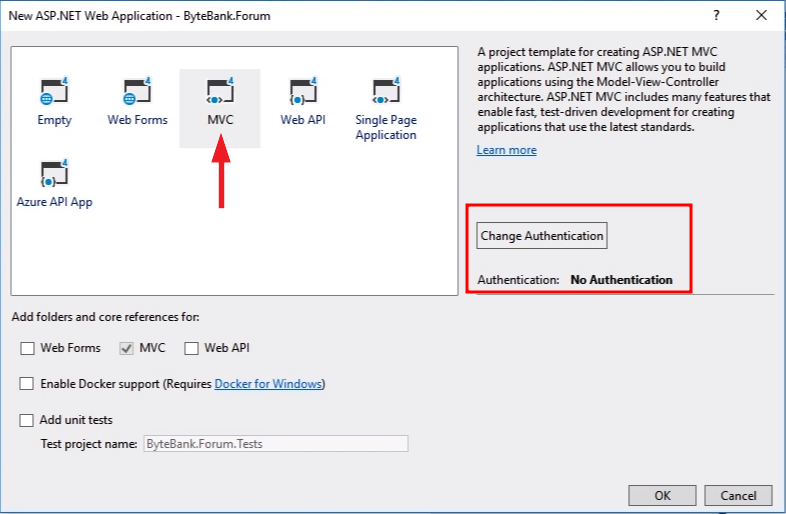




Na janela seguinte, teremos mais uma etapa desse Wizard de criação de projeto, na qual selecionaremos o template para nossa aplicação. Dentre as seguintes opções:

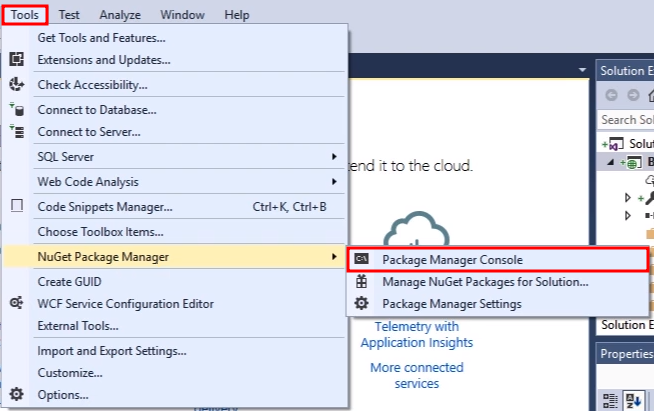
* Empty (Vazio)
* Web Forms
* MVC
* Web API
* Single Page Application
* Azure API App

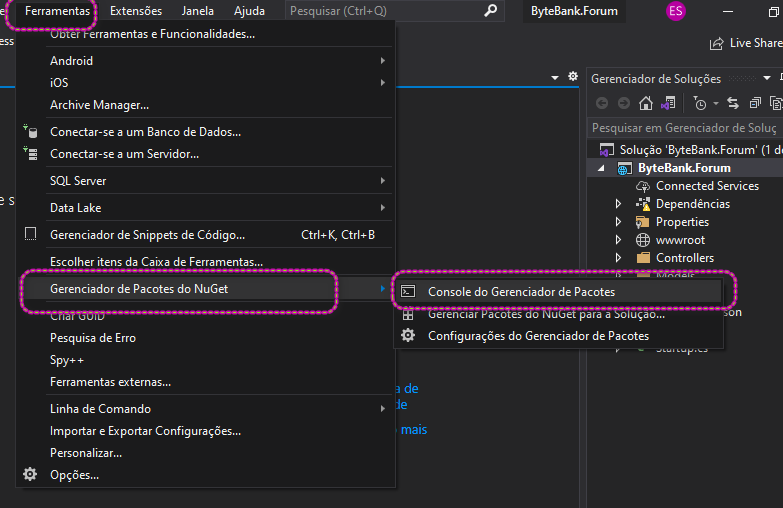
Escolheremos MVC. Precisamos considerar que teremos a preocupação de gerenciar as contas de usuárias e usuários do fórum. E, por mais que, à direita das opções, encontremos um botão para alterar a autenticação ("Change Authentication"), não utilizaremos o do template do Visual Studio, pois entenderemos como esse recurso deve ser utilizado, construindo-o. Portanto, manteremos acionada a opção "No Authentication".



Feito isso, clicaremos em "OK", no canto inferior direito, e o projeto será criado.

Bom, optamos por utilizar a opção "No Authentication". No entanto, manteremos a preocupação com o gerenciamento de contas do fórum, no caso: em cadastros e como será feito o login. Para isso, utilizaremos o ASP.NET Identity. Então, no menu superior, acessaremos "Tools > NuGet Package Manager > Package Manager Console".





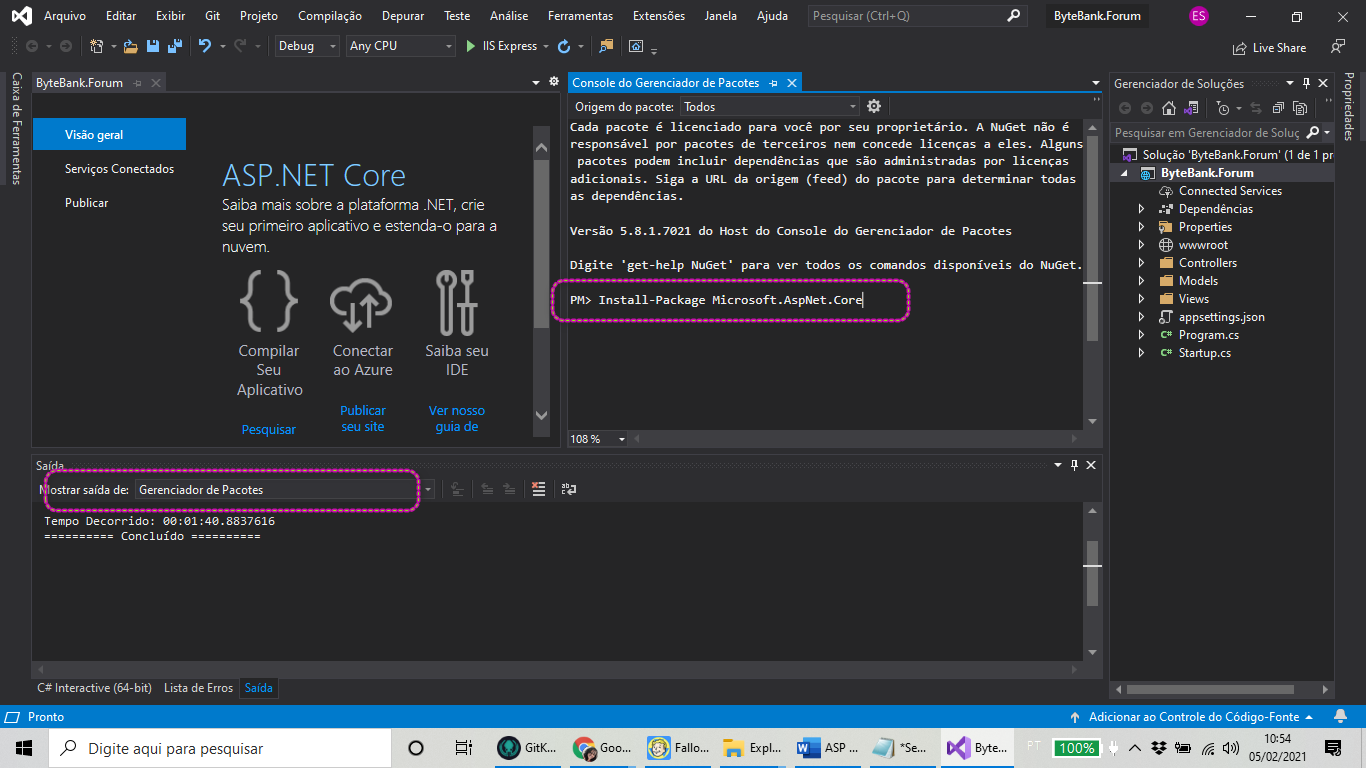
Dessa forma, poderemos incluir um pacote de dependência em nosso projeto, ByteBank.Forum. Com o console aberto, instalaremos alguns pacotes do Identity, adicionando após PM>:

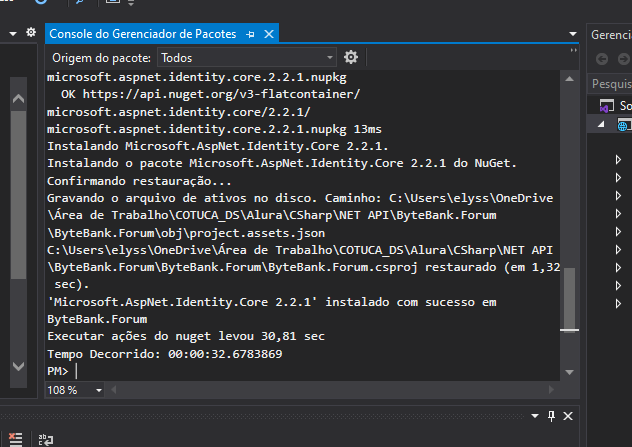
PM> Install-Package Microsoft.AspNet.Identity.Core -Version 2.2.1

Especificamos Core ao final, considerando que o ASP.NET Identity possui vários pacotes, dentro do Nuget. Esse pacote Core contém os principais recursos e bibliotecas do Identity.

Além disso, apontamos a versão (Version 2.2.1), que é a mais atual — pelo menos durante a gravação desse curso. No momento em que você iniciar o curso e implementar isso na sua máquina, para evitar problemas, recomendo que utilize a mesma versão, para entendermos juntos como funciona o Identity.

Após digitar a linha de instalação, teclaremos "Enter"

 e aguardaremos a instalação de todos os pacotes relacionados ao Identity.



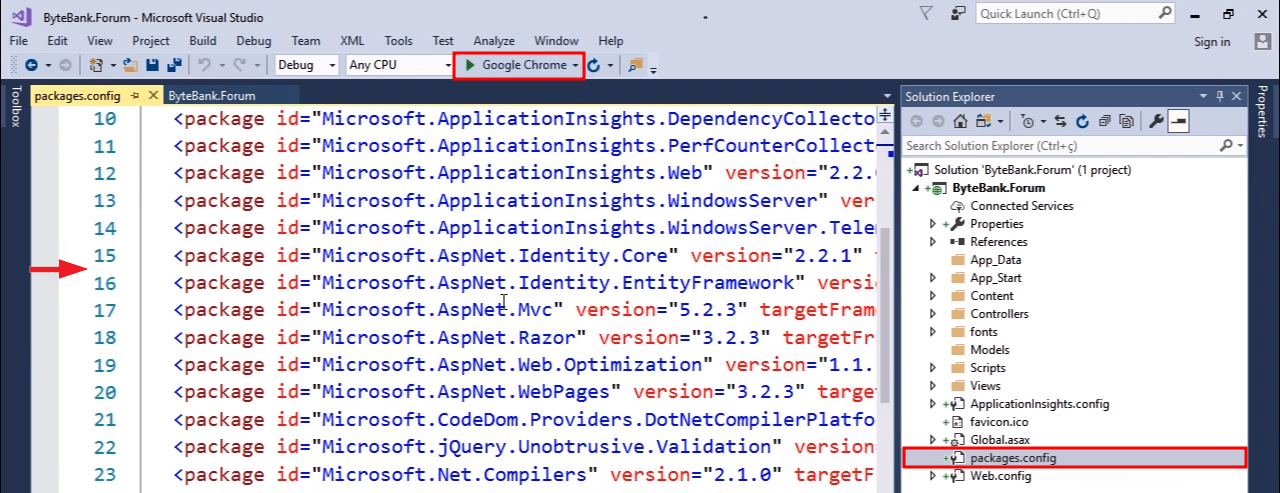
Enquanto isso, podemos pensar no local em que serão armazenados essas usuárias e usuários, além das informações a respeito do gerenciamento deles. Já que estamos trabalhando com ASP.NET, ASP.NET MVC e ASP.NET Identity, ou seja, todos produtos da Microsoft, em cima do .NET, é uma escolha natural armazenar esses dados no Entity Framework. Sendo assim, instalaremos mais um pacote, relacionado ao Entity Framework e à interface dele com o nosso ASP.NET Identity.

Para isso, abaixo da instalação do pacote anterior, após PM>, acionaremos a seta para cima do teclado para verificar o histórico de pacotes. Com isso, encontraremos a linha que utilizamos para a primeira instalação, que reutilizaremos, aplicando algumas alterações. Bastará substituirmos Core por EntityFramework. A versão permanecerá a mesma:

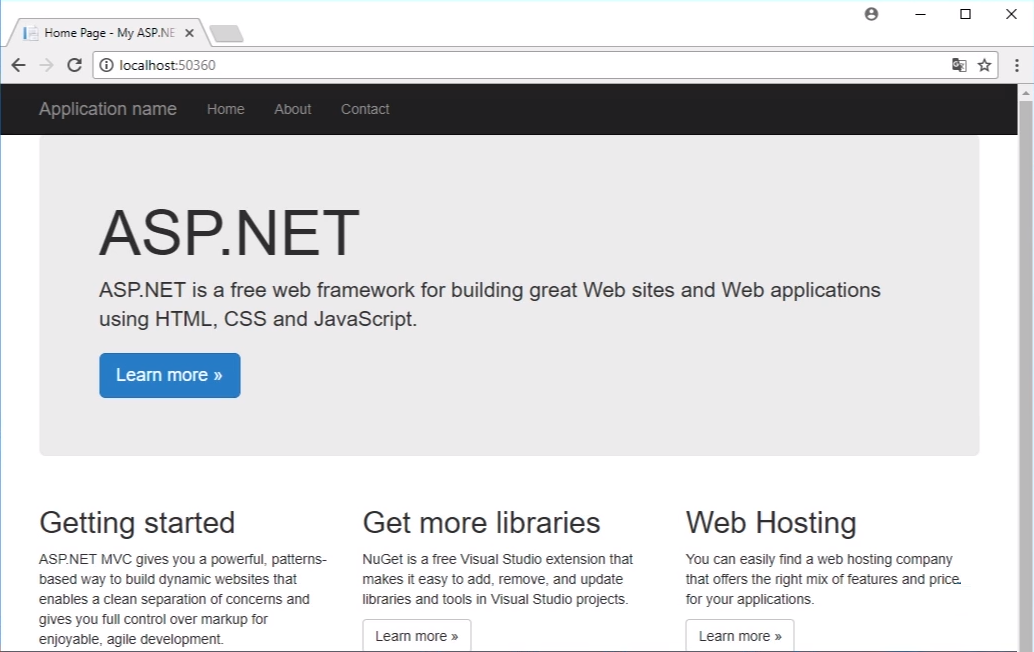
PM> Install-Package Microsoft.AspNet.Identity.EntityFramework -Version 2.2.1

Teclaremos "Enter" e aguardaremos. Maravilha! Após um tempinho, teremos os pacotes instalados. Para verificar se a instalação foi feita corretamente, podemos acessar o menu direito e clicar em packages.config.

Nas linhas 15 e 16, encontraremos as duas instalações que acabamos de fazer. Na sequência, no topo, clicaremos em "Google Chrome", para dar um Start e conferirmos que está compilando e está tudo bonitinho com essas dependências também.



Aguardaremos a compilação da aplicação. Uma vez concluída, abrirá uma janela de navegador, que talvez demore mas um tempinho para carregar, até exibir o seguinte conteúdo:



Maravilha! Nossa aplicação está rodando, com os pacotes instalados. Sendo assim, vamos começar a programar?

#### Pacote obrigatório

Para usarmos o AspNet Identity em nosso projeto AspNet, foi obrigatório a instalação de qual pacote?

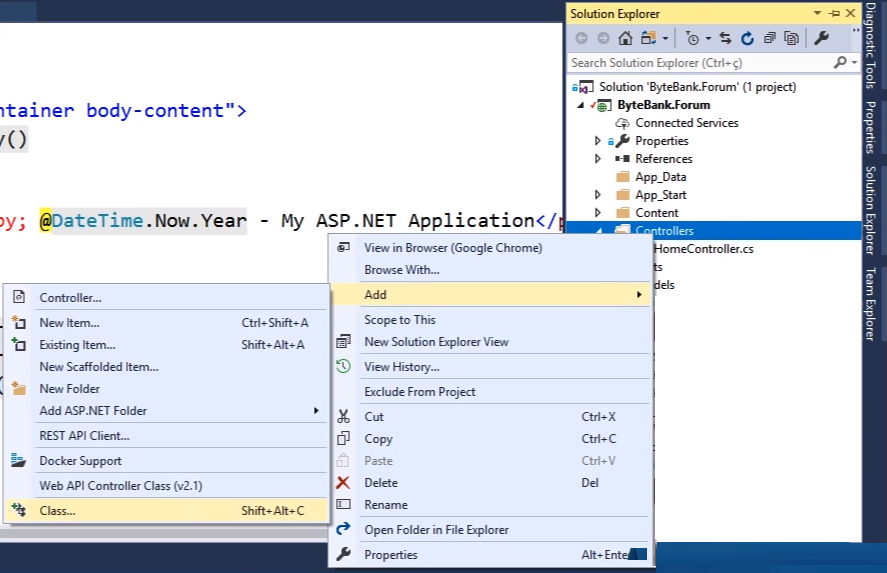
Microsoft.AspNet.Identity.Core

Apenas o Core do AspNet Identity é necessário para começar a usá-lo.

#### Criando a pagina de registro

Legal! Nossa aplicação está criada e com os pacotes instalados. Sendo assim, vamos criar as páginas para usuárias e usuários que, antes de tudo, precisarão se registrar para ter acesso ao fórum do ByteBank. Então, vamos adicionar uma **página de cadastro**.

Considerando que essa página de registro fará parte do controle, de um gerenciamento, da conta de usuário, vamos criar uma Controller com essa finalidade. Para isso, com o menu de "Solution Explorer" aberto à direita, clicaremos com o botão direito do mouse sobre "Controllers" e adicionaremos uma nova classe, selecionando "Add > Class".



Na janela de configuração que se abrirá, nomearemos a classe como ContaController. Para ser uma Controller de verdade, ela precisa herdar a classe Controller do ASP.NET MVC. Mas, ao aplicar a herança, um erro é apontado, e Controller fica sublinhado em vermelho:

using System;

using System.Collections.Generic;

using System.Linq

using System.Web;

namespace ByteBank.Forum.Controllers

{

public class ContaController : Controller

{

}

}

Esse erro acontece porque a classe não foi encontrada. Para corrigi-lo, acionaremos o atalho "Ctrl + **.**" e encontraremos o uso da diretiva System.Web.Mvc como sugestão. A selecionaremos e o código ficará da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq

using System.Web;

using System.Web.Mvc;

namespace ByteBank.Forum.Controllers

{

public class ContaController : Controller

{

}

}

Feito isso, o erro estará corrigido. Continuando, temos ContaController. A motivação para chegar até aqui é ter uma Action para que usuárias e usuários possam se registrar. Sendo assim, adicionaremos Registrar() à nova classe. Considerando que nesse local que usuárias e usuários acessarão e obterão (get) essa página de registro, retornaremos uma View(), até que essa página esteja construída.

public class ContaController : Controller

{

public ActionResult Registrar()

{

return View();

}

}

Bom, mas usuárias e usuários acessarão a página Registrar(), preencherão os campos necessários e, depois, submeterão (submit) as informações do formulário. Para receber essas informações, adicionaremos outra Action, chamada Registrar(). Portanto, podemos fazer uma cópia da primeira e adornar o segundo método com o atributo [HttpPost], já que a diferença entre essas duas é que:

* a primeira será utilizada para acessar a página;
* a segunda será utilizada para dar um post, ou seja, submeter as informações inseridas no formulário.

Assim, com [HttpPost], indicaremos o **verbo** que nossa aplicação ASP.NET deve aceitar nessa Action. Sendo assim, o método Registrar() precisa receber os argumentos preenchidos obtidos no formulário. Então, **vamos criar um objeto** para isso. Como nome — já que estamos falando da Action (Registrar()), da classe ContaController —, escolheremos ContaRegistrarViewModel. O sufixo ViewModel ajudará a manter o código mais organizado. Em seguida, nomearemos o parâmetro como modelo.

public class ContaController : Controller

{

public ActionResult Registrar()

{

return View();

}

[HttpPost]

public ActionResult Registrar(ContaRegistrarViewModel modelo)

{

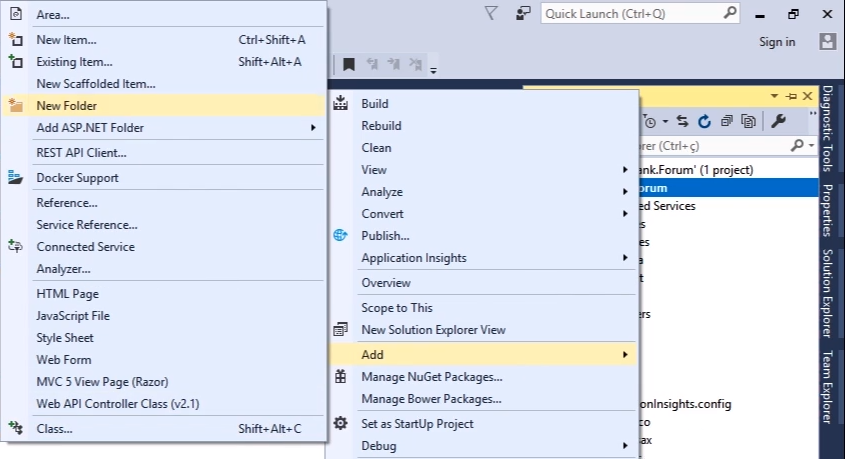
return View();

}

}

Feito isso, ContaRegistrarViewModel ficará sublinhado em vermelho, pois, de fato, ela ainda não existe. Sendo assim, vamos criá-la. É muito comum encontrar esse tipo de classe no diretório "Models". Mas, vamos reservá-lo para modelos que serão compartilhados adiante com o Entity Framework e com o banco de dados. Por enquanto, estamos preocupados em montar a página.

Portanto, para guardar a classe ContaRegistrarViewModel, criaremos um novo diretório, chamado ViewModels. Em "Solution Explorer", à direita, clicaremos com o botão direito do mouse sobre "ByteBank.Forum" e selecionaremos "Add > New Folder".



Na sequência, clicaremos com o botão direito do mouse sobre a nova pasta "ViewModels" e criaremos a classe ContaRegistrarViewModel, selecionando "Add > Class". Com isso, a nova classe se abrirá:

using System;

using System.Collections.Generic;

using System.Linq

using System.Web;

namespace ByteBank.Forum.ViewModels

{

public class ContaRegistrarViewModel

{

}

}

Começaremos definindo os atributos que precisam ser preenchidos no formulário, por exemplo:

* User Name, como no Twitter, no qual utiliza-se @ para mencionar alguém;
* Nome Completo, para mostrar ao lado do título do tópico ou de uma resposta;
* E-mail, para que seja possível entrar em contato com usuárias e usuários;
* Senha, que será definida por usuárias e usuários, por questão de segurança.

Então, vamos adicioná-los como string:

namespace ByteBank.Forum.ViewModels

{

public class ContaRegistrarViewModel

{

public string UserName { get; set; }

public string NomeCompleto { get; set; }

public string Email { get; set; }

public string Senha { get; set; }

}

}

Podemos tornar essa classe ainda mais rica, mais elaborada. Para isso, podemos adornar esses atributos, com características particulares. Por exemplo, todos esses atributos são **requeridos**, ou seja, são obrigatórios. Vamos indicar isso para o ASP.NET com o **data mutation**, um atributo, indicando que são campos de preenchimento obrigatório.

Sendo assim, acima de UserName, adicionaremos [Required].

namespace ByteBank.Forum.ViewModels

{

public class ContaRegistrarViewModel

{

[Required]

public string UserName { get; set; }

public string NomeCompleto { get; set; }

public string Email { get; set; }

public string Senha { get; set; }

}

}

[Required] ficará sublinhado em vermelho, pois o Visual Studio não o localiza. Sendo assim, acionaremos "Ctrl + **.**" para ter acesso à dica de correção, que apontará a necessidade de adicionar a diretiva using de DataAnnotations.

Selecionaremos a devida opção e a lista de diretivas using ficará da seguinte forma:

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.Linq

using System.Web;

Feita a correção, vamos copiar e colar o adorno para todos os atributos:

namespace ByteBank.Forum.ViewModels

{

public class ContaRegistrarViewModel

{

[Required]

public string UserName { get; set; }

[Required]

public string NomeCompleto { get; set; }

[Required]

public string Email { get; set; }

[Required]

public string Senha { get; set; }

}

}

Podemos enriquecer ainda mais esse detalhe sobre os atributos. NomeCompleto, por exemplo, é um termo interno, pois o nome amigável para usuárias e usuários, ou seja, o display que será exibido no momento do cadastro é "Nome", espaço, "Completo". É diferente de NomeCompleto. Então, vamos adicionar o atributo [Display()], abaixo de [Required], que serve para mudar esse tipo de informação. Entre parênteses, adicionaremos Name, que é o atributo que estamos alterando para "Nome completo".

[Required]

[Display(Name = "Nome completo")]

public string NomeCompleto { get; set; }

Continuando, Email também merece um tratamento especial, pois possui input e validação diferentes. Portanto, abaixo do respectivo [Required], adicionaremos [EmailAdress] para indicar para o ASP.NET que trata-se de um campo de e-mail.

[Required]

[EmailAdress]

public string Email { get; set; }

Faremos o mesmo para Senha, indicando o tipo de dado, com [DataType()], ao qual passaremos DataType.Password.

[Required]

[DataType(DataType.Password)]

public string Senha { get; set; }

Muito bem! Criamos a classe ContaRegistrarViewModel — que está bem completa — e a nossa Controller, por meio da classe ContaController.

Inclusive, antes de continuar, em ContaController, precisamos adicionar uma diretiva using para localizar ContaRegistrarViewModel. Sendo assim, com o cursor posicionado no erro apontado, acionaremos "Ctrl + **.**" para adicionar a seguinte diretiva à lista de using:

using ByteBank.Forum.ViewModels;

Com isso, o erro será corrigido. Analisando o que fizemos, modificamos ContaController para ter as duas Actions. Agora, falta só criar a View(). Para isso, em "Solution Explorer", à direita, clicaremos com o botão direito do mouse sobre "Views" e selecionaremos "Add > New Folder" para adicionar um diretório chamado "Conta". Assim, teremos um local para armazenar as Views de ContaController.

Com o diretório criado, clicaremos com o botão direito do mouse sobre e selecionaremos "Add > View" para adicionar a View do método da Action Registrar().Sendo assim, na janela que se abrirá, digitaremos Registrar em "View Name". Manteremos as demais configurações default do Visul Studio.

Antes de clicaremos em "Add", para concluir e adicionar, repare em "Use a layout page:", pois em alguns casos, esse campo está vazio. No caso, temos o padrão Views/Shared/\_Layout.cshtml, mas caso não esteja preenchido, podemos preencher, clicando no botão com reticências (...), à direita.

![(<https://s3.amazonaws.com/caelum-online-public/802-aspnet-identity-parte-1/01/01.05_003_add-view.png>)

Na janela que se abrirá, podemos navegar pelos diretórios "View > Shared", à esquerda. À dreita, encontraremos o \_Layout.cshtml. Ao selecioná-lo, o campo será preenchido. Esse layout é o responsável pelo menu e nota de rodapé.

Feitas as configurações, vamos alterar a view Registrar.cshtml que, ao abrir, estará assim:

@{

ViewBag.Title = "Registrar";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>Registrar</h2>

Começaremos definindo o modelo, na primeira linha, com:

@model ByteBank.Forum.ViewModels.ContaRegistrarViewModel

Na sequência, **criaremos o formulário**. Para isso, abaixo de <h2>Registrar</h2>, utilizaremos o bloco using, com Html.BeginForm.

@using (Html.BeginForm())

{

}

Assim, iniciamos o formulário. Entre as chaves ({}), podemos inserir uma mensagem de boas vindas, com orientações do que deve ser feito. Utilizaremos <hr /> para pular uma linha, melhorando a estética da página.

O código inteiro ficará da seguinte forma:

@model ByteBank.Forum.ViewModels.ContaRegistrarViewModel

@{

ViewBag.Title = "Registrar";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>Registrar</h2>

@using (Html.BeginForm())

{

<h4> Complete o cadastro de sua conta! </h4>

<hr />

}

Criamos diversos atributos para mostrar informação de validação, mas ainda precisamos mostrar o resultado dessa validação em algum lugar da nossa página. Para isso, em using com Html.BeginForm, utilizaremos ValidationSummary(), que resumirá todas as mensagens de erro, caso alguma informação esteja errada nessa div, referente ao HTML Helper, que será criado na página.

Utilizaremos uma sobrecarga diferente, a utilizaremos com parâmetros, de forma que permita mudar os atributos da div que será renderizada pelo HTML Helper. Sendo assim, manteremos o primeiro parâmetro, referente à mensagem, vazio, porque ele não nos importa. O que nos importa é o segundo parâmetro, que permitirá a alteração da classe que será utilizada nessa div. Para isso, utilizaremos um objeto anônimo e, entre as chaves ({}), colocaremos um atributo chamado class ao qual atribuiremos o valor que deve ser renderizado na classe dessa div, que é do Bootstrap "text-danger. Assim, daremos um visual mais bonito, uma cara mais interessante para nossa aplicação.

@model ByteBank.Forum.ViewModels.ContaRegistrarViewModel

@{

ViewBag.Title = "Registrar";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>Registrar</h2>

@using (Html.BeginForm())

{

<h4> Complete o cadastro de sua conta! </h4>

<hr />

@Html.ValidationSummary("", new { @class = "text-danger" } )

}

Agora, precisamos, de fato, de um formulário para que usuárias possam editar e preencher. Para isso, adicionaremos outro HTML Helper, no caso, o EditorForModel() e, dessa forma, com base em tudo que foi preenchido, o ASP.NET será capaz de criar todos os campos.

@model ByteBank.Forum.ViewModels.ContaRegistrarViewModel

@{

ViewBag.Title = "Registrar";

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>Registrar</h2>

@using (Html.BeginForm())

{

<h4> Complete o cadastro de sua conta! </h4>

<hr />

@Html.ValidationSummary("", new { @class = "text-danger" } )

@Html.EditorForModel()

}

Legal! Usuárias e usuários já possuem o campo de validação e uma forma de preencher os dados. Agora, só falta um botão para clicar e submeter (submit) as informações. Para isso, abaixo de EditorForModel(), criaremos uma div, adicionando outra classe do Bootstrap, no caso, "form-group". Para posicionar, criaremos mais uma div, utilizndo col-md-offset-2 e o col-md-10. Abaixo, colocaremos, enfim, nosso input, que será do tipo submit, com um valor (value) igual a "Registrar" e também colocaremos uma classe Bootstrap, "btn btn-default", para deixar o visual mais legal.

<div class="form-group">

<div class="col-md-offset-2 col-md-10">

<input type="submit" value="Registrar" class="btn btn-default" />

</div>

</div>

}

Então, já criamos nossa ViewModels e a Controller. Agora, como essa página será acessada? Precisaremos adicionar a opção de "Registrar" ao menu. Vamos colocá-la no canto superior direito. Para isso, em "Solution Explorer", acessaremos "Views > Share > [@] \_Layout.cshtml". Nesse arquivo, encontraremos o menu superior e a nota de rodapé da página. Sendo assim, na parte de menu, na div de navbar-collapse collapse, adicionaremos uma opção.

Como essa opção será posicionada à direita, vamos separá-la das outras opções. Para facilitar, podemos copiar e colar uma das linhas do código acima para indicar que colocaremos "Registrar". Como:

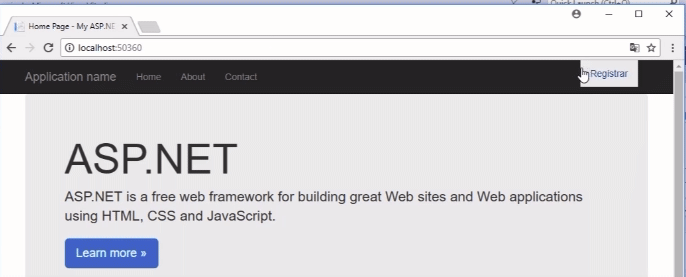
* primeiro parâmetro, passaremos o texto que será visualizado por usuárias e usuários, no caso, "Registrar";
* segundo parâmetro, passaremos a Action para a qual a URL apontará, no caso, Registrar, da forma que definimos na Controller.
* terceiro parâmetro, passaremos a nossa Controller, ContaController.

<ul class="nav navbar-right">

<li>@Html.ActionLink("Registrar", "Registrar", "Conta")</li>

</ul>

Beleza! Vamos executar? Daremos um start, clicando em "Google Chrome" no menu superior. Com a aplicação rodando, perceberemos que o estilo de botão de "Registrar" ficou diferente dos outros.



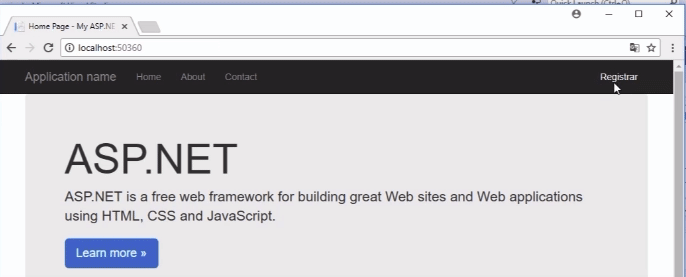
Ao analisarmos o código para entender o que aconteceu, perceberemos que, ao criar a lista não enumerada nav navbar-nav navbar-right, esquecemos de adicionar a classe navbar-nav. Inserimos somente a navbar-right. Sendo assim, antes dessa última, adicionaremos a que falta:

<ul class="nav navnavbar-nav navbar-right">

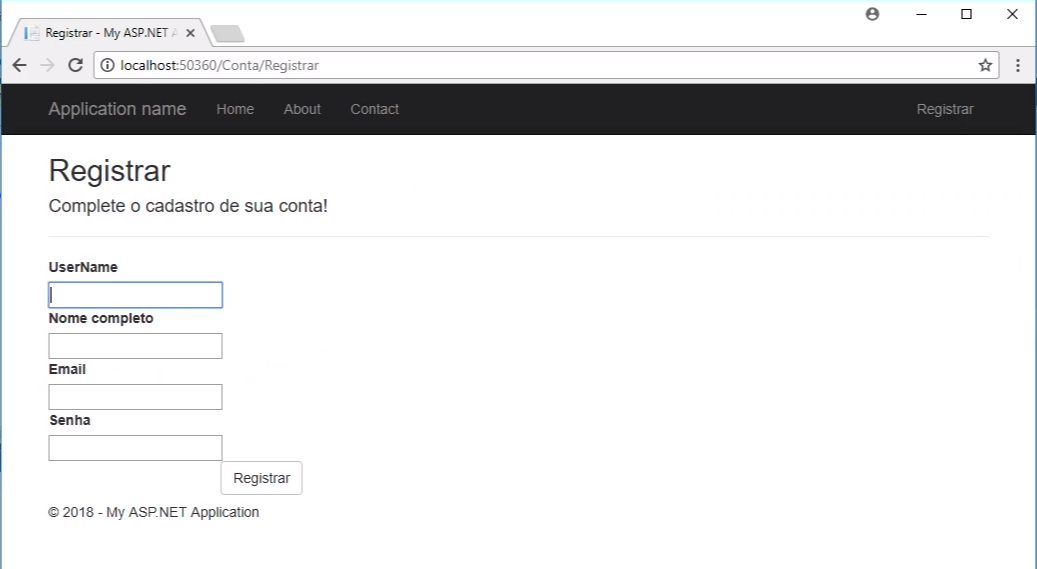
<li>@Html.ActionLink("Registrar", "Registrar", "Conta")</li>

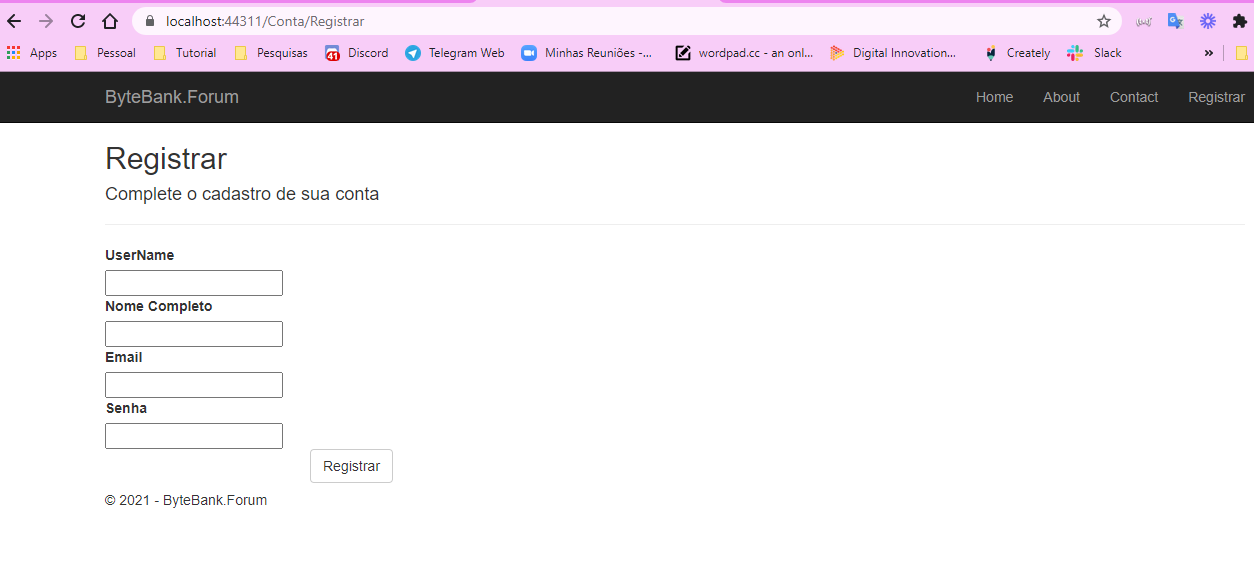
</ul>

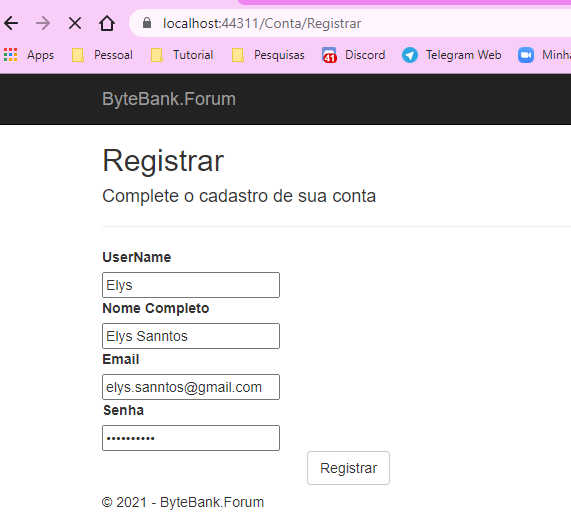
Feito isso, salvaremos a alteração, voltaremos à página e a atualizaremos ("F5"). Beleza! Agora o botão "Registrar" ficou igual aos outros.



Ao clicar nele, encontraremos e poderemos preencher todos os campos, de acordo com o que criamos na ContaRegistrarViewModelViewModel.

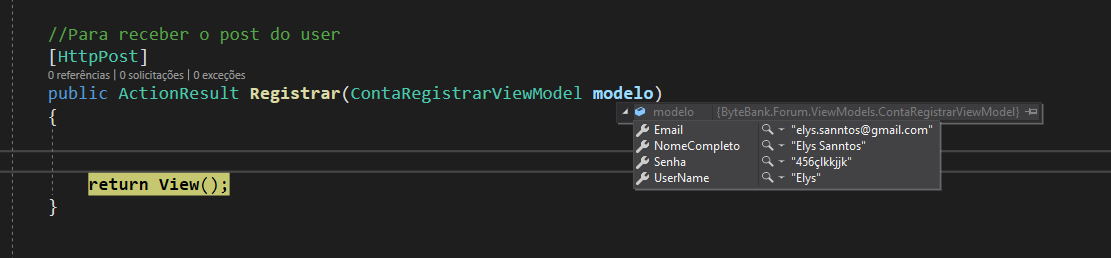






Vamos adicionar um break point em ContaController para ter certeza de que está batendo nela, como esperamos. Feito isso, voltaremos à página com os campos preenchidos e clicaremos em "Registrar".

Maravilha! Ao voltar ao código de ContaController, o break point terá interrompido a execução. Ao posicionar o mouse sobre modelo, será possível visualizar os campos com suas respectivas informações, caso suas caixas tenham sido preenchidas.



Beleza! Adiante, vamos implementar uma lógica, para podermos salvar e persistir as informações de usuárias e usuários.

#### O que aprendemos?

Neste capítulo, progredimos solucionando nossos problemas e aprendemos bastante:

**O que é o AspNet Identity:**

* Vimos que o AspNet Identity é um framework para o gerenciamento de identidades de usuários e tarefas como verificar senhas, nomes de usuários, e-mails, etc ficam muito mais simples!.

**Instalamos o AspNet Identity em um projeto criado do zero:**

* Vimos que para instalar o AspNet Identity em um novo projeto basta usar o comando Install-Package Microsoft.AspNet.Identity.Core.

#### Consolidando seu conhecimento

Chegou a hora de você seguir todos os passos realizados por mim durante esta aula. Caso já tenha feito, excelente. Se ainda não, é importante que você implemente o que foi visto no vídeo para poder continuar com o próximo capítulo que tem como pré-requisito todo código aqui escrito.

Se por acaso você já domina esta parte, em cada capítulo você terá a opção de baixar o projeto feito até aquele ponto. Você encontrará o link para download na próxima explicação do capítulo.

#### Aula 02: Arquitetura do AspNet Identity

#### Projeto da aula anterior

Aqui você pode baixar o projeto do curso no ponto em que paramos na aula anterior.

[Baixe o zip](https://github.com/alura-cursos/aspnet-identity-parte-1/archive/aula-01.zip) ou [visualize os arquivos](https://github.com/alura-cursos/aspnet-identity-parte-1/tree/aula-01) no Github!

#### IdentityDbContext e IdentityUser

Antes de incluir usuárias e usuários no banco de dados, precisamos ter certeza de que as informações inseridas no formulário estão corretas. Ou seja, é necessário conferir se:

* os campos que são requeridos — obrigatórios — foram preenchidos;
* o campo de e-mail foi preenchido com o formato esperado.

Como estamos utilizando o ASP.NET MVC, em ContaController, podemos acionar uma propriedade chamada ModelState. Com ela, é possível saber se o estado do nosso modelo é válido ou não, por meio do atributo IsValid. Se IsValid for verdadeiro (True), é sinal de que os campos estão devidamente preenchidos. Caso contrário, alguma informação está errada. Sendo assim, vamos colocar isso dentro de um if.

Se (if) ModelState.IsValid, poderemos incluir o usuário. Vamos adicionar esse comentário (//) ao bloco. Depois que incluirmos usuárias e usuários, poderemos direcioná-los à *home page* do fórum, com return RedirectToAction() e passar a Index de Home, no caso, a *home* da Controller.

[HttpPost]

public ActionResult Registrar(ContaRegistrarViewModel modelo)

{

if(ModelState.IsValid)

{

// Podemos incluir o usuario

return RedirectToAction("Index", "Home");

}

return View();

}

Legal! Agora, se esse bloco não for executado, algo errado aconteceu. Podemos comentar (//) logo após o fechamento das chaves (}) de if. Depois do comentário, mostraremos às usuárias e usuários, devolvendo o modelo que recebemos.

[HttpPost]

public ActionResult Registrar(ContaRegistrarViewModel modelo)

{

if(ModelState.IsValid)

{

// Podemos incluir o usuario

return RedirectToAction("Index", "Home");

}

// Alguma coisa de errado aconteceu!

return View(modelo);

}

Com essa lógica preparada, vamos nos preocupar em **como incluir** usuárias e usuários no banco de dados. O nosso ORM — ou seja, esse nosso banco de dados — é o Entity Framework, o que é natural, considerando que estamos trabalhando com o ASP.NET MVC e o Identity. Mas isso não significa que o uso desse banco de dados é obrigatório. Inclusive, como sabemos, a implementação do Entity Framework para o Identity é feita por pacotes separados no NuGet. Isso significa que o núcleo — *core*, parte central — do Identity é independente da tecnologia de ORM que utilizamos. A adequação pode ser feita tanto para o Entity Framework, quanto para o NHibernate, ou para outra solução com banco de dados não relacional.

Neste curso, como utilizaremos o Entity Framework, precisaremos de um dbContext em if. Portanto, será que devemos criar uma classe derivada de dbContext e, nessa classe derivada de dbContext, teremos que colocar um IDbSet do usuário e, então, definir para esse usuário todas essas coisas inerentes ao uso de um dbContext? Na verdade, não. Usamos o pacote do NuGet com o Identity implementado no Entity Framework, então podemos usar essa solução em dbContext, sem precisar criar (new) uma classe derivada dela. Podemos usar o próprio IdentityDbContext().

[HttpPost]

public ActionResult Registrar(ContaRegistrarViewModel modelo)

{

if(ModelState.IsValid)

{

var dbContext = new IdentityDbContext();

// Podemos incluir o usuario

return RedirectToAction("Index", "Home");

}

// Alguma coisa de errado aconteceu!

return View(modelo);

}

Maravilha! Vamos conferir o conteúdo dele? Abaixo da declaração de dbContext, digitaremos dbContext, ponto (**.**) e procuraremos por Users, por exemplo:

dbContext.Users

Com isso, encontraremos um DbSet de Users, já nesse IdentityDbContext(), sem precisar nos preocupar com a modelagem de usuário. E qual é o tipo? Se analisarmos o DbSet, veremos que o tipo é <IdentityUser>:

System.Data.Entity.IDbSet<IdentityUser> IdentityDbContext<IdentityUser, IdentityRole, string, IdentityUserLogin, IdentityUserRole, IdentityUserClaim>.Users{get; set;}

IDbSet of Users

Ah, então, vamos utilizar <IdentityUser>. Para deixar o código mais limpo, utilizaremos uma nova variável, novoUsuario, abaixo da variável dbContext.

[HttpPost]

public ActionResult Registrar(ContaRegistrarViewModel modelo)

{

if(ModelState.IsValid)

{

var dbContext = new IdentityDbContext();

var novoUsuario = new IdentityUser();

dbContext.Users

// Podemos incluir o usuario

return RedirectToAction("Index", "Home");

}

// Alguma coisa de errado aconteceu!

return View(modelo);

}

Assim, seguiremos a definição que consultamos em IdentityDbContext(). Na linha abaixo, vamos digitar novoUsuario.Email para conferir se novoUsuario possui Email, e constataremos que já tem. Maravilha! Nem precisaremos alterar o IdentityUser, pois já possui Email. Sendo assim, podemos continuar a escrita da linha, atribuindo (=) modelo.Email:

[HttpPost]

public ActionResult Registrar(ContaRegistrarViewModel modelo)

{

if(ModelState.IsValid)

{

var dbContext = new IdentityDbContext();

var novoUsuario = new IdentityUser();

novoUsuario.Email = modelo.Email;

dbContext.Users

// Podemos incluir o usuario

return RedirectToAction("Index", "Home");

}

// Alguma coisa de errado aconteceu!

return View(modelo);

}

E será que novoUsuario tem UserName também? Aquele campo, que funciona como o do Twitter, para fazer referências? Vamos conferir digitando novoUsuario.UserName, abaixo de novoUsuario.Email. Feito isso, constataremos que também possui UserName. Sendo assim, vamos pegar o nosso modelo.UserName e atribuir a novoUsuario.UserName:

[HttpPost]

public ActionResult Registrar(ContaRegistrarViewModel modelo)

{

if(ModelState.IsValid)

{

var dbContext = new IdentityDbContext();

var novoUsuario = new IdentityUser();

novoUsuario.Email = modelo.Email;

novoUsuario.UserName = modelo.UserName;

dbContext.Users

// Podemos incluir o usuario

return RedirectToAction("Index", "Home");

}

// Alguma coisa de errado aconteceu!

return View(modelo);

}

Agora , falta o só NomeCompleto. Vamos conferir? Abaixo de UserName, digitaremos novoUsuario.FullName — utilizamos o termo inglês para "Nome completo" — e teremos um problema, pois nem aparecerá a opção de autocompletar para nós. Mesmo se tentarmos somente com Name, não encontraremos o que estamos procurando. Então, nosso IdentityUser não possui o que precisamos, no caso, um equivalente para NomeCompleto.

Vamos fazer uma análise? Posicionaremos o cursor em IdentityUser e pressionaremos a tecla "F12". Na nova aba que se abrirá, repare que IdentityUser é uma classe bem simples e só possui esses dois construtores:

public IdentityUser();

public IdentityUser(string UserName);

Podemos pensar que é porque, na verdade, tudo está na IdentityUser de tipo genérico, que armazena esses dois construtores:

namespace Microsoft.AspNet.Identity, EntityFramework

{

public class IdentityUser : IdentityUser< string, IdentityUserLogin, IdentityUserRole, IdentityUserClaim>, IUser, IUser<string>

{

public IdentityUser();

public IdentityUser(string UserName);

}

}

Se posicionarmos o cursor nessa IdentityUser e pressionarmos "F12" para analisar, aí sim, encontraremos um monte de argumentos, que são ainda mais restritivos e herdam de outra classe, que também é genérica. Está ficando complicado. Mas tudo bem. Vamos voltar para ContaController. Seremos obrigados a utilizar IdentityUser porque dbContext.Users é uma tabela de IdentityUser, como vimos no IDbSet de Users anteriormente. Não adianta estendermos essa classe porque dbContext não encontrará nossas novas propriedades.

Mas será que é necessária toda essa dificuldade para estender o Identity? Vamos posicionar o cursor em IdentityDbContext e pressionar "F12" para analisá-lo. Constataremos que também é uma classe bem simples, composta somente por construtores. No entanto, repare na herança (:) dela, que é um IdentityDbContext com diversos tipos genéricos. O primeiro deles é o TUser, tipo que será utilizado para representar usuários no banco de dados. Na seguinte linha, por exemplo, ele é utilizado naquele IDbSet que vimos de Users:

public virtual IDbSet<TUser> Users { get; set; }

Legal! Poxa, mas para alterar o tipo de usuário, teremos que nos preocupar com todos esses outros tipos e amarrações complexas? Na verdade, isso é algo que nos depararemos com muita frequência aqui no Identity. Temos a **solução mais genérica possível** para que seja fácil estender e testar. Mas também temos uma que é mais **geral**, no caso, a IdentityDbContext, que não tem tipos genéricos. E também encontraremos diversas vezes a **solução intermediária**, que possui somente um tipo genérico, no caso, o tipo do usuário (TUser).

Sendo assim, vamos criar um tipo de usuário que deriva de IdentityUser, que terá NomeCompleto? Em "Solution Explorer", à direita, clicaremos com o botão direito do mouse sobre o diretório "Models" e selecionaremos "Add > Class". A nomearemos como UsuarioAplicacao e clicaremos em "Add". Feito isso, teremos a nova classe:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace ByteBank.Forum.Models

{

public class UsuarioAplicacao

{

}

}

Com base nas análises que fizemos, essa classe deve herdar IdentityUser, afinal, nela encontramos e-mail, telefone e todos os campos que serão necessários futuramente. Após aplicar a herança, acionaremos "Ctrl + **.**" para adicionar a diretiva using necessária e adicionaremos a UsuarioAplicacao uma propriedade, uma string de NomeCompleto.

using Microsoft.AspNet.Identity.EntityFramework;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace ByteBank.Forum.Models

{

public class UsuarioAplicacao : IdentityUser

{

public string NomeCompleto { get; set; }

}

}

Então, em ContaController, usaremos UsuarioAplicacao como tipo genérico (<>) do nosso IdentityDbContext. Lembrando de adicionar a respectiva diretiva using, acionando o "Ctrl + **.** " para corrigir o erro apontado pelo sublinhado vermelho abaixo de UsuarioAplicacao.

Além disso, novoUsuario não é mais IdentityUser. Agora é UsuarioAplicacao. Manteremos Email e UserName, e ganharemos um novo campo, no caso, NomeCompleto, que é igual a modelo.NomeCompleto. E, em dbContext, podemos adicionar Add() o novoUsuario, após Users. Feito isso, precisaremos salvar as alterações, não é mesmo? Sendo assim, abaixo da adição de novoUsuario, salvaremos as informações com SaveChanges.

[HttpPost]

public ActionResult Registrar(ContaRegistrarViewModel modelo)

{

if(ModelState.IsValid)

{

var dbContext = new IdentityDbContext<UsuarioAplicacao>();

var novoUsuario = new UsuarioAplicacao();

novoUsuario.Email = modelo.Email;

novoUsuario.UserName = modelo.UserName;

novoUsuario.NomeCompleto = modelo.NomeCompleto;

dbContext.Users.Add(novoUsuario);

dbContext.SaveChanges();

// Podemos incluir o usuario

return RedirectToAction("Index", "Home");

}

// Alguma coisa de errado aconteceu!

return View(modelo);

}

Bom, estamos estudando bastante sobre o Entity Framework com todas essas alterações que fizemos, utilizando SaveChanges, IDbSet, convenções de nome DbContext. Conferimos que o Entity Framework **não é obrigatório** para o Identity, porém o utilizamos nesse curso para esse projeto.

**Dica:** Caso queira aprofundar seu conhecimento em Entity Framework, aqui na [Alura](http://www.alura.com.br/" \t "_blank) você encontra uma [Carreira para Desenvolvedor .Net com Entity Framework](https://www.alura.com.br/carreira-desenvolvedor-net-com-entity-framework). Nela, encontram-se diversos cursos para se aprofundar nesse assunto.

Mas, nesse curso, entenderemos somente aquilo que utilizarmos do Entity Framework. Caso queira saber mais sobre, recomendo fortemente a carreira indicada logo acima.

Continuando, entendemos que podemos utilizar o IdentityDbContext para não nos preocuparmos com todas as tabelas do Identity, nem com a modelagem de usuário. Além disso, descobrimos que as implementações do Identity que fizemos são bem genéricas, então podemos usar com o nosso tipo de usuário, no caso, UsuarioAplicacao.

No entanto, estamos acoplando nossa aplicação com o Entity Framework, porque estamos criando nosso contexto (IdentityDbContext) e lidando com SaveChanges, propriedade desse banco de dados, e não está legal. Lembra que a intenção de utilizar o Identity era descomplicar? Adiante, exploraremos formas de melhorar esse código.

#### xxxxxxxx

#### Aula 03: Owin

#### Aula 04: Validações de usuário

#### Aula 05: Serviços de email

# Principais comandos

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Conflitos

# Conclusão