**Introdução:**

Olá!

Sou Marcelo Oliveira, bem-vindo ao curso Asp.NET Core 2.0 parte II da **Alura**. Nesse curso, continuaremos incrementando o e-commerce do nosso cliente "Casa do Código" com a ajuda dos recursos do Asp.NET Core 2.0. Começaremos implementando o clique dos botões "+" e "-" do carrinho de compras, para adicionar ou remover itens.

Essa implementação irá nos proporcionar muitos conhecimentos novos, por exemplo, chamar uma função JavaScript pelo evento do clique no clique do HTML. Selecionaremos elementos do HTML por meio do plug-in do Jquery. Aprenderemos a diferença entre uma requisição com método post() e get(). Veremos como realizar uma requisição através do método httppost(), com a ajuda da função ajax() do Jquery.

Aprenderemos a organizar melhor nosso código por meio de uma classe do JavaScript, e começaremos a lidar com o evento de digitação da quantidade de elementos no carrinho de compras.

Ainda, prepararemos no servidor o recebimento da função ajax() pelo controller, em segida guardaremos as informações no banco de dados. Aprenderemos a implementar um padrão chamado view model, isto é, um modelo específico para fornecer informações para a <view>. Retornaremos um objeto específico para a requisição ajax(), e por meio da resposta obtida atualizaremos toda a <view>.

Posteriormente, trabalharemos com os campos de cadastro. Aprenderemos a redirecionar o usuário para outra action caso tenhamos algum problema de validação nos campos, assim garantimos que todos eles estejam preenchidos e com dados corretos, pois criaremos validações customizadas para cada item.

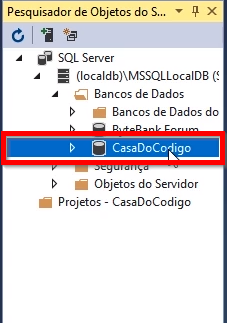
Criaremos um formulário no Razor por meio de uma TagHelper. Pegaremos a informação que foi enviada para o servidor e gravar no banco de dados, e depois veremos como proteger nossa action contra acessos indevidos utilizando o recurso de validação do tolken. Iremos aprender a proteger o método que receberá a chamada ajax() contra acessos maliciosos.

Finalmente, renderizaremos a <view> de resumo com os dados de cadastro do usuário.

Espero que vocês gostem do curso e não deixem de participar do fórum!

Vamos lá?!

**Aula 1 – Resumo 1:** Usaremos o mesmo projeto iniciado no curso Asp.NET Core 2.0 parte I, portanto copiaremos esse arquivo e o utilizaremos na segunda etapa do curso. Havíamos criado um banco de dados na primeira parte do curso, mas ele será apagado nesta etapa. No Visual Studio, clicaremos sobre "Exibir" no cabeçalho de ferramentas, e selecionaremos a opção "Pesquisador de Objetos no SQL Server". Assim, localizaremos o banco de dados CasaDoCodigo e o excluiremos. Basta clicar sobre ele com o botão direito e selecionar a opção "Excluir".

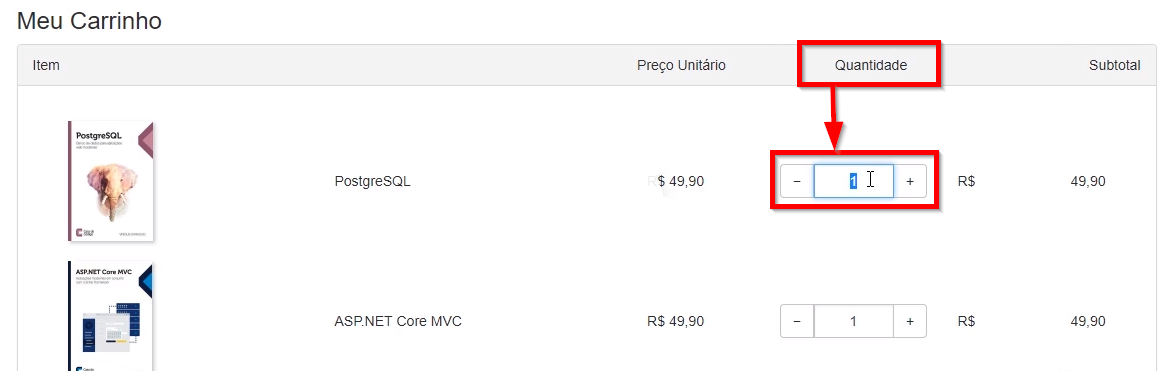


Deletamos esse banco de dados porque nós fizemos uma cópia do arquivo e trocamos o nome do *name space*, isso pode gerar problemas no momento em que precisarmos trabalhar com o banco.

Depois que deletamos o banco de dados iremos executar a aplicação.A aplicação rodará no *browser* ,e teremos o catálogo de livros da Casa do Código e a opção de adiciona-los ao carrinho de compras por via do botão "Adicionar". Na primeira opção apresentada, o livro "ASP.NET Core MVC", clicaremos sobre o botão "Adicionar".



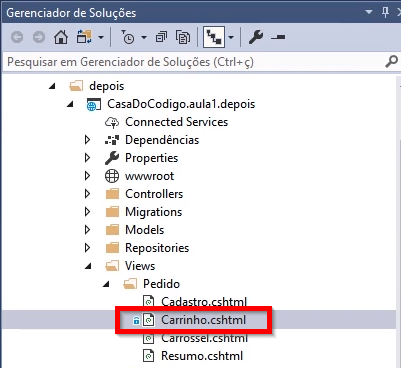
Adicionaremos mais alguns livros diferentes no carrinho. Ao realizarmos essa ação, somos direcionados para a página que exibe todos os itens que estamos comprando. Contudo, temos a opção de "Quantidade" de livros, e esta não pode ser alterada diretamente ao clicarmos no botão "+", ficando sempre com o valor 1 fixo.



Nosso objetivo é fazer com que a aplicação responda ao clique no botão "+" para podermos aumentar as quantidades de itens no carrinho.

Até o momento trabalhamos com o código apenas do lado do servidor, o chamado Server Side. Neste ponto do curso, iremos fazer uma implementação do lado do cliente, utilizaremos a linguagem JavaScript, que somente pode ser executada o *browser*.

Abriremos novamente o Visual Studio e acessaremos a view de carrinho. Na área de gerenciamento de soluções, clicaremos sobre "Views > Carrinho.cshtml".



Iremos até o final da view e inseriremos o código JavaScript que será executado do browser. Não podemos simplesmente adicionar o script no código , primeiramente precisamos adicionar a tag <script> que por sua vez necessita de um tipo (type), no caso, text/javascript

@{

ViewData["Title"] = "Carrinho";

}

<div class="row">

<div class="col-md-12">

<div class="pull-right">

<a class="btn btn-success" asp-action="carrossel">

Adicionar Produtos

</a>

<a class="btn btn-success" asp-action="resumo">

Finalizar Pedido

</a>

</div>

</div>

</div>

<script type="text/javascript">

</script>

Dentro dessa tag iremos inserir o código JavaSript. Porém, há um detalhe: não podemos inserir uma tag <script> dentro de uma view do Razor, como é o casso de Carrinho.cshtml. Precisamos criar uma sessão para incluir esse script, para isso, deveremos criar uma diretiva @section que terá o nome de Scripts. Essa sessão terá duas chaves, e dentro delas inseriremos a tag <script>.

@{

ViewData["Title"] = "Carrinho";

}

<div class="row">

<div class="col-md-12">

<div class="pull-right">

<a class="btn btn-success" asp-action="carrossel">

Adicionar Produtos

</a>

<a class="btn btn-success" asp-action="resumo">

Finalizar Pedido

</a>

</div>

</div>

</div>

@section Scripts

{

<script type="text/javascript">

</script>

}

Com isso, podemos realmente começar a escrever nosso código JavaScript.

O objeto é fazer com que aplicação responde ao clique do usuário no botão "+", tal responde deve ser criada por meio de uma função (function), que terá o nome de clickIncremento().

@{

ViewData["Title"] = "Carrinho";

}

@section Scripts

{

<script type="text/javascript">

function clickIncremento(){

}

</script>

}

Iremos até o trecho de código responsável pelo botão "+" na view, que é button class="btn btn-default".

@{

ViewData["Title"] = "Carrinho";

}

@foreach (var item in Model)

{

<div class="row row-center linha-produto">

<div class="col-md-3">

<img class="img-produto-carrinho" src="/images/produtos/large\_@(item.Produto.Codigo).jpg" />

</div>

<div class="col-md-3">@(item.Produto.Nome)</div>

<div class="col-md-2 text-center">R$ @(item.PrecoUnitario)</div>

<div class="col-md-2 text-center">

<div class="input-group">

<span class="input-group-btn">

<button class="btn btn-default">

<span class="glyphicon-minus"></span>

</button>

</span>

<input type="text" value="@(item.Quantidade)"

class="form-control text-center" />

<span class="input-group-btn">

<button class="btn btn-default">

<span class="glyphicon-plus"></span>

</button>

</span>

</div>

</div>

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

R$ <span class="pull-right" subtotal>

@(item.Quantidade \* item.PrecoUnitario)

</span>

</div>

</div>

}

Adicionaremos um atributo ao elemento <button> para o evento do click. Esse atributo é chamado de onclick, e ele receberá a função JavaScript que acabamos de criar, a clickIncremento().

@{

ViewData["Title"] = "Carrinho";

}

<button class="btn btn-default">

onclick="clickIncremento()">

<span class="glyphicon-plus"></span>

</button>

Voltaremos ao browser a atualizaremos a página utilizando o atalho "Ctrl + F5" e testaremos se a nossa modificação foi bem sucedida. Ao clicaremos sobre o botão de "+" veremos que nenhuma ação é realizada. Isso ocorre porque não solicitamos nenhuma ação, nós iremos inserir um *break point* no código JavaScript da view. Inseriremos o comando debugger.

@{

ViewData["Title"] = "Carrinho";

}

@section Scripts

{

<script type="text/javascript">

function clickIncremento(){

debugger;

}

</script>

}

Ao testarmos novamente o botão "+", notaremos que nada ocorre. Para que o comando debugger funcione, devemos entrar nas ferramentas do desenvolvedor utilizando o atalho "F12" no browser. Uma nova área irá surgir na parte inferior da tela. Clicaremos no botão "+" e teremos acesso ao seguinte trecho de código:

<script type="text/javascript">

function clickIncremento() {

debugger;

}

</script>

</body>

</html>

Estamos parados da linha do comando debugger. Veremos nas próximas aulas como implementar o código JavaScript em nossa aplicação.

**Resumo 2:** Realizamos o primeiro teste de interação entre o usuário e botão "+", que por sua vez chama função JavaScript por meio do evento onclick, entenderemos o porquê devemos sempre inserir o script dentro de uma view do Asp.NET Core dentro de uma sessão de Script.

@{

ViewData["Title"] = "Carrinho";

}

@section Scripts

{

<script type="text/javascript">

function clickIncremento() {

debugger;

}

</script>

}

Voltemos até o browser, na página de carrinho que exibe todos os itens que compramos. Iremos analisar o código html gerado da página, para isso, basta clicar com o botão direito do mouse e selecionar a opção "Exibir código fonte da página" ou pressionar o atalho "Ctrl + U". Ao final do código, veremos a função clickIncremento() inserida. Temos os botõesAdicionar Produtos e Finalizar Pedido,em seguida temos o rodapé da nossa página que foi gerado pelo layout.cshtml, a view que envolve todo o código html. Dois, temos os scripts utilizados na página, e ao final teremos o script específico para nossa página.

<!DOCTYPE html>

<\*\*\*\*!\*\*\*\*>

<div class="row">

<div class="col-md-12">

<div class="pull-right">

<a class="btn btn-success" asp-action="carrossel">

Adicionar Produtos

</a>

<a class="btn btn-success" asp-action="resumo">

Finalizar Pedido

</a>

</div>

</div>

</div>

<hr />

<footer>

<p>&copy; 2018 - Casa do Código</p>

</footer>

</div>

<script src="/lib/jquery/dist/jquery.js"></script>

<script src="/lib/jquery-validation/dist/jquery.validade.js"></script>

<script src="/lib/jquery-validation-

unobtrusive/jquery.validade.unobtrusive.js"></script>

<script src="/lib/bootstrap/dist/js/bootstrap.js"></script>

<script src"/js/site.js?v=8ZRc1sGeVrPBx41D717BgRaQeKyh78QKV9SKsdt638U">

</script>

<script type="text/javascript">

function clickIncremento() {

debugger;

}

</script>

}

</body>

</html>

O que Asp.Net Core faz é lançar a tag de script para o final da página, caso a ordem fosse outra, seria, executados os outros scripts principais da aplicação, e isso geraria uma série de erros e teremos dependências que ainda não forma executadas. Por isso é muito importante inserir o código JavaScript dentro de uma sessão.

De volta ao código html da página, notaremos os itens do carrinho que são exibidos como uma série de <div>s e <span>s, e não mencionam exatamente quais são os itens que estamos trabalhando.

<!DOCTYPE html>

<\*\*\*\*!\*\*\*\*>

<div class="col-md-3">PostgreeSQL</div>

<div class="col-md-2 text-center">R$ 49,90</div>

<div class="col-md-w text-center">

<div class="input-group">

<span class="input-group-btn">

<button class="btn btn-default">

<span class='glyphicon-minus"></span>

</button>

<\*\*\*\*!\*\*\*\*>

A única informação que possuímos é o nome, no caso PostgreeSQL, mas isso não é relevante. O que precisamos saber é qual o item que foi clicado no botão "+". Para identificarmos cada um dos itens iremos colocar dentro de um elemento <div> um atributo com o identificador de cada item.

Na view Carrinho.cshmtl localizaremos a <div> que envolve cada um dos itens. Essa <div> possui a classe linha produto.

@{

ViewData["Title"] = "Carrinho";

}

@foreach (var item in Model)

{

<div class="row row-center linha-produto">

<div class="col-md-3">

<img class="img-produto-carrinho" src="/images/produtos/large\_@(item.Produto.Codigo).jpg" />

</div>

<\*\*\*\*!\*\*\*\*>

Adicionaremos outro atributo, que chamaremos de item-id, que será igual ao id do item pedido. Inseriremos, portanto, `@item.Id`.

@{

ViewData["Title"] = "Carrinho";

}

@foreach (var item in Model)

{

<div class="row row-center linha-produto" item-id="@item.Id">

<div class="col-md-3">

<img class="img-produto-carrinho" src="/images/produtos/large\_@(item.Produto.Codigo).jpg" />

</div>

<\*\*\*\*!\*\*\*\*>

Desse modo estamos inserido um código que será obtido do banco de dados, ou seja, estamos lançando uso do Razor para que ele renderize o número de identificação do item correspondente.

Salvaremos as modificações realizadas, atualizaremos a página e novamente iremos verificar o código fonte. Notaremos a presença do novo atributo item-id.

<!DOCTYPE html>

<\*\*\*\*!\*\*\*\*>

<div class="row row center linha-produto" item-id"8">

<div class="col-md-3">

Na view Carrinho.cshtml passaremos o item-id para o código JavaScript, de forma que esse item possa ser trabalhado e o servidor possa receber a informação e atualizar a quantidade de itens no carrinho. No evento onclick teremos a função clickIncremento(), para esta passaremos o contexto por meio da palavra this,.

@{

ViewData["Title"] = "Carrinho";

}

<input type="text" value="@(item.Quantidade)"

class="form-control text-center" />

<span class="input-group-btn">

<button class="btn btn-default">

onclick="clickIncremento(this)">

<span class="glyphicon-plus"></span>

</button>

</span>

Além disso, devemos modificar a assinatura da função JavaScript, passando para clickIncremento() um parâmetro btn, isto é, o botão que foi clicado pelo usuário. Em seguida, removeremos o comando debugger.

@{

ViewData["Title"] = "Carrinho";

}

@section Scripts

{

<script type="text/javascript">

function clickIncremento(btn) {

}

</script>

}

Para acessarmos o atributo id a partir desse elemento, usaremos a biblioteca **jQuery**, que é um componetente que acompanha a instalação do Asp.NET Core. O jQuery fornece diversas ferramentas que são úteis para, por exemplo, fazer requisições ao servidor. Inseriremos o caractere $ e passaremos o elemento a ser manipulado, no caso trata-se do btn.

@{

ViewData["Title"] = "Carrinho";

}

@section Scripts

{

<script type="text/javascript">

function clickIncremento(btn) {

$(btn)

}

</script>

}

E seguida, acessaremos a função attr , uma das várias disponibilizadas pelo jQuery, e ela receberá o nome do atributo que desejamos trabalhar, ou seja, item-id. Todos esses valores serão atribuídos a uma variável que chamaremos de idemId. Finalmente, adicionaremos novamente o comando debugger.

@{

ViewData["Title"] = "Carrinho";

}

@section Scripts

{

<script type="text/javascript">

function clickIncremento(btn) {

var itemId = $(btn).attr('item-id');

debugger;

}

</script>

}

Atualizaremos a página no browser, abriremos o código fonte e clicaremos sobre o botão "+". Nos atentaremos ao seguinte trecho do código:

<!DOCTYPE html>

<\*\*\*\*!\*\*\*\*>

<script type="text/javascript">

function clickIncremento(btn) { btn = button.btn.btn-default

var itemI = $(btn).attr('item-id'); itemId = undefined

debugger;

}

</script>

</body>

</html>

Note que o valor trazido é indefinido (undefinded), não foi possível encontrar o atributo item-id. O que fizemos foi clicar no botão, mas não é no botão que está o atributo.

Ao analisarmos o código fonte mais detalhadamente, perceberemos que o atributoitem-id foi armazenado em uma <div>, e mais abaixo no código teremos a função clickIncremento(), e <button class="btn btn-default> foi passado como evento onclick. Porém, ao analisarmos a estrutura do código hmtl perceberemos que item-id está bem acima.

<!DOCTYPE html>

<\*\*\*\*!\*\*\*\*>

<div class="row row-center linha-produto"item-id="8">

<div class="col-md-3">

<img class="img-produto-carrinho" src="/images/produtos/large\_036.jpg" />

</div>

<div class="col-md-3">@Business Intelligence</div>

<div class="col-md-2 text-center">R$ 49,90</div>

<div class="col-md-2 text-center">

<div class="input-group">

<span class="input-group-btn">

<button class="btn btn-default">

<span class="glyphicon-minus"></span>

</button>

</span>

<input type="text" value="1"

class="form-control text-center" />

<span class="input-group-btn">

<button class="btn btn-default">

onclick="clickIncremento(this)">

<span class="glyphicon-plus"></span>

</button>

<\*\*\*\*!\*\*\*\*>

Nas próximas aulas veremos como localizar o item-id="8" por meio da hierarquia de elementos html por funções do jQuery.

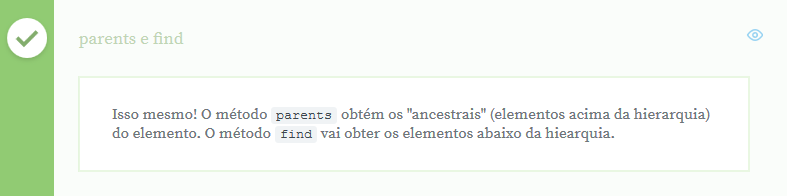
**Exercicio:**

**Obtendo elementos na hierarquia do HTML**

**Você está desenvolvendo uma aplicação de comércio eletrônico, usando a biblioteca JavaScript JQuery.**

**A página de *carrinho* da aplicação possui um trecho contendo informações importantes sobre os itens do pedido, e você precisa obter informações nos elementos desse documento HTML.**

**Quais métodos jQuery você usaria para obter os elementos acima e abaixo de um determinado elemento, respectivamente? Escolha a melhor alternativa.**



**Resumo 3:** Temos em nosso código JavaScript que obtém as informações necessárias para realizar a alteração da quantidade de itens no carrinho. Devemos passar as informações para o servidor, mas antes de fazer a requisição devemos prepara-lo para receber essa requisição.

Faremos a preparação em PedidoController.cs. Ao final do código, criaremos um novo método público, e a principio não declararemos nada. O nome desse método será UpdateQuantidade()

namespace CasaDoCodigo.Controllers

{

<\*\*\*\*!\*\*\*\*>

public IActionResult Resumo()

{

return View(pedidoRepository.GetPedido());

}

public void UpdateQuantidade()`

{

}

}

}

O método UpdateQuantidade() irá tratar a requisição, e receberá como parâmetros um valor interio (int) de itemPedidoId e a nova quantidade de itens, que chamaremos de quantidade.

namespace CasaDoCodigo.Controllers

{

<\*\*\*\*!\*\*\*\*>

public IActionResult Resumo()

{

return View(pedidoRepository.GetPedido());

}

public void UpdateQuantidade(int item PedidoId, int quantidade)`

{

}

}

}

Adicionaremos um break point, e faremos uma requisição diretamente no browser pela barra de endereços, dessa forma podemos testar nosso método. Na barra de endereço digitaremos:

http://localhost:50040/pedido/updatequantidade

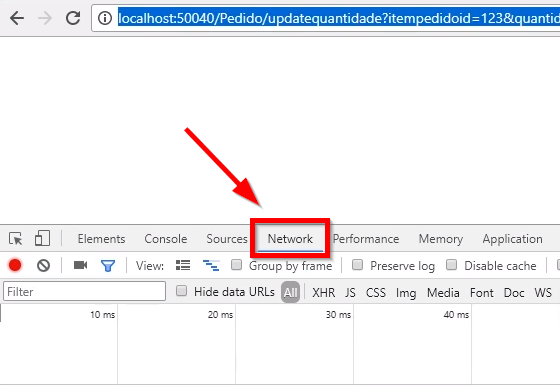
Veremos que a aplicação parou no break point do método updatequantidade(), passando como parâmetro itemPedidoId, que é 0 e quantidade também 0, pois estes são os valores dafault já que não passamos nada para esses parâmetros.

Faremos uma nova requisição diretamente no browser, mas dessa vez passando parâmetros. Utilizaremos a ?, que marca o começo da dos parâmetros, depois inseriremos itemPedidoId e configuraremos um valor qualquer de 123. Em seguida inseriremos o segundo parâmetro quantidade de valor 7, e que será dividido do primeiro parâmetro pelo caractere &,

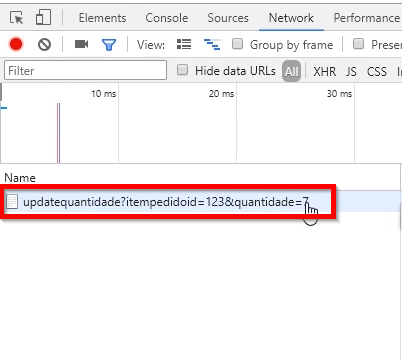
http://localhost:50040/pedido/updatequantidade?itempedidoid=123&quantidade=7

Conseguimos passar com sucesso os parâmetros para o método. No entanto, quando formos realizar de fato a requisição não a passaremos diretamente na **query string**, isto é, todo o conteúdo após a ? que contém os parâmetros da chamada.

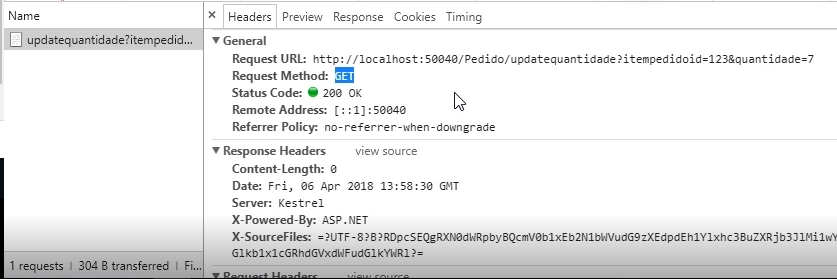
Queremos que esses dados sejam passados no corpo da chamada. Executaremos novamente a aplicação e acessaremos o corpo do desenvolvedor pressionando o atalho "F12". Antes de fazermos a chamada para a requisição, selecionaremos a opção "Network" no browser Chrome.



Agora que a aba network foi selecionada, faremos a requisição novamente. Voltando ao browser, veremos a requisição registrada.



Ao clicarmos sobre ela, teremos acesso a alguns detalhes, como sua URL e seu método, no caso, "GET".



GET é um verbo http, isto é, uma das formas de realizar uma requisição http. Os métodos mais comuns são GET, POST, PUT, DELETE e alguns outros. O método de requisição é a maneira como ela é realizada para o servidor.

Como já havíamos dito, queremos que as informações no corpo da requisição e reforçaremos essa regra no nosso método. Ao invés de realizarmos uma requisição GET, iremos fazer uma do tipo POST, que exige que os parâmetros sejam enviados no corpo da requisição. Além disso, uma requisição da natureza http POST é ideal para realizar alterações em alguma informação do sistema, enquanto a GET é mais interessante para simplesmente obter dados.

Em PedidoController adicionaremos um atributo de método que chamaremos de HttpPost .

namespace CasaDoCodigo.Controllers

{

<\*\*\*\*!\*\*\*\*>

public IActionResult Resumo()

{

return View(pedidoRepository.GetPedido());

}

[HttpPost]

public void UpdateQuantidade(int item PedidoId, int quantidade)`

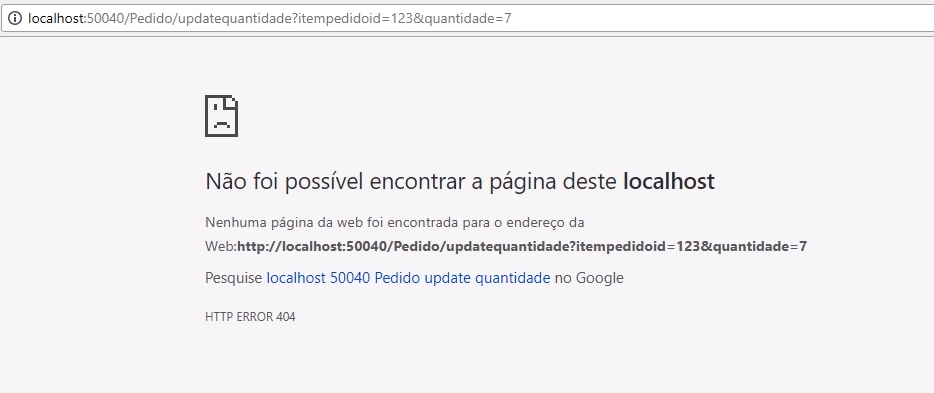
{

}

}

}

Ao realizarmos novamente a requisição no browser, veremos que a aplicação não conseguiu identificar uma página para responder ao endereço updatequantidade.



Nas próximas aulas veremos como fazer uma requisição com o método http POST a partir do código JavaScript.

**Resumo 4:** O método UpdadeQuantidade() do PedidoController está pronto para receber requisições http POST, mas essas requisições irão enviar dados no corpo da requisição e não na url. Precisamos fazer uma adaptação nos parâmetros do método: ao invés de dois parâmetros, passaremos apenas um que será um objeto que conterá as duas informações id e quantidade.

Inseriremos uma instância da classe ItemPedido.

namespace CasaDoCodigo.Controllers

{

<\*\*\*\*!\*\*\*\*>

public IActionResult Resumo()

{

return View(pedidoRepository.GetPedido());

}

[HttpPost]

public void UpdateQuantidade(itemPedido itemPedido)`

{

}

}

}

ItemPedido contém tanto o id, que está em BaseModel como Quantidade, uma propriedade do ItemPedido.

public class ItemPedido: BaseModel

{

[Required]

public Pedido Pedido { get; private set; }

[Required]

public Produto Produto { get; private set; }

[Required]

public int Quantidade {get; private set; }

[Required]

public decimal PrecoUnitario { get; private set; }

<\*\*\*\*!\*\*\*\*>

Como iremos passar ItemPedido a partir do corpo da requisição, devemos anotar esse parâmetro com um atributo que indica sua proveniência, portanto adicionaremos FromBody.

namespace CasaDoCodigo.Controllers

{

<\*\*\*\*!\*\*\*\*>

public IActionResult Resumo()

{

return View(pedidoRepository.GetPedido());

}

[HttpPost]

public void UpdateQuantidade([FromBody]itemPedido itemPedido)`

{

}

}

}

Com isso, podemos voltar ao nosso código JavaScript e programar a requisição. Esse tipo de requisição utiliza uma técnica chamada ajax do jQuery, uma função que exige parâmetros.

@{

ViewData["Title"] = "Carrinho";

}

<\*\*\*\*!\*\*\*\*>

@section Scrips

{

<script type="text/javascript">

function clickIncremento(btn) {

var linhaDoItem = $(btn).parents('[item-id]');

var itemId = $(linhaDoItem).attr('idem-id');

var novaQtde = $(linhaDoItem).find('input').val();

$.ajax({

});

debugger;

}

</script>

Passaremos paraajax as propriedades do objeto JavaScript, mas antes vamos compreender melhor esta função. Ajax é uma sigla: **A\**synchronous \**Ja**vascript **X**ml. Inicialmente tratava-se de uma técnica que utilizava o JavaScript e XML para fazer requisições a sincronizar o servidor, o que significa que o seu código não vai aguardar a resposta da requisição para continuar rodando. O XML caiu em desuso para o Ajax, e ultimamente se usa o JSON para formatação de objetos.

**Para fazermos uma requisição ajax utilizando o jQuery precisaremos quatro parâmetros:**

1. **url: endereço do nosso método do controller(pedido/updatequantidade)**
2. **type: o tipo de requisição http que escolhemos (POST)**
3. **contentType: o formado dos dados (JSON)**
4. **data : o objeto que levará os dados do cliente para o servidor**

Isso esclarecido, iremos inserir esses quatro parâmetros em nossa função ajax:

@{

ViewData["Title"] = "Carrinho";

}

<\*\*\*\*!\*\*\*\*>

@section Scrips

{

<script type="text/javascript">

function clickIncremento(btn) {

var linhaDoItem = $(btn).parents('[item-id]');

var itemId = $(linhaDoItem).attr('idem-id');

var novaQtde = $(linhaDoItem).find('input').val();

$.ajax({

url:'/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data:

});

debugger;

}

</script>

Antes de adicionarmos o objeto em data, iremos definir esse objeto em uma variável: ele terá a informação Id e Quantidade, essa segunda terá o conteúdo da variável novaQtdade. No entanto, não podemos utilizar o objeto sem antes transforma-lo em uma string, portanto adicionaremos JSON.stringfy para realizar essa conversão.

@{

ViewData["Title"] = "Carrinho";

}

<\*\*\*\*!\*\*\*\*>

@section Scrips

{

<script type="text/javascript">

function clickIncremento(btn) {

var linhaDoItem = $(btn).parents('[item-id]');

var itemId = $(linhaDoItem).attr('idem-id');

var novaQtde = $(linhaDoItem).find('input').val();

var data = {

Id: itemId,

Quantidade: novaQtde

};

$.ajax({

url:'/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

});

debugger;

}

</script>

Executaremos a aplicação para averiguar se tudo opera como o esperado. Entraremos na página de compras e escolheremos um produto qualquer e clicaremos no botão "Adicionar".

Voltaremos para a página do carrinho e clicaremos sobre o botão "+", como de costume. Seremos redirecionados para o código de PedidoController e poderemos investigar o parâmetro que foi recebido pelo método UpdateQuantidade(). O resultado foi :

id: 7009

Pedido: null

PrecoUnitario: 0

Produto: null

Quantidade: 0

A Quantidade não foi passada, precisamos descobrir a causa. Iremos até a classe itemPedido averiguar o que pode ter acontecido. O que foi passado para o servidor foi apenas o Id, mas essa informação não se encontra na classe, e sim em BaseModel.

public class ItemPedido: BaseModel

{

[Required]

public Pedido Pedido { get; private set; }

[Required]

public Produto Produto { get; private set; }

[Required]

public int Quantidade {get; private set; }

[Required]

public decimal PrecoUnitario { get; private set; }

<\*\*\*\*!\*\*\*\*>

Ao investigarmos a classe BaseModel, perceberemos que ela possui o Id anotado com o atributo DataMember. Além disso, há também o atributo DataContract. Nós só conseguimos passar o Id porque tanto a classe BaseModel como a propriedade Id estão marcadas com atributos que permitem a sua seralização. Como recebemos uma string a partir do corpo da requisição, tal string pôde ser convertida corretamente em objeto. Em ItemPedido não conseguimos realizar o mesmo procedimento, precisamos anotar tanto a classe quanto as propriedades ItemPedido com os atributos.

namespace CasaDoCodigo.Models

{

[DataContract]

public abstract class BaseModel

{

[DataMember]

pubic int Id { get; protected set; }

}

<\*\*\*\*!\*\*\*\*>

Navegaremos até a classe ItemPedido e coemçaremos a anotar com o atributo DataContract, depois, para cada propriedade do pedido marcaremos como DataMember.

[DataContract]

public class ItemPedido: BaseModel

{

[Required]

[DataMember]

public Pedido Pedido { get; private set; }

[Required]

[DataMember]

public Produto Produto { get; private set; }

[Required]

[DataMember]

public int Quantidade {get; private set; }

[Required]

[DataMember]

public decimal PrecoUnitario { get; private set; }

<\*\*\*\*!\*\*\*\*>

Executaremos a aplicação, e na página carrossel escolheremos um novo produto, seremos direcionados para a página de carrinho e clicaremos sobre o botão "+" para aumentar a quantidade de itens. Feito isso, conseguimos ver as informações passadas para ItemPedido:

Id : 8009

Pedido : null

PrecoUnitario : 0

Produto : null

Quantidade : 1

Temos a Quantidade com valor 1, obtido por meio do código JavaScript. Continuaremos nosso projeto e veremos como essas informações serão utilizadas na aplicação.

**Exercicio:**

# Método no Controller para a Chamada AJAX

**Você está desenvolvendo um e-commerce, e uma das páginas permite aumentar/diminuir a quantidade de itens através de chamadas AJAX que são direcionadas para o método UpdateQuantidade na classe PedidoController:**

**public void UpdateQuantidade(ItemPedido itemPedido)**

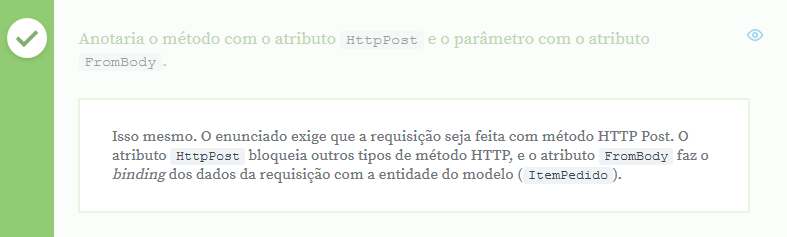
**{**

**//aqui vai o código para alterar quantidades**

**}**

**Porém, esse método necessita de algumas alterações, pois os dados da requisição precisam ser passados no corpo da requisição AJAX.**

**Que alterações você faria nesse método? Escolha a melhor resposta.**



**Exercicio:**

**Monte a Chamada AJAX**

**Você está desenvolvendo uma aplicação ASP.NET Core MVC.**

**Você precisa montar uma chamada AJAX para o método no servidor que altera as quantidades de itens:**

**[HttpPost]**

**public void UpdateQuantidade([FromBody]ItemPedido itemPedido)**

**{**

**//aqui vai o código para alterar quantidades**

**}**

**Como você montaria essa requisição AJAX com JavaScript e jQuery? Escolha a melhor alternativa.**

Selecione uma alternativa:

"Ah, você programa em jQuery?"

Neste livro, Plínio Balduino parte de um código JavaScript, que estava prestes a tornar impossível de se manter, para simplificá-lo com o uso de jQuery, enquanto apresenta o funcionamento do framework por debaixo dos panos. A aplicação JavaScript tem aquele sabor que tanto conhecemos: muita manipulação de DOM e invocações AJAX, de onde começam a brotar IFs e técnicas pouco ortodoxas (as conhecidas gambiarras) para se proteger das incompatibilidades entre navegadores e suas idiossincrasias.



[**https://www.casadocodigo.com.br/pages/sumario-javascript-jquery**](https://www.casadocodigo.com.br/pages/sumario-javascript-jquery)

## Dominando JavaScript com jQuery

**Plínio Balduino**

O que aprendemos?

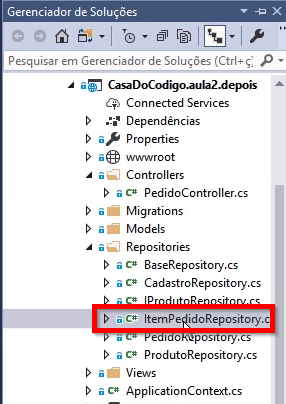
**Aula 1 - Implementando a Chamada AJAX**

* Introdução
* Chamando Função JavaScript a Partir de um evento HTML
  + O operador dólar ($)
* Usando Filtros jQuery Para Acessar Elementos
  + Encontrando um elemento com find
* Navegando Pela Hierarquia de Elementos
  + find e parents
* Criando Método POST no Servidor para Receber Requisição
  + O atributo HttpPost
* Ajax com jQuery
  + O método jQuery.ajax

**Aula 2 – Resumo 1:** Iniciamos a nossa aula 2 com um novo projeto para a solução. Seguiremos esse padrão de projeto por aula até o final do curso, dessa forma podemos comparar isoladamente o que foi produzido em cada atividade.

Em PedidoController nós temos o método UpdateQuantidade() que é acessodo com sucesso a partir do código JavaScript. A próxima etapa é coletar as informações de Id, ItemPedido e Quantidade e fazer as alterações necessárias no banco de dados. Para isso, precisaremos utilizar como meio o **Entity Framework** e, como já aprendemos na primeira parte do curso, para acessar o Entity utilizamos o conceito de repositório, portanto adicionaremos um novo método para alterar a quantidade que irá cair na classe repositório de ItemPedido.

No gerenciador de soluções, abriremos o repositório ItemPedidoRepository, localizado na pasta Repositories.



Teremos acesso à interface que usaremos para declarar o novo método.

namespace CasaDoCodigo.Repositories

{

public interface ItemPedidoRepository

{

}

public class ItemPedidoRepository: BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicatonContext) : base(contexto)

{

}

}

}

Nada será retornado, então adicionaremos void e o nome o mesmo nome proveniente de PedidoController, isto é, UpdateQuantidade que receberá o parâmetro ItemPedido.

namespace CasaDoCodigo.Repositories

{

public interface ItemPedidoRepository

{

void UpdateQuantidade(ItemPedido itemPedido);

}

public class ItemPedidoRepository: BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicatonContext) : base(contexto)

{

}

}

}

Faremos essa classe que implementa a interface, também implementar o método UpdateQuantidade(). Clicaremos sobre o nome da interface IItemPedidoRepository, digitamos o atalho "Ctrl + ." e selecionamos a opção "Implementar Interface".

Dessa forma, será criado um método novo, o NotImplementedException()

namespace CasaDoCodigo.Repositories

{

public interface ItemPedidoRepository

{

void UpdateQuantidade(ItemPedido itemPedido);

}

public class ItemPedidoRepository: BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicatonContext) : base(contexto)

{

}

public void UpdateQuantidade(ItemPedido itemPedido)

{

throw new NotImplementedException();

}

}

}

Mesmo não existindo nenhum código dentro desse novo método, nós iremos realizar sua chamada a partir de PedidoController. Para fazer isso, **utilizaremos a injeção de dependências.**

Declararemos um novo campo privado, chamado itemPedidoRepository.

namespace CasaDoCodigo.Controllers

{

public class PedidoController : Controller

{

private readonly IProdutoRepository produtoRepository;

private readonly IPedidoRepository pedidoRepository;

private readonly IItemPedidoRepository itemPedidoRepository;

public PedidoController(IProdutoRepository produtoRepository,

IPedidoRepository pedidoRepository)

{

this.produtoRepository = produtoRepository;

this.pedidoRepository = pedidoRepository;

}

O novo campo privado que criamos receberá o valor da injeção de dependência. Iremos incluir um novo parâmetro no construtor da classe, que será IItemPedidoRepository itemPedidoRepository. Em seguida faremos a atribuição, de modo que o campo privado receba o valor do parâmetro que foi injetado, portanto escreveremos this.itemPedidoRepository = itemPedidoRepository

namespace CasaDoCodigo.Controllers

{

public class PedidoController : Controller

{

private readonly IProdutoRepository produtoRepository;

private readonly IPedidoRepository pedidoRepository;

private readonly IItemPedidoRepository itemPedidoRepository;

public PedidoController(IProdutoRepository produtoRepository,

IPedidoRepository pedidoRepository,

IItemPedidoRepository itemPedidoRepository)

{

this.produtoRepository = produtoRepository;

this.pedidoRepository = pedidoRepository;

this.itemPedidoRepository = itemPedidoRepository;

}

Assim feito, no método UpdateQuantidade() acessaremos o novo método UpdateQuantidade() e passaremos o parâmetro itemPedido

namespace CasaDoCodigo.Controllers

<\*\*\*\*!\*\*\*\*>

[httpPost]

public void UpdateQuantidade([FromBody]ItemPedido itemPedido)

{

itemPedidoRepository.UpdateQuantidade(itemPedido);

}

}

}

Acessaremos a página de produtos, selecionaremos qualquer um e clicaremos sobre o botão "+" quando estivermos na página de carrinho. Seremos redirecionados para o método UpdateQuantidade() do PedidoController. Usaremos o atalho "F11" e chegaremos à linha throw new NotImplementedException().

Public class ItemPedidoRepository : BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

public void UpdateQuantidade(ItemPedido itemPedido)

{

throw new NotImplementedException();

}

}

A partir desse ponto começaremos a implementar o novo método para gravar informações no banco de dados.

**Resumo 2:** Chegamos no ItemPedidoRepository, e começaremos a implementar o método UpdateQuantidade() para gravarmos informações no banco de dados.

Public class ItemPedidoRepository : BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

public void UpdateQuantidade(ItemPedido itemPedido)

{

throw new NotImplementedException();

}

}

Removeremos a linha throw new NotImplementedException() e começaremos a acessar a entidade ItemPedido. Vamos entender como faremos isso:

ItemPedidoRepository herda da classe BaseRepository, quando navegamos por essa classe - pasta seleciona-la e pressionar o atalho "F12" - encontraremos um campo privado chamado dbSet do tipo Dbset<T>, isto é, um tipo genérico que é o item pedido. No construtor da classe dbSet recebe o valor contexto.Set<T>(), método do Entity Framework que permite a manipulação da entidade em questão, no caso, ItemPedido.

namespace CasaDoCodigo.Repositories

{

public abstract class BaseRepository<T> where T : BaseModel

{

protected readonly ApplicationContext contexto;

protected readonly DbSet<T> dbSet;

public BaseRepository(ApplicationContext contexto)

{

this.contexto = contexto;

dbSet = contexto.Set<T>();

}

}

}

Voltaremos ao ItemPedidoRepository e acessaremos o dbSet no UpdateQuantidade(). Aplicaremos um filtro, de forma que localizemos somente o pedido que nos interessa, ou melhor, aquele que possui o id passado pelo objeto. Para aplicarmos esse filtro utilizaremos um método do Linq chamado Where(), que por sua vez receberá uma expressão lambda ip (que representa item pedido), ip.id que deverá ser igual ao id do parâmetro de itemPedido, portanto escreveremos itemPedido.Id.

public class ItemPedidoRepository: BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

public void UpdateQuantidade(ItemPedido itemPedido)

{

dbSet

.Where(ip => ip.Id == itemPedido.Id)

Como estamos buscando apenas um item, usaremos outro método do Linq chamado SingleOrDefault() que retornará um elemento, caso esse elemento não seja encontrado será retornado um valor nulo.

public class ItemPedidoRepository: BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

public void UpdateQuantidade(ItemPedido itemPedido)

{

dbSet

.Where(ip => ip.Id == itemPedido.Id)

.SingleOrDefault();

Precisamos armazenar esse valor em algum lugar, portanto declararemos uma variável nova chamada itemPedidoDB.

public class ItemPedidoRepository: BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

public void UpdateQuantidade(ItemPedido itemPedido)

{

var itemPedidoDB =

dbSet

.Where(ip => ip.Id == itemPedido.Id)

.SingleOrDefault();

Uma vez que estamos obtendo a variável ItemPedidoDB, precisamos verificar se o seu valor não é nulo, e caso seja, queremos ignorá-la. Adicionaremos uma verificação If, e caso o valor da variável seja diferente de nulo nós iremos atualizar o banco de dados.

Para atualizarmos os dados basta acessar o objeto itemPedidoBD e modificar sua quantidade, atribuindo a nova quantidade a ser recebida do parâmetro de ItemPedido.

public class ItemPedidoRepository: BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

public void UpdateQuantidade(ItemPedido itemPedido)

{

var itemPedidoDB =

dbSet

.Where(ip => ip.Id == itemPedido.Id)

.SingleOrDefault();

if (itemPedidoDB != null)

{

itemPedidoDB.Quantidade = itemPedido.Quantidade;

}

No entanto, perceberemos que não podemos fazer atribuição a Quantidade, porque a propriedade ItemPedido.Quantidade não pode ser utilizada nesse contexto, porque o "acessador set é inacessível", segundo a mensagem no Visual Studio. Isso quer dizer que esta é apenas uma propriedade de leitura. Ao invés de modificarmos a propriedade para deixá-la pública, criaremos um novo método para modificar a Quantidade. Criaremos um novo método chamado AtualizaQuantidade(), que receberá como parâmetro a nova quantidade.

public class ItemPedidoRepository: BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

public void UpdateQuantidade(ItemPedido itemPedido)

{

var itemPedidoDB =

dbSet

.Where(ip => ip.Id == itemPedido.Id)

.SingleOrDefault();

if (itemPedidoDB != null)

{

itemPedidoDB.AtualizaQuantidade (itemPedido.Quantidade);

}

Nós escrevemos o método, mas ele ainda não existe e por isso forçaremos o Visual Studio a criá-lo: selecionaremos o método e teclaremos o atalho "Ctrl + .", depois escolheremos a opção "Gerar método 'ItemPedido.AtualizaQuantidade'". Dessa forma não será mais indicado erro no código, mas não temos o método implementado. Selecionaremos AtualizaQuantidade() e pressionaremos "F12".

Implementaremos o método atribuindo à propriedade Quantidade o novo valor da quantidade recebida no método

namespace CasaDoCodigo.Models

{

<\*\*\*\*!\*\*\*\*>

public ItemPedido(Pedido pedido, Produto produto, int quantidade, decimal precoUnitario)

{

Pedido = pedido;

Produto = produto;

Quantidade = quantidade;

PrecoUnitario = precoUnitario;

}

internal void AtualizaQuantidade(int quantidade)

{

Quantidade = quantidade;

}

}

De volta ao ItemPedidoRepository, teremos alterações feitas em memória, portanto precisamos descarregar essas alterações no banco dados. Para isso, utilizaremos o contexto.SaveChanges().

Feito isso, adicionaremos breaking points na linha if (itemPedidoDB != null) e itemPedidoDB.AtualizaQuantidade (itemPedido.Quantidade);. Adicionaremos outro breaking point na linha Quantidade = quantidade; do método AtualizaQuantidade().

public class ItemPedidoRepository: BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

public void UpdateQuantidade(ItemPedido itemPedido)

{

var itemPedidoDB =

dbSet

.Where(ip => ip.Id == itemPedido.Id)

.SingleOrDefault();

if (itemPedidoDB != null)

{

itemPedidoDB.AtualizaQuantidade (itemPedido.Quantidade);

contexto.SaveChanges();

}

Iremos executar a aplicação e, como de praxe, clicaremos sobre um item qualquer e depois sobre o botão "+" quando estivermos na página do carrinho. Quando essas ações forem finalizadas, seremos direcionados para a PedidoController. Veremos que para itemPedido teremos as seguintes informações:

Id : 12009

Pedido: null

PrecoUnitario : 0

Produto: null

Quantidade : 1

Pressionaremos o atalho "F5" para navegarmos até o próximo método. ItemPedidoDB, que foi carregado do banco de dados, terá as seguintes informações:

Id: 12009

Pedido: null

PrecoUnitario : 49,90

Produto: null

Quantidade : 1

Em seguida iremos até o método AtualizaQuantidade(), e veremos que a quantidade continua sendo 1. Devemos inserir um valor que nos permita acompanhar se houve mudança no banco de dados, alteraremos o valor de quantidade para 117.

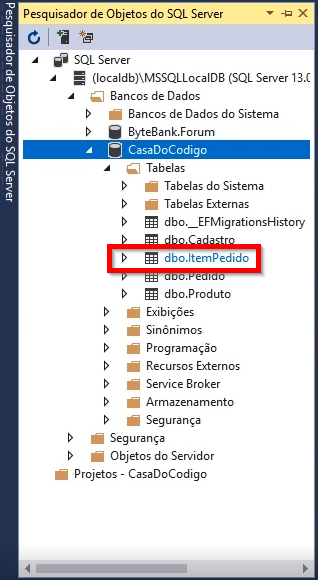
internal void AtualizaQuantidade(int quantidade);

{

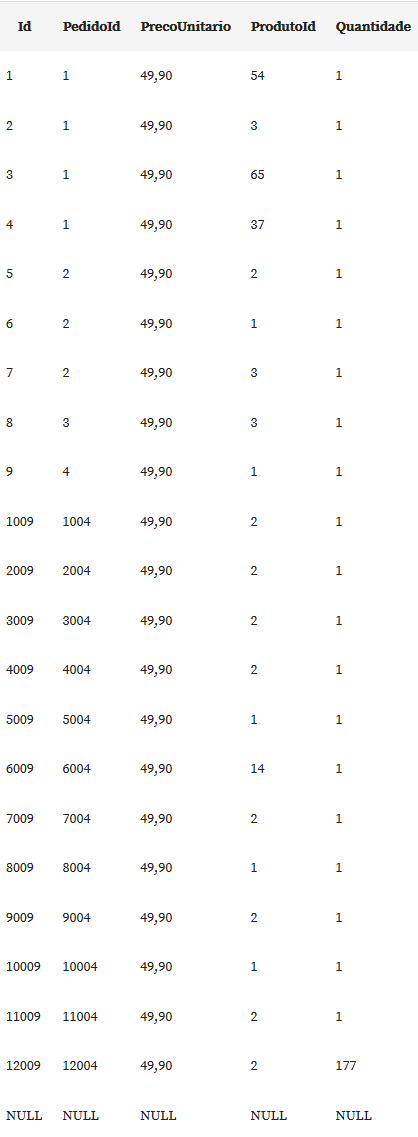
Quantidade = quantidade;

}

Pressionaremos "F5" e iremos verificar se de fato houve mudança no banco de dados. Na área "Pesquisador de Objetos do SQL Server", clicaremos sobre "CasaDoCódigo > Tabelas > dbo.ItemPedido", clicaremos sobre o arquivo e selecionaremos a opção "Exibir Dados".



Como podemos ver por meio da tabela, os dados foram atualizados corretamente.

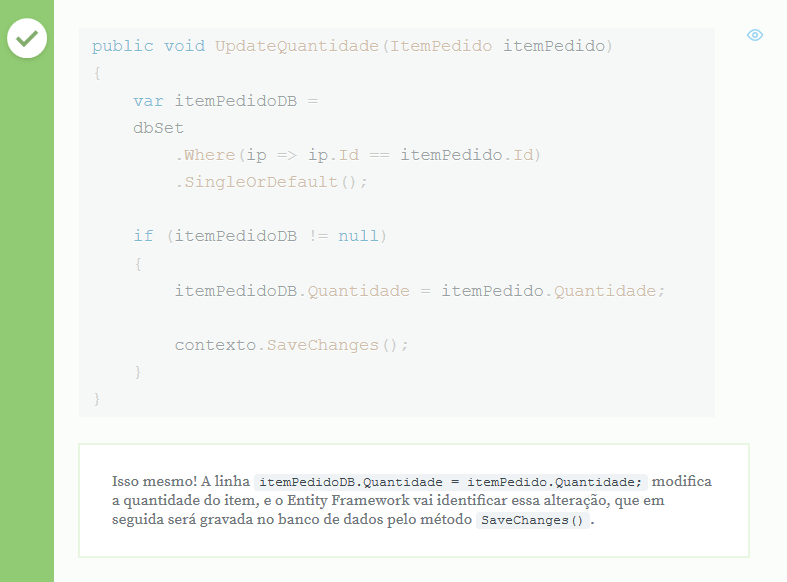


# Exercício:

# Altere a Quantidade no Banco de Dados

**Você precisa desenvolver um método para tomar como parâmetro um item de pedido contendo o Id do item e também a nova quantidade que será gravada no banco de dados para esse item.**

**Quais dos trechos abaixo deve ser usado para realizar essa tarefa? Escolha a melhor alternativa.**



**Resumo 3:** Conseguimos atualizar as quantidades no banco de dados. Voltaremos ao código JavaScript para organiza-lo e precisaremos criar uma nova função, para que possamos tornar o botão "-" também funcional na página do carrinho de compras.

O nosso código JavaScript está se tornando muito extenso, o que prejudica sua organização e manutenção.

@{

ViewData["Title"] = "Carrinho";

}

@section Scripts

{

<script type="text/javascript">

function clickIncremento(btn) {

var linhaDoItem = $(btn).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQtde = $(linhaDoItem).find('input').val();

var data = {

Id: itemId,

Quantidade: novaQtde

};

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

});

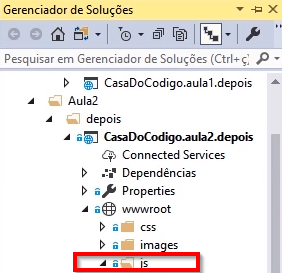
debugger;

}

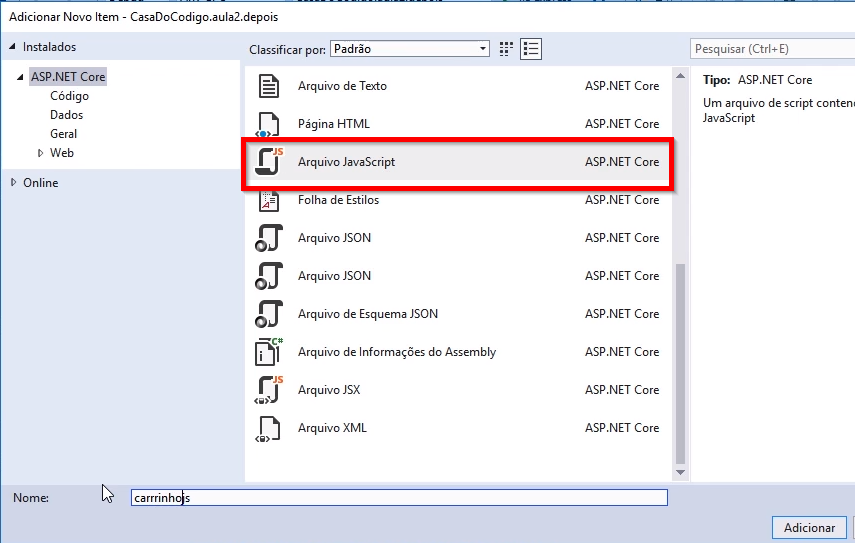
</script>

}

Criaremos um novo arquivo que armazenará essas informações. Na área "Gerenciador de Soluções" do Visual Studio, abriremos a pasta "wwwroot > js".



Com Js selecionado, clicaremos com o botão direito e escolheremos as opções "Adicionar > Novo Item". Na nova caixa de diálogo, configuraremos para que este novo arquivo seja JavaScript, e seu nome será carrinho.js.



Moveremos para dentro do novo arquivo o código JavaScript, que contém somente a função clickIncremento(). Como nós movemos esse código, precisamos avisar a view onde está localizado o arquivo js, portanto devemos passar a fonte do arquivo (src). Teremos a subpasta js e o arquivo carrinho.js.

@section Scripts

{

<script src="~/js/carrinho.js">

</script>

}

Voltando para carrinho.js, teremos uma função clickIncremento() que chamos de **escopo global**, o que indica que esta função está visível para todo o código JavaScript que referencia esse arquivo. O que desejamos fazer é organizar esse código, deixando a função dentro de um outro escopo, logo, ciaremos uma nova classe para encapsular a função.

function clickIncremento(btn) {

var linhaDoItem = $(btn).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQtde = $(linhaDoItem).find('input').val();

var data = {

Id: itemId,

Quantidade: novaQtde

};

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

});

debugger;

A classe é um conceito que entrou no JavaScript a partir do ECMAScript versão 6, e ele irá permitir que organizemos melhor nosso código. o Nome dessa classe será Carrinho.Para utilizar um função dentro de uma classe, não podemos ter a palavra function dentro dela, portanto a deletaremos.

class Carrinho {

clickIncremento(btn) {

var linhaDoItem = $(btn).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQtde = $(linhaDoItem).find('input').val();

var data = {

Id: itemId,

Quantidade: novaQtde

};

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

});

debugger;

}

}

Para acessarmos a função clickIncremento() que está na classe precisamos criar uma instância, que ficará ao final do código e se chamará carrinho.carrinhoserá uma nova instância da classe Carrinho().

class Carrinho {

clickIncremento(btn) {

var linhaDoItem = $(btn).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQtde = $(linhaDoItem).find('input').val();

var data = {

Id: itemId,

Quantidade: novaQtde

};

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

});

debugger;

}

}

var carrinho = new Carrinho();

Criaremos uma nova função para decrementar a quantidade de itens no carrinho. Essa nova função se chamará clickDecremento(), e receberá o botão como parâmetro (btn).

class Carrinho {

clickIncremento(btn) {

var linhaDoItem = $(btn).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQtde = $(linhaDoItem).find('input').val();

var data = {

Id: itemId,

Quantidade: novaQtde

};

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

});

debugger;

}

clickDecremento(btn) {

}

}

var carrinho = new Carrinho();

O código de clickDecremento() será praticamente o mesmo de clickIncremento(), portanto iremos realizar um reaproveitamento do código atribuindo-lhe novas funções. Observem este trecho:

clickIncremento(btn) {

var linhaDoItem = $(btn).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQtde = $(linhaDoItem).find('input').val();

var data = {

Id: itemId,

Quantidade: novaQtde

};

Sabemos que esta parte de código é responsável pela obtenção do objeto data. Podemos criar uma nova função denominada getData() ao final da classe Carrinho. Essa nova função deverá receber um elemento html, portanto escreveremos elemento como parâmetro. Feito isso, inseriremos o trecho de código que havíamos destacado para dentro de getData().

class Carrinho {

clickIncremento(btn) {

var linhaDoItem = $(btn).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQtde = $(linhaDoItem).find('input').val();

var data = {

Id: itemId,

Quantidade: novaQtde

};

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

});

debugger;

}

clickDecremento(btn) {

}

getData(elemento) {

var linhaDoItem = $(btn).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQtde = $(linhaDoItem).find('input').val();

var data = {

Id: itemId,

Quantidade: novaQtde

};

}

}

var carrinho = new Carrinho();

No método getData() substituiremos o parâmetro btn em linhaDoItem() para elemento.

getData(elemento) {

var linhaDoItem = $(elemento).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQtde = $(linhaDoItem).find('input').val();

var data = {

Id: itemId,

Quantidade: novaQtde

};

}

}

Iremos consumir a função GetData() em clickIncremento(), para isso acessaremos a instância this.getData() e passaremos o parâmetro btn. Depois, atribuiremos esse conteúdo a uma variável local denominada let data.

class Carrinho {

clickIncremento(btn) {

let data = this.getData(btn);

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

});

Vejamos o próximo trecho de código restante:

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

});

Sabemos que sua função é postar a quantidade do pedido, logo, criaremos ao final da classe Carrinho, uma nova função denominada postQuantidade() que receberá o objeto data. Dentro desta função alocaremos o trecho do código destacado.

postQuantidade(data) {

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

});

Assim feito, chamaremos esse código por meio do this.postQuantidade(), passando como parâmetro a variável data

class Carrinho {

clickIncremento(btn) {

let data = this.getData(btn);

this.postQuantidade(data);

debugger;

}

Ao observarmos atentamente a função getData(), veremos que ela não retorna nada. Substituiremos var data por return

getData(elemento) {

var linhaDoItem = $(elemento).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQtde = $(linhaDoItem).find('input').val();

return = {

Id: itemId,

Quantidade: novaQtde

};

}

}

Nosso código está bem mais organizado e funcional. Copiaremos a código de clickIncremento() para clickDecremento() e removeremos o debugger.

class Carrinho {

clickIncremento(btn) {

let data = this.getData(btn);

this.postQuantidade(data);

}

clickDecremento(btn) {

let data = this.getData(btn);

this.postQuantidade(data);

}

Precisamos diferenciar os dois: no caso de clickIncremento() o valor de Quantidadedeve aumentar, já em clickDecremento() deve diminuir. Faremos isso adicionando, respectivamente data.Quantidade++ e data.Quantidade--.

class Carrinho {

clickIncremento(btn) {

let data = this.getData(btn);

data.Quantidade++;

this.postQuantidade(data);

}

clickDecremento(btn) {

let data = this.getData(btn);

data.Quantidade--;

this.postQuantidade(data);

}

Para conseguirmos utilizar a instância da classe Carrinho, que está na variável carrinho devemos fazer algumas modificações no código html. Iremos até Carrinho.cshtml e modificaremos o evento onclick, referenciando a instância carrinho.

<input type="text" value="@(item.Quantidade)"

class="form-control text-center" />

<span class="input-group-btn">

<button class="btn btn-default"

onclick="carrinho.clickIncremento(this)">

<span class="glyphicon-plus"></span>

Copiaremos a linha onclick="carrinho.clickIncremento(this)" e a colocaremos mais acima do código onde se localiza o botão "-" e copiaremos esse tracho. Depois, alteraremos o nome da função clickIncremento() para clickDecremento().

<span class="input-group-btn">

<button class="btn btn-default">

onclick="carrinho.clickDecremento(this)"

<span class="glyphicon-minus"></span>

</button>

</span>

Com isso, temos o código para aumentar e diminuir a quantidade de itens no carrinho de compras.

**Exercício:**

# Consuma Função de uma Classe JavaScript a Partir de evento HTML

**Você acabou de implementar uma classe carrinho em JavaScript, que contém um método clickIncremento que precisa ser acessado por um evento HTML:**

**class Carrinho {**

**clickIncremento(botao) {**

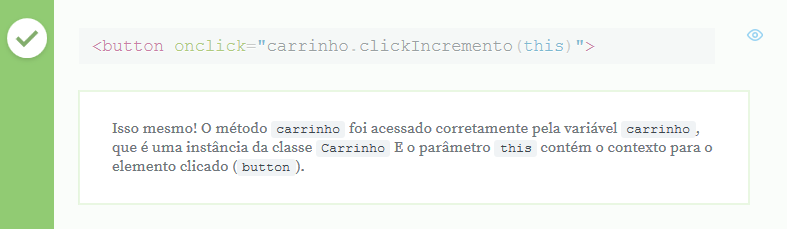
**//o código do clique vai aqui...**

**}**

**}**

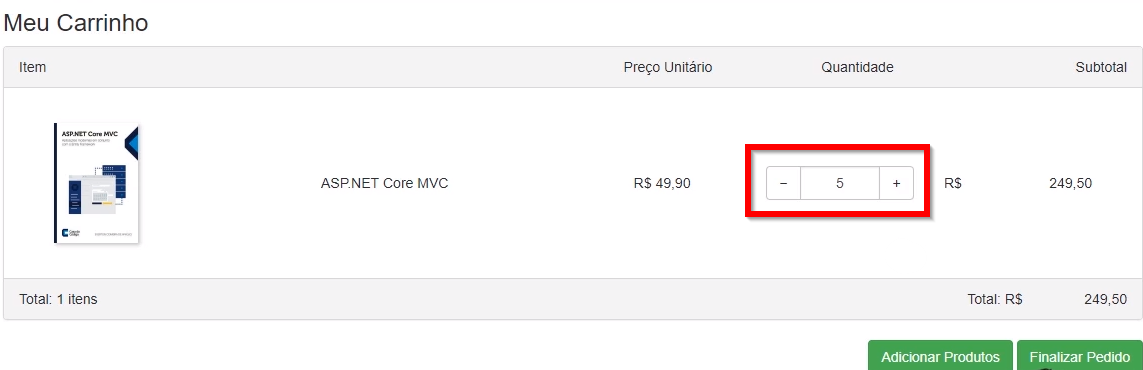
**var carrinho = new Carrinho();**

**Como o evento HTML deve acessar esse método? Escolha a melhor alternativa.**



**Resumo 4:** Temos a função clickIncremento() e clickDecremento() em nosso código. Iremos testá-las na tela de carrinho, clicando sobre os botões de "+" e "-", depois que ela for atualizada por meio do atalho "F5".

Ao clicarmos sobre o botão "+" e atualizarmos a página, veremos que o valor de itens do carrinho aumenta, e ao clicarmos no botão "-" diminui. A numeração na caixa de texto está responsiva aos cliques do usuário, exatamente o que queríamos.



Para finalizar a configuração da caixa de texto, queremos que o usuário possa digitar a quantidade de itens sem precisar fazer uso dos botões de "+" ou "-", gerando uma modificação direta no banco de dados.

Modificaremos a classe Carrinho, adicionando uma nova função chamada updateQuantidade(), que receberá o elemento html que será a própria caixa de texto, isto é, o nome do parâmetro será input. Em seguida, aproveitaremos trechos de código da mesma classe, retirando a linha data.Quantidade. Faremos outra modificação: para getData() passaremos input e não btn.

class Carrinho {

clickIncremento(btn) {

let data = this.getData(btn);

data.Quantidade++;

this.postQuantidade(data);

}

clickDecremento(btn) {

let data = this.getData(btn);

data.Quantidade--;

this.postQuantidade(data);

}

updateQuantidade(input) {

let data = this.getData(input);

this.postQuantidade(data);

}

Para acessarmos updateQuantidade() iremos para Carrinho.cshtml e localizarmos o input e adicionaremos um novo evento chamado onblur, que será acionado para chamarmos na instância carrinho a função que criamos. Passaremos como parâmetro para updateQuantidade() o elemento this.

<input type="text" value="@(item.Quantidade)"

class="form-control text-center" />

onblur="carrinho.updateQuantidade(this)"/>

<span class="input-group-btn">

<button class="btn btn-default"

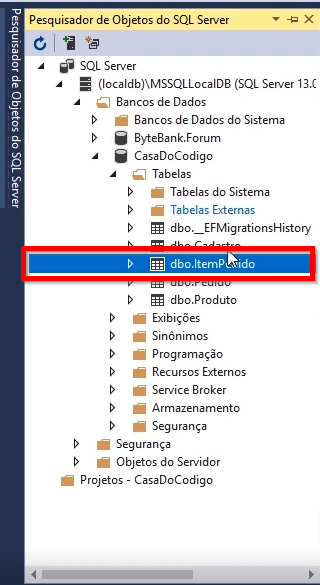
onclick="clickIncremento(this)">

<span class="glyphicon-plus"></span>

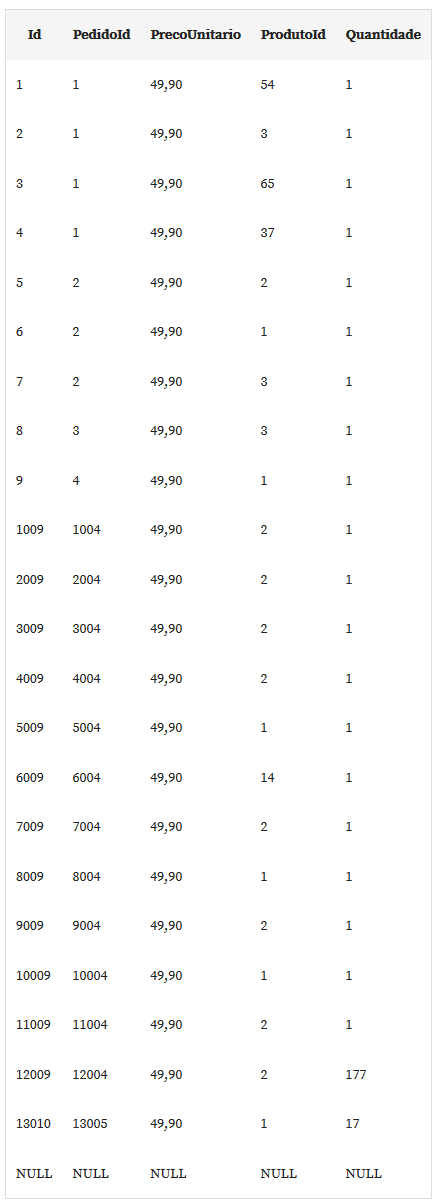
</button>

Feitas essas modificações, poderemos ir até a página de carrinho e digitar um valor qualquer no campo de quantidade. Digitaremos 17 e apertaremos a tecla "Tab", desse modo a chamada será realizada para o banco de dados.

De volta ao Visual Studio, aa área "Pesquisador de Objetos", inspecionaremos a tabela dbo.ItemPedido.



Ao final da tabela teremos o valor 17. O registro ocorrerá da mesma maneira se adicionarmos valores diferentes na caixa de texto, o que revela que conseguimos implementar a alteração de quantidades diretamente.



**Resumo 5: A QUEM SE DESTINA ESTE LIVRO**

Este livro é destinado aos desenvolvedores web que tenham, pelo menos, conhecimentos básicos da linguagem JavaScript. A seguir, seguem os principais tópicos que é necessário ter pelo menos um conhecimento básico para melhor aproveitamento do conteúdo: Declaração de funções e variáveis; Estrutura de dados: objetos e arrays; Funções anônimas e de callback; Escopos; Herança por prototipagem.



[**https://www.casadocodigo.com.br/pages/sumario-ecmascript6**](https://www.casadocodigo.com.br/pages/sumario-ecmascript6)

## ECMAScript 6 - Entre de cabeça no futuro do JavaScript

**Diego Martins de Pinho**

O que aprendemos?

**Aula 2 - Alterando a Quantidade de Itens**

* Chamando Repositório de Itens a Partir do Controller
  + Injeção de dependência
* Alterando a Quantidade no Banco de Dados
  + O evento HTML onclick
  + Salvando dados com DbContext.SaveChanges()
* Organizando Código JavaScript com Classe
  + Montando uma classe com EcmaScript 6 (ES6)
* Digitando a Nova Quantidade
  + O evento HTML onblur

**Aula 3 – Resumo 1:** Estamos visualizando a view de Carrinho.cshtml que exibe corretamente as informações e tratando dos eventos de click do botão de "+" e de "-", assim como da digitação de quantidade diretamente na caixa de texto, que será gravada no banco de dados.

Nesta view fazemos o cálculo do total geral de itens do carrinho utilizando uma expressão em C#, A @(Model.Sum(i => i.Quantidade \* i.PrecoUnitario)), que utiliza a quantidade do item e multiplica por seu preço unitário, ao final é realizada a soma desses valores, dessa forma é obtida o total no carrinho. Essa é uma regra de negócio, e essa informação não deveria estar contida dentro da view, que por definição só abriga regras de apresentação.

</div>

<div class="panel-footer">

<div class="row">

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

<span numero-itens>

Total: @(Model.Count())

itens

</span>

</div>

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

Total: R$ <span class="pull-right" total>

@(Model.Sum(i => i.Quantidade \* i.PrecoUnitario))

</span>

</div>

</div>

</div>

</div>

O ideal seria mover essa regra de negócio para um local mais adequeado, para isso criaremos uma classe nova que fornecerá os dados para a view. No começo da nossa view teremos a declaração do modelo com diretiva @Model, que recebe uma lista ItemPedido. Trocaremos essa informação por uma nova classe que fornecerá um modelo novo e específico para esta view, isto é, uma *view model*.

@{

ViewData["Title"] = "Carrinho";

}

@model IList<ItemPedido>;

<h3>Meu Carrinho</h3>

Criaremos a *view model* dentro da pasta Models do nosso projeto. Criaremos um novo diretório chamado ViewModels e dentro dele adicionaremos numa nova classe cujo nome será CarrinhoViewModel.cs.

A vantagem de uma view model é que ela não precisa ser associada ao Entity Framework, pois diferentes das classes do modelo, não será gravada no banco de dados como tabela.

namespace CasaDoCodigo.Models.ViewModels

{

public class CarrinhoViewModel

{

}

}

CarrinhoViewModel, portanto, irá fornecer uma série de dados para view, como uma lista de itens de pedido. Vamos declarar essa informação como uma propriedade pública, uma lista de pedidos que chamaremos de Itens e será somente de leitura, por isso a declararemos como get.

namespace CasaDoCodigo.Models.ViewModels

{

public class CarrinhoViewModel

{

public IList<ItemPedido> Itens { get; }

}

}

A próxima etapa é gerar o construtor para que essa classe receba a lista de itens. Selecionaremos a linha public IList<ItemPedido> Itens { get; }, acionaremos o atalho "Ctrl + ." e escolheremos a opção "Gerar Construtor".

namespace CasaDoCodigo.Models.ViewModels

{

public class CarrinhoViewModel

{

public CarrinhoViewModel(IList<ItemPedido> itens)

{

itens = itens;

}

public IList<ItemPedido> Itens { get; }

}

}

Forneceremos para a view uma propriedade que calculará o valor total disponível no carrinho, para isso declararemos uma propriedade pública chamada Total que retornará um valor que contém casas decimais, isto é, a soma do subtotal de itens. Usaremos o método Sum() e como parâmetro passaremos uma expressão lambda.

namespace CasaDoCodigo.Models.ViewModels

{

public class CarrinhoViewModel

{

public CarrinhoViewModel(IList<ItemPedido> itens)

{

Itens = itens;

}

public IList<ItemPedido> Itens { get; }

public decimal Total => Itens.Sum(i => i.Quantidade \* i.PrecoUnitario);

}

}

Utilizaremos essa nova classe como modelo da view Carrinho.cshtml.

@{

ViewData["Title"] = "Carrinho";

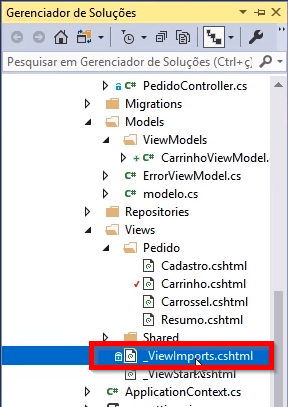
}

@model CarrinhoViewModel;

<h3>Meu Carrinho</h3>

No entanto, veremos uma marcação de erro do Visual Studio em CarrinhoViewModel, isso se deve porque colocamos essa classe em outro namespace que não foi reconhecido. Temos duas alternativas: inserir antes do nome da classe seu respectivo namespace, ou incluir uma nova diretiva using no arquivo \_ViewImports.cshtml. Recorreremos a segunda opção.

Na área "Gerenciador de Soluções" abriremos o arquivo \_ViewImports.cshtml.



Dentro desse arquivo teremos algumas diretivas using.

@using CasaDoCodigo

@using CasaDoCodgio.Models

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

Adicionaremos uma nova diretiva referente a ViewModels.

@using CasaDoCodigo

@using CasaDoCodgio.Models

@using CasaDoCodigo.Models.ViewModels

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

Com isso, não teremos mais uma mensagem de erro em CarrinhoViewModel na view Carrinho.cshtml.

O próximo passo é modificar algumas informações da view de modo que ela se adeque à CarrinhoViewModel, com por exemplo o laço foreach que obtém os itens a partir de Model, que costumava ser uma lista, mas atualmente trata-se de um objeto que contém uma lista.

<div class="panel-body">

@foreach (var item in Model)

{

<div class="row row-center linha-produto" item-id="@item.Id">

Substituiremos Model por Model.Itens.

<div class="panel-body">

@foreach (var item in Model.Itens)

{

<div class="row row-center linha-produto" item-id="@item.Id">

Mais abaixo no código da view, teremos a quantidade total de itens Total, que era fornecido por Model.Count. Substituiremos Model.Count por Model.Itens.Count.

</div>

<div class="panel-footer">

<div class="row">

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

<span numero-itens>

Total: @(Model.Itens.Count())

itens

</span>

</div>

Mais abaixo, teremos o cálculo do valor total que nos referimos no começo da aula.

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

Total: R$ <span class="pull-right" total>

@(Model.Sum(i => i.Quantidade \* i.PrecoUnitario))

Nós removeremos este cálculo e o substituiremos por Model.Total.

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

Total: R$ <span class="pull-right" total>

@(Model.Total)

Precisamos, ainda, fornecer uma instância a nova classe CarrinhoViewModel a partir do cotroller, afinal é ele que irá retornar uma view e injetar o objeto que servirá de modelo. Em gerenciador de soluções abriremos o arquivo PedidoController.cs e localizaremos a action que retorna Carrinho. Percebam que ela está injetando na view o parâmetro pedidoRepository.GetPedido().Itens, que é uma lista de pedidos. Podemos substituir este trecho por CarrinhoViewModel.

public IActionResult Carrinho(string codigo)

{

if (!string.IsNullOrEmpty(codigo))

{

pedidoRepository.AddItem(codigo);

}

return View(pedidoRepository.GetPedido().Itens);

}

Selecionaremos pedidoRepository.GetPedido().Itens e extrairemos uma variável local, para que nosso código se torne mais legível. Pressionaremos "Ctrl + ." e escolheremos a opção "Introduzir local para "pedidoRepository.GetPedido().Itens".

public IActionResult Carrinho(string codigo)

{

if (!string.IsNullOrEmpty(codigo))

{

pedidoRepository.AddItem(codigo);

}

List<ItemPedido> itens = pedidoRepository.GetPedido().Itens;

return base.View(itens);

}

Com a variável criada que capta os itens, montaremos uma instância de CarrinhoViewModel, que receberá como parâmetro itens.

public IActionResult Carrinho(string codigo)

{

if (!string.IsNullOrEmpty(codigo))

{

pedidoRepository.AddItem(codigo);

}

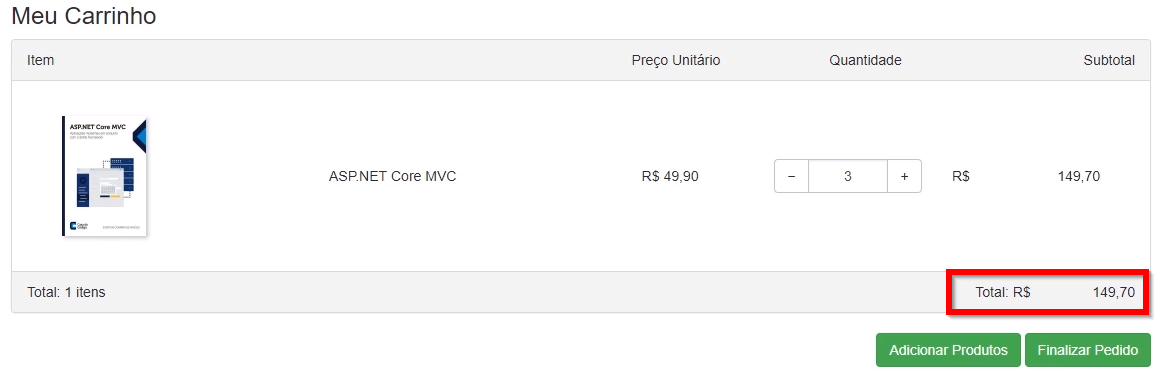
List<ItemPedido> itens = pedidoRepository.GetPedido().Itens;

CarrinhoViewModel carrinhoViewModel = new CarrinhoViewModel(itens);

return base.View(carrinhoViewModel);

}

Executaremos nossa aplicação pressionando o atalho "F5". Seremos direcionados para a página principal e selecionaremos um produto para adicionarmos o carrinho. Já na página de carrinho aumentaremos o número de itens e veremos que o valor total é modificado cada vez que um intem é incluído.



Conseguimos aplicar o padrão novo que fornece informações para a view com a vantagem de não precisar ser gravado no banco de dados como tabela.

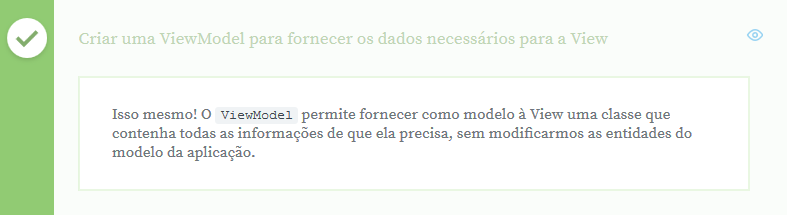
**Exercício:**

# Fornecendo os Dados Necessários Para a View

**Você está desenvolvendo uma aplicação ASP.NET Core MVC.**

**Você precisa fazer a action de um controller passar para uma View informações que estão em duas classes diferentes do modelo do MVC. Nenhuma classe, sozinha, possui atualmente todas as informações necessárias para a view.**

**Qual modificação você faria na aplicação para atender esse requisito? Escolha a melhor alternativa.**



**Resumo 2:** Nesta aula iremos resolver mais um problema da nossa aplicação: quando aumentamos a quantidade de itens no carrinho de compras, nós não dispomos da atualização automática da página, o que nos obriga a pressionar o tempo todo a tecla "F5" para vermos os valores corretos. Para deixarmos nossa página dinâmica, modificaremos o JavaScript.

No Visual Studio, abriremos o arquivo Carrinho.js dentro da pasta wwwroot, subpasta js. Nós teremos o método ajax() que realiza a chamada para o servidor para que os dados sejam atualizados.

class Carrinho {

<\*\*\*\*!\*\*\*\*>

postQuantidade(data) {

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

});

}

}

var carrinho = new Carrinho();

A página deve ser atualizada no momento em que obtivermos a resposta de sucesso na chamada ajax(). Faremo isso ao adicionar uma chamada para outro método, logo depois doajax(). Esse novo método indicará que a chamada foi completada, e seu nome é done(), que significa em inglês "completo" ou "terminado". done() receberá como parâmetro uma função (function()) , que por sua vez receberá a resposta do servidor(response), enviada por pedidoController. Dentro do corpo da função nós inseriremos o código de *refresh* da página, isto é, o comando JavaScript location.reload().

class Carrinho {

<\*\*\*\*!\*\*\*\*>

postQuantidade(data) {

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

}).done(function (response) {

location.reaload();

});

}

}

var carrinho = new Carrinho();

Feitas as alterações no código, voltaremos a página inicial e selecionaremos um item qualquer para dispormos no carrinho. Já na página de carrinho, atualizaremos o JavaScript pressionando a tecla "F5" E clicaremos sobre o botão "+" como fizemos outras vezes, para avaliar se as modificações surtiram efeito. Notaremos que a quantidade de itens e valores é atualizada automaticamente, contudo esse ainda não é o estado ideal, pois quando fazemos o location.reload() estamos obrigando o servidor a recriar toda a página e mandá-la novamente para o navegador, o que aumenta o consumo de banda larga por usuário. O ideal é realizarmos uma chamada ajax() que receba uma informação precisa do item que foi alterado e o valor total do carrinho. Deletaremos location.reload() e começaremos a montar uma nova classe que fornecerá um objeto de resposta para o método updateQuantidade().

Abriremos o arquivo PedidoController.cs, onde teremos o método que fornece a resposta para a chamada ajax(), isto é, UpdateQuantidade().

[HttpPost]

public void UpdateQuantidade([FromBody]ItemPedido itemPedido)

{

itemPedidoRepository.UpdateQuantidade(itemPedido);

}

}

}

UpdateQuantidade() não retorna nada, isto é, temos void como resposta. O que queremos fazer é retornar um novo objeto. Criaremos uma nova classe chamada UpdateQuantidadeResponse dentro da pasta Models do nosso projeto. A nova classe irá conter o item de pedido que foi alterado ao clicarmos no botão de "+" ou "-" no carrinho. Esse novo itemPedido será apenas de leitura, portanto escreveremos a palavra get.

namespace CasaDoCodigo.Models

{

public class UpdateQuantidadeResponse

{

public ItemPedido ItemPedido { get; }

}

}

Quando alteramos algum valor, é importante que retornemos todo o carrinho com os valores totais, portanto declararemos uma outra propriedade pública que será o próprio carrinho (CarrinhoViewModel), caso seja necessário incluiremos o namespace.

namespace CasaDoCodigo.Models

{

public class UpdateQuantidadeResponse

{

public ItemPedido ItemPedido { get; }

public CarrinhoViewModel CarrinhoViewModel { get; }

}

}

Criaremos um construtor para a classe: selecionaremos as duas propriedades existentes e acionaremos o atalho "Ctrl + ." e a opção "Gerar Construtor".

namespace CasaDoCodigo.Models

{

public class UpdateQuantidadeResponse

{

public UpdateQuantidadeResponse(ItemPedido itemPedido, CarrinhoViewModel carrinhoViewModel)

{

ItemPedido = itemPedido;

CarrinhoViewModel = carrinhoViewModel;

}

public ItemPedido ItemPedido { get; }

public CarrinhoViewModel CarrinhoViewModel { get; }

}

}

Dessa forma, temos a classe que fornecerá o objeto de resposta quando o usuário clicar nos botões "+", "-" ou alterar a quantidade de itens diretamente na caixa de texto.

Exercício:

# Tratando Resposta da Chamada AJAX

**Você acabou de desenvolver um método UpdateQuantidade no controller Pedido, para receber a chamada de requisições AJAX a partir do código cliente.**

**Esse método devolve um objeto da classe UpdateQuantidadeResponse, que tem duas propriedades:**

## ItemPedido

## CarrinhoViewModel

**Depois que a requisição é feita, o código JavaScript precisa tratar a resposta do método para "imprimir" as informações no console do browser.**

**Qual trecho de código é o mais adequado para essa tarefa?**



**Resumo 3:** Criamos a classe UpdateQuantidadeResponse que será retornada para Carrinho.js na função ajax(), que contém a função done() que por sua vez, receberá como resposta uma instância dessa nova classe.

Iremos preparar o método UpdateQuantidadade() , que está na PedidoController.cs para retornar uma instância da nova classe.

Iremos até a PedidoController.cs para retornar o UpdateQuantidadeResponse. Adicionaremos a palavra return logo a frente de itemPedidoRepository.

[HttpPost]

public void UpdateQuantidade([FromBody]ItemPedido itemPedido)

{

return itemPedidoRepository.UpdateQuantidade(itemPedido);

}

}

}

No entanto, o método UpdadeQuantidade() retorna um void, portanto iremos até itemPedididoRepository e encontraremos a assinatura do método na interface IItemPedidoRepository.

namespace CasaDoCodigo.Repositories

{

public interface IItemPedidoRepository

{

void UpdateQuantidade(ItemPedido itemPedido);

}

Mais abaixo no código encontraremos a implementação concreta do método UpdateQuantidade().

public void UpdateQuantidade(ItemPedido itemPedido)

{

var itemPedidoDB =

dbSet

.Where(ip => ip.Id == itemPedido.Id)

.SingleOrDefault();

if (itemPedidoDB != null)

{

itemPedidoDB.AtualizaQuantidade(itemPedido.Quantidade);

contexto.SaveChanges();

}

}

Trocaremos a palavra void por UpdateQuantidadeResponse. É importante notarmos que o objeto UpdateQuantidadeResponse retorna não apenas o item que é alterado, mas também o CarrinhoViewModel que contém informações do pedido. Como estamos no repositório de ItemPedido não teremos como acessar o pedido a partir daqui, por isso moveremos o método UpdateQuantidade() para o repositório de pedidos, isto é, ItemPedidoRepository.cs. Colaremos todo o trecho destacado ao final da classe, e o removeremos de seu local original.

public UpdateQuantidadeResponse UpdateQuantidade(ItemPedido itemPedido)

{

var itemPedidoDB =

dbSet

.Where(ip => ip.Id == itemPedido.Id)

.SingleOrDefault();

if (itemPedidoDB != null)

{

itemPedidoDB.AtualizaQuantidade(itemPedido.Quantidade);

contexto.SaveChanges();

}

}

Moveremos, ainda, a assinatura da interface. Copiaremos a linha void UpdateQuantidade(ItemPedido itemPedido) em ItemPedidoRepository.cs, depois podemos remover essa assinatura do código.

namespace CasaDoCodigo.Repositories

{

public interface IItemPedidoRepository

{

void UpdateQuantidade(ItemPedido itemPedido);

}

Depois colaremos em PedidoRepository.cs, ou seja, na interface do repositório de pedidos.

namespace CasaDoCodigo.Repositories

{

public interface IPedidoRepository

{

Pedido GetPedido();

void AddItem(string codigo);

void UpdateQuantidade(ItemPedido itemPedido);

}

Iremos retornar a parte do UpdateQuantidade, modificando o retorno da interface de void para UpdateQuantidadeResponse

namespace CasaDoCodigo.Repositories

{

public interface IPedidoRepository

{

Pedido GetPedido();

void AddItem(string codigo);

UpdateQuantidadeResponse UpdateQuantidade(ItemPedido itemPedido);

}

Mais abaixo no código, começaremos a implementar o UpdateQuantidade que foi movida para PedidoRepository.cs.

Temos ItemPedidoDB que será obtido a partir do banco de dados e que inicialmente acessávamos a partir do dbset, que era de ItemPedido, inicalmente. Aqui o dbset é referente a Pedido. Teremos de obter por meio do itemPedidoRepository a informação de ItemPedido.

public UpdateQuantidadeResponse UpdateQuantidade(ItemPedido itemPedido)

{

var itemPedidoDB =

dbSet

.Where(ip => ip.Id == itemPedido.Id)

.SingleOrDefault();

if (itemPedidoDB != null)

{

itemPedidoDB.AtualizaQuantidade(itemPedido.Quantidade);

contexto.SaveChanges();

}

}

Criaremos em ItemPedidoRepository um novo método para obter somente ItemPedido a partir do itemPedidoId. Iseriremos uma assinatura de método que retornará ItemPedido e criaremos um novo método chamado GetItemPedido(), que receberá como parâmetro int itemPedidoId.

namespace CasaDoCodigo.Repositories

{

public interface IItemPedidoRepository

{

ItemPedido GetItemPedido(int itemPedidoId);

}

public class ItemPedidoRepository : BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

}

}

Feito isso, implementaremos o método na classe concreta IItemPedidoRepository. Dessa forma teremos GetItemPedido recebendo itemPedidoId

namespace CasaDoCodigo.Repositories

{

public interface IItemPedidoRepository

{

ItemPedido GetItemPedido(int itemPedidoId);

}

public class ItemPedidoRepository : BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

public ItemPedido GetItemPedido(int itemPedidoId)

{

throw new NotImplementedException();

}

}

}

Dessa forma teremos GetItemPedido() recebendo itemPedidoId. Esse método irá retornar a seguinte expressão registrada em PedidoRepository.cs.

<\*\*\*\*!\*\*\*\*>

dbSet

.Where(ip => ip.Id == itemPedido.Id)

.SingleOrDefault();

<\*\*\*\*!\*\*\*\*>

Iremos copiar essa expressão e colá-la no método GetItemPedido() e adicionar a expressão return. Além disso, substituiremos itemPedido.Id por ItemPedidoId.

namespace CasaDoCodigo.Repositories

{

public interface IItemPedidoRepository

{

ItemPedido GetItemPedido(int itemPedidoId);

}

public class ItemPedidoRepository : BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

public ItemPedido GetItemPedido(int itemPedidoId)

{

return

dbSet

.Where(ip => ip.Id == itemPedidoId)

.SingleOrDefault();

}

}

}

Consumiremos o método GetItemPedido() em PedidoRepository.cs, faremos isso chamando um repositório a partir do outro, realizando uma injeção de dependência por meio da interface. Em PedidoRepository declararemos um campo privado que será chamado de IItemPedidoRepository itemPedidoRepository. Esse item será fornecido via injeção de dependência, portanto precisamos criar um novo parâmetro no construtor da classe, que será itemPedidoRepository. Forneceremos no corpo do construtor this.itemPedidoRepository, atribuindo a este campo local o parâmetro itemPedidoRepository.

public class PedidoRepository : BaseRepository<Pedido>, IPedidoRepository

{

private readonly IHttpContextAccessor contextAccessor;

private readonly IItemPedidoRepository itemPedidoRepository;

public PedidoRepository(ApplicationContext contexto,

IHttpContextAccessor contextAccessor,

IItemPedidoRepository itemPedidoRepository) : base(contexto)

{

this.contextAccessor = contextAccessor;

this.itemPedidoRepository = itemPedidoRepository;

}

Ao final do código, trabalharemos no método UpdateQuantidade(). Removeremos o seguinte trecho do código:

dbSet

.Where(ip => ip.Id == itemPedido.Id)

.SingleOrDefault();

Assim, obteremos itemPedidoDB a parir de ItemPedidoRepository.GetItemPedido, passando como parâmetro itemPedido.Id. A nova forma do código ficará desse modo:

public UpdateQuantidadeResponse UpdateQuantidade(ItemPedido itemPedido)

{

var itemPedidoDB = itemPedidoRepository.GetItemPedido(itemPedido.Id);

if (itemPedidoDB != null)

{

itemPedidoDB.AtualizaQuantidade(itemPedido.Quantidade);

contexto.SaveChanges();

}

Conseguimos pegar ItemPedidoDB e salvando a quantidade de itens no banco de dados. Teremos de retornar do método UpdateQuantidade() uma nova instância de UpdateQuantidadeResponse(), para isso passaremos como parâmetro no construtor ItemPedidoDB e CarrinhoViewModel, que declararemos mais acima do código em uma variável local para que este se torne mais legível. Para CarrinhoViewModel() iremos acionar o método GetPedido() e depois acessaremos a propriedade Itens. Assim feito, passaremos carrinhoViewModel para como outro parâmetro do método UpdateQuantidadeResponse().

public UpdateQuantidadeResponse UpdateQuantidade(ItemPedido itemPedido)

{

var itemPedidoDB = itemPedidoRepository.GetItemPedido(itemPedido.Id);

if (itemPedidoDB != null)

{

itemPedidoDB.AtualizaQuantidade(itemPedido.Quantidade);

contexto.SaveChanges();

var carrinhoViewModel = new CarrinhoViewModel(GetPedido().Itens);

return mew UpdateQuantidadeResponse(itemPedidoDB, carrinhoViewModel);

}

Ainda resta resolver um problema: quando itemPedidoDB for igual a nulo, isto é, quando não for encontrado o Id correto, o que faremos? Precisamos criar uma exceção. Escreveremos trow new ArgumentException(), passando como parâmetro a mensagem "ItemPedido não encontrado".

public UpdateQuantidadeResponse UpdateQuantidade(ItemPedido itemPedido)

{

var itemPedidoDB = itemPedidoRepository.GetItemPedido(itemPedido.Id);

if (itemPedidoDB != null)

{

itemPedidoDB.AtualizaQuantidade(itemPedido.Quantidade);

contexto.SaveChanges();

var carrinhoViewModel = new CarrinhoViewModel(GetPedido().Itens);

return mew UpdateQuantidadeResponse(itemPedidoDB, carrinhoViewModel);

}

throw new ArgumentExecption("ItemPedido não encontrado");

}

}

}

Voltemos à PedidoController e veremos o método UpdateQuantidade() que é chamado no ItemPedidoRepository.

[HttpPost]

public UpdateQuantidadeResponse UpdateQuantidade([FromBody]ItemPedido itemPedido)

{

return itemPedidoRepository.UpdateQuantidade(itemPedido);

}

}

}

Faremos uma pequena alteração e chamaremos diretamente o método em pedidoRepository

[HttpPost]

public UpdateQuantidadeResponse UpdateQuantidade([FromBody]ItemPedido itemPedido)

{

return pedidoRepository.UpdateQuantidade(itemPedido);

}

}

}

Dessa forma temos todas as chamadas operando corretamente. Nas próximas aulas veremos como utilizar o código JavaScript para tratar os retornos e fazer a atualização na tela de carrinho.

# Adicione a propriedade Subtotal em ItemPedido

# IMPORTANTE

Antes de continuar no próximo vídeo, adicione esta propriedade à classe ItemPedido no arquivo CasaDoCodigo/Models/modelo.cs:

**[DataMember]**

**public decimal Subtotal => Quantidade \* PrecoUnitario;**

Esse código nada mais é o do que uma propriedade "read only", isto é, propriedade com get mas sem set, que poderia ser escrita de outra forma:

**[DataMember]**

**public decimal Subtotal**

**{**

**get**

**{**

**return Quantidade \* PrecoUnitario;**

**}**

**}**

Com a propriedade Subtotal, a entidade ItemPedido tem pode fazer ela mesma o cálculo do subtotal, e assim evitamos passar essa responsabilidade de regra de negócio para outros trechos de código da aplicação, como é o caso do código JavaScript que você irá estudar agora.

**Resumo 4:** Começaremos a modificar o valor e quantidade total de itens no carrinho na tela via JavaScript, mas primeiro, veremos uma situação em que removemos a quantidade de itens do carrinho até que ela fique zerada.

Quando zeramos a quantidade de itens no carrinho devemos remover esse item do banco de dados, portanto voltaremos ao ItemPedidoRepository.cs, onde criaremos um método novo para realizar essa remoção quando necessária.

Na interface adicionaremos o método RemoveItemPedido() que retornará um void. Esse método terá como parâmetro int itemPedidoId

namespace CasaDoCodigo.Repositories

{

public interface IItemPedidoRepository

{

ItemPedido GetItemPedido(int itemPedidoId);

void RemoveItemPedido(int itemPedidoId);

}

Assim feito, implementaremos o método na classe concreta: selecionamos IItemPedidoRepository e utilizando o atalho "Ctrl + ." escolheremos a opção "Implementar interface".

public class ItemPedidoRepository : BaseRepository<ItemPedido>, IItemPedidoRepository

{

public ItemPedidoRepository(ApplicationContext contexto) : base(contexto)

{

}

Desas forma, ao final do código teremos:

public void RemoveItemPedido(int itemPedidoId)

{

throw new NotImplementedException();

}

Para removermos o item precisaremos obte-lo no banco de dados, faremos isso por meio do método GetItemPedido(), que receberá como parâmetro itemPedidoId. Iremos remover esse itemPedidoId do dbSet, que é o conjunto de itens do banco de dados. O dbSet possui um método para excluir itens do banco, chamado Remove(), que por sua vez receberá GetItemPedido(itemPedidoId).

public void RemoveItemPedido(int itemPedidoId)

{

dbSet.Remove(GetItemPedido(itemPedidoId));

}

Iremos até PedidoRepository.cs e chamaremos o método RemoveItemPedido() para excluir ItemPedido quando necessário. Adicionaremos a condicional if, e quando o Quantidade foi igual a 0, chamaremos o itemPedidoRepository.RemoveItemPedido(itemPedido.id).

public UpdateQuantidadeResponse UpdateQuantidade(ItemPedido itemPedido)

{

var itemPedidoDB = itemPedidoRepository.GetItemPedido(itemPedido.Id);

if (itemPedidoDB != null)

{

itemPedidoDB.AtualizaQuantidade(itemPedido.Quantidade);

if (itemPedido.Quantidade == 0)

{

itemPedidoRepository.RemoveItemPedido(itemPedido.Id);

}

No código JavaScript, atualizaremos o valor total do carrinho. Procuraremos o html que contém o elemento com a quantidade total de itens em nosso carrinho. Ao inspecionarmos o código html veremos que essa quantidade é armazenada em um <span>, precisamente no atributo numero-itens.

<span numero-itens> Total: 1 itens</span>

Localizaremos esse tributo por meio de uma filtragem do jQuery. No código JavaScript inseriremos o seletor para localizado atributo numero-itens. Quando esse elemento for encontrado, trocaremos o html com a função html(), que receberá Total: 12345 itens

postQuantidade(data) {

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

}).done(function (response) {

let itemPedido = response.itemPedido;

let linhaDoItem = $('[item-id=' + itemPedido.id + ']')

linhaDoItem.find('input').val(itemPedido.quantidade);

linhaDoItem.find('[subtotal]').html((itemPedido.subtotal).duasCasas());

$('[numero-itens]').html('Total: 12345 itens');

debugger;

});

Iremos intercalar a string Total: 12345 itens de forma que possamos substituir os caracteres numéricos pela quantidade total de itens no em carrinhoViewModel. Acessaremos a lista de itens e em seguida obteremos a quantidade de elementos da lista, que conseguimos acessar por meio da função JavaScript lenght.

postQuantidade(data) {

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

}).done(function (response) {

let itemPedido = response.itemPedido;

let linhaDoItem = $('[item-id=' + itemPedido.id + ']')

linhaDoItem.find('input').val(itemPedido.quantidade);

linhaDoItem.find('[subtotal]').html((itemPedido.subtotal).duasCasas());

$('[numero-itens]')

.html('Total:' + carrinhoViewModel.itens.length + 'itens');

debugger;

});

Contudo, carrinhoViewModel ainda não foi declarado, faremos isso na linha acima.

postQuantidade(data) {

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

}).done(function (response) {

let itemPedido = response.itemPedido;

let linhaDoItem = $('[item-id=' + itemPedido.id + ']')

linhaDoItem.find('input').val(itemPedido.quantidade);

linhaDoItem.find('[subtotal]').html((itemPedido.subtotal).duasCasas());

**let carrinhoViewModel = response.carrinhoViewModel;**

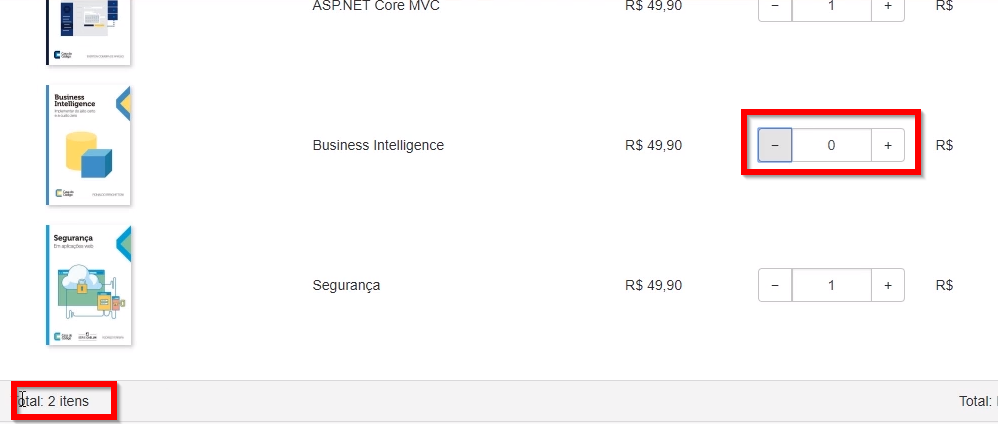
**$('[numero-itens]')**

**.html('Total:' + carrinhoViewModel.itens.length + 'itens');**

debugger;

});

Pressionaremos o atalho "F5" para compilar a aplicação. Depois, iremos até a página inicial e colocaremos três itens diferentes no carrinho de compras. Reduziremos a quantidade de um dos itens até que na caixa de texto seja exibido o valor 0. A quantidade total de itens foi atualizada, contudo o item que apresenta 0 de quantidade não foi removido.



Isso aconteceu porque não configuramos para que o item seja removido da tela. Faremos isso nesta etapa. De volta ao código JavaScript, iremos verificar a quantidade de itens que foram alterados. Incluiremos a condicional if para fazer a verificação de itemPedido, caso ela seja igual a 0, linhaDoItem será removida por meio do método remove().

postQuantidade(data) {

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

}).done(function (response) {

let itemPedido = response.itemPedido;

let linhaDoItem = $('[item-id=' + itemPedido.id + ']')

linhaDoItem.find('input').val(itemPedido.quantidade);

linhaDoItem.find('[subtotal]')

.html((itemPedido.subtotal).duasCasas());

let carrinhoViewModel = response.carrinhoViewModel;

$('[numero-itens]')

.html('Total: ' + carrinhoViewModel.itens.length + ' itens');

**if (itemPedido.quantidade == 0) {**

**linhaDoItem.remove();**

**}**

debugger;

});

}

}

Dessa forma, quando um item estiver com uma quantidade igual a 0, ele será removido da tela e da quantidade total de itens. Contudo, falta atualizarmos o valor total do pedido. Para isso, inseriremos $('[total]').html((carrinhoViewModel.total), e para que a formatação seja correta, inseriremos a função duasCasas() que envolverá toda a expressão.

postQuantidade(data) {

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

}).done(function (response) {

let itemPedido = response.itemPedido;

let linhaDoItem = $('[item-id=' + itemPedido.id + ']')

linhaDoItem.find('input').val(itemPedido.quantidade);

linhaDoItem.find('[subtotal]').html((itemPedido.subtotal).duasCasas());

let carrinhoViewModel = response.carrinhoViewModel;

$('[numero-itens]').html('Total: ' + carrinhoViewModel.itens.length + ' itens');

**$('[total]').html((carrinhoViewModel.total).duasCasas());**

if (itemPedido.quantidade == 0) {

linhaDoItem.remove();

}

debugger;

});

}

}

Com isso, o valor total e quantidade de itens são atualizados.

Exercício:

# Redefina o Valor do Input

# Considere um código JavaScript, onde input é objeto representando um elemento <input> que foi capturado por uma função do jQuery, e itemPedido é uma variável objeto contendo duas propriedades:

**var itemPedido = {**

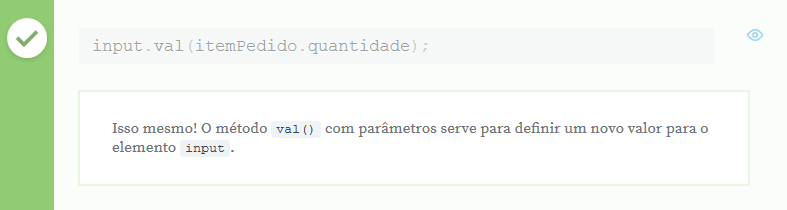
**id: 123,**

**quantidade: 7**

**}**

**var input = $('input.total-do-item');**

**Como você modificaria o valor do elemento input a partir da quantidade da variável itemPedido?**



# Para saber mais - ASP.NET Core MVC

Neste livro, Everton Coimbra apresenta o ASP.NET Core MVC de uma maneira que o leitor já possa aprender com a prática desde o início. O estudo é dirigido ao desenvolvimento de uma aplicação que envolve persistência em uma base de dados, fazendo uso do Entity Framework Core. Você verá desde as operações CRUD até como desenvolver seus layouts e tabelas, separando sua aplicação em camadas seguindo os conceitos de coesão e acoplamento. Dentre as técnicas e recursos apresentados, estão associações, personalização das classes de modelo, controle de acesso de usuários, tratamento de erros, controles DropDownList aninhados e o uso de sessão para o armazenamento de dados.



[**https://www.casadocodigo.com.br/products/livro-aspnet-core-mvc**](https://www.casadocodigo.com.br/products/livro-aspnet-core-mvc)

## ASP.NET Core MVC

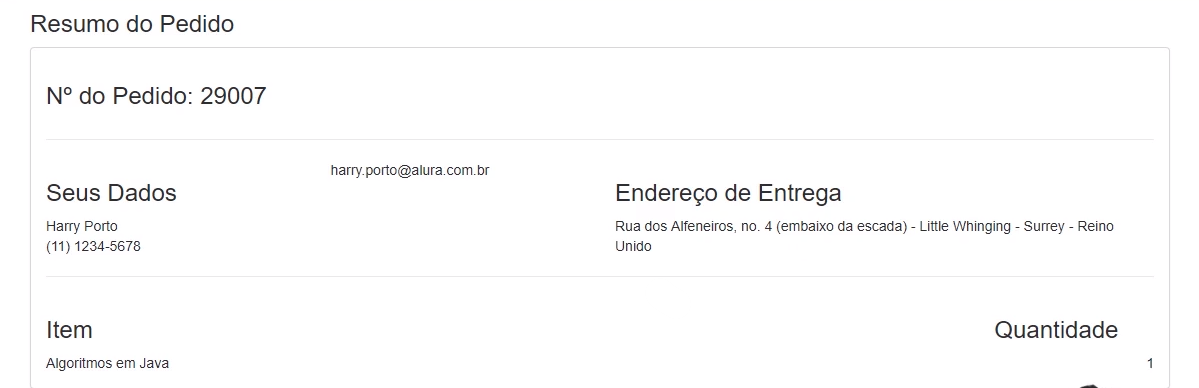
**Aplicações modernas em conjunto com o Entity Framework** **Everton Coimbra de Araújo**

**Aula 3 - Adicionando e Removendo Itens**

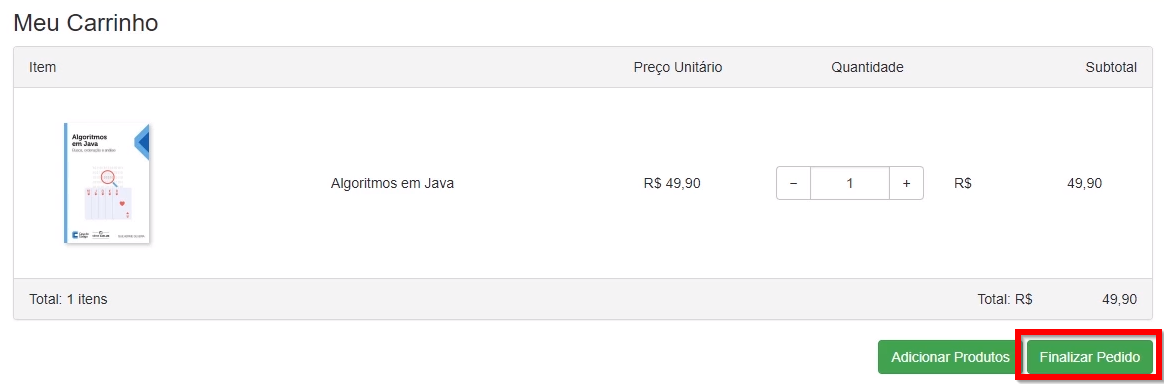
* Criando uma ViewModel
  + Fornecendo dados customizados para uma view
* Preparando o Objeto de Resposta Para a Chamada AJAX
* Refatorando Repositórios
  + Delegando funções entre repositórios com injeção de dependência
* Atualizando o Item na View com JavaScript
  + Os métodos jQuery html() e val()
* Atualizando Totais do Carrinho na View com JavaScript
  + Movendo lógica de negócios para fora da view

**Aula 4 – Resumo 1: //method="post" serve para enviar dados de entrada da view**

Quando terminamos de inserir os produtos no carrinho e queremos finalizar a compra clicando no botão "Finalizar Pedido", somos direcionamos para o resumo de compra, que conterá informações do pedido e do cadastro do cliente.



Contudo, as informações que dispomos são falsas e servem apenas de modelo. Queremos exibir informações que foram coletadas do cliente, para isso precisamos criar uma tela de cadastro. Faremos uma modificação no fluxo de navegação e trocaremos o botão "Finalizar Pedido" da tela de carrinho por outra que direcione o usuário para uma tela de cadastro.



Acessaremos Cadastro.cshtml , isto é, nossa view de cadastramento de usuários. Teremos todos os elementos que compõe o formulário, que até o momento não passa de um conjunto de tags html, portanto precisamos criar os campos para serem preenchidos.

Incluiremos o elemento <form> no começo do código. Tal elemento precisa de algumas informações, como a action, que é o endereço para onde enviaremos as informações, ou seja, /pedido/resumo. Inseriremos, ainda, o método post() cuja função é enviar dados e realizar mudanças na aplicação.

@{

ViewData["Title"] = "Cadastro";

}

<h3>Cadastro</h3>

<form action="/pedido/resumo" method="post">

</form>

<div class="panel panel-default">

Incluiremos em <form> a <div> que contém as informações do cadastro de usuário.

@{

ViewData["Title"] = "Cadastro";

}

<h3>Cadastro</h3>

<form action="/pedido/resumo" method="post">

<div class="panel panel-default">

<div class="panel-body">

<div class="col-md-4">

<div class="form-group">

<label class="control-label" for="nomeCliente">Nome do Cliente</label>

<input type="text" class="form-control" id="nomeCliente" placeholder="Nome do Cliente">

</div>

<div class="form-group">

<label class="control-label" for="email">Email</label>

<input type="email" class="form-control" id="email" placeholder="Email">

</div>

<div class="form-group">

<label class="control-label" for="telefone">Telefone</label>

<input type="text" class="form-control" id="telefone" placeholder="Telefone">

</div>

<\*\*\*\*!\*\*\*\*>

class="btn btn-success">

Finalizar Pedido

</button>

</div>

</div>

</div>

</div>

</form>

Estamos trabalhando com a forma html que será executada no browser. Contudo, existe outra maneira de criar o formulário por meio da *tag helper*. Para isso, trocaremos o atributo action para asp-action. Assim feito, mudaremos o nome da nossa antiga action para uma que se encontra em PedidoController.cs, que será a resumo.

@{

ViewData["Title"] = "Cadastro";

}

<h3>Cadastro</h3>

<form asp-action="resumo" method="post">

<div class="panel panel-default">

<\*\*\*\*!\*\*\*\*>

Começaremos a colocar os campos do formulário e associá-los ao nosso modelo, contudo ainda não definimos um modelo para esta view, faremos isso nesse momento. No começo do código, criaremos uma diretiva model, em seguida, o nome da classe Cadastro.

@{

ViewData["Title"] = "Cadastro";

}

@model Cadastro

<h3>Cadastro</h3>

<form asp-action="resumo" method="post">

<div class="panel panel-default">

<\*\*\*\*!\*\*\*\*>

Precisamos associar cada campo do formulário com uma propriedade da classe Cadastro. Faremos esse procedimento ao modificar cada um dos elementos input. O primeiro é aquele que recebe nomeCliente.

<input type="text" class="form-control" id="nomeCliente" placeholder="Nome do Cliente">

</div>

Declacaremos como tag helper a propriedade que está associada a este input, faremos isso ao criar um novo atributo chamado asp-for. Assim feito, criaremos a associção entre uma propriedade do modelo: `@Model.Nome`

<input type="text" class="form-control" id="nomeCliente" placeholder="Nome do Cliente"

asp-for="@Model.Nome">

</div>

Faremos o mesmo procedimento para todos os inputs do formulário, são eles Email, Telefone, Endereço,Complemento,Bairro, Município e CEP.

<input type="text" class="form-control" id="nomeCliente" placeholder="Nome do Cliente"

asp-for="@Model.Nome">

<span asp-validation-for="@Model.Nome" class="text-danger"></span>

</div>

<div class="form-group">

<label class="control-label" for="email">Email</label>

<input type="email" class="form-control" id="email" placeholder="Email"

asp-for="@Model.Email">

<span asp-validation-for="@Model.Email" class="text-danger"></span>

</div>

<div class="form-group">

<label class="control-label" for="telefone">Telefone</label>

<input type="text" class="form-control" id="telefone" placeholder="Telefone"

asp-for="@Model.Telefone">

<span asp-validation-for="@Model.Telefone" class="text-danger"></span>

</div>

</div>

<\*\*\*\*!\*\*\*\*>

O elemento responsável pelo estado (UF) não se trata de um input, mas de select. Temos a caixa de listagem com todos os estados brasileiros disponíveis. Incluiremos `@Model.UF`.

<div class="form-group">

<label class="control-label" for="UF">UF</label>

<select class="form-control single-select" name="UF"

asp-for="@Model.UF">

<option selected value="">[Selecione o estado]</option>

<option value="AC">Acre</option>

<option value="AL">Alagoas</option>

<option value="AP">Amapá</option>

<option value="AM">Amazonas</option>

<\*\*\*\*!\*\*\*\*>

Nas próximas aulas prepararemos o envio do modelo para a view.

**Exercício:**

# Defina o Binding dos Campos do Formulário

**Você está desenvolvendo uma aplicação MVC com ASP.NET Core.**

**Uma das views possui um formulário de cadastro de dados de cliente, e seu modelo é definido como o tipo Cadastro:**

**@model Cadastro**

**Essa classe possui Cadastro possui propriedades como Nome, Email e Telefone.**

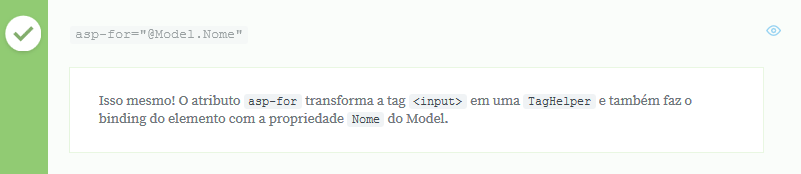
**Para o campo Nome, você cria um <label> e um <input>, conforme segue:**

**<label class="control-label" for="nomeCliente">Nome do Cliente</label>**

**<input type="text" class="form-control" id="nomeCliente" placeholder="Nome do Cliente">**

**Porém, você precisa fazer um binding dessa campo com o modelo da view.**

**Qual atributo você adiciona à tag <input> para fazer o binding com a propriedade Cadastro.Nome e transformá-la em uma TagHelper?**



**Resumo 2:** Nesta aula, faremos com o que PedidoController.cs passe o modelo para a view Cadastro.cshtml. Em PedidoController.cs, trabalharemos no método de Cadastro().

public IActionResult Cadastro()

{

return View();

}

Primeiramente pegaremos o objeto cadastro que se encontra em Pedido, o acessaremos por meio do repositório. Escreveremos pedidoRepository e coletemos o pedido utilizando o método getPedido(). Armazenaremos o objeto dentro de uma variável local pedido.

public IActionResult Cadastro()

{

var pedido = pedidoRepository.GetPedido();

return View();

}

Em seguida, passaremos para a View() o cadastro acessado por meio de pedido.Cadastro. Contudo, existe uma situação em que pedido pode ser nulo, nesse caso não podemos permtir que o usuário prossiga na compra, portanto o direcionaremos para a página inicial do projeto, isto é, a Carrossel. Utilizaremos o método RedirectToAction() e passaremos como parâmetro a action de destino Carrossel.

public IActionResult Cadastro()

{

var pedido = pedidoRepository.GetPedido();

if (pedido == null)

{

return RedirectToAction("Carrossel");

}

return View(pedido.Cadastro);

}

Executaremos a nossa aplicação partido da página inicial e escolheremos um item qualquer para adicioná-lo ao carrinho. Assim feito, clicaremos sobre o botão "Preencher Cadastro" para acessarmos a tela de formulário.



Na página de cadastro teremos os formulários em branco a serem preenchidos



Clicaremos com o botão direito sobre a página e selecionaremos a opção "Inspecionar", desse modo conseguiremos visualizar o código html gerado pelo servidor.

Encontraremos a tag <form> gerada a partir da tag helper. Teremos a action marcada como resumo, mas o Asp.NET gerou a action corretamente no html como /Pedido/resumo.

<h3>Cadastro</h3>

<form method="post" action="/Pedido/resumo" novalidate="novalidate"> == $0

Mais abaixo veremos que para cada um dos campos foi criado um input html com uma série de informações. Veremos que o input de Nome possui uma série de validações que se encontram no modelo.cs, como, por exemplo, que o nome é obrigatório e precisa ser preenchido. ("The Nome field is required.")

<input type="text" class="form-control" id="nomeCliente" placeholder="Nome do Cliente" placeholder="Nome do Cliente" data-val="true" data-val-required="The Nome field is required." name="Nome" value> == $0

Ao observarmos o nosso arquivo modelo.cs veremos que em Cadastro() teremos os campos marcados como obrigatórios.

public Cadastro()

{

}

public virtual Pedido Pedido { get; set; }

[Required]

public string Nome { get; set; } = "";

[Required]

public string Email { get; set; } ="";

[Required]

public virtual Telefone { get; set; } = "";

[Required]

<\*\*\*\*!\*\*\*\*>

O Asp.NET Core facilita nosso trabalho ao inserir uma série de tags html para indicar que um campo é obrigatório.

**Resumo 3:** Conseguimos passar modelo para a view, e de volta recebemos no PedidoController as informações do cadastro por meio da action de resumo, contudo precisamos prepará-la para lidar corretamente com esse tipo de requisição.

Ao entrarmos no browser, perceberemos que é possível acessar a action por meio de uma url digitada na barra de endereço: localhost:50040/Pedido/resumo.

Contudo, essa não deveria ser uma possibilidade, pois deveríamos conseguir chegar até resumo a partir do envio de um formulário, feito por uma requisição POST.

Por conseguinte iremos restringir o acesso à action inserindo uma notação com o atributo http na PedidoController.cs, o que irá impedir a chamada direta do browser.

[HttpPost]

public IActionResult Resumo()

{

return View(pedidoRepository.GetPedido());

}

Outra modificação a ser realizada na action é a inserção de um parâmetro para recebermos os dados de cadastro. Inseriremos o parâmetro cadastroe criaremos um break point na linha return View(pedidoRepository.GetPedido());.

[HttpPost]

public IActionResult Resumo(Cadastro cadastro)

{

return View(pedidoRepository.GetPedido());

}

Pressionaremos a tecla "F5" para compilar a aplicação. Feito isso, iremos abrir a página principal no browser e selecionaremos um produto qualquer, seremos direcionados à página de carrinho e clicaremos sobre o botão "Preencher Cadastro". Com isso, seremos direcionados para a página de formulário.

Iremos preencher alguns campos do cadastro, são eles "Nome do Cliente", "Email" e "Telefone". Preenchidos os campos, clicaremos sobre o botão "Finalizar Pedido", para avaliarmos o resultado.

Quando clicamos no botão de finalização de pedido, não conseguimos acessar a action de resumo. Isso ocorre porque temos uma série de validações sendo feitas pelo cliente. Caso inspecionemos a caixa de texto "Endereço" por exemplo, veremos que há um elemento hmtl input com uma série de atributos de validação. Tais atributos são utilizados pelo JavaScript do jQuery Validation Unobtrusive - um dos frameworks utilizados no projeto - como condição para que o cliente avance nas próximas etapas de compra.

<inuot type="text" class="form-control input-validation-error" id="endereco" placeholder="Endereço" data-val="true" data-val-required="The Endereco field is required." name="Endereco" value aria-required="true" aria-describedby="endereco-error" aria-invalid="true"> == $0

Precisamos exibir para o usuário uma mensagem de que os campos de cadastro não estão corretamente preenchidos, para isso incluiremos alguns elementos html na view de cadastro. Abriremos Cadastro.cshtml e para cada campo do formulário, incluiremos um elemento hmtl que indicará que o campo possui um problema de preenchimento.

Logo depois do <input>, incluiremos um elemento <span> que terá uma classe CSS denominada text-danger, que por sua vez irá marcar o texto com a cor vermelha. Feito isso, transformaremos a classe em uma tag helper, incluindo o atributo específico asp-validation-for que receberá o valor `@Model.Nome`.

<div class="form-group">

<label class="control-label" for="telefone">Telefone</label>

<input type="text" class="form-control" id="telefone" placeholder="Telefone"

asp-for="@Model.Telefone">

<span asp-validation-for="@Model.Telefone" class="text-danger"></span>

</div>

</div>

<div class="col-md-4">

<div class="form-group">

<label class="control-label" for="endereco">Endereço</label>

<input type="text" class="form-control" id="endereco" placeholder="Endereço"

asp-for="@Model.Endereco">

<span asp-validation-for="@Model.Endereco" class="text-danger"></span>

</div>

<\*\*\*\*!\*\*\*\*>

No caso de UF temos um select, portando depois dele iremos inserir o <span> com a propriedade `@Model.UF`.

<div class="form-group">

<label class="control-label" for="UF">UF</label>

<select class="form-control single-select" name="UF"

asp-for="@Model.UF">

<option selected value="">[Selecione o estado]</option>

<option value="AC">Acre</option>

<option value="AL">Alagoas</option>

<option value="AP">Amapá</option>

<option value="AM">Amazonas</option>

<\*\*\*\*!\*\*\*\*>

</select>

<span asp-validation-for="@Model.UF" class="text-danger"></span>

Com as modificações realizadas em todos os campos, iremos atualizar a página e clicaremos na opção "Finalizar Pedido", na página de cadastro. Veremos que todos os campos que não foram preenchidos surgem com a mensagem em vermelho "The field is required", dessa forma orientamos o usuário a preencher os campos corretamente.

# Exercicio: Valide o Campo do Formulário

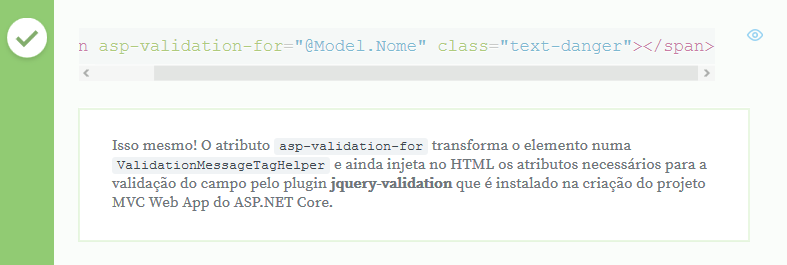
**Você está montando um cadastro de clientes em uma aplicação MVC usando ASP.NET Core. Na view contendo o cadastro, você cria o label e também o input para o campo Nome:**

**<label class="control-label" for="nomeCliente">Nome do Cliente</label>**

**<input type="text" class="form-control" id="nomeCliente" placeholder="Nome do Cliente"**

**asp-for="@Model.Nome">**

**Agora é necessário implementar a validação do campo no lado do cliente (no navegador). Qual linha de código de validação você acrescentaria? Escolha a melhor resposta.**



**Resumo 4:** Temos a validação do lado do cliente, e já podemos começar a tratar do cadastro quando ele for recebido no servidor por meio da action resumo. Essa action receberá o parâmetro Cadastro, e ao final gravaremos essa informação no banco de dados.

Para realizar a gravação no banco, utilizamos CastroRepositoy.cs. Neste arquivo, teremos a interface ICadastroRepository .

namespace CasaDoCodigo.Repositoires

{

public interface ICadastroRepository

{

}

public class CadastroRepository : BaseRepository<Cadastro> ICadastroRepository

{

public CadastroRepository(ApplicationContext contexto) : base(contexto)

{

}

}

}

Criaremos um novo método chamado Update() para atualizar as informações do cadastro. Tal método retornará um objeto Cadastro e receberá duas informações: cadastroId e Cadastro novoCadastro.

namespace CasaDoCodigo.Repositoires

{

public interface ICadastroRepository

{

Cadastro Update(int cadastroId, Cadastro novoCadastro)

}

public class CadastroRepository : BaseRepository<Cadastro>, ICadastroRepository

{

public CadastroRepository(ApplicationContext contexto) : base(contexto)

{

}

}

}

Assim feito, criaremos um método na classe concreta CadastroRepository a partir da interface. Clicaremos com o botão direito sobre ICadastroRepository e selecionaremos a opção "Ações Rápidas e Refatorações > Implementar interface". Dessa forma teremos um método novo, mas por enquanto não inseriremos o seu código definitivo, apenas um break point na linha throw new NotImplementedException();.

namespace CasaDoCodigo.Repositoires

{

public interface ICadastroRepository

{

Cadastro Update(int cadastroId, Cadastro novoCadastro)

}

public class CadastroRepository : BaseRepository<Cadastro>, ICadastroRepository

{

public CadastroRepository(ApplicationContext contexto) : base(contexto)

{

}

public Cadastro Update(int CadastroId, novoCadastro)

{

throw new NotImplementedException();

}

}

}

Para acessarmos esse novo método do repositório de cadastro, acessaremos o repositório de pedido, afinal é ele que contém informações do pedido que está na sessão do Asp.NET Core. Portanto iremos até PedidoRepository.cs e criaremos uma referência para CadastroRepository.cs por meio de um campo privado. Esse campo será alimentado via injeção de dependência, logo criaremos um novo parâmetro no construtor (cadastroRepository) que será alimentando no corpo do construtor no momento em que acessarmos this.cadastroRepository = cadastroRepository;

public class PedidoRepository : BaseRepository<Pedido>, IPedidoRepository

{

private readonly IHttpContextoAccessor contextAccessor;

private readonly IItemPedidoRepository itemPedidoRepository;

private readonly ICadastroRepository cadastroRepository;

public PedidoRepository(ApplicationContext contexto,

IHttpContextoAccessor contextAccessor,

IItemPedidoRepository itemPedidoRepository,

ICadastroRepository castroRepository) : base(contexto)

{

this.contextAccessor = contextAccessor;

this.itemPedidoRepository = itemPedidoRepository;

this.cadastroRepository = cadastroRepository;

}

Precisamos criar outro método na PedidoRepository que será responsável pela atualização do cadastro. Na interface, criaremos esse novo método chamado UpdateCadastro() que retornará Pedido. Por sua vez, o método receberá o cadastro sendo modificado na view (Cadastro cadastro).

public interface IPedidoRepository

{

Pedido GetPedido();

void AddItem(string codigo);

UpdateQuantidadeResponse UpdateQuantidade(ItemPedido itemPedido);

Pedido UpdateCadastro(Cadastro cadastro);

}

Faremos a classe concreta implementar o novo método clicando sobre IPedidoRepository com o botão direito e selecionando as opções "Ações Rápidas e Refatorações > Implementar Interface". Ao final da classe, teremos o método UpdateCadastro().

<\*\*\*\*!\*\*\*\*>

public Pedido UpdateCadastro(Cadastro cadastro)

{

throw new NotImplementedException():

}

Acessaremos cadastroRepository.Update(), que receberá Cadastro.Id, que está associado a pedido, portanto primeiramente pegaremos pedido, declarando uma variável pedido = GetPedido(). O segundo parâmetro do método Update() será cadastro. Ao final das alterações, deveremos retornar Pedido a partir do método UpdateCadastro(), faremos isso escrevendo return pedido, isto é, a variável local.

<\*\*\*\*!\*\*\*\*>

public Pedido UpdateCadastro(Cadastro cadastro)

{

var pedido = GetPedido();

cadastroRepository.Update(pedido.Cadastro.Id, cadastro);

return pedido;

}

Votaremos à PedidoController.cs e consumiremos o método UpdateCadastro(). Trabalharemos no seguinte trecho do código:

[HttpPost]

public IActionResult Resumo(Cadastro.castro)

{

return View(pedidoRepository.GetPedido());

}

Ao invés de utilizarmos return View(pedidoRepository.GetPedido()); inseriremos return View(pedidoRepository.UpdateCadastro(cadastro));

[HttpPost]

public IActionResult Resumo(Cadastro.castro)

{

return View(pedidoRepository.UpdateCadastro(cadastro));

}

Contudo, temos de levar em consideração o seguinte quadro: pode ser que não tenhamos preenchido corretamente as informações na view, isto é, Cliente.cshtml. Por isso é sempre importante validar os dois lados, tanto cliente quanto servidor. Para termos certeza de que o modelo foi preenchido corretamente e que as regras de negócio presentes são válidas, devemos verificar o estado do modelo. Para realizarmos essa verificação, inseriremos if e acessaremos o objeto ModelState, em seguida faremos uso da propriedade IsValid. Caso o modelo esteja correto, o cadastro será atualizado.

[HttpPost]

public IActionResult Resumo(Cadastro.castro)

{

if(ModelState.IsValid)

{

return View(pedidoRepository.UpdateCadastro(cadastro));

}

}

Se o modelo não for válido, retornaremos o usuário para a view de cadastro até que as informações tenham sido preenchidas corretamente. Para fazer esse redirecionamento do usuário, utilizamos return RedirectToAction(), que receberá como parâmetro o nome action correta, no caso, "Cadastro".

[HttpPost]

public IActionResult Resumo(Cadastro.castro)

{

if(ModelState.IsValid)

{

return View(pedidoRepository.UpdateCadastro(cadastro));

}

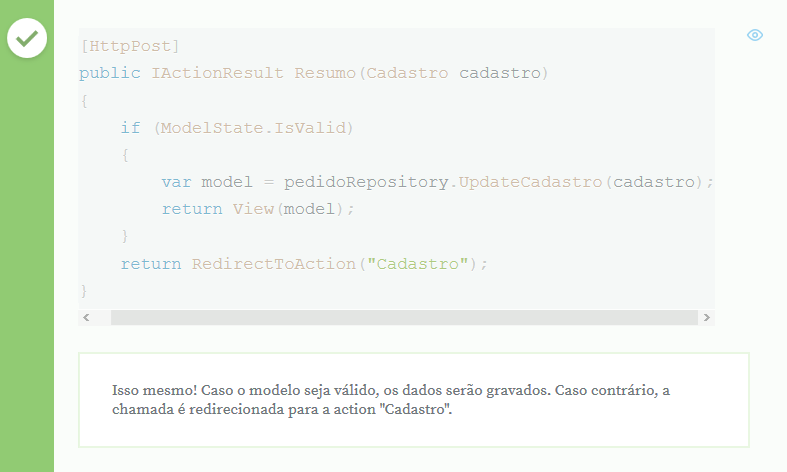
return RedirectToAction("Cadastro");

}

Nas próximas aulas iremos programar a gravação do cadastro no banco de dados.

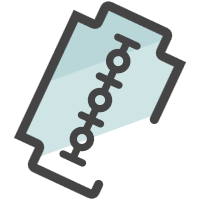
**Exercicio:** Retornando uma View Diferente

**Você precisa fazer uma action de uma aplicação MVC com ASP.NET Core para receber uma requisição partindo de um formulário de cadastro de cliente. Caso os dados do modelo sejam válidos, a action deve gravar os dados em banco de dados. Caso contrário, uma view diferente a requisição deve ser retornada a partir da action. Qual alternativa contém o código necessário para realizar essa tarefa? Escolha a melhor resposta.**



# Para saber mais

No curso de Asp.Net MVC tivemos uma visão geral sobre o funcionamento de uma aplicação web, desde o controller até a camada de visualização. Nesse curso daremos o foco à camada de visualização da aplicação e aprenderemos como utilizar efetivamente o Razor para gerar as views da aplicação.



[**https://www.alura.com.br/curso-online-razor**](https://www.alura.com.br/curso-online-razor)

## Curso online de ASP.NET: Razor

**Victor Kendy Harada**

***Faça esse curso e***...

* Use o Razor para trabalhar com o seu HTML
* Utilize Templates no seu dia a dia
* Trabalhe com internacionalização
* Lide com views fortemente tipadas

**Aula 4 - Validando e Enviando Formulário de Cadastro**

* Montando a View de Cadastro
  + Entendendo formulários ASP.NET Core
  + Binding de campos com propriedades do modelo com o atributo asp-for
* Injetando o Cadastro na View e Redirecionando se Necessário
  + O método RedirectToAction()
* Validando Campos do Formulário no Cliente
  + Validando campos do formulários com o atributo asp-validation-for
* Customizando Validações no Modelo de Cadastro
  + Validando entidades do modelo com MinLength, MaxLength e Required
* Validando Dados do Formulário no Servidor
  + Consultando ModelState.IsValid()

**Aula 5 – Resumo 1:** Para gravamos o cadastro no banco de dados, precisaremos passar por algumas etapas: primeiramente preencheremos na view de cadastro as informações, depois os dados serão recebidos na action Resumo, local em que iremos validar os dados no lado do servidor. Depois que o modelo for validado, chamaremos o método UpdateCadastro(), que está em PedidoRepository.cs.

O método UpdateCadastro() obterá pedido a partir do método GetPedido(), depois, navegará para Cadastro coletará o Id para passar para o método Update() do repositório.

public Pedido UpdateCadastro(Cadastro cadastro)

{

var pedido = GetPedido();

cadastroRepository.Update(pedido.Cadastro.Id, cadastro);

return pedido;

}

Como pedido saberá qual é o Id de Cadastro? Primeiramente, faremos uma consulta para averiguar a informação. Navegaremos para GetPedido(), em PedidoRepository.cs para estudar melhor essa questão.

Obtemos o pedidoId que está na sessão utilizando o método GetPedidoId(). Uma vez que esta informação for coletava, faremos uma consulta Linq, que procurará pedidoId e retornará algumas entidades, como p.Itens e i.Produto, além do Idem si.

public Pedido GetPedido()

{

var pedidoId = GetPedidoId();

var pedido = dbSet

.Include(p => p.Itens)

.ThenInclude(i => i.Produto)

.Where(p => p.Id == pedidoId)

.SingleOrDefault();

if (pedido == null)

{

pedido = new Pedido();

dbSet.Add(pedido);

contexto.SaveChanges();

SetPedidoId(pedido.Id);

}

return pedido;

}

Não especificamos em nenhum momento do código que Cadastro também deveria ser consultado. Utilizaremos o método Include() para que os dados de Cadastro, inclusive o Id, sejam verificados.

public Pedido GetPedido()

{

var pedidoId = GetPedidoId();

var pedido = dbSet

.Include(p => p.Itens)

.ThenInclude(i => i.Produto)

.Inclue(p =>p.Cadastro)

.Where(p => p.Id == pedidoId)

.SingleOrDefault();

if (pedido == null)

{

pedido = new Pedido();

dbSet.Add(pedido);

contexto.SaveChanges();

SetPedidoId(pedido.Id);

}

return pedido;

}

Começaremos a implementar a gravação de Cadastro no banco de dados. Em CadastroRepository.cs teremos o método Updade().

public Cadastro Update(int cadastroId, Cadastro novoCadastro)

{

throw new NotImplementedException();

}

O primeiro passo é armazenarmos em uma variável local o cadastro proveniente do banco de dados, chamaremos esta variável de cadastroDB, que será um objeto de cadastro a partir de dbSet. Contudo, devemos realizar um filtro a partir do cadastroId. Para isso, escreveremos o método Where() com uma expressão lambda.

Em outra linha adicionaremos o método SingeOrDefault(), para que seja trazido um cadastro ou nenhum, e caso a informação não seja encontrada, quer dizer que o objeto é nulo e precisamos tratá-lo de alguma maneira.

Inseriremos a condicional If para tratarmos do objeto com valor nulo. Declararemos uma exceção ArgumentNullException(),que receberá como parâmetro "cadastro"

public Cadastro Update(int cadastroId, Cadastro novoCadastro)

{

var cadastroDB = dbSet.Where(c => c.Id == cadastroId)

.SingleOrDefault();

if (cadastroDB == null)

{

throw new ArgumentNullException("cadastro");

}

}

Prosseguiremos coletando o novoCadastro, recebido como parâmetro, para atualizar o banco de dados. Escreveremos ao final do código cadastroDB.Update(), e passaremos como parâmetro novoCadastro, que possui os novos dados. Em seguida, usaremos o método SaveChanges() para gravarmos as informações no banco. Ao final, retornaremos CadastroDB.

public Cadastro Update(int cadastroId, Cadastro novoCadastro)

{

var cadastroDB = dbSet.Where(c => c.Id == cadastroId)

.SingleOrDefault();

if (cadastroDB == null)

{

throw new ArgumentNullException("cadastro");

}

}

cadastroDB.Update(novoCadastro);

contexto.SaveChanges();

return CadastroDB;

Contudo, não possuímos o método Update() no modelo de cadastro, isto é, não possuímos um método para atualizar informações a partir de um segundo cadastro. Iremos implementar o método Update(): com ele selecionado,pressionaremos o atalho "Ctrl + ." e escolheremos a opção "Gerar método 'Cadastro.Update'".

Dessa forma, o método Updade() já foi gerado na classe Cadastro, em modelo.cs.

internal void Update(Cadastro novoCadastro)

{

throw new NotImplementedException();

}

Iremos implementar o método Update(). Inseriremos this.bairro - a primeira propriedade - como sendo igual a novoCadastro.Bairro. Faremos esse mesmo procedimento para todas as outras propriedades.

internal void Update(Cadastro novoCadastro)

{

this.Bairro = novoCadastro.Bairro;

this.CEP = novoCadastro.CEP;

this.Complemento = novoCadastro.Complemento;

this.Email = novoCadastro.Email;

this.Endereco = novoCadastro.Endereco;

this.Municipio = novoCadastro.Municipio;

this.Nome = novoCadastro.Nome;

this.Telefone = novoCadastro.Telefone;

this.UF = novoCadastro.UF;

}

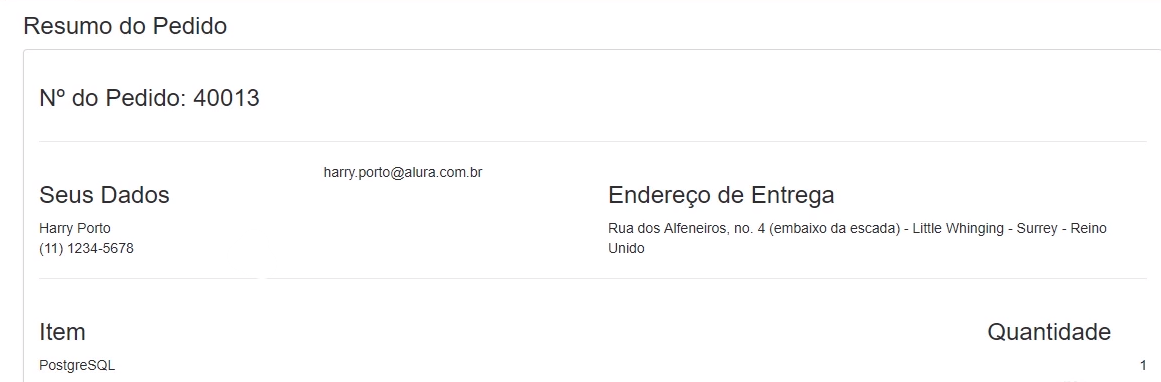
}

Nas próximas aulas executaremos a aplicação para verificar se conseguimos gravar as informações no banco de dados.

**Resumo 2:** Testaremos a gravação das informações cadastrais no banco de dados. No browser, preencheremos todos os campos obrigatórios na página de cadastro e ao final clicaremos sobre o botão "Finalizar Pedido".



Uma vez que finalizarmos o cadastro, seremos encaminhados para a view de resumo. Neste ponto, todo o processo de transferência de dados do cadastro para o repositório e gravação no banco de dados já foi realizado.



De volta ao Visual Studio, acessaremos os dados da tabela dbo.Cadastro para conferir os dados gravados. Essa arquivo encontra-se na pasta Tabelas do projeto. Veremos todas as informações que foram preenchidas nos campos de cadastro.

| **Id** | **Bairro** | **CEP** | **Complemento** | **Email** | **Endereco** | **Municipio** | **Nome** | **Telefone** | **UF** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 40009 |  |  |  |  |  |  |  |  |  |
| 40010 |  |  |  |  |  |  |  |  |  |
| 40012 |  |  |  |  |  |  |  |  |  |
| 40013 | Vila Mariana | 15151515151 | fundos | fulano@gmail.com | Rua Vergueiro,... | sao paulo | fulado de tal | 12345678 | SP |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL |

Contudo devemos refletir: \**será que nosso sistema está bem protegido contra acessos externos? \**

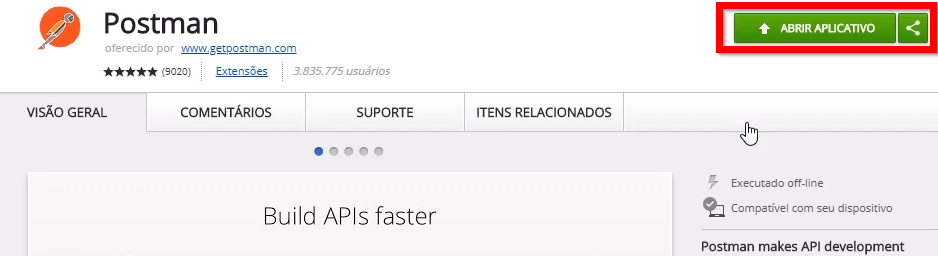
Marcamos a action com o acesso do tipo "POST", o que impede que qualquer usuário acesse a action de resumo por meio da URL, mas isso não é o suficiente, pois podemos sofrer um ataque conhecido como **CSRF**, ***Cros-site request forgery***, em uma tradução livre "Falsificação de solicitação entre sites". Quando recebemos uma requisição proveniente de fora do nosso site, existe a possibilidade de que se trata de um agente malicioso, com interesse no furto de dados ou algo do gênero.

De volta a tela de cadastro veremos a existência de vários campos, mas temos um campo oculto. Ao inspecionarmos a página de cadastro, veremos um <input> marcado com o valor hidden, isto é, "oculto". Em seguida, teremos uma informação criptografada, conhecida por **token de verificação**. Essa informação precisa ser validada pelo servidor, caso contrário podemos sofrer um ataque externo.

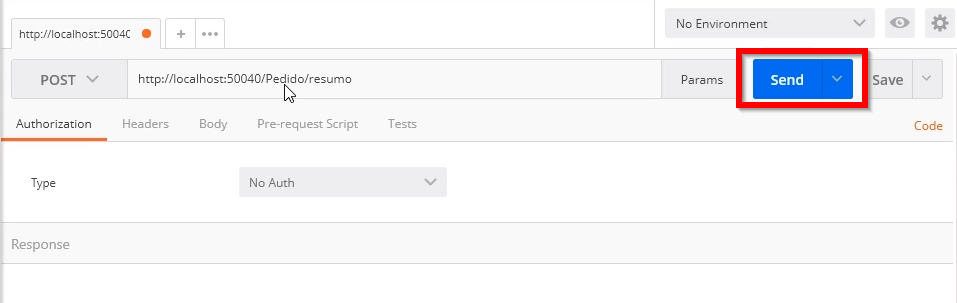
<input name="\_\_RequestVerificationToken" type="hidden" value=

"CfDJ8DGA9YtWRddDkFJM71llQFM7JaUU49wEH-y8MjecfFu7xM9EMQTS39cHlzZDBJI5\_QppOv3dSWg5fGyvg11QWqImS4IOPR6ia8QrFashYO9u3LxLGNDUQ6sQ1gc-tJU49Mp2qH5C\_\_axg"> == $0

Simularemos um ataque por meio de uma ferramenta chamada **Postman** para Chrome. Basta procurar o termo no Google e realizar o download e depois abrir o programa.



Este é um recurso bem interessante para simularmos requisições. Na página inicial do programa, inseriremos no campo "POST" - que configuramos como sendo o tipo de acesso que utilizamos - o endereço da action de resumo: http://localhost:50040/Pedido/resumo. Depois, clicaremos sobre o botão "Send".



Obteremos a resposta da action de resumo.

Em PedidoController.cs inseriremos um break point na linha if (ModelState.IsValid).

[HttpPost]

public IActionResult Resumo(Cadastro cadastro)

{

if (ModelState.IsValid)

{

return View(pedidoRepository.UpdateCadastro(cadastro));

}

return RedirectToAction("Cadastro");

}

Faremos novamente a requisição no programa Postman clicando sobre o botão "Send". Seremos redirecionados novamente para a action Resumo. Isso pode ser um agente mal intencionado querendo acessar o site. Nesses casos, devemos utilizar o token criptografado na view de cadastro para garantirmos a segurança da aplicação.

Pararemos o programa e inseriremos na action o atributo ValidateAntiFogeryToken que valida o tolken no momento em que recebemos uma requisição.

**[HttpPost]**

**[ValidateAntiForgeryToken]**

**//ValidateAntiForgeryToken => Reforço de segurança em requisições por token**

**//Utilize o Postman para fazer testes de segurança**

**//CSRF (Cross-site Request Forgery, ou Falsificação de solicitação entre sites)**

**public IActionResult Resumo(Cadastro cadastro)**

**{**

**if (ModelState.IsValid)**

**//Se o estado do Model (Modelo formulário e entrada de dados) for válida/true**

**{**

**return View(pedidoRepository.updateCadastro(cadastro));**

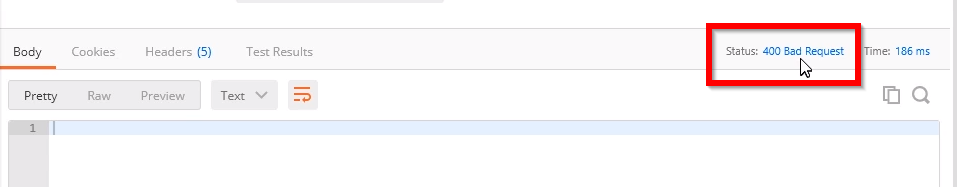
**}**

**return RedirectToAction("Cadastro");**

**}**

Pressionaremos o atalho "F5" para compilarmos a aplicação e seremos direcionados para a página principal, escolheremos um produto qualquer e preencheremos o formulário de cadastro.

Iremos até o Postman e tentearemos fazer uma nova requisição. Dessa vez,o programa acusou o status da requisição como "400 Bad Request", afinal não foi permitido o acesso pelo servidor, pois não fornecemos o token por meio do Postman.



**Exercicio:** Proteja a Action do Controller

**Você está desenvolvendo uma aplicação de comércio eletrônico (MVC Web App) com ASP.NET Core.Em determinado momento, você necessita proteger uma action do controller contra ataques CSRF (Cross-site Request Forgery, ou Falsificação de solicitação entre sites).**

**Qual estratégia você tomaria para proteger a action abaixo? Escolha a melhor alternativa.**

**public IActionResult Resumo(Cadastro cadastro)**

**{**

**if (ModelState.IsValid)**

**{**

**return View(pedidoRepository.UpdateCadastro(cadastro));**

**}**

**return RedirectToAction("Cadastro");**

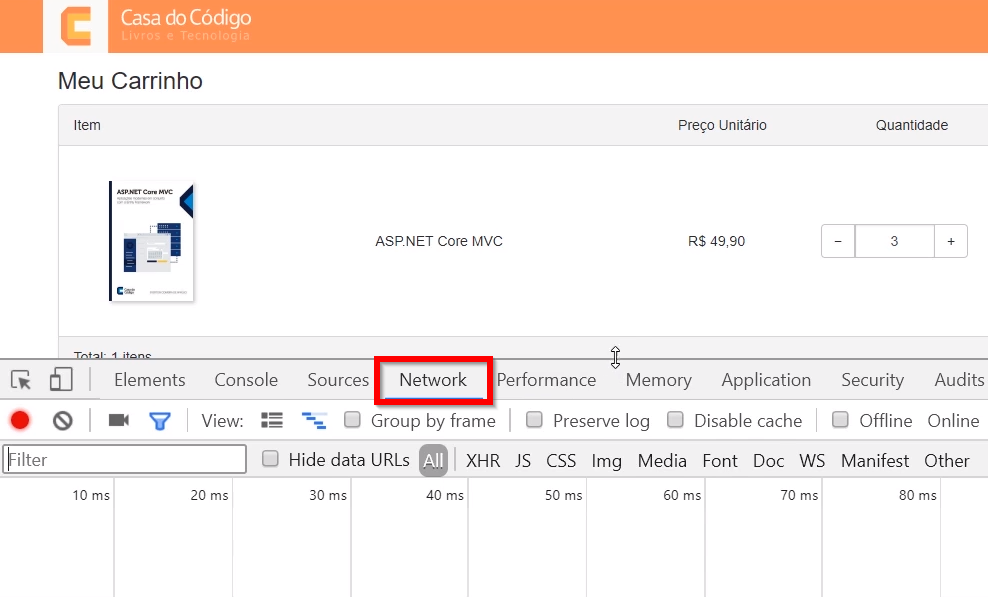
**}**



**Resumo 3:** Conseguimos proteger com sucesso a action de resumo contra o ataque de algum agente malicioso, mas ainda falta um outro ponto critico da aplicação: o método que recebe a requisição da chamada ajax().

Quando clicamos nos botões "+" ou "-" para alterarmos a quantidade de itens no carrinho de compras, temos na PedidoController.cs o método UpdateQuantidade() que recebe essa requisição e faz a alteração no número de itens. Precisamos proteger esse método contra ataques externos. É importante que sempre validemos o token anti-falsificação. Antes disso, simularemos um ataque feito contra o método UpdateQuantidade().

Na página de carrinho aberta no browser, pressionaremos o atalho "F12" e teremos acesso às ferramentas do desenvolvedor. Selecionaremos a aba "Network", desse modo poderemos ver todas as requisições feitas dentro da página.



Clicaremos sobre o botão "+" do carrinho para verificarmos quais são as execuções da página e qual endereço será acessado. Somos direcionados para o método UpdateQuantidade() em PedidoController.cs

[Http]

public UpdadeQuantidadeResponse UpdateQuantidade([FromBody]ItemPedido itemPedido)

{

return pedidoRepository.UpdateQuantidade(itemPedido);

}

Pressionaremos "F5" e voltaremos ao navegador. Pressionaremos "F5" e voltaremos ao navegador. Verificaremos que a aplicação no cliente se interrompeu na linha debugger de carrinho.js.

<\*\*\*\*!\*\*\*\*>

if (itemPedido.quantidade == 0) {

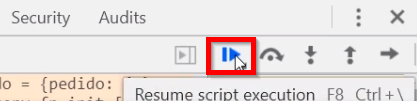
linhaDoItem.remove();

}

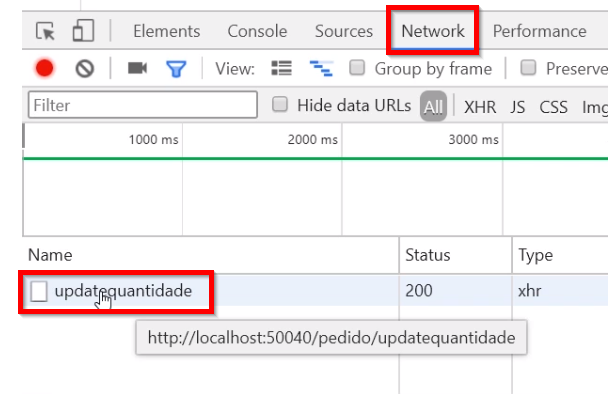
debugger;

});

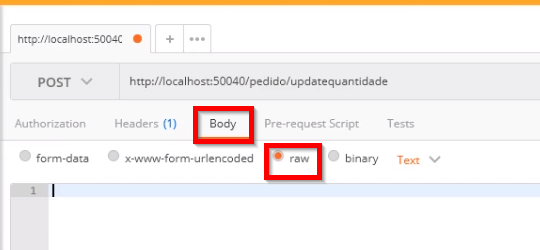
Na área de ferramentas do desenvolvedor, pressionaremos o botão "Resume Script Execution" para dar prosseguimento à execução do JavaScript.



Clicaremos sobre a aba "Network" , e encontraremos o endereço chamado, isto é, a URL de updatequantidade que é http://localhost:50040/pedido/updatequantidade. Clicaremos sobre a opção updatequantidade e copiraremos esta URL, para isso basta clicar com o botão direitos obre a opção e selecionar "copy > copy link address".



Voltaremos ao Postman e montaremos uma nova requisição, simulando um ataque de fora da aplicação. Modificaremos o método http para "POST" e colocaremos a URL no campo correspondente. Assim feito, selecionaremos as opções "Body" - que corresponde ao corpo da requisição - e depois marcaremos a opção "raw", que se fere ao texto puro a ser enviado.



Nesse texto puro o que será enviado? Será copiado do browser o Request Payload, que contém o texto enviado no corpo da requisição.

Request Payload

{Id: "41019", Quantidade: 4}

Id: "41019""

Quantidade:4

Copiaremos a linha {Id: "41019", Quantidade: 4} e colaremos no campo de texto do Postman. Assim feito, modicaremos a opção "Text" para "JSON(application/json)".

Desse modo nossa requisição está pronta e podemos pressionar o botão "Send".

Seremos direcionados à PedidoController.cs e veremos que a execução se interrompeu no break point, na linha return pedidoRepository.UpdateQuantidade(itemPedido);.

[HttpPost]

public UpdadeQuantidadeResponse UpdateQuantidade([FromBody]ItemPedido)

{

return pedidoRepository.UpdateQuantidade(itemPedido);

}

Temos algumas informações passadas no parâmetro ItemPedido: o Id e a Quantidade. Voltaremos a requisição no Postman e alteraremos o número de Quantidade de 4 para 4000 e enviaremos a requisição.

{Id: "41019", Quantidade: 4000}

Na página no carrinho no browser, veremos que a quantidade de itens é 4000 e o valor total da compra é de 196600,00. Percebemos que Postman está conseguindo acessar a nossa aplicação e nosso trabalho é impedir esse acesso indevido, aplicando um atributo de validação do token anti-falsificação (ValidateAntiForgeryToken) em PedidoController.

[HttpPost]

[`ValidateAntiForgeryToken`]

public UpdadeQuantidadeResponse UpdateQuantidade([FromBody]ItemPedido)

{

return pedidoRepository.UpdateQuantidade(itemPedido);

}

Executaremos a aplicação no browser. Escolheremos um item qualquer e seremos direcionados para a página do carrinho. No Postman enviaremos novamente a requisição {Id: "41019", Quantidade:"4000"}. Dessa vez, o acesso é impedido e a requisição recebe o status de erro "400 Bad Request".

Precisamos pensar em como passar esse tolken por meio do carrinho, uma vez que nessa configuração, ao pressionarmos o botão de "+" a quantidade de itens não é alterada.

Pressionaremos o atalho "F12" para acessarmos as ferramentas do desenvolvedor. Feito isso, selecionaremos a aba "Console", e verificaremos um erro Failed toload resource: the serve responded with a status of 400 (Bad Request).

Ao inspecionarmos os elementos da página de carrinho, não encontraremos o token necessário para realizar a validação, como havia na tela de cadastro.

O que precisamos fazer é fornecer o token para view Carrinho.cshtml. Logo depois do cabeçalho criaremos uma tag helper <form>, com a informação de método (method) que será o post. Inseriremos, ainda, a informação asp-action, ou seja, para onde o formulário irá enviar as informações, no caso inseriremos o próprio carrinho.

@{

ViewData["Title"] = "Carrinho";

}

@model CarrinhoViewModel;

<h3>Meu Carrinho</h3>

<form method="post" asp-action="carrinho">

</form>

Não inseriremos mais nenhuma informação dentro do formulário, pois <form> foi criado apenas com o objetivo de fornecer para view o tolken anti-falsificação.

De volta à página de carrinho e pressionaremos a telca "F5". Inspecionaremos os elementos e procuraremos o elemento <form>. Veremos que ele carrega um elemento a mais que é oculto (*hidden* ), mas carrega o valor do token, uma informação bastante importante.

<form method="post" action="/Pedido/carrinho/035">

input name="\_RequestVerificationToken" type="hidden" value=

"CfDJ8DGA9YtwEdDkFJM71llQFN\_JsCz06wU.Iuks-

U\_8Wgi7XM8GoelLqoLkfp12\_XS\_4UT19AKL2dlwodoifFSFssdemmskAI92femsowdaqwb\_DpC8mFYjdsIWfzjfe

iEFIFlWOX2oDCVEFEUHWKDEI3rfjzcxfi2g"> == $0

</form>

Enviaremos o tolken para o servidor. Para isso, iremos para o arquivo carrinho.js e procuraremos a requisição ajax(), pois nesse monto do código nós montaremos um objeto que conterá um cabeçalho onde iremos inserir o token.

Escreveremos: let headers, e este objeto de cabeçalho será vazio, e atribuiremos a ele uma chave RequestVerificationToken. Passaremos o token. Em seguida, criaremos uma variável token que será alimentada com o conteúdo do input que vimos anteriormente, cujo nome é RequestVerificationToken. O obteremos por meio de um seletor do JQuery, por isso inseriremos o caractere $. Resta incluirmos o nome do atributo, que será name.

postQuantidade(Data) {

let token = $('[name=\_\_RequestVerificationToken]').val()

let headers = {};

headers['RequestVerificationToken'] = token;

$ajax({

url:'/pedido/updatequantidade',

type: 'POST',

data: JSON.stringify(data)

})done(function (response) {

<\*\*\*\*!\*\*\*\*>

Em seguida, devemos acessar o valor do token, para isso incluiremos a função do JQuery val(). Desse modo pegaremos o token, o passaremos para o objeto headers e atribuiremos este objeto abaixo da linha data: JSON.stringify(data).

postQuantidade(Data) {

let token = $('[name=\_\_RequestVerificationToken]')

let headers = {};

headers['RequestVerificationToken'] = token;

$ajax({

url:'/pedido/updatequantidade',

type: 'POST',

data: JSON.stringify(data),

headers: headers

})done(function (response) {

<\*\*\*\*!\*\*\*\*>

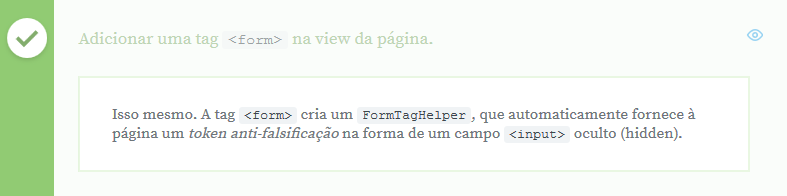
Salvaremos as modificações e voltaremos a página de carrinho. Clicaremos sobre o botão "+" e verificaremos que as quantidades do item são alteradas e o método UpdadateQuantidade()consegue ser acessado. Dessa forma, nossa aplicação está mais segura e funcional.

**Exercicio:** Forneça um Token Anti-Falsificação para a Página

**Você precisa fazer uma requisição AJAX no JavaScript para um método no servidor que exige um token anti-falsificação.**

**Porém, a view de onde o JavaScript é executado não contém o token anti-falsificação.**

**Que estratégia você escolheria para fornecer um token anti-falsificação à página.**



**Exercício:** Passe o Token Anti-Falsificação na Chamada Ajax

**Você precisa enviar um token anti-falsificação através de uma requsição AJAX no código JavaScript. Observe o trecho de código abaixo, e escolha a modificação mais adequada para realizar essa tarefa.**

**var data = {**

**Id: 123,**

**Quantidade: 7**

**};**

**$.ajax({**

**url: '/pedido/updatequantidade',**

**type: 'POST',**

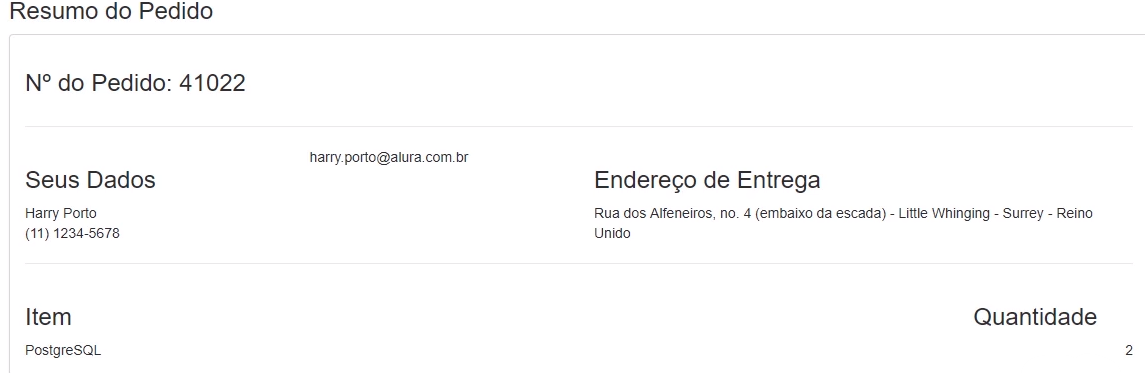
**contentType: 'application/json',**

**data: JSON.stringify(data)**

**})**



**Resumo 4:** Nosso projeto está quase finalizado, mas resta resolvermos a view de resumo, já que não temos dados do cadastro real do nosso cliente. Os dados concretos da página de resumo são apenas "Número do Pedido" e os itens que foram comprados. Todo o resto das informações são falsas, como "Seus Dados" e "Endereço de Entrega".



No arquivo Resumo.cshtml iremos substituir os campos que estão fixos no código. Começaremos com o nome do usuário Harry Porto:

<div class="col-md-3">

<h3>Seus Dados</h3>

<div>Harry Porto</div>

<div>(11) 1234-5678</div>

</div>

Acessaremos modelo utilizando @Model, pois queremos utilizar um código C# dentro da view. Utilizaremos Cadastro para acessar Nome.

<div class="col-md-3">

<h3>Seus Dados</h3>

<div>@Model.Cadastro.Nome</div>

<div>(11) 1234-5678</div>

</div>

Faremos o mesmo procedimento para o campo de telefone e email:

<div class="row">

<div class="col-md-3">

<h3>Seus Dados</h3>

<div>@Model.Cadastro.Nome</div>

<div>@Model.Cadastro.Telefone</div>

</div>

<div class="col-md-3">

@Model.Cadastro.Email

</div>

Mais abaixo teremos o campo de endereço, que é composto por múltiplas informações.

<div class="col-md-6">

<h3>Endereço de Entrega</h3>

<div>Rua dos Alfeneiros, no. 4 (embaixo da escada) - Little Whinging - Surrey - Reino Unido</div>

</div>

Trocaremos o nome da rua por @.Model.Cadastro.Endereco, o complemento por `@Model.Cadastro.Complemento` e seguiremos a mesma lógica para os outros dados.

<div class="col-md-6">

<h3>Endereço de Entrega</h3>

<div>@Model.Cadastro.Endereco @Model.Cadastro.Complemento - @Model.Cadastro.Bairro - CEP: @Model.Cadastro.CEP - @Model.Cadastro.Municipio - @Model.Cadastro.UF - Brasil</div>

</div>

Feitas as alterações a página de resumo do pedido conterá os dados reais do usuário, coletados a partir do cadastro.

**Resumo 5:** Para darmos um toque final à aplicação, abriremos o arquivo JavaScript carrinho.js.

Iremos refatorar a classe Carrinho, trocando os nomes das viriáveis com a finalidade de deixar nossa aplicação mais clara. Vamos observá-la em seu atual estado:

class Carrinho {

clickIncremento(btn) {

let data = this.getData(btn);

data.Quantidade++;

this.postQuantidade(data);

}

clickDecremento(btn) {

let data = this.getData(btn);

data.Quantidade--;

this.postQuantidade(data);

}

updateQuantidade(input) {

let data = this.getData(input);

this.postQuantidade(data);

}

getData(elemento) {

var linhaDoItem = $(elemento).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQtde = $(linhaDoItem).find('input').val();

return {

Id: itemId,

Quantidade: novaQtde

};

}

Em clickIncremento(), temos como parâmetro btn, mas isso não confere muita clareza ao nosso código. Tocaremos esse parâmetro pelo nome button. Faremos o mesmo procedimento para clickDecremento() e nos outras linhas de código que apresentam btn.

class Carrinho {

clickIncremento(button) {

let data = this.getData(button);

data.Quantidade++;

this.postQuantidade(data);

}

clickDecremento(button) {

let data = this.getData(button);

data.Quantidade--;

this.postQuantidade(data);

}

Mais abaixo teremos a função getData(), que contém algumas variáveis. Uma delas é novaQtde, também presente em return.

getData(elemento) {

var linhaDoItem = $(elemento).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQtde = $(linhaDoItem).find('input').val();

return {

Id: itemId,

Quantidade: novaQtde

};

}

Substituiremos novaQtde por novaQuantidade.

getData(elemento) {

var linhaDoItem = $(elemento).parents('[item-id]');

var itemId = $(linhaDoItem).attr('item-id');

var novaQuantidade = $(linhaDoItem).find('input').val();

return {

Id: itemId,

Quantidade: novaQuantidade

};

}

Mais adiante encontraremos a chamada ajax(), e ao final veremos a linha com a instrução debugger, responsável por parar a aplicação quando o programa estive sendo depurado no browser.

$.ajax({

url: '/pedido/updatequantidade',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify(data)

}).done(function (response) {

let itemPedido = response.itemPedido;

let linhaDoItem = $('[item-id=' + itemPedido.id + ']')

linhaDoItem.find('input').val(itemPedido.quantidade);

linhaDoItem.find('[subtotal]').html((itemPedido.subtotal).duasCasas());

let carrinhoViewModel = response.carrinhoViewModel;

$('[numero-itens]').html('Total: ' + carrinhoViewModel.itens.length + ' itens');

$('[total]').html((carrinhoViewModel.total).duasCasas());

if (itemPedido.quantidade == 0) {

linhaDoItem.remove();

}

debugger;

});

}

}

Contudo, essa instrução não é necessária quando formos enviar o projeto para a produção, por isso retiraremos a linha debuggerdo código.

<\*\*\*\*!\*\*\*\*>

if (itemPedido.quantidade == 0) {

linhaDoItem.remove();

}

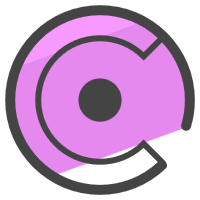
});

}

}

Dessa forma, concluímos a refatoração do nosso código.

# Para saber mais - Curso Entity Framework Core



[**https://cursos.alura.com.br/course/entity-framework-core**](https://cursos.alura.com.br/course/entity-framework-core)

## Curso Entity Framework Core: Banco de dados de forma eficiente

**Daniel Portugal**

**Aulas**

* Introdução ao Entity Framework Core
* Manipulando objetos
* Como o Entity monta o SQL?
* Sincronizando o banco de dados com o modelo de classes
* Relacionamentos Um para Muitos no Entity
* Relacionamentos Muitos para Muitos no Entity
* Relacionamentos Um para Um no Entity
* Recuperando objetos relacionados

O que aprendemos?

**Aula 5 - Recebendo e Gravando Dados de Cadastro**

* Gravando Cadastro no Banco de Dados
* Protegendo uma Action com ValidateAntiForgeryToken
  + Defendendo sua aplicação contra ataques CSRF
* Protegendo a Action do AJAX Contra Ataques Maliciosos
  + Usando token anti-falsificação numa requisição AJAX
* Aplicando os Campos do Cadastro do Modelo na View de Resumo
  + Renderizando modelo do Cadastro nos campos com Razor
* Refatorando o Arquivo Carrinho.js
* Conclusão