

Criando aplicações para o seu

Windows Phone



© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

Sumário

1	Antes de tudo	1
1.1	Sobre o autor	1
1.2	Agradecimentos	2
2	Estamos na Era Mobile	3
2.1	E esse tal de Windows Phone?	4
2.2	Por que desenvolver para Windows Phone?	5
2.3	O que você encontrará neste livro	6
2.4	Antes de começarmos	6
3	Conhecendo o SDK e criando nossa primeira aplicação	9
3.1	Criando nossa primeira aplicação e conhecendo o emulador	11
3.2	Configurando seu dispositivo e sua conta como desenvolvedor	14
3.3	Blend	15
4	Utilizando componentes do sistema para melhorar a experiência do usuário	17
4.1	Princípios de design	18
4.2	Começando nosso aplicativo	20
4.3	Gerenciadores de layout	20
4.4	Criando a página de boas-vindas	26
5	Design de aplicações	39
5.1	Melhorias de usabilidade	39
5.2	Conhecendo os controles Panorama, ApplicationBar e Pivot	46

6	LocalDB, ApplicationSettings e utilização de dados no aplicativo	55
6.1	Carregando os dados para a aplicação	56
6.2	Vinculando os dados com os controles de usuário	61
6.3	LocalDB	75
6.4	ApplicationSettings e os impactos em nosso aplicativo	84
7	Navegação e ciclo de vida	89
7.1	Navegação entre páginas	89
7.2	Ciclo de vida da aplicação	97
7.3	PageState	99
7.4	ApplicationState e inicialização rápida	103
8	Integração com o sistema operacional	109
8.1	Finalizando as funcionalidades já criadas	109
8.2	Entendendo os Live tiles	115
8.3	Criando tiles secundários	122
8.4	Launchers e Choosers	126
9	Mapas e localização	131
9.1	Criando a página para finalizar compra	131
9.2	Entendendo o componente Map	137
9.3	Finalizando nosso aplicativo	139
10	Dicas para publicação na loja do Windows Phone	147
10.1	Passos para publicação	147
10.2	Conclusão	148

CAPÍTULO 1

Antes de tudo

Esta obra é resultado do trabalho e esforço direto e indireto de diversas pessoas. Este primeiro capítulo é dedicado a agradecimentos às pessoas envolvidas e também para que você possa saber mais sobre o autor.

1.1 SOBRE O AUTOR

Eu, Gabriel Schade Cardoso, sou um jovem de 23 anos (2014), graduado em Ciência da Computação, estudante de mestrado em Inteligência Artificial e amante da tecnologia, em especial da área de desenvolvimento de software. Meu primeiro computador foi aos 10 anos de idade, um presente que mudou minha vida. Como quase todos os garotos dessa idade, eu gostava muito de vídeo games e de jogar RPG com meus amigos. Isso me influenciou para utilizar meu computador para pesquisar sobre como os jogos eram criados e, mesmo sem perceber, aos 11 anos eu já estava estudando programação e criando jogos simples.

Depois disso não foi difícil decidir o que eu queria fazer da vida. Comecei a explorar a programação fora da área de jogos e sou fascinado por isso até hoje. Tenho

sorte de conseguir trabalhar com o que gosto, estou com cinco anos de experiência na área de desenvolvimento de software e nesse tempo acumulei conhecimentos nas linguagens Java, C++, Javascript e claro, C#.

Atualmente, além de trabalhar na área de desenvolvimento de software, também escrevo com frequência no blog: www.programadopoliglota.com.br e busco participar de eventos na comunidade técnica e científica, através de artigos, palestras e apresentações.

1.2 AGRADECIMENTOS

Seria muito egoísmo de minha parte não agradecer a todas as pessoas que me ajudam direta e indiretamente em todas as minhas tarefas.

Agradeço: À minha mãe, Eulália, por sempre estar comigo e por conseguir suportar pesos absurdos para manter seus filhos seguros e saudáveis, mesmo nos momentos mais difíceis.

À minha melhor amiga e irmã Daniela, por ser minha conselheira e confidente.

A toda minha família, por ser a luz que me guia, até nos momentos mais escuros.

Aos meus amigos da época de faculdade, Jhony, Rodrigo e João, pessoas das quais eu tenho orgulho de chamar de amigo e com certeza as pessoas mais brilhantes que eu conheço.

A todos os meus amigos e pessoas queridas, que são a estrutura vital que me empurra para cima e me faz ser quem eu sou.

À Editora Casa do Código, por oferecer mais uma oportunidade, criar toda a estrutura necessária para a realização desta obra e por acreditar em mim e neste projeto.

A todas essas pessoas fica o meu mais sincero **obrigado**.

CAPÍTULO 2

Estamos na Era Mobile

Estamos na era **mobile**: hoje em dia não é nada surpreendente ter um smartphone. O atual cenário despertaria bastante curiosidade a pouco tempo atrás, mas o fato é que agora carregar um smartphone em um bolso da calça é quase tão comum quanto a carteira que se carrega no outro bolso. As pessoas estão cada vez mais conectadas, compartilhando coisas, lendo notícias e e-mails, gravando vídeos, conversando com os amigos distantes e fazendo até videoconferências.

Com um dispositivo razoavelmente pequeno conseguimos fazer grande parte das tarefas que executamos em um computador, que é bem mais pesado e menos portátil. De acordo com a Forbes, estima-se que em 2017 mais de 80% dos dispositivos conectados à internet serão tablets e smartphones. Para nós, desenvolvedores de software, é importantíssimo investir nesse mercado e manter-se atualizado para este novo paradigma.

2.1 E ESSE TAL DE WINDOWS PHONE?

Você já ouviu falar do Windows Phone? Caso não, saiba que ele é o sistema operacional para dispositivos smartphones da Microsoft, sendo uma alternativa aos sistemas mais conhecidos e já consolidados, como o iOS da Apple e Android da Google.

Apesar de serem diferentes, dispositivos Android e iOS possuem muitas similaridades em questão de interface e de experiência com o usuário. O Windows Phone chegou praticamente três anos depois desses sistemas com uma proposta bem diferente. Na figura 2.1 podemos comparar a tela inicial das últimas versões até a data da publicação deste livro dos sistemas supracitados, Android, iOS e Windows Phone, respectivamente.



Figura 2.1: Sistemas operacionais móveis

Este sistema foca bastante na experiência do usuário e em uma proposta diferenciada de interface. Na tela inicial podemos ver diversos quadros, no lugar de ícones — estes quadros são conhecidos por **tiles**. Estes tiles podem mostrar informações atualizadas a todo momento, tanto com informações locais do sistema, quanto com informações online.

Diferente dos ícones das aplicações, os tiles não se limitam a um por aplicativo. O usuário pode decidir fixar tiles diversos para o mesmo aplicativo. Um exemplo disso seria o aplicativo **pessoas**, que agrupa as informações sobre seus contatos. Caso você

queira, pode inserir um tile para abrir este aplicativo, ou até mesmo para abri-lo diretamente no perfil de determinada pessoa. Por estes motivos a Microsoft considera seu sistema operacional como o sistema mais personalizável do mercado.

Antes deste sistema, a Microsoft já havia criado uma proposta para sistemas operacionais móveis, tanto para smartphones quanto para pocket PCs. Este sistema era conhecido como **Windows Mobile**. Diferente do que muitas pessoas pensam, o Windows Phone não é uma continuação do Windows Mobile; ele é um sistema novo, totalmente reescrito e até mesmo incompatível com seu “irmão mais velho”. Entretanto, agora o Windows Phone 8 possui um *kernel* compartilhado com o próprio sistema operacional Windows, conhecido como **Windows NT**.

2.2 POR QUE DESENVOLVER PARA WINDOWS PHONE?

Essa é uma pergunta muito válida e que todo desenvolvedor deve se fazer antes de investir tempo para aprender uma nova plataforma. A resposta para esta pergunta pode parecer simplória, mas é totalmente válida. Este é um sistema novo e pouco explorado, diferente das lojas de aplicativos já saturadas nos concorrentes. Claro que isso não quer dizer que você não deva desenvolver para as outras plataformas, pelo contrário, na minha opinião, as três plataformas que citei são totalmente válidas para um desenvolvedor nelas apostar.

Além disso, o marketshare do Windows Phone vem aumentando mais de 100% ao ano desde de 2011, o que faz com que ele se torne o segundo sistema operacional mais popular em diversos países (incluindo Brasil), perdendo apenas para o Android da Google. Algumas pesquisas indicam que o Windows Phone deve ultrapassar o IOS da Apple em larga escala até 2015.

Caso você já conheça C# e a plataforma .NET, você já possui uma grande vantagem, pois como há este kernel compartilhado entre o Windows e o Windows Phone, é possível compartilhar boa parte de seu código entre as diferentes plataformas. Se já conhece aplicações WPF ou aplicações Windows 8/8.1 que utilizam a linguagem XAML como interface, você só terá de aprender como utilizar as APIs disponíveis para consumir os recursos do dispositivo.

É possível publicar seus aplicativos facilmente através da loja Windows Phone Store. Todos os usuários podem acessá-la através de seus dispositivos e encontrar seus aplicativos lá. Caso você já possua uma conta de desenvolvedor para a loja de aplicativos do Windows 8/8.1, você pode utilizar a mesma conta para publicar seus aplicativos móveis.

2.3 O QUE VOCÊ ENCONTRARÁ NESTE LIVRO

Este livro é escrito para desenvolvedores que já possuem conhecimento dos principais conceitos relacionados à orientação a objetos e que já tiveram contato com a linguagem C#. Para facilitar a compreensão de desenvolvedores menos experientes nessa linguagem, serão dadas pequenas explicações sobre alguns aspectos da linguagem. No capítulo inicial iremos desenvolver uma aplicação de teste para termos nossa primeira experiência, e depois vamos construir uma aplicação até o fim do livro.

Os capítulos seguintes irão abordar conceitos que envolvem desde a criação de uma boa interface até a conexão com dados, GPS, integração de sistemas e muito mais. É importante fomentar que as aplicações neste livro não vão considerar todos os possíveis padrões arquiteturais e melhorias de performance. Como este é um guia para iniciantes na plataforma, serão priorizados exemplos com clareza, facilidade e funcionalidades diversas do sistema.

Todos os exemplos apresentados aqui poderão ser encontrados no repositório:

<https://github.com/gabrielschade/LivroWindowsPhone>.

Qualquer dúvida que você encontrar você é só buscar ajuda no grupo de discussão:

<http://bit.ly/WPgrupo>

A meta deste livro é encorajar e iniciar o leitor a criar aplicações móveis, não só no sistema abordado, mas em qualquer outro, sendo um primeiro passo para o desenvolvimento nesta nova era de dispositivos. Espero que você consiga aproveitar o conteúdo deste livro de forma fluida, simples e divertida, e que ele possa ser um bom guia para sua nova jornada.

2.4 ANTES DE COMEÇARMOS

Antes de começarmos a falar de aplicações móveis, paradigmas de programação e tudo mais, é necessário que você já tenha feito o download das ferramentas necessárias. Primeiro precisamos ter um ambiente de desenvolvimento: neste livro será utilizado o **Visual Studio 2013 Ultimate** — caso não possua esta ferramenta, você pode baixar uma versão gratuita do Visual Studio 2012:

<http://www.visualstudio.com/en-us/downloads#d-express-windows-phone>

E o SDK (software development kit) pode ser baixado aqui:

<http://dev.windowsphone.com/en-us/downloadsdk>

É importante salientar que você não precisa de um dispositivo Windows Phone para testar suas aplicações, já que o Visual Studio possui um emulador que irá nos ajudar com isso. Entretanto, apesar de ser um bom emulador, eu aconselho ao desenvolvedor ter, sim, um dispositivo com este sistema para testar suas aplicações que possuam fins comerciais, bem como para garantir uma boa experiência e usabilidade.

Tendo essas ferramentas em mãos e uma boa noção de programação, você estará apto a seguir neste livro, criando no próximo capítulo sua primeira aplicação para Windows Phone.

CAPÍTULO 3

Conhecendo o SDK e criando nossa primeira aplicação

Neste capítulo iremos conhecer um pouco melhor o kit de desenvolvimento para Windows Phone (SDK) e seus principais recursos. Teremos uma introdução ao ambiente de desenvolvimento no Visual Studio e criaremos uma primeira aplicação. Dessa forma, vamos ter a experiência de testar nosso aplicativo e conhecer o emulador, bem como outras ferramentas que nos ajudarão ao longo do processo de desenvolvimento.

No Visual Studio, você deve ser capaz de criar projetos que possuem a plataforma Windows Phone como alvo. Para fazer isso, selecione a opção `File -> New -> Project`, e depois, o tipo de projeto **Windows Phone**, conforme a figura [3.1](#).

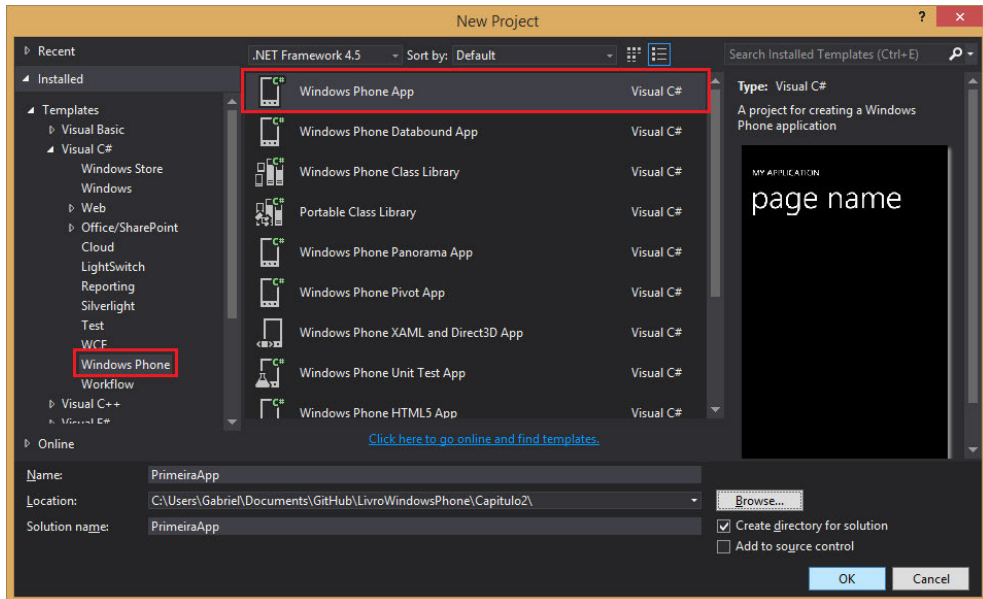


Figura 3.1: Novo projeto para Windows Phone

Você verá que existem diversos templates de aplicativos, mas não se assuste: todos eles têm um propósito definido — alguns irão apenas criar atalhos para projetos, como por exemplo, *Panorama App* ou *Pivot App*. Mas mesmo que você escolha a opção *Windows Phone App*, você não estará privado destes recursos. Abordaremos alguns tipos de projetos ao longo do livro, porém agora iremos escolher a opção *Windows Phone App* como template para nosso primeiro projeto.

Após criar seu projeto, você verá que esta IDE fornece tudo que precisamos para construirmos nossa App, desde a criação da interface, de seu *code behind* e até ferramentas de teste. Ao longo do livro iremos explorar cada um destes recursos para que você possa criar o melhor aplicativo possível.

Agora você já pode executar sua aplicação! Quando fizer isso, será aberto um emulador do Windows Phone em seu Windows Desktop, que ficará em um ambiente isolado e virtualizado, imitando um dispositivo totalmente separado de seu computador. O botão para iniciar o navegador também é uma lista de seleção; nela poderemos escolher diversas opções diferentes de emulador e até mesmo testar em nosso dispositivo. A figura 3.2 ilustra isso.

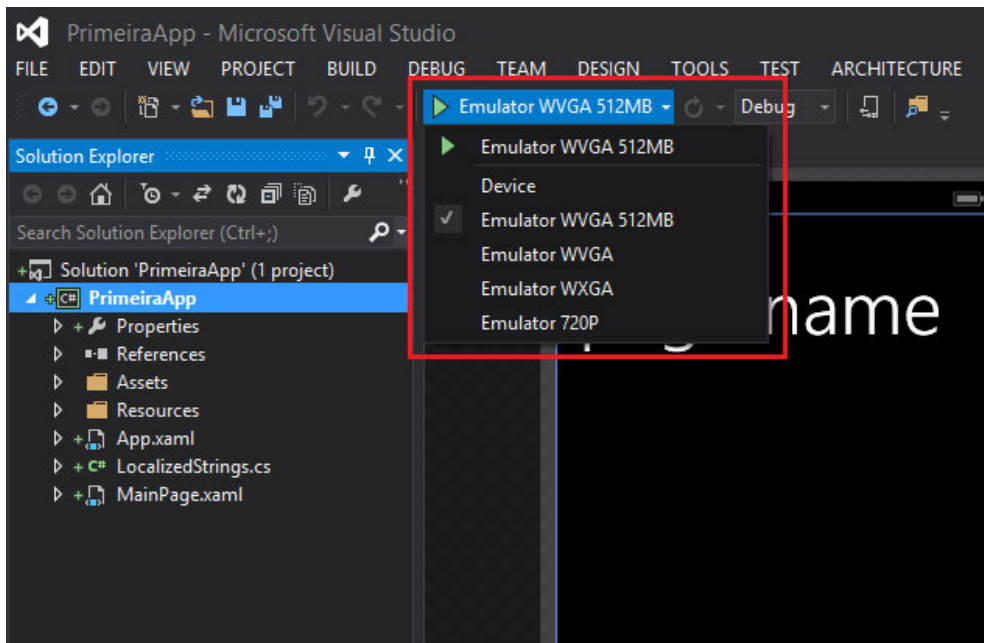


Figura 3.2: Emuladores para Windows Phone

3.1 CRIANDO NOSSA PRIMEIRA APLICAÇÃO E CONHECENDO O EMULADOR

Como você deve ter notado ao criar o projeto, nós iniciamos com o arquivo `MainPage.xaml` aberto. Como o nome sugere, ele é a página principal do nosso aplicativo, mas como você pode ver, não é o único arquivo que possuímos no projeto. Vamos agora conhecer a estrutura básica de um aplicativo Windows Phone.

O arquivo `MainPage`, não é o único que possui extensão `XAML` (*eXtensible Application Markup Language*), nosso projeto também conta com o arquivo `App.xaml`. Ele permite que declaremos recursos globais, ou seja, que poderão ser acessados através de diferentes páginas e controles em nosso aplicativo. Como padrão, já existe o recurso `LocalizedStrings`, mas por enquanto basta sabermos que ele está aqui. Outro ponto importantíssimo é que, neste arquivo, são definidos os manipuladores dos eventos disparados durante o ciclo de vida da aplicação, como por exemplo, `Closing` e `Deactivated`. Além do `App.xaml` também há um `App.xaml.cs`, que é o *code-behind* deste arquivo, ou seja, é nele que estão as definições dos métodos C#

que manipulam os eventos citados anteriormente e é esta classe que se encarrega de inicializar o aplicativo.

Você também poderá notar a pasta **Assets** na raiz de seu aplicativo. Ela já estará com alguns arquivos, como por exemplo, as imagens que serão utilizadas nos tiles e como ícone de sua aplicação. Também há a pasta **Resources**, que possui como conteúdo um arquivo de recursos, que pode ser utilizado para internacionalização de seu aplicativo.

Além dos já citados, há um componente importantíssimo para qualquer aplicação Windows Phone: trata-se do arquivo `WMAppManifest.xml`. Ele possui os metadados sobre sua aplicação, o que envolve o ícone de sua aplicação, o tipo do tile que será exibido, quais recursos do sistema operacional sua aplicação precisa para funcionar corretamente (lista de contatos, por exemplo), recursos de hardware (giroscópio ou NFC, por exemplo) e até os dados sobre o pacote de seu aplicativo na loja. Apesar de ser um arquivo XML, você não precisa se preocupar em conhecer as tags e tudo mais, o Visual Studio possui uma interface bastante amigável para alterar estas configurações, como ilustra a figura 3.3.

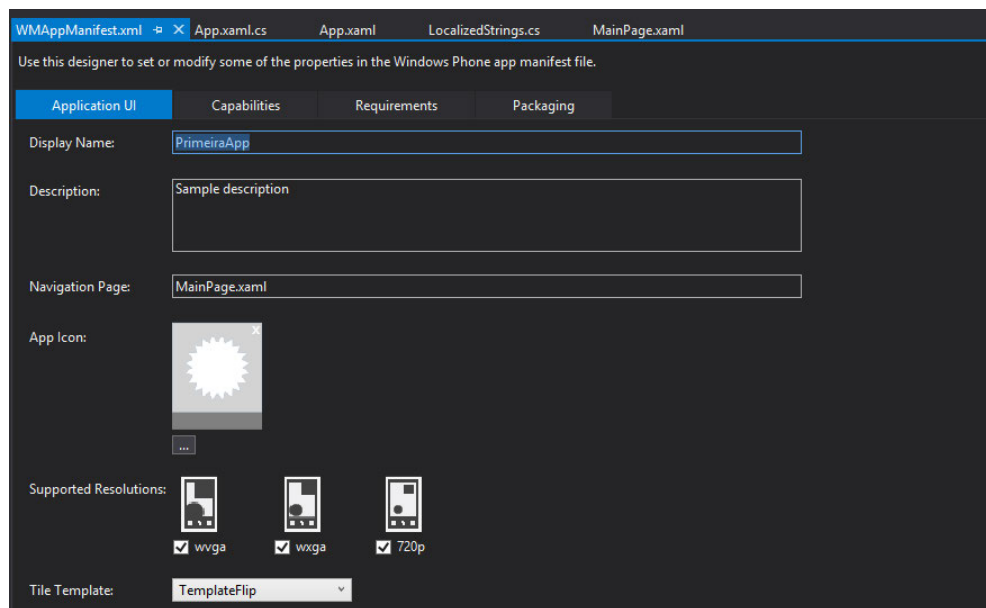


Figura 3.3: Arquivo Manifest

Agora que já conhecemos a estrutura de nosso projeto, vamos abrir novamente

o arquivo `MainPage.xaml` e alterar o título de nossa página de “page name” para “Hello World”, e alterar o título da aplicação para “PRIMEIRA APP”. Se quiser, também pode explorar um pouco a caixa de ferramentas e inserir alguns controles na página do aplicativo. É interessante incluir um campo de texto para vermos o comportamento do teclado. Agora vamos começar a conhecer o emulador, para isso precisamos executar nossa aplicação!

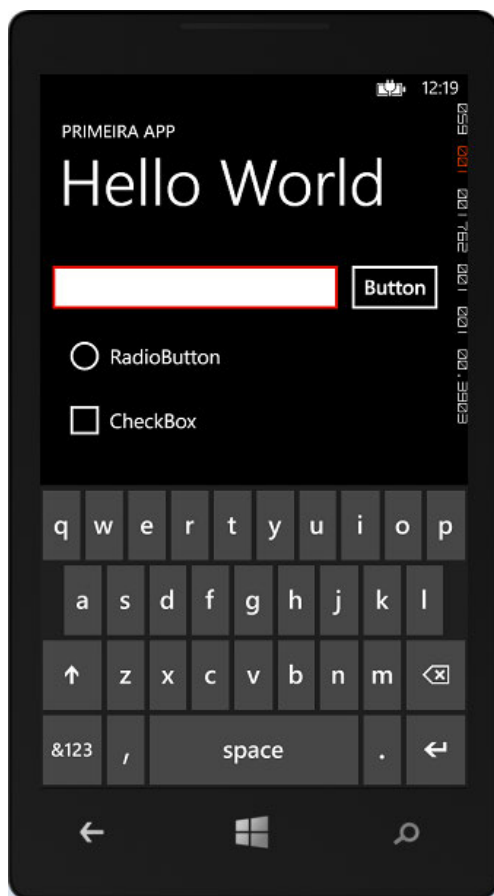


Figura 3.4: Primeira aplicação executando

Neste momento é interessante que você explore um pouco o emulador e dê uma passeada pelo sistema. Para voltar à tela inicial do smartphone emulado, pressione a tecla `Windows`. Você também pode utilizar o mouse para simular eventos de touch; ao clicar, manter o clique pressionado e mover o mouse para a esquerda, você verá a

lista de aplicativos instalados no emulador e irá encontrar, inclusive, a nossa aplicação.

DICA: UTILIZAR O TECLADO DO COMPUTADOR

Na figura 3.4, você pode notar que o teclado está visível e que para digitar você precisa pressionar as teclas correspondentes no emulador com o mouse. Mas há uma maneira mais fácil para inserir textos: você pode pressionar a tecla `Page Down` para recolher o teclado do emulador e utilizar o teclado do seu computador. Caso queira ativar o teclado do emulador novamente, utilize a tecla `Page Up`. Apesar de ser um recurso interessante, não é bom mantê-lo sempre ligado, pois você poderá esquecer de utilizar o teclado correto para determinado campo. Falaremos disso quando abordarmos o assunto *User Experience* 4.

Observe que ao lado do emulador do smartphone há uma série de botões que executam funções como girar o smartphone virtual ou aplicar zoom. O último botão abre uma nova janela chamada *Additional Tools*, na qual você pode testar o acelerômetro, GPS e bater screenshots de seu aplicativo.

3.2 CONFIGURANDO SEU DISPOSITIVO E SUA CONTA COMO DESENVOLVEDOR

Certo, agora nós já experimentamos de uma forma muito breve como é desenvolver para Windows Phone e testar o aplicativo através de um emulado. Entretanto, como eu já havia mencionado, pode ser importante testar seus aplicativos em um dispositivo real, principalmente operações que envolvem câmeras e sensores do smartphone. Mas para isso, você precisará de uma conta de desenvolvedor Windows Phone, o que lhe dará direito a fazer o deploy de sua aplicação em um dispositivo real e, claro, permitirá que você publique seu aplicativo na loja.

Primeiro é necessário ter uma conta Microsoft, como um e-mail do Outlook, por exemplo. Tendo isso, basta preencher o registro na página <http://migre.me/hCfN4> e pagar a quantia necessária. Após já ter uma conta de desenvolvedor, você pode procurar pelo aplicativo *Windows Phone Developer Registration*, que foi instalado junto com o SDK. Conectando seu dispositivo e informando os dados de sua conta

de desenvolvedor, é possível cadastrá-lo, de modo que você possa fazer os testes de seu aplicativo em seu próprio dispositivo.

Com a conta de desenvolvedor ativa, você também pode acessar um dashboard no portal de desenvolvedores do Windows Phone, no link <http://dev.windowsphone.com/> nela é possível publicar suas aplicações e verificar relatórios sobre suas vendas.

3.3 BLEND

Um outro componente do SDK que você também vai gostar de conhecer é o **Blend for Visual Studio**. Esta ferramenta deve ser utilizada para projetar e criar todo o design de sua aplicação. O Blend e o Visual Studio são totalmente integrados, ou seja, trabalham com os mesmos tipos de arquivos e manipulam os mesmos elementos de interface. Você pode, por exemplo, abrir a aplicação que acabou de criar no Blend e verá uma tela parecida com a da figura 3.5.

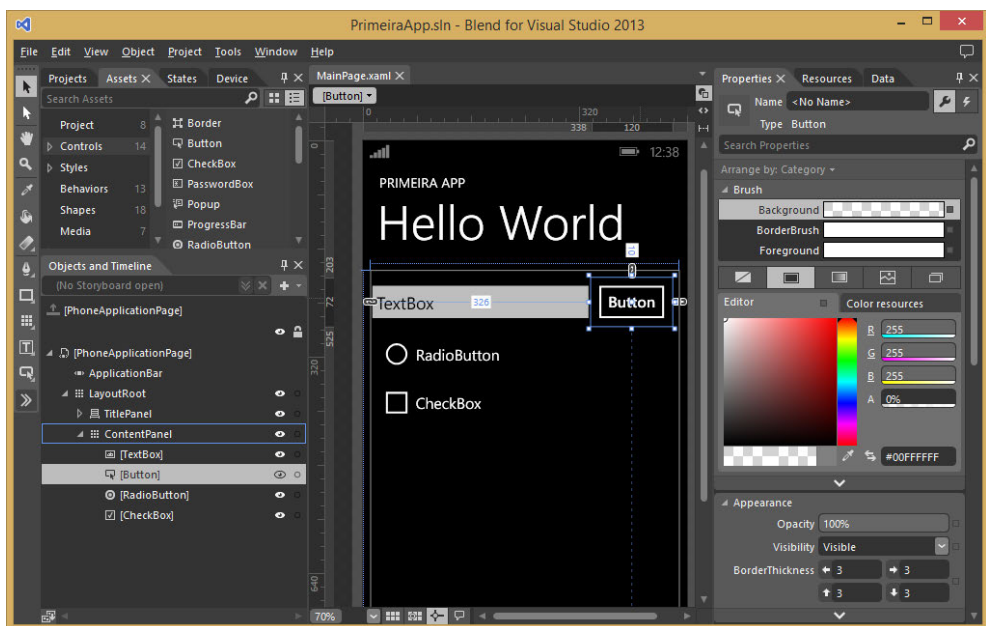


Figura 3.5: Blend para Windows Phone

Apesar deste visual de “software para designers”, não se engane: o Blend também é, sim, um componente importantíssimo para desenvolvedores. Assim como o Vi-

sual Studio, ele faz parte da construção de seu aplicativo, durante as mudanças na sua interface. Você pode testar e executá-lo no emulador, assim como faz no Visual Studio.

Apesar de ser possível criar animações através de código no Visual Studio, você pode criar um Story Board e trabalhar com animações de forma visual e mais amigável no Blend. As abas *Projects* e *Assets* são muito similares às abas existentes no Visual Studio, então você pode explorá-las por si só e perceber as pequenas diferenças entre elas. A aba *States* serve para criar estados diferentes em sua interface, sendo que cada estado possui um conjunto de valores para um determinado grupo de propriedades. Lá você também pode criar transições entre eles, o que é outra funcionalidade interessante para se testar agora. Por fim, na aba *Device*, pode-se alterar as configurações do dispositivo virtual exibido para testar sua interface em diferentes temas e configurações. Faça algumas alterações em sua página e teste-a novamente. Caso você volte para o Visual Studio, será notificado de que houve mudanças no projeto e que ele será recarregado.

Esta foi nossa primeira aplicação! Você pode encontrá-la em

<https://github.com/gabrielschade/LivroWindowsPhone/tree/master/Capitulo3/PrimeiraApp>

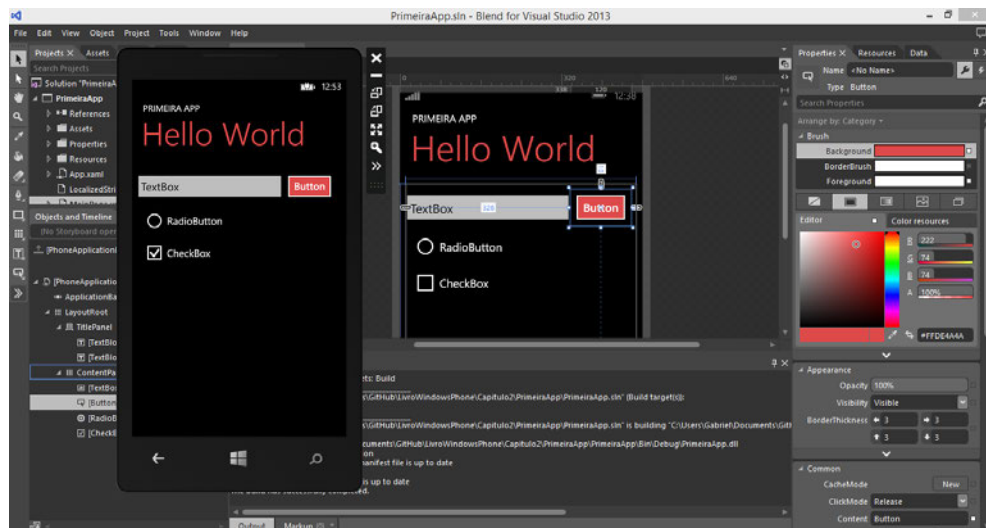


Figura 3.6: Primeira aplicação para Windows Phone

CAPÍTULO 4

Utilizando componentes do sistema para melhorar a experiência do usuário

Você já ouviu falar do conceito UX (*User eXperience*)? Este conceito, muito difundido na área de engenharia de usabilidade, trata-se de como sua aplicação interage com o usuário e de como ele se sente ao usar seu aplicativo. A experiência do usuário é algo importantíssimo para qualquer aplicação, mas no mercado de aplicativos móveis isso é ainda mais crítico, pois você concorre com outros milhares de aplicativos que são muito fáceis de serem instalados e testados. Você precisa garantir que a experiência do usuário com *seu* aplicativo seja ótima. Neste capítulo veremos alguns pontos que podem ajudá-lo nisso.

É importante você sempre ter em mente que os princípios de design e UX definidos para o Windows Phone não são regra, e sim um guia. Aplicativos como o Facebook, por exemplo, utilizam poucos componentes do sistema, mas trazem uma

experiência agradável. Então lembre-se, são guias e dicas, não padrões e regras.

4.1 PRINCÍPIOS DE DESIGN

Caso você possua um smartphone Windows Phone ou já tenha utilizado um, você deve ter notado que a tipografia e as páginas desenhadas para Windows Phone possuem um aspecto semelhante. Elas utilizam muito do conceito de *clean*, ou seja, a tela possui poucas linhas, grades, bordas etc., focando mais no conteúdo que deve ser mostrado. Para a separação destes conteúdos, normalmente é utilizado o espaçamento. A imagem *AlignmentGrid* na pasta **Assets** de nosso projeto possui este propósito. Vamos voltar na página que criamos no capítulo anterior — ao olhar o fim do nosso arquivo XAML, você verá o trecho de código comentado a seguir:

```
<phone:PhoneApplicationPage>
.....

    <Grid>
        .....
        <!--Uncomment to see an alignment grid to help ensure your
            controls are aligned on common boundaries.
            The image has a top margin of -32px to account for the
            System Tray.
            Set this to 0 (or remove the margin altogether) if the
            System Tray is hidden.

            Before shipping remove this XAML and the image itself.-->
        <!--<Image Source="/Assets/AlignmentGrid.png"
            VerticalAlignment="Top" Height="800" Width="480"
            Margin="0,-32,0,0" Grid.Row="0" Grid.RowSpan="2"
            IsHitTestVisible="False" />-->
    </Grid>

</phone:PhoneApplicationPage>
```

Caso você remova o comentário desta imagem, verá a imagem sobre sua página e poderá conferir o espaçamento entre os elementos, conforme a figura 4.1.

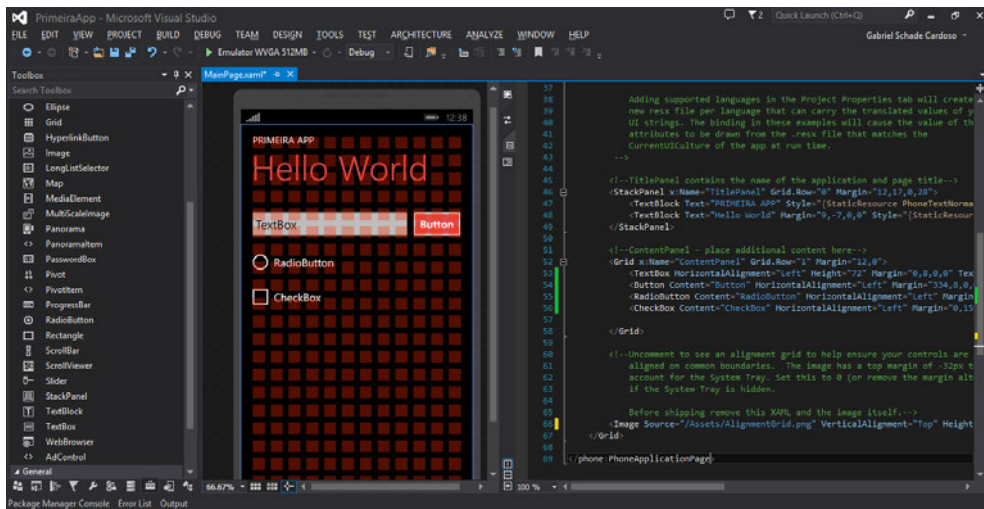


Figura 4.1: Imagem ilustrando o espaçamento

As animações e transições também são amplamente utilizadas nas aplicações e no próprio sistema operacional, o que você pode notar utilizando o emulador. As próprias *live tiles* das aplicações também estão constantemente se alterando para mostrar novas informações ao usuário. Isso faz com que o sistema tenha mais vida, criando uma interação mútua, do usuário com o sistema e do sistema com o usuário.

O último ponto e talvez o mais importante para a experiência do usuário nesta plataforma é: **tire proveito do sistema operacional**. Existem dois recursos conhecidos como **Launchers** e **Choosers** que veremos ao longo deste livro 8.4, que permitem que você interaja com as outras aplicações já instaladas no dispositivo de forma indireta através do sistema operacional. Por exemplo, se você criar um aplicativo de fotos, não precisará criar integrações com o Facebook ou Instagram, mas sim apenas se comunicar com o sistema operacional e ele saberá onde você pode publicar suas fotos, sendo uma maneira bastante simples e conveniente de integrar seus aplicativos.

Os princípios de design não terminam aqui. Vamos vê-los a partir da construção de um aplicativo ao longo deste livro, mas agora iremos começar o nosso novo aplicativo.

4.2 COMEÇANDO NOSSO APLICATIVO

Vamos começar a construir nosso aplicativo real. Ele será desenvolvido ao longo de todo o livro e com ele exemplificarei as principais funcionalidades para o desenvolvimento de aplicativos nesta plataforma. Criaremos um aplicativo de compras, no qual você poderá procurar por produtos, cadastrar um perfil de comprador, indicar os produtos a um amigo e uma série de funcionalidades que iremos ver ao longo do livro.

Um aplicativo móvel nesta linha normalmente vem acompanhado de uma infraestrutura para os web services, para, por exemplo, fazer a consulta dos produtos. Estas informações precisam estar em alguma base de dados — geralmente localizada na nuvem — para que possa ser acessada de qualquer local a partir do smartphone. Vamos criar uma estrutura local com poucos dados para que você possa visualizar como seria a aplicação real, mas deixaremos claro via comentário no código-fonte os pontos-chaves onde chamaríamos os serviços web, para que você compreenda como seria o cenário em uma aplicação real.

Para começar, criaremos uma página inicial, onde o usuário poderá decidir entre criar uma conta de comprador, entrar com uma conta já existente, ou ainda entrar na loja sem criar uma conta (neste último modo ele não poderá fazer compras sem efetuar o login). Como você deve ter percebido, esta tela será extremamente simples, precisando apenas de três botões e do título da nossa aplicação. Vamos criar um novo projeto e construir a página do zero.

Já vimos o processo de criação anteriormente. Você deverá repeti-lo selecionando novamente o template *Windows Phone App*, mas desta vez daremos o nome **CompreAqui** para o aplicativo. Ao fazer isso, teremos a mesma estrutura de projeto que conhecemos no capítulo anterior. Vamos editar o arquivo `MainPage.xaml` para que cumpra nossas necessidades, mas antes de sair alterando o layout como fizemos na outra aplicação, vamos conhecer os gerenciadores de layout.

4.3 GERENCIADORES DE LAYOUT

Os gerenciadores de layout são componentes de tela não visuais que nos auxiliam na organização dos itens na página. Eles são especialmente úteis quando falamos de dispositivos que podem conter tamanhos e resoluções de tela diferentes. Para que os controles não fiquem distorcidos ou em locais de difícil acesso para o usuário, nós os utilizamos.

Todos os gerenciadores de layout se encontram na mesma caixa de ferramenta

que você utiliza para adicionar um botão na página, por exemplo. Eles são tratados da mesma forma que os outros controles; além disso, cada gerenciador funciona de uma forma particular. Irei explicá-los para que você possa aproveitar o melhor de cada um deles.

O controle **Grid** é um gerenciador de layout bastante utilizado. Ao criar uma página padrão, já podemos ver que ela utiliza este componente. Ele serve basicamente para separar a sua página em um layout tabular, ou seja, em formato de tabela. Você pode definir linhas e colunas e cada controle irá preencher o conteúdo de sua respectiva célula, como a figura 4.2 ilustra.

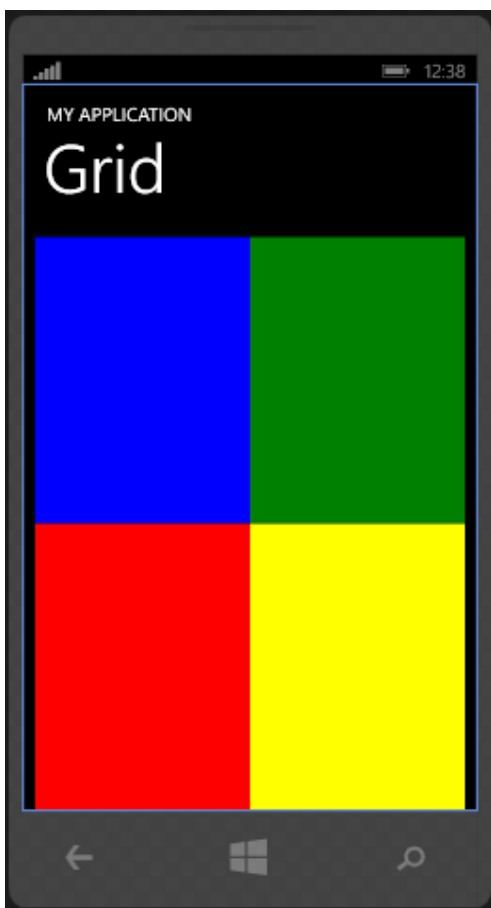


Figura 4.2: Gerenciador de layout Grid

Como você pode ver na figura anterior, o Grid está definido para ter duas linhas e duas colunas, e cada célula possui um retângulo de uma cor diferente. Isso tudo é definido no código XAML a seguir.

```
<Grid x:Name="ContentPanel">

    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Rectangle Fill="blue"></Rectangle>
    <Rectangle Grid.Column="1" Fill="green"></Rectangle>
    <Rectangle Grid.Row="1" Fill="Red"></Rectangle>
    <Rectangle Grid.Row="1" Grid.Column="1" Fill="yellow"></Rectangle>

</Grid>
```

Não há nenhum mistério no código para gerar um gerenciador deste tipo: basta criá-lo, definir suas configurações para linhas e colunas e inserir os componentes. Entretanto, algumas observações devem ser levadas em conta. Note que na definição de colunas eu não especifiquei a propriedade *Width*; desta forma, elas ocupam exatamente o mesmo espaço. O mesmo acontece com a altura das linhas definidas com o caractere * (asterisco), que é utilizado como coringa para informar que a altura da linha (ou largura da coluna) deverá ocupar todo o espaço restante. No caso de as duas utilizarem esse recurso, o espaço restante é compartilhado entre elas.

Um ponto importante que deve ser lembrado sempre é que estes gerenciadores de layout comumente são aninhados, ou seja, é muito comum que haja vários destes componentes dentro de outros gerenciadores ou até dentro de um componente do mesmo tipo. No exemplo a seguir veremos os comportamentos do *StackPanel* e o aninharemos a um componente *Grid*.

O *StackPanel*, como o próprio nome já sugere, é um painel para empilhar componentes. O comportamento deste componente está fortemente vinculado à sua propriedade *Orientation*, que define se ele vai empilhar os componentes um acima do

outro ou um ao lado do outro.

Neste gerenciador de layout, diferente do *Grid*, é obrigatório o preenchimento das propriedades de dimensões dos elementos. No exemplo anterior, não precisamos definir nem a altura e nem a largura dos retângulos no *Grid* — eles ocuparam a célula toda automaticamente. Mas como não há o conceito de célula em um *StackPanel*, é necessário informarmos a propriedade referente à orientação do painel, ou seja, caso o painel esteja na vertical, precisamos preencher o valor de altura, e caso esteja na horizontal, precisamos informar ao menos o valor da largura.

A figura 4.3 ilustra dois painéis deste tipo, cada um com a orientação diferente, e ambos estão separados em linhas de um *Grid* que os circunda.



Figura 4.3: Gerenciador de layout StackPanel

Apesar de não ter sido exemplificado, quando a orientação do painel está definida como *Horizontal*, nós podemos escolher se os componentes são empilhados da esquerda para a direita ou da direita para a esquerda, através da propriedade *Flow-Direction*. Veja o código para criar a página ilustrada:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>

  <StackPanel Grid.Row="1">

    <Rectangle Fill="blue" Width="75" Height="75"></Rectangle>
    <Rectangle Fill="green" Width="75" Height="75"></Rectangle>
    <Rectangle Fill="Red" Width="75" Height="75"></Rectangle>
    <Rectangle Fill="yellow" Width="75" Height="75"></Rectangle>

  </StackPanel>

  <StackPanel Grid.Row="2" Orientation="Horizontal">

    <Rectangle Fill="blue" Width="75" Height="75"></Rectangle>
    <Rectangle Fill="green" Width="75" Height="75"></Rectangle>
    <Rectangle Fill="Red" Width="75" Height="75"></Rectangle>
    <Rectangle Fill="yellow" Width="75" Height="75"></Rectangle>

  </StackPanel>

</Grid>
```

O terceiro componente que iremos mostrar aqui é o *Canvas*. Ele possui layout livre, ou seja, a posição dos elementos é absoluta e é definida pelas propriedades *Canvas.Top* e *Canvas.Left*, que indicam o deslocamento do elemento da borda superior e da borda esquerda, respectivamente.

Este gerenciador geralmente é utilizado para painéis onde há movimentação dos itens por meio de um gesto, ou para fazer desenhos. Deve-se ter cuidado ao utilizar este tipo de componente, pois ele pode se comportar de maneira inesperada, devido à posição absoluta dos elementos nele contidos. A figura 4.4 mostra os retângulos em um painel do tipo *Canvas*.

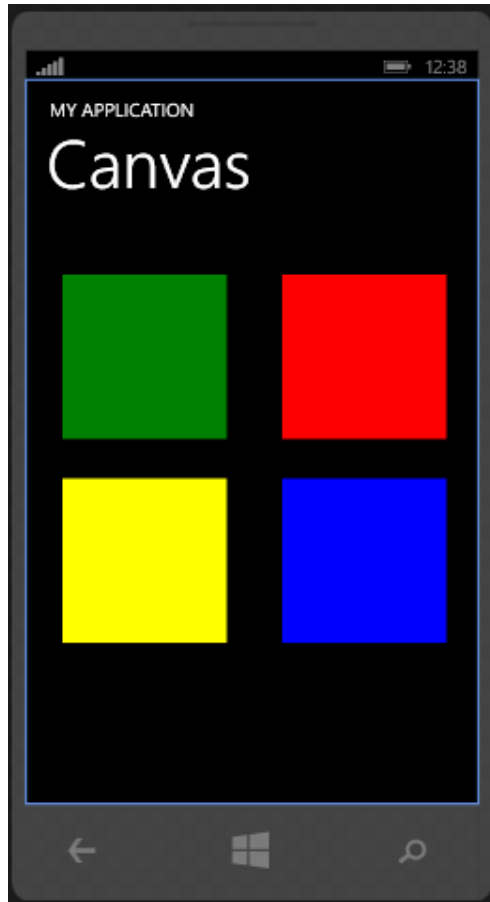


Figura 4.4: Gerenciador de layout Canvas

O Canvas é bastante livre e, como podemos ver, o código para inserir elementos nele é bem simples: basta colocar o elemento aninhado e definir as propriedades supracitadas. Também é possível mover os componentes livremente com o mouse através da interface do Visual Studio. Para gerar a figura 4.4, foi utilizado o código a seguir.

```
<Canvas Grid.Row="1">
```

```
  <Rectangle Fill="blue" Width="175" Height="175"  
    Canvas.Left="272" Canvas.Top="262"></Rectangle>
```

```
<Rectangle Fill="green" Width="175" Height="175"  
  Canvas.Left="38" Canvas.Top="45"></Rectangle>  
  
<Rectangle Fill="Red" Width="175" Height="175"  
  Canvas.Left="272" Canvas.Top="45"></Rectangle>  
  
<Rectangle Fill="yellow" Width="175" Height="175"  
  Canvas.Left="38" Canvas.Top="262"></Rectangle>  
  
</Canvas>
```

4.4 CRIANDO A PÁGINA DE BOAS-VINDAS

Com isso, nós já conhecemos os principais gerenciadores de layout que iremos utilizar para a criação das páginas de nosso aplicativo. Como citamos anteriormente, a primeira página será a página de título do aplicativo. Nela, o usuário poderá escolher entre fazer login, criar uma conta de usuário ou entrar na loja sem criar uma conta.

Não será exigido que você crie a página exatamente igual à que iremos mostrar como exemplo, mas lembre-se de utilizar os gerenciadores para que seu layout comporte-se de maneira correta. Futuramente veremos que eles são ainda mais úteis quando tratamos a orientação do dispositivo. Ao terminar sua página, ela deve se parecer com a figura a seguir.

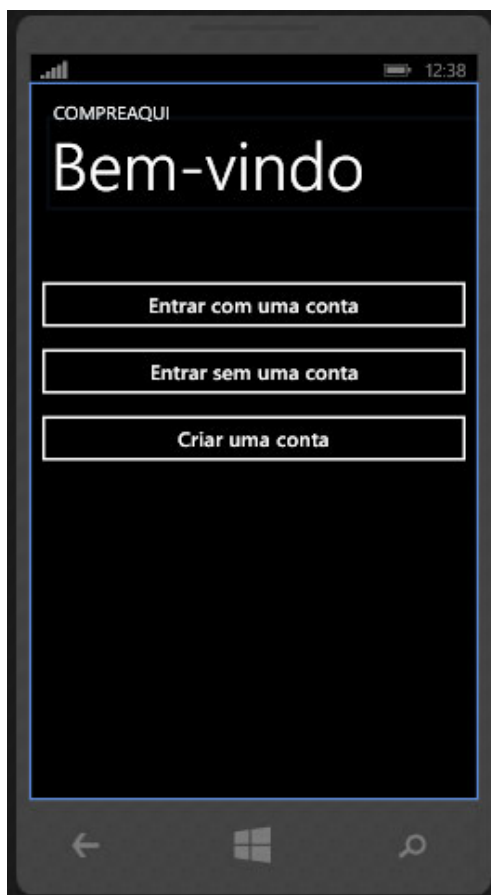


Figura 4.5: Página de boas-vindas

```
<phone:PhoneApplicationPage>
  <!--Declaração de Namespaces-->
  <!--Declaração do Grid e StackPanel padrão da página-->

  <StackPanel Grid.Row="1" Margin="0,40,0,0">
    <Button>Entrar com uma conta</Button>
    <Button>Entrar sem uma conta</Button>
    <Button>Criar uma conta</Button>
  </StackPanel>
</phone:PhoneApplicationPage>
```

O projeto pode ser encontrado no meu github pelo link: <http://bit.ly/>

GITcap4Partel

Observe a figura 4.5, ela é a nossa primeira página do aplicativo. Como você pode ver, ela segue corretamente o template de páginas disponíveis e utiliza os gerenciadores de layout, conforme o código citado anteriormente. Entretanto, esta página não é atrativa para o usuário, ela está sem vida, extremamente simplória e sem nenhum atrativo.

Podemos melhorar isso através de imagens, cores e, claro, ideias. Novamente peço para que tente pensar em algo antes de seguir com o livro — tente pensar em como esta tela pode se tornar mais atraente para o usuário e tente implementar o que você pensou utilizando o Visual Studio e o Blend. Apesar de este livro ser um guia inicial, é bastante importante você confiar em suas ideias e experimentar por conta própria.

A primeira coisa que faremos é utilizar uma imagem de fundo e criar um ícone para nossa aplicação. Também vamos alterar um pouco o formato do layout da página, remover o cabeçalho com o nome da aplicação e os dizeres “Bem-vindo”. Deixemos que o *Grid* principal nomeado por padrão de *LayoutRoot* continue tendo duas linhas, mas vamos fixar o tamanho da linha inferior para 270.

O painel inferior com os botões de interação deverão ser jogados para a segunda linha do *Grid*, o que faz com que eles fiquem na parte inferior da página, sobrando um espaço em branco considerável. Vamos preencher esse espaço com o plano de fundo e com o título da aplicação “CompreAqui”. Veja na figura 4.6 como nossa página já melhorou bastante.



Figura 4.6: Página de boas vindas 2.0

Link do projeto no meu github: <http://bit.ly/GITcap4ParteII>

No link do projeto supracitado, você poderá encontrar as imagens que foram utilizadas para o plano de fundo e como ícone. Como já vimos anteriormente, elas estarão na pasta *Assets* do nosso projeto. Até agora fizemos alterações baseadas em layout e inserimos algumas imagens, o que já muda bastante a aparência desta janela.

Ainda assim, podemos ver que os botões de ação permanecem iguais e também podemos dar-lhes uma aparência diferente do padrão. Para isso, podemos utilizar dois conceitos diferentes: **estilo** e **template**.

Se você já trabalhou com Web, você deve estar familiarizado com este conceito. Ele também está presente no XAML através da tag `Style`. Com novos estilos, po-

demos criar aparências diferenciadas para componentes já existentes, e esses estilos podem ser utilizados em diferentes níveis. Podemos criar um estilo para um único componente (quando alteramos suas propriedades, como por exemplo, *Background*), podemos criar um estilo e reutilizá-lo em uma página toda ou podemos criar um estilo que poderemos acessar em todas as páginas da aplicação.

Template de componentes também é um estilo — é um conceito parecido, porém mais poderoso. O estilo vai se limitar a definir as propriedades dos componentes do sistema. Por exemplo, você pode alterar a cor de fundo do botão, mas com templates você poderá alterar a forma do botão, ou até mesmo inserir novos componentes nele. A definição de um template para um componente também é feita através da tag `Style`.

Agora vamos utilizar um template de componente para alterarmos os botões de nossa página. Para fazer isso, utilize a janela *Document Outline* do Visual Studio (o atalho para abri-la é `Ctrl + Alt + T`). Esta janela é muito útil para visualizarmos a organização dos controles e gerenciadores de layout em nossa página. Para facilitar a criação de um template, procure por um dos botões de nossa página e pressione o botão direito do mouse. No menu flutuante, selecione a opção `Edit Template > Edit a Copy...` conforme figura 4.7.

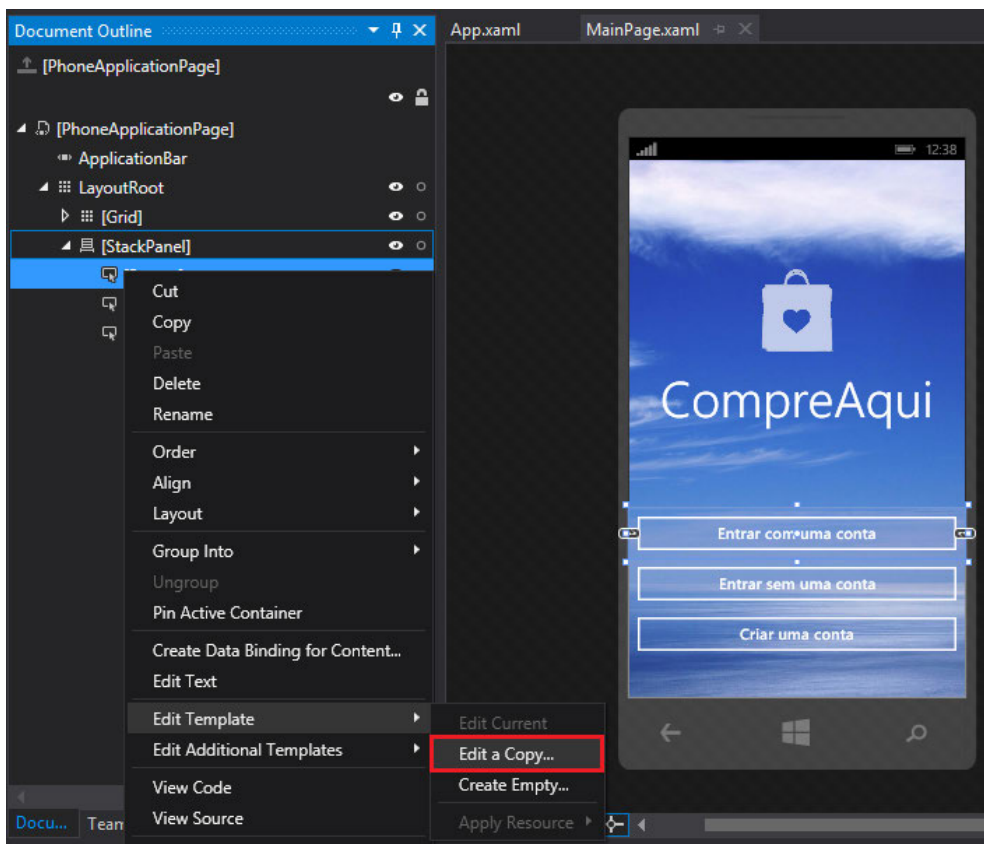


Figura 4.7: Editar uma cópia do template de um componente

DOCUMENT OUTLINE

Esta janela é muito útil para organizarmos os controles visuais e os gerenciadores de layout de nossa página. Além disso, através dela, utilizando o botão direito do mouse podemos editar templates e estilos. Pode ser bastante útil caso você não conheça os estilos que fazem parte do sistema operacional — você pode, por exemplo, selecionar um controle `TextBlock` e escolher a cor, tamanho ou nome da fonte através deste atalho.

Esta opção é bastante útil pois você poderá fazer as edições no componente a par-

tir do componente padrão, garantindo que só perderá o comportamento dos componentes que remover. Ao selecioná-la, será exibida uma janela para que sejam informados alguns dados a respeito do template que pretende criar. Insira o nome do template que preferir e selecione a opção `Application` no grupo `Define in`, isso irá garantir que o template possa ser usado por toda a aplicação.

Você será redirecionado para o arquivo `App.xaml`, já que é lá onde ficam todos os recursos que podem ser utilizados em toda a aplicação. Você verá que há um estilo definido com a propriedade `TargetType`, com o valor `Button`, e com a propriedade `x:Key`, com o valor que você informou na janela anterior. Esta é a definição do template que estamos editando. Vamos fazer algumas modificações para que o botão fique da maneira como pretendemos.

Antes de fazermos as alterações, vamos conhecer um pouco mais sobre os estilos. Segue o código-fonte resumido do estilo que define o template padrão dos botões:

```
<Style x:Key="LargeButton" TargetType="Button">

    <Setter Property="Background" Value="Transparent"/>
    <Setter Property="BorderBrush"
        Value="{StaticResource PhoneForegroundBrush}"/>

    <Setter Property="Template">

        <Setter.Value>
            <ControlTemplate TargetType="Button">

                <Grid Background="Transparent">

                    <VisualStateManager.VisualStateGroups>
                        <VisualStateGroup x:Name="CommonStates">

                            <VisualState x:Name="Normal"/>

                            <VisualState x:Name="Pressed">
                                <Storyboard>
                                    <ObjectAnimationUsingKeyFrames
                                        Storyboard.TargetProperty="Foreground"
                                        Storyboard.TargetName="ContentContainer">

                                        <DiscreteObjectKeyFrame
                                            KeyTime="0"
```

```
        Value="{StaticResource
        PhoneButtonBasePressedForegroundBrush}"/>

    </ObjectAnimationUsingKeyFrames>

    </Storyboard>
</VisualState>

<Border x:Name="ButtonBackground"
        BorderBrush="{TemplateBinding BorderBrush}">

    <ContentControl x:Name="ContentContainer"
        ContentTemplate="{TemplateBinding ContentTemplate}"
        Content="{TemplateBinding Content}"
        Foreground="{TemplateBinding Foreground}" />

</Border>
</Grid>

</ControlTemplate>
</Setter.Value>
</Setter>

</Style>
```

Agora vamos destrinchar este código para compreendermos o significado de cada coisa. As primeiras tags que aparecem são os *Setters*, que simplesmente aplicam o valor a alguma propriedade. O exemplo anterior indica que a propriedade *Background* do botão inicia com o valor *Transparent*. Alguns *Setters* utilizam uma notação um pouco diferente (“StaticResource Phone~”), que é utilizada para obter as cores, tamanhos e padrões do próprio sistema operacional. Note que também podemos estender a tag *Setter*, para o caso de editarmos o template do botão. Dentro desta tag, nós definimos os componentes visuais que formam o botão, como por exemplo, o *Grid* principal e suas bordas (*Border*).

Você também notará que há um grupo de tags chamado *VisualState*, sobre os quais falaremos melhor nos capítulos futuros. Por enquanto, basta sabermos que ele trata de um estado do componente. Isso quer dizer que, cada *VisualState* pode conter diferentes valores para as mesmas propriedades, e durante o fluxo de nossa aplicação podemos alterar entre estes estados, que a página irá se reorganizar completamente

de acordo com as novas propriedades.

Também há a tag *Storyboard*, que se refere a animações de elementos. No caso do exemplo anterior, a cor da fonte do botão é alterada para uma determinada fonte quando ele é pressionado. Também falaremos melhor disso futuramente.

Após esta explicação sobre os estilos, você deve estar apto a fazer pequenas mudanças na aparência do botão. Em nosso exemplo, vamos remover o *Setter* da propriedade *Padding*, pois queremos que o botão ocupe todo o espaço disponível. Também criaremos um *Setter* para a propriedade *Height* e definiremos seu valor padrão como 90. Além disso, vamos alterar o valor dos *Setters* *FontFamily* e *FontSize* para *PhoneFontFamilyLight* e *PhoneFontSizeLarge*, respectivamente.

Removeremos os estados (VisualState) *Pressed* e *Disabled*, pois não vamos criar uma situação onde este botão possa ser desabilitado e ele não deverá alterar sua cor para os padrões do sistema ao ser pressionado. Por fim, removeremos todas as propriedades do componente *Border*, exceto pela propriedade *Background*.

Voltando ao nosso formulário e aplicando o template para todos os botões, podemos ver mudanças em sua aparência, mas eles estão todos com o fundo transparente, dando uma sensação de vazio. Para finalizar esta etapa, vamos preencher a propriedade *Background* dos botões com os respectivos valores: `#B2C80000`, `#B20000C8` e `#BF0E6400`. Estes valores são tons transparentes de vermelho, azul e verde, dando um pouco mais de vida à nossa tela inicial, que neste ponto deve estar parecida com a figura 4.8.



Figura 4.8: Página de boas vindas versão final

```
<StackPanel Grid.Row="3">
```

```
<Button Style="{StaticResource LargeButton}"  
  Background="#B2C80000" >Entrar com uma conta</Button>
```

```
<Button Style="{StaticResource LargeButton}"  
  Background="#B20000C8" >Entrar sem uma conta</Button>
```

```
<Button Style="{StaticResource LargeButton}"  
  Background="#BF0E6400" >Criar uma conta</Button>
```

```
</StackPanel>
```

Link do projeto no meu github: <http://bit.ly/GITcap4ParteIII>

O principal ponto deste capítulo é que você consiga dominar os conceitos básicos sobre templates e estilos, além de entender que os protótipos que o Visual Studio entrega feito não são regras, ele são apenas guias. Pode ser interessante para seu aplicativo utilizar as cores do sistema operacional e o tema do usuário — isso pode sim oferecer uma ótima experiência e com uma boa imersão —, mas neste capítulo quis mostrar que esta não é a única forma correta.

Existem diversos tutoriais pela internet falando sobre como seguir os padrões e guias de interface do Windows Phone, mas podemos ver claramente que aplicativos como o Facebook, Twitter e tantos outros possuem sua própria identidade. **É mais importante ser um aplicativo que oferece uma boa experiência do que seguir todos os padrões já propostos.**

A figura 4.9 ilustra como as páginas para criar conta e entrar com uma conta devem se parecer. Tente fazê-las sozinhos e no próximo capítulo passaremos por alguns pontos explicando melhor como podemos criar as páginas deste tipo.



Figura 4.9: Páginas para criar uma conta e entrar no aplicativo

CAPÍTULO 5

Design de aplicações

Já falamos sobre experiência de usuário e design de aplicações, mas vamos continuar seguindo esta trilha para que nossos aplicativos transmitam uma boa experiência para o usuário. No capítulo anterior foram mostradas duas páginas já criadas, agora vamos avaliar a aplicação sendo executada e após isso faremos melhorias no código-fonte.

Neste ponto, peço que execute a aplicação que se encontra em meu github através do link: <http://bit.ly/GITcap5ParteI> e faça alguns testes com ela, como por exemplo, navegar nas páginas, preencher os formulários etc.

5.1 MELHORIAS DE USABILIDADE

Depois de fazer estes testes, apontarei alguns problemas de usabilidade que talvez você também tenha percebido. **Orientação:** nestas novas páginas, a orientação do dispositivo influencia a organização dos elementos na página. A página `Entrar.xaml` não possui nenhum problema, entretanto a `CriarConta.xaml`

possui um problema seriíssimo. Quando o dispositivo se encontra no modo paisagem, ou seja, está na horizontal, alguns campos estão se tornando inalcançáveis ao usuário.

Felizmente, há um gerenciador de layout do qual ainda não falamos, chamado *ScrollView*, que circunda elementos ou outros gerenciadores e ativa barras de rolagem quando necessário. Ao circundar o painel que contém os elementos nesta página, nosso problema estará resolvido.

```
<!-- Início da página -->

<ScrollView>

    <StackPanel Margin="10,0">

        <TextBlock Style="{StaticResource PhoneTextLargeStyle}">
            Email
        </TextBlock>

        <TextBox></TextBox>

        <!--Outros componentes-->

    </StackPanel>
</ScrollView>

<!-- Continuação da página -->
```

Nesta mesma página, também há um outro pequeno problema: ao selecionar o campo e-mail, é exibido o teclado convencional, mas existem outros tipos de teclado. O contexto da situação poderá definir qual tipo de informação que o teclado deverá inserir, por exemplo, para um campo somente numérico não faz sentido que o teclado apresente letras.

Para definir o tipo de teclado que será exibido, utilizaremos a propriedade *InputScope*. Quando selecionamos o valor desta propriedade como `EmailUserName`, ao ser exibido, o teclado exibe uma tecla para `@` e outra para `.com`, conforme ilustra figura 5.1.



Figura 5.1: Escopos de teclado

No capítulo anterior, vimos o conceito de estilo e template de componentes. Várias propriedades definidas nos templates dos componentes padrões do Windows Phone possuem sua coloração vinculada ao próprio smartphone. Em nossa aplicação não é diferente: entrando nas configurações do dispositivo e alterando as cores do tema poderemos ver mudanças em nossa interface.

Caso você prefira que os componentes se mantenham iguais mesmo com a alteração do tema, é necessário que você crie estilos/templates que não estejam vinculados ao sistema operacional. Como já vimos a forma de criar um template para um componente e definir padrões para as propriedades dele, não iremos entrar em detalhes agora.

Outro ponto que pode ser notado, apesar de não ser exatamente um erro, é a

falta de transições e animações. O sistema operacional Windows Phone é bastante vivo, ou seja, praticamente tudo nele possui uma animação, o que pode melhorar a experiência. Para isso, iremos aprender como criar transições para quando o usuário alterar a orientação do dispositivo ou navegar entre as páginas.

Primeiro, vamos inserir a animação para quando navegarmos entre as páginas. Para isso, precisamos incluir um pacote de ferramentas que facilitem este trabalho. Este pacote é disponibilizado pela própria Microsoft e pode ser incorporado ao projeto diretamente pelo Visual Studio. Vamos utilizar o **NuGet**.

NUGET

O NuGet é um gerenciador de pacotes incorporado ao Visual Studio, que é capaz de buscar bibliotecas através da internet em um repositório, incorporando as referências mais atualizadas da biblioteca ao seu projeto no Visual Studio.

Para adicionar este pacote de ferramentas, pressione o botão direito do mouse sobre o elemento **References** de nosso projeto e selecione a opção **Manage NuGet Packages . . .** Pesquise online por **WPToolkit** e obterá como resultado da busca a biblioteca **Windows Phone Toolkit**. Após isso, você deve instalá-la em seu projeto. A figura 5.2 ilustra os passos que você deve seguir.

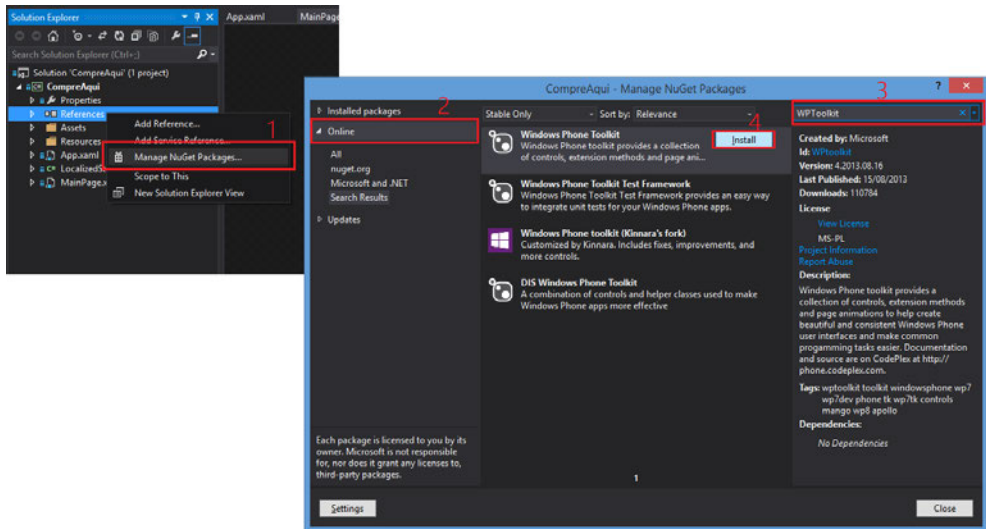


Figura 5.2: Adicionando o pacote de ferramentas

Agora que já temos o pacote de ferramentas incluso em nosso projeto, vamos utilizá-lo! Abra a página `CriarConta.xaml`. A primeira coisa a se fazer é incluir o namespace do pacote de ferramentas em nossa página. Você já deve ter notado que as linhas iniciais da página são utilizadas para a declaração dos namespaces que são utilizados nela. Após a declaração do namespace `shell`, vamos adicionar nosso namespace e chamá-lo de `toolkit`, conforme o código a seguir.

```
<phone:PhoneApplicationPage
    x:Class="CompreAqui.Paginas.CriarConta"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;
        assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;
        assembly=Microsoft.Phone"

    xmlns:toolkit="clr-namespace:Microsoft.Phone.Controls;
        assembly=Microsoft.Phone.Controls.Toolkit"

    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

```
xmlns:mc=
"http://schemas.openxmlformats.org/markup-compatibility/2006"

FontFamily="{StaticResource PhoneFontFamilyNormal}"
FontSize="{StaticResource PhoneFontSizeNormal}"
Foreground="{StaticResource PhoneForegroundBrush}"
SupportedOrientations="PortraitOrLandscape" Orientation="Portrait"
mc:Ignorable="d"
shell:SystemTray.IsVisible="True">
```

Esse código mostra a declaração da tag que define a página `CriarConta`. Com o namespace toolkit já inserido nela poderemos utilizá-lo nesta página à vontade.

Este pacote de ferramentas conta com um namespace chamado `TransitionService`. Trata-se de um conjunto de funcionalidades para criar transições entre as páginas, tanto para quando você entra nela, quanto para quando você sai. Além disso, você pode definir transições diferentes para quando o usuário entra na página através de um link ou através do botão voltar do dispositivo.

Esta página terá transição apenas na entrada do usuário, tanto através de um link quanto através do botão voltar. Para fazer isso, você deve utilizar os objetos do tipo *Transition*; são eles: *RollTransition*, *RotateTransition*, *SlideTransition*, *SwivelTransition* e *TurnstileTransition*. Eles possuem a propriedade *Mode* que define como será feita a animação. Você pode fazer testes para verificar como cada transição se comporta, mas utilizaremos a transição *SlideTransition*. Basta inserir este trecho de código logo abaixo da declaração da tag da página, mostrada no código anterior.

```
<toolkit:TransitionService.NavigationInTransition>
  <toolkit:NavigationInTransition>

    <toolkit:NavigationInTransition.Backward>
      <toolkit:SlideTransition Mode="SlideUpFadeIn" />
    </toolkit:NavigationInTransition.Backward>

    <toolkit:NavigationInTransition.Forward>
      <toolkit:SlideTransition Mode="SlideUpFadeIn" />
    </toolkit:NavigationInTransition.Forward>

  </toolkit:NavigationInTransition>
</toolkit:TransitionService.NavigationInTransition>
```

Antes de essa alteração ter efeito, teremos que fazer uma pequena mo-

dificação na classe `App.xaml.cs`. Nesta classe você encontrará a declaração do objeto `RootFrame` da seguinte maneira: `RootFrame = new PhoneApplicationFrame();`. Você terá de alterar o tipo do objeto para `RootFrame = new TransitionFrame();`.

Com isso já poderemos ver a animação funcionando ao entrar nesta página, porém a saída permanece “sem graça”. Podemos resolver isso colocando uma animação de entrada na página principal, porém com o efeito na direção contrária a este: na página principal, usaremos o modo `SlideDownFadeIn`. Você também deve inserir o mesmo trecho de código da página `CriarConta.xaml` na página `Entrar.xaml` para termos a mesma animação em todas as páginas.

Agora que resolvemos o problema de transição entre as páginas, nossa aplicação parece mais viva e interessante aos olhos do usuário, mas ainda falta a transição para quando a orientação do dispositivo é alterada. Infelizmente, o pacote de ferramentas que baixamos ainda não possui esta implementação, entretanto há uma implementação de um desenvolvedor da Microsoft para fazer isso. Você pode encontrá-la no link: <http://bit.ly/transitionMS>.

Adicione esta classe ao seu projeto. Você poderá ver que ela é uma sub-classe de uma classe contida no pacote de ferramentas e que acabamos de utilizar (`TransitionFrame`). Com isso, deve ficar claro que você terá de alterar novamente a declaração do objeto `RootFrame`, desta vez para a nova classe que você acabou de inserir no projeto, com uma pequena diferença: este construtor pede um parâmetro para definir a duração desta animação. Neste exemplo, utilizarei 400 milissegundos, conforme o código: `RootFrame = new HybridOrientationChangesFrame(TimeSpan.FromMilliseconds(400));`.

Já podemos ver todas estas melhorias em nosso aplicativo. Para acessar o código-fonte do projeto completo com estas implementações acesse: <http://bit.ly/GITcap5ParteII>.

Vamos seguir adiante, mas fique à vontade para fazer ainda mais melhorias no aplicativo, como por exemplo, fazer com que seja possível acessar a página de criação de conta através da página para entrar com uma conta existente (isso bem comum, pois cria um atalho para o usuário que ainda não possui conta e entrou nesta página).

5.2 CONHECENDO OS CONTROLES PANORAMA, APPLICATIONBAR E PIVOT

Até este ponto já construímos nossas páginas de boas-vindas, criação de conta e para entrar no aplicativo utilizando uma conta. Agora iremos construir a página principal de nosso aplicativo, onde o usuário poderá ver os produtos, as promoções e tudo mais que faz parte do conceito principal da aplicação.

Para a tela principal, utilizaremos um *Panorama*. Este é um controle bastante característico do Windows Phone que já engloba boa parte dos conceitos de design do sistema operacional.

O conceito do *Panorama* é bastante simples de compreender: basta imaginar que a página da aplicação é mais larga que o próprio smartphone. Apesar de você ter de navegar nela na horizontal, trata-se do mesmo conteúdo. Este conceito é utilizado em diversos aplicativos nativos, como por exemplo, o hub de pessoas ou de fotos, que está ilustrado na figura 5.3.



Figura 5.3: Hub de fotos do Windows Phone

Toda a implementação de gestos para navegar através do *Panorama* é implemen-

tada nativamente no controle, então é bastante simples de usá-lo e você verá que o efeito visual dele é muito bacana. Vamos agora criar uma nova página em nosso aplicativo e chamá-la de `ProdutosHub`. No momento da criação, você pode utilizar o template *Windows Phone Portrait Page* normalmente, mas para facilitar o trabalho, você também pode selecionar a opção *Windows Phone Panorama Page*, que já com o controle *Panorama* incluso na página.

De qualquer forma, caso você crie uma página comum, basta adicionar o controle *Panorama* que se encontra no namespace `phone` e adicionar os itens de *panorama*. Cada item corresponde a uma “subpágina” dentro do *Panorama*. Criaremos os seguintes: “classificações”, “promoção” e “produtos”, conforme o código a seguir.

```
<phone:Panorama Title="CompreAqui">

    <phone:PanoramaItem Header="classificações">

    </phone:PanoramaItem>

    <phone:PanoramaItem Header="promoção">

    </phone:PanoramaItem>

    <phone:PanoramaItem Header="produtos">

    </phone:PanoramaItem>

</phone:Panorama>
```

Por padrão, o fundo do *Panorama* é vazio, assumindo a cor de fundo do seu tema (preto ou branco), mas podemos preenchê-lo com uma imagem para melhorar sua aparência. Por enquanto, deixaremos os itens sem nenhum conteúdo, e o resultado deve ser semelhante à figura 5.4.



Figura 5.4: Panorama

Agora vamos conhecer o controle `ApplicationBar`, que, como o nome sugere, é uma barra de comandos para sua aplicação. Esta barra de comandos fica na parte inferior da página no modo retrato, e ou no canto direito no modo paisagem.

Na `ApplicationBar` devemos inserir ações que, apesar de fazerem parte do contexto, não se encaixam em algum ponto na página. Em nosso exemplo, vamos colocar em nosso componente um botão para acessarmos a página de pesquisa e um item de menu para acessarmos a página para gerenciamento de conta. Apesar de este componente fazer parte do namespace `shell`, precisamos adicioná-lo na propriedade `ApplicationBar` da página. Para fazer isso, utilizamos o trecho de código a seguir.

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar>

    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

Dentro da tag `<shell:AppBar>`, podemos inserir nossos componentes. Como já dito antes, será feito um botão para pesquisa e um item de menu para acessar as configurações de sua conta. Para utilizar o botão, você deve usar o componente `AppBarIconButton`, um botão circular que foi projetado para ser utilizado neste componente.

Ao adicioná-lo na barra de comandos, você verá que é obrigatório que ele possua um ícone. Felizmente, o SDK do Windows Phone já possui alguns ícones e entre eles está o ícone para pesquisas. Ele se encontra no grupo *features icons*. Neste ponto, sua `AppBar` deve estar similar ao código que segue.

```
<phone:PhoneApplicationPage.AppBar>
  <shell:AppBar>

    <shell:AppBarIconButton Text="pesquisar"
      IconUri="/Assets/AppBar/feature.search.png"/>

  </shell:AppBar>
</phone:PhoneApplicationPage.AppBar>
```

Para finalizar, vamos incluir o componente `AppBarMenuItem`, porém, diferente do botão, ele não pode ser inserido diretamente na barra, da mesma forma que a própria barra não pode ser incluída diretamente na página. Então iremos adicionar este item de menu na propriedade `MenuItems` do componente `AppBar`.

Por padrão, a barra possui uma cor de fundo escura. Vamos mantê-la, mas diminuiremos sua opacidade em 50%, dando um efeito transparente. Agora nossa barra de ações está finalizada, conforme o código e a figura 5.5.

```
<phone:PhoneApplicationPage.AppBar>
  <shell:AppBar Opacity="0.5" >

    <shell:AppBarIconButton Text="pesquisar"
      IconUri="/Assets/AppBar/feature.search.png"/>

    <shell:AppBar.MenuItems>
      <shell:AppBarMenuItem Text="sua conta"/>
    </shell:AppBar.MenuItems>
  </shell:AppBar>
</phone:PhoneApplicationPage.AppBar>
```

```
</shell:ApplicationBar>  
</phone:PhoneApplicationPage.ApplicationBar>
```



Figura 5.5: ApplicationBar

O último componente do sistema operacional que iremos explicar é o `Pivot`, que é um agrupador de páginas com um funcionamento parecido com o *Panorama*, porém com propósitos diferentes. Enquanto o *Panorama* serve como um hub, o `Pivot` serve como um separador de conteúdos, fazendo com que cada item seu possua itens relacionados em um contexto mais amplo, mas diferentes entre si.

Vamos criar agora a página para gerenciamento de conta. Ela contará com um componente do tipo `Pivot` para organizar seus conteúdos, então se preferir, tam-

bém pode selecionar o tipo *Windows Phone Pivot Page* no momento de criação da página `SuaConta.xaml`.

Nesta página, vamos utilizar o mesmo plano de fundo da página anterior, e por hora só haverá um item chamado “Senha”. Sua página deve conter o trecho de código semelhante ao que segue:

```
<Grid x:Name="LayoutRoot">

    <Grid.Background>
        <ImageBrush Stretch="UniformToFill"
            ImageSource="/Assets/Images/panoramaBackground.jpg"/>
    </Grid.Background>

    <phone:Pivot Title="SUA CONTA">

        <phone:PivotItem Header="Senha">

            </phone:PivotItem>

        </phone:Pivot>
    </Grid>
```

Neste ponto, vamos preencher o item “Senha” com os campos para que o usuário digite sua senha atual, uma nova senha e uma confirmação para a nova senha. Além disso, é claro, vamos fazer um botão para efetuar a troca de senha, conforme a figura 5.6.



Figura 5.6: Pivot

Agora vamos fazer com que o item de menu que inserimos na `ApplicationBar` nos leve até este formulário. Para isso precisamos inserir um manipulador para o evento `Click`. Isso é feito tanto no arquivo XAML quanto no arquivo C#. Veja os códigos a seguir:

```
<shell:ApplicationBarItem
  Text="sua conta"
  Click="SuaConta_Click"/>
```

```
private void SuaConta_Click(object sender, EventArgs e)
{
```



```
}
```

Falaremos melhor sobre navegação nos capítulos futuros, então não se preocupe com todos os conceitos envolvidos nesta funcionalidade. Neste ponto, basta você saber que há um objeto encarregado deste tipo de função. Ele se chama `NavigationService` e, nele, há o método `Navigate`, no qual temos de informar um objeto do tipo `Uri` por parâmetro. Observe o código:

```
private void SuaConta_Click(object sender, EventArgs e)
{
    NavigationService.Navigate(
        new Uri("/Paginas/SuaConta.xaml", UriKind.Relative));
}
```

Para finalizarmos este capítulo, utilize o pacote de ferramentas do capítulo anterior para criar transições entre as novas páginas.

Você pode acessar o código-fonte do aplicativo até aqui através do meu github: <http://bit.ly/GITcap5ParteIII>.

CAPÍTULO 6

LocalDB, ApplicationSettings e utilização de dados no aplicativo

Apesar de já termos criado diversas páginas, nosso aplicativo ainda não possui nenhuma ligação com os dados. Como já foi mencionado, vamos criar uma base de dados fictícia simulando o acesso a um serviço. Para fazer isso, vamos resgatar os dados de um arquivo JSON que, junto com XML, são as formas mais comuns de entregas de dados através de serviços. No caso de aplicativos móveis, a utilização de JSON é mais recomendada devido a um consumo menor de banda.

JSON: JAVASCRIPT OBJECT NOTATION

JSON é um padrão aberto para transferência de dados muito comum. É utilizado principalmente para transferir dados de um servidor para um aplicativo web ou móvel. Este padrão utiliza uma linguagem simples através do formato **chave:valor** e é uma alternativa ao padrão XML.

6.1 CARREGANDO OS DADOS PARA A APLICAÇÃO

Antes de podermos utilizar os dados, precisamos de uma representação deles em nossa aplicação. Para fazer isso, criaremos classes C# que definam nossos dados. Em alguns padrões arquiteturais, esses dados são chamados de **modelos**, e o padrão mais comum utilizado em aplicativos para Windows Phone é o **MVVM**.

MVVM: MODEL-VIEW-VIEWMODEL

MVVM é um padrão de arquitetura de aplicativos que pode ser utilizado na plataforma .NET. Ele consiste em três camadas principais: **Model**, que é a camada de dados, **View**, que é a camada de interface (podendo ser páginas web, Windows 8 ou Windows Phone) e **ViewModel**, que é uma abstração de um modelo que possui interação com a interface da aplicação. É um padrão similar aos mais conhecidos: MVC (Model-View-Controller) e MVP (Model-View-Presenter).

É recomendado, porém não obrigatório, separar as camadas do aplicativo em pastas, como já fizemos com nossas páginas. Vamos criar a pasta `Modelos` e os modelos: `Produto`, `Categoria` e `Loja`, conforme o código a seguir.

```
public class Produto
{
    public int Id { get; set; }

    public string Descricao { get; set; }

    public double Preco { get; set; }

    public string DescricaoDetalhada { get; set; }

    public double AvaliacaoMedia { get; set; }

    public Categoria Categoria { get; set; }

    public string Icone { get; set; }
}

public class Categoria
```

```
{
    public int Id { get; set; }

    public int Descricao { get; set; }

    public int DescricaoDetalhada { get; set; }

    public List<Produto> Produtos { get; set; }
}

public class Loja
{
    private Loja()
    {}

    private static Loja dados;
    public static Loja Dados
    {
        get
        {
            if (dados == null)
                dados = new Loja();

            return dados;
        }
        set
        {
            dados = value;
        }
    }

    public List<Produto> Produtos { get; set; }
}
```

A classe `Loja` declarada aqui utiliza o padrão de projeto **Singleton**, que é utilizado para garantir que a classe possua somente uma instância durante toda a aplicação. Caso você queira saber mais sobre isso, acesse o link: <http://bit.ly/padraoSingleton>.

Além destas classes, também precisamos criar uma forma de extrair os dados do arquivo JSON que será embutido em nosso aplicativo. Para isso, utilizamos as classes `Stream` e `StreamReader` do .NET — não entrarei em detalhes para explicar esta

funcionalidade, pois ela será feita da mesma forma que qualquer outra aplicação .NET. Veja o código a seguir.

```
public static class LeitorArquivo
{
    public static string Ler(string caminhoArquivo)
    {
        StreamResourceInfo resourceInfo =
            Application.GetResourceStream(
                new Uri(caminhoArquivo, UriKind.Relative));

        if (resourceInfo != null)
        {
            Stream arquivo = resourceInfo.Stream;
            if (arquivo.CanRead)
            {
                StreamReader leitor =
                    new StreamReader(arquivo,
                        System.Text.Encoding.GetEncoding("iso8859-1"));

                return leitor.ReadToEnd();
            }
        }

        throw new InvalidOperationException
            ("Problema ao ler o arquivo de dados");
    }
}
```

A classe `LeitorArquivo` possui a declaração de `static`. Caso você não esteja familiarizado com este termo, trata-se de uma classe que não pode ser instanciada; você utiliza seus métodos a partir dela própria, por exemplo: `LeitorArquivo.Ler("caminhoArquivo")`. Se quiser obter mais informações, siga a leitura em: <http://bit.ly/staticClass>.

Para continuar a desenvolver a aplicação, é necessário que você faça download do arquivo que contém nossos dados. Ele pode ser baixado através do link <http://bit.ly/dataJSON> e deve ser incluído na pasta `Resources` de seu aplicativo. Além disso, também é necessário fazer o download da pasta que contém os ícones dos nossos produtos, que podem ser encontrados no link <http://bit.ly/dataIcones>.

Depois de incorporar os arquivos para o projeto, precisamos fazer uma pequena alteração no arquivo `dados.txt` para que ele se torne um recurso de fato. É necessário ir até a aba propriedades deste arquivo (através do Visual Studio) e alterar a opção `Build Action`, conforme ilustra a figura 6.1.

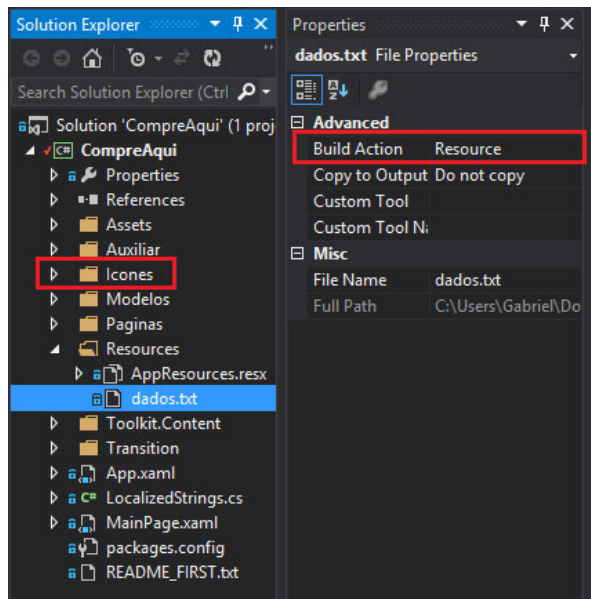


Figura 6.1: Transformando os dados em recursos

Com isso feito, já conseguiremos trabalhar com os dados do arquivo, então devemos inserir o trecho de código que carrega os dados durante a inicialização de nossa aplicação. Isso seria feito da mesma forma caso estivéssemos chamando um serviço real — a diferença é que, em vez de fazermos uma requisição para uma URL na internet, vamos utilizar um leitor de arquivo local.

O código para isso deve ser inserido no método `Application_Launching` da classe `App.xaml.cs`. Este método é executado sempre que a aplicação é iniciada. Neste ponto, devemos utilizar a classe `LeitorArquivo`, conforme o código:

```
private void Application_Launching
(object sender, LaunchingEventArgs e)
{
    string dados =
        LeitorArquivo
```

```
.Ler("/CompreAqui;component/Resources/dados.txt");  
}
```

Com esse código já possuímos uma string contendo todo o JSON de nossos dados, entretanto ainda precisamos converter este texto para objetos C#. É o que faremos utilizando a classe `JsonConvert`. Ela não é disponibilizada no pacote inicial, então teremos de buscá-la em pacotes NuGet, da mesma forma que fizemos com o pacote de ferramentas anteriormente.

Pressione o botão direito sobre as referências de seu projeto e selecione a opção `Manage NuGet Packages`. Após isso, pesquise por `Json.NET` — atualmente ela está na lista de bibliotecas mais baixadas, então é bastante fácil encontrá-la. Feito isso, podemos finalmente concluir nossa extração de dados do arquivo com o seguinte código:

```
private void Application_Launching  
(object sender, LaunchingEventArgs e)  
{  
    string dados =  
        LeitorArquivo.Ler("/CompreAqui;component/Resources/dados.txt");  
  
    Loja.Dados = JsonConvert.DeserializeObject<Loja>(dados);  
}
```

Antes de prosseguirmos, vamos seguir mais uma dica de usabilidade: em um cenário real, nós poderíamos estar carregando uma grande quantidade de dados neste momento, portanto, podemos adicionar uma tela de boas-vindas para evitar a frustração do usuário de permanecer na página de carregamento de nossa aplicação.

Esta tela é chamada de *Splash Screen*. No Windows Phone existem dois tipos de *Splash Screen*. Você pode escolher criar uma imagem diferente para cada possível resolução do sistema operacional, ou criar apenas uma imagem com exatos 768 x 1280 pixels e deixar que o sistema a redimensione para cada resolução.

Vamos optar pela segunda forma, utilizando apenas uma imagem. Para fazer com que nosso aplicativo entenda que deva utilizar esta imagem como *Splash Screen* é necessário inseri-la no diretório raiz de nossa aplicação e nomeá-la como `SplashScreenImage.jpg`. A figura 6.2 já apresenta a *Splash Screen* de nossa aplicação sendo exibida!

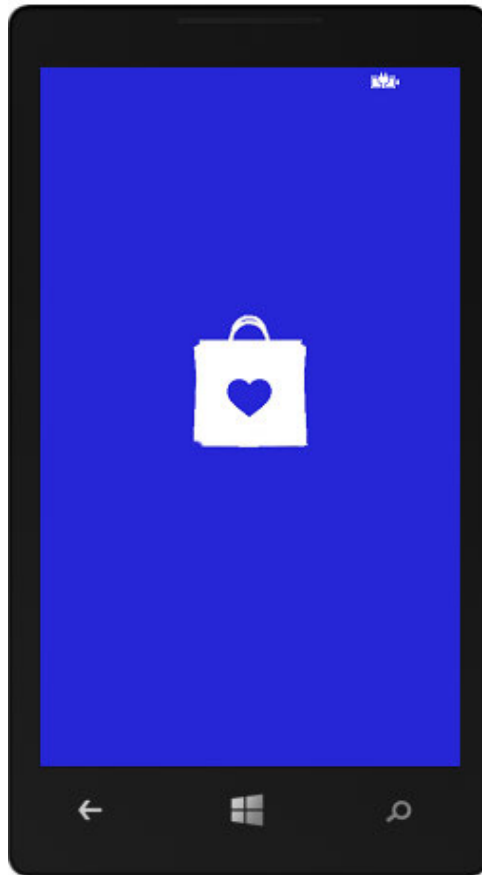


Figura 6.2: Splash Screen

Você pode encontrar o código-fonte da aplicação já apta a carregar os dados e com a Splash Screen pronta em meu github através do link: <http://bit.ly/GITcap6ParteI>.

6.2 VINCULANDO OS DADOS COM OS CONTROLES DE USUÁRIO

Neste ponto, já estamos apto para utilizar os dados em nosso aplicativo. A primeira coisa a fazer é voltarmos à nossa página `ProdutosHub.xaml` e adicionarmos um manipulador para o evento `PageLoaded`. Este evento é disparado quando a pá-

gina termina de ser construída e neste momento vamos buscar as informações para preencher as páginas do nosso panorama.

Vamos ao `PanoramaItem` classificações. Aqui exibiremos uma lista contendo a descrição das categorias dos produtos de nossa loja. Um controle de usuário excelente para este tipo de necessidade é o *LongListSelector*, que permite ao usuário organizar uma lista de informações, onde cada elemento irá replicar um template criado na página. Veja o código:

```
<phone:PanoramaItem Header="classificações">
    <Grid Margin="10,0">
        <phone:LongListSelector>
            <phone:LongListSelector.ItemTemplate>
                <DataTemplate>

                    </DataTemplate>
            </phone:LongListSelector.ItemTemplate>
        </phone:LongListSelector>
    </Grid>
</phone:PanoramaItem>
```

Dentro da tag `DataTemplate` podemos criar uma forma de visualização de nosso dado. Neste exemplo será bastante simples, pois como desejamos mostrar apenas a descrição das categorias, vamos adicionar somente um componente do tipo `TextBlock`. Também é necessário preencher a propriedade `Name` do componente `LongListSelector`, pois através dela poderemos acessá-lo no código-fonte e então vincular os dados.

Já sabemos que temos que adicionar um `TextBlock` como template da lista, mas como vamos preencher o valor do texto deste componente? A resposta para essa pergunta está no conceito de **Binding** implementado pelo XAML. O valor do texto será vinculado com a propriedade `Descricao` dos dados que serão conectados a este componente. Para fazer isso, utilize o trecho de código a seguir:

```
<phone:PanoramaItem Header="classificações">
    <Grid Margin="10,0">

        <phone:LongListSelector x:Name="Categorias">
            <phone:LongListSelector.ItemTemplate>
                <DataTemplate>

                    <TextBlock Text="{Binding Path=Descricao}"
```

```
Style="{StaticResource PhoneTextExtraLargeStyle}"/>

</DataTemplate>
</phone:LongListSelector.ItemTemplate>
</phone:LongListSelector>

</Grid>
</phone:PanoramaItem>
```

Agora só precisamos conectar nossos dados a este componente no manipulador do evento `Loaded` que criamos anteriormente. Todos os nossos dados estão armazenados em memória na propriedade `Dados` do nosso objeto `Loja`. Há uma forma bastante simples de buscá-los no C#, conhecida como **LINQ**.

LINQ: LANGUAGE-INTEGRATED QUERY

LINQ é um padrão de consulta desenvolvido na plataforma .NET para buscar, unir, armazenar e atualizar dados. Através dele você pode manipular dados tanto em coleções de objetos, banco de dados e até XMLs. Para saber mais sobre LINQ acesse: <http://bit.ly/msdnLINQ>.

Nesta primeira consulta, iremos selecionar o `Id` e a descrição das categorias dos produtos, mas sem repeti-las, portanto lembre-se de utilizar o método `Distinct()`, conforme exemplo.

```
Categorias.ItemsSource = (from produtos in Loja.Dados.Produtos
                           select new
                           {
                               Id = produtos.Categoria.Id,
                               Descricao = produtos.Categoria.Descricao
                           }).Distinct().ToList();
```

Com isso, já podemos ver os dados em nossa aplicação!



Figura 6.3: Primeiros dados em nossa aplicação

Como você pode ver no código anterior, a lista de objetos vinculada ao controle `Categorias` é de um objeto anônimo, ou seja, um objeto resultante de uma consulta que não possui uma classe definida. Entretanto, é interessante que você utilize objetos tipados, apenas lembre-se de que os objetos de modelo não devem possuir nenhuma ligação com a página. Neste momento, entra o já apresentado *view model*.

Vamos criar uma pasta chamada *ViewModels* e adicionar nela a classe `ViewModelBase`. Esta classe será utilizada como base para todos os view models de nossa aplicação. Para que um objeto view model notifique a interface quando o valor de suas propriedades for alterado, é necessário que a classe implemente a interface `INotifyPropertyChanged`. Além disso, já criaremos um método para notificar a propriedade que foi alterada, conforme o código a seguir.

```
public class ViewModelBase : INotifyPropertyChanged
{
    protected void NotificarAlteracao(string propriedade)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propriedade));
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```

Neste primeiro exemplo de view model, não será necessária a utilização do método `NotificarAlteracao`, pois o usuário não vai conseguir alterar os dados dos produtos ou das categorias, então a nossa classe ficará bastante similar a uma classe comum. Veja o código.

```
public class CategoriaVM : ViewModelBase
{
    public int Id { get; set; }

    public string Descricao { get; set; }

    public override bool Equals(object obj)
    {
        if (obj is CategoriaVM)
            return Id == ((CategoriaVM)obj).Id;

        return false;
    }

    public override int GetHashCode()
    {
        return this.Id;
    }
}
```

Além das propriedades, nós sobrescrevemos os métodos `Equals` e `GetHashCode`, para alterar a forma com que o C# compara objetos deste tipo. Portanto, não teremos repetição ao utilizar o método `Distinct`. Agora já

podemos alterar a consulta LINQ para retornar dados do tipo `CategoriaVM`, conforme o código:

```
Categorias.ItemsSource = (from produtos in Loja.Dados.Produtos
                          select new CategoriaVM
                          {
                              Id = produtos.Categoria.Id,
                              Descricao = produtos.Categoria.Descricao
                          }).Distinct().ToList();
```

Agora vamos criar um template de dados mais complexo para o item *promoção*. Aqui vamos listar todos os itens que possuem preço promocional, com seu título, sua imagem e com o percentual de desconto.

Uma boa prática para desenvolver templates mais complexos é criá-los fora da lista, como se fossem componentes fixos, até você ficar contente com o design. Somente depois disso insira-o no template e altere o valor das propriedades para as respectivas propriedades via *Binding*.

Primeiro vamos criar um `Grid` e dividi-lo em duas colunas: na esquerda ficará a imagem do produto e na direita ficarão os dados que citamos anteriormente. Daremos uma transparência ao `Grid` principal e, na coluna da imagem, vamos inserir um `Grid` com a cor branca, para dar um fundo ao ícone do produto. Observe o código:

```
<Grid Height="150" Background="#4C000000">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="130"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>

    <Grid Background="White">
        <Image Source="/Icones/camera.png" Height="75" />
    </Grid>
</Grid>
```

Agora vamos inserir os componentes na coluna da direita. Como exibiremos diversos dados, vamos criar um `Grid` aninhado da mesma forma que fizemos com a imagem, conforme o código a seguir.

```
<Grid Grid.Column="1" Margin="10">
    <Grid.RowDefinitions>
```

```
<RowDefinition Height="90"/>
<RowDefinition Height="*/>
</Grid.RowDefinitions>

<TextBlock Text="Teste produto exemplo exemplo"
  FontSize="24" TextWrapping="Wrap" Grid.RowSpan="2"/>

<StackPanel Orientation="Horizontal" Grid.Row="1">

  <TextBlock Text="Desconto de: " VerticalAlignment="Center"/>
  <TextBlock Text="50" FontSize="24" VerticalAlignment="Center"/>
  <TextBlock Text="%" FontSize="24" VerticalAlignment="Center"/>

</StackPanel>

</Grid>
```

Perceba como o conceito de gerenciadores de layout aninhados é utilizado com bastante frequência. Para criarmos uma simples listagem já utilizamos diversos Grids e StackPanels. Com os códigos finalizados, já temos nosso template de dados, basta fazer o *binding* das informações.

O primeiro item é o `TextBlock` com a fonte maior, que representa a descrição do produto, logo, faremos o binding com a propriedade `Descricao`. Neste ponto você já deve ter notado que precisamos fazer o binding com o percentual de desconto do produto e, na verdade, não temos este dado do produto.

No entanto, podemos extrair esta informação a partir das propriedades `Preco` e `PrecoPromocao`. Para podermos fazer o binding, vamos criar um view model para a classe `Produto` que possua esta propriedade, sendo que ela será somente leitura e deve retornar o valor calculado pelos preços. Veja o código:

```
public class ProdutoVM : ViewModelBase
{
    public int Id { get; set; }

    public string Descricao { get; set; }

    public double Preco { get; set; }

    public double PrecoPromocao { get; set; }
```

```
public string DescricaoDetalhada { get; set; }

public double AvaliacaoMedia { get; set; }

public int CategoriaId { get; set; }

public string CategoriaDescricao { get; set; }

public string Icone { get; set; }

public double Desconto
{
    get
    {
        return Math.Round(100 - (PrecoPromocao * 100 / Preco), 2);
    }
}
```

Como você pode ver, nossa propriedade possui apenas o método `get`, já que não é possível inserir um valor nela. Agora você pode utilizar esta propriedade para fazer o binding normalmente.

Por fim, teremos que vincular a imagem do produto ao componente `Image`. Aqui temos um problema diferente no momento de fazer o binding. O componente espera que o valor vinculado seja uma `Uri` ou algum outro tipo de fonte de imagens, enquanto que a propriedade do nosso objeto é um tipo `string`.

Para solucionar esse problema, utilizaremos um recurso chamado *Value Converter*, que facilita o vínculo de dados de tipos diferentes. Vamos criar uma pasta chamada `Converter` e adicionar nela uma classe chamada `ImageSourceConverter` — não há nenhuma obrigatoriedade quanto à palavra “Converter” no nome da classe, mas é muito comum que elas se chamem assim.

Nossa classe deve implementar a interface `IValueConverter`, que está no namespace `System.Windows.Data`. Ao fazer isso, você terá de implementar os dois métodos dessa interface, conforme o código:

```
public class ImageSourceConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
        object parameter, System.Globalization.CultureInfo culture)
    {
```



```
        throw new NotImplementedException();
    }

    public object ConvertBack(object value, Type targetType,
        object parameter, System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

Estes métodos são utilizados para converter o valor para a interface XAML e converter novamente para o código C#, respectivamente. Em nosso exemplo, não precisaremos implementar o método `ConvertBack`, mas o método `Convert` é essencial. Teremos de utilizar o valor da propriedade que virá através do parâmetro `value` e transformá-lo em uma `Uri` para encontrar o ícone do produto, conforme codificação a seguir.

```
public object Convert(object value, Type targetType,
    object parameter, System.Globalization.CultureInfo culture)
{
    string valor = value as string;
    return new Uri( string.Concat("/Icones/", valor),
        UriKind.Relative);
}
```

É um código bastante simples, porém altamente necessário para o funcionamento de nosso binding. Agora voltaremos à página que estávamos implementando e adicionaremos o namespace de nosso converter.

```
xmlns:converter="clr-namespace:CompreAqui.Converter"
```

Para finalizar, devemos preencher a propriedade `Source` do objeto `Image` fazendo o binding com a propriedade `Ícone`. Porém, utilizaremos nosso conversor de valores também e, para fazer isso, é preciso declará-lo como recurso da aplicação, página ou componente. Neste caso, optaremos por fazer dele um recurso apenas do componente, como ilustra o código a seguir.

```
<Grid Background="White">
    <Grid.Resources>

        <converter:ImageSourceConverter
```

```

        x:Key="converter"/>

</Grid.Resources>

<Image Source="{Binding Path=Icone,
                        Converter={StaticResource converter}}"
        Height="75" />

</Grid>

```

Por fim, voltaremos ao método `PhoneApplicationPage_Loaded` e faremos a consulta para encontrar os produtos que estão com o preço em promoção, ordenando-os do maior desconto para o menor:

```

Promocoes.ItemsSource = (from produtos in Loja.Dados.Produtos
                          where produtos.PrecoPromocao != 0
                          select new ProdutoVM
                          {
                              Id = produtos.Id,
                              Descricao = produtos.Descricao,
                              Preco = produtos.Preco,
                              PrecoPromocao = produtos.PrecoPromocao,
                              Icone = produtos.Icone
                          })
                          .OrderByDescending(produto => produto.Desconto)
                          .ToList();

```

Agora já podemos ver os dados dos produtos que estão em promoção, como ilustra a figura 6.4.



Figura 6.4: Produtos em promoção

Como você deve ter notado na imagem, a lista de produtos ultrapassa a altura da página, então não se esqueça de adicionar o componente `ScrollView` neste ponto, como já fizemos anteriormente.

Os dois primeiros itens do *Panorama* já estão montados. Agora vamos preencher o terceiro item com uma nova lista, porém com um template semelhante ao template de promoções. Mostraremos a imagem e o título, mas em vez de exibir a quantidade de desconto, vamos mostrar o preço.

Apesar do nome deste item ser **produtos**, não vamos mostrar todos os produtos nesta tela, mas apenas pegar dois produtos sorteados e adicionar um link para que o usuário possa visualizar todos os produtos em uma outra página. Em um cenário

real, esta técnica é geralmente utilizada para que o usuário não espere muito tempo durante a consulta de dados.

A primeira coisa a fazer é adicionar o `Grid` neste item e dividi-lo em duas linhas. Já podemos preencher a linha inferior com um `TextBlock` com o texto “todos os produtos”.

```
<phone:PanoramaItem Header="produtos">
  <Grid Margin="10,0">
    <Grid.RowDefinitions>
      <RowDefinition Height="*/"/>
      <RowDefinition Height="160"/>
    </Grid.RowDefinitions>

    <TextBlock Text="todos os produtos"
      Style="{StaticResource PhoneTextExtraLargeStyle}"
      Grid.Row="1" Margin="0"/>

  </Grid>
</phone:PanoramaItem>
```

Agora vamos utilizar o mesmo template na linha superior, fazendo algumas pequenas modificações para que ele encaixe melhor nesta página. Altere a altura do `Grid` que define o template de cada item da lista de 150 para 130, e diminua as margens verticais de 10 para 5. Por último, altere o conteúdo do `StackPanel` que exibia o desconto para exibir o preço do produto.

```
<phone:LongListSelector x:Name="Produtos">
  <phone:LongListSelector.ItemTemplate>
    <DataTemplate>
      <Grid Height="130" Margin="0,5"
        Background="#4C000000" VerticalAlignment="Top">

        <!-- Conteúdo sem alterações -->

        <StackPanel Orientation="Horizontal" Grid.Row="1">
          <TextBlock Text="por: " VerticalAlignment="Center"/>

          <TextBlock Text="{Binding Path=PrecoAPagar}"
            FontSize="24" VerticalAlignment="Center"/>

          <TextBlock Text=" R$" FontSize="24"
```

```
        VerticalAlignment="Center"/>
    </StackPanel>
</Grid>
</Grid>
</DataTemplate>
</phone:LongListSelector.ItemTemplate>
</phone:LongListSelector>
```

Neste trecho, a parte que não teve alterações em relação ao template utilizado no item “promoção” foi suprimida para facilitar a leitura do código e para dar mais ênfase ao que foi alterado. Você pode ter notado que um dos textos está vinculado à propriedade `PrecoAPagar` e ela ainda não existe em nossa classe `ProdutoVM`.

Vamos criá-la para retornar o preço que o comprador irá pagar no produto. A regra é bastante simples: esta propriedade vai retornar o valor da propriedade `Preco` do produto quando a propriedade `PrecoPromocao` estiver com o valor zero; caso contrário, o valor retornado deve ser o da propriedade `PrecoPromocao`.

```
public double PrecoAPagar
{
    get
    {
        double valor;
        if (PrecoPromocao != 0)
            valor = PrecoPromocao;
        else
            valor = Preco;

        return Math.Round(valor, 2);
    }
}
```

Agora vamos voltar ao método `PhoneApplicationPage_Loaded` e adicionar a fonte de dados para esta lista. Esteja à vontade para desenvolver um algoritmo que busque produtos aleatórios, mas para fins de demonstração vamos apenas selecionar os produtos com os identificadores 3 e 4. Dessa forma, sempre serão selecionados os mesmos produtos, já que nossa base de dados é estática.

```
Produtos.ItemsSource = (from produtos in Loja.Dados.Produtos
                        where produtos.Id == 3 || produtos.Id == 4
                        select new ProdutoVM
```

```
{  
    Id = produtos.Id,  
    Descricao = produtos.Descricao,  
    Icone = produtos.Icone,  
    Preco = produtos.Preco,  
    PrecoPromocao = produtos.PrecoPromocao  
})  
.ToList();
```

A figura 6.5 ilustra o resultado de nossa implementação.



Figura 6.5: Listagem de produtos

É sempre uma boa prática circundar as listas com gerenciadores de layout do

tipo `ScrollView`, pois as listas podem aumentar de tamanho ao decorrer do uso da aplicação.

Caso você queira acessar o código-fonte do projeto até este ponto, você pode encontrá-lo no github através do link: <http://bit.ly/GITcap6ParteII>.

6.3 LOCALDB

Já mostramos como utilizar dados para exibir informações. Agora vamos verificar como um aplicativo Windows Phone pode armazenar informações localmente. Em um cenário real, toda a parte de criação de contas de usuários deveria ser armazenada em um banco de dados na nuvem. Para fins de exemplo, utilizaremos o banco de dados local do Windows Phone para guardar este tipo de informação.

Cada aplicativo em um dispositivo Windows Phone possui direito a um espaço em disco nomeado *Isolated Storage*, utilizado para a criação de um banco de dados local. Este banco de dados é individual por aplicativo, o que impossibilita o compartilhamento de dados armazenados neste banco de dados entre diferentes aplicativos.

O Windows Phone utiliza o gerenciador de banco de dados SQL **Server Compact Edition**, que é uma versão muito mais simples e leve que a alternativa SQL Server para servidores.

Para acessar este tipo de banco de dados em nosso aplicativo, utilizaremos a API **LINQ to SQL**. Já vimos anteriormente que o LINQ é uma forma de criar consultas em coleções e em dados na linguagem C#. Esta API, habilita o LINQ a fazer consultas, inserções e alterações no banco de dados através do C#.

Também vamos mapear objetos C# e suas propriedades para uma tabela e suas colunas. Podemos dizer que o **LINQ to SQL** é dividido em dois principais componentes: **DataContext** e **Runtime**.

O **DataContext** funciona como uma representação do banco de dados em nosso aplicativo, isso é, toda a interação com o banco de dados é feita através dele. O **Runtime** é responsável por traduzir as consultas LINQ escritas no aplicativo para consultas Transact-SQL que executam no banco de dados SQL Server e também por traduzir os resultados da consulta novamente para a linguagem C#.

Agora vamos voltar para nosso aplicativo e criar as classes `Usuario`, na pasta `Modelos`, e `BancoDados`, na pasta `Resources`, conforme o código:

```
public class Usuario
{
    public int Id { get; set; }
```

```

    public string Email { get; set; }
    public string NomeUsuario { get; set; }
    public string Senha { get; set; }
    public bool EntrarAutomaticamente { get; set; }
}

public class BancoDados : DataContext
{
    public BancoDados(string stringConexao)
        : base(stringConexao)
    {}

    public static string StringConexao
    {
        get
        {
            return "isostore:/compreaqui.sdf";
        }
    }

    public Table<Usuario> Usuarios;
}

```

Obrigatoriamente a classe `BancoDados` deve possuir um construtor que receba uma string. Isso é necessário pois a classe `DataContext` precisa da localização em que o banco de dados será armazenado no dispositivo.

O valor a ser inserido neste construtor pode ser recuperado através da propriedade estática `StringConexao`, como você pode visualizar no código. Para acessarmos o *isolated storage* do aplicativo, utilizamos o prefixo `isostore:/`.

Nossa classe `Usuario` ainda precisa sofrer alterações para que ela se torne de fato uma tabela. As próximas alterações são chamadas de *Annotations* ou *Atributos de Classe / Propriedade / Método*. Precisamos identificar a classe como tabela e as propriedades como colunas.

Estas anotações servem para identificar quais propriedades são mapeadas e também para informar dados a respeito do banco, por exemplo, qual das propriedades é a chave primária da tabela, tipo do campo no banco de dados, tamanho máximo do campo etc. Depois das alterações, a classe `Usuario` ficará da seguinte forma:

```
[ Table(Name="Usuarios") ]
```



```
public class Usuario
{
    [Column(IsPrimaryKey=true,IsDbGenerated=true)]
    public int Id { get; set; }

    [Column]
    public string Email { get; set; }

    [Column]
    public string NomeUsuario { get; set; }

    [Column]
    public string Senha { get; set; }

    [Column]
    public bool EntrarAutomaticamente { get; set; }
}
```

Agora vamos executar a criação do nosso banco de dados durante a inicialização da aplicação. Também vamos garantir que sempre haja um usuário cadastrado, o qual nomearemos de **admin**. Volte à classe `App.xaml.cs` e crie o método `CriarBancoDados`, que fará as tarefas que citamos anteriormente.

```
private void CriarBancoDados()
{
    using (BancoDados bancoDados =
        new BancoDados(BancoDados.StringConexao))
    {
        if (!bancoDados.DatabaseExists())
        {
            bancoDados.CreateDatabase();

            Usuario admin = new Usuario();
            admin.NomeUsuario = "admin";
            admin.Senha = "admin";
            admin.Email = "admin@compreaqui.com.br";

            bancoDados.Usuarios.InsertOnSubmit(admin);
            bancoDados.SubmitChanges();
        }
    }
}
```

```
}
```

Deve haver uma chamada para este método no fim do construtor desta classe, para garantir que o banco de dados seja criado na primeira vez que o aplicativo for aberto.

Agora vamos voltar para a página de criação de contas e fazer a ligação inversa dos dados. Assim faremos com que o nosso objeto receba os valores da página, seguindo o já mencionado padrão *MVVM*. Vamos criar mais uma classe na pasta `ViewModels` que se chamará `UsuarioVM`.

Lembre-se que o view model deve conter apenas as propriedades que são acessíveis pela interface, então esta classe possuirá uma propriedade para o campo “confirmação de senha”. O código a seguir mostra a estrutura que as propriedades devem seguir, exceto pela propriedade `EntrarAutomaticamente`, pois ela possui o valor `true` como padrão.

```
public class UsuarioVM : ViewModelBase
{
    private string _email;
    public string Email
    {
        get { return _email; }
        set
        {
            _email = value;
            NotificarAlteracao("Email");
        }
    }

    //Outras propriedades
    //...

    private bool _entrarAutomaticamente = true;
    public bool EntrarAutomaticamente
    {
        get { return _entrarAutomaticamente; }
        set
        {
            _entrarAutomaticamente = value;
            NotificarAlteracao("EntrarAutomaticamente");
        }
    }
}
```

```
}  
  
}
```

Com a classe `UsuarioVM` criada, já estamos aptos a vincular os dados na página de criação de contas. Como já vimos como fazer o binding das propriedades, não precisamos entrar em detalhes neste ponto.

Depois de ligar os controles de usuário com as devidas propriedades da classe, você precisa criar manipuladores para os eventos `Loaded` da página e `Tap` do botão “Criar conta”. No código C# de nossa página, vamos criar um atributo do tipo `UsuarioVM`, que será atribuído na propriedade `DataContext` da página durante o evento `Loaded`.

```
private void PhoneApplicationPage_Loaded  
(object sender, RoutedEventArgs e)  
{  
    if (_usuarioVM == null)  
        _usuarioVM = new UsuarioVM();  
  
    this.DataContext = _usuarioVM;  
}
```

Após fazer isso, vamos criar um método chamado `ValidarCamposCadastro` na classe `UsuarioVM`. Ele deve ser do tipo `string` e deverá validar o preenchimento de todas as propriedades, bem como a igualdade entre os campos `Senha` e `ConfirmacaoSenha`. Vamos concatenar uma mensagem com os problemas que encontrarmos e retornar mensagem completa no método.

```
public string ValidarCamposCadastro()  
{  
    StringBuilder validacoes = new StringBuilder();  
  
    if (string.IsNullOrEmpty(Email))  
        validacoes.AppendLine("- É necessário preencher o campo Email");  
  
    if (string.IsNullOrEmpty(Nome))  
        validacoes.AppendLine("- É necessário preencher o campo Usuário");  
  
    if (string.IsNullOrEmpty(Senha) ||  
        string.IsNullOrEmpty(ConfirmacaoSenha))
```

```

        validacoes.AppendLine(
            "- É necessário preencher os campos Senha e Confirmação de Senha");

    else
        if (Senha != ConfirmacaoSenha)
            validacoes.AppendLine(
                "- Os campos Senha e Confirmação de senha estão com valores diferentes");

    return validacoes.ToString();
}

```

A criação do método `ValidarCamposCadastro` na classe de view model pode ser questionável — alguns desenvolvedores preferem criar um método dentro da classe modelo — mas, pessoalmente, prefiro que a classe modelo tenha responsabilidade pelas regras de negócio e pelas informações, enquanto o view model faz todo o intermédio da página com o modelo, inclusive em sua validação.

Agora voltaremos para o código da página de criação de contas e preencheremos o método `Button_Tap`. A primeira parte deste método consiste em utilizar a validação que criamos e exibir uma mensagem ao usuário quando houver algum problema.

```

private void Button_Tap
(object sender, System.Windows.Input.GestureEventArgs e)
{
    string validacoes = _usuarioVM.ValidarCamposCadastro();
    if (!string.IsNullOrEmpty(validacoes))
    {
        string mensagem = string.Concat(
            "Não foi possível gravar esta conta por um ou mais motivos abaixo:",
            Environment.NewLine, validacoes);

        MessageBox.Show(mensagem);
    }
    else
    {
    }
}

```

Vamos testar nossa aplicação para verificar como ela se comporta.

Se você tentou executar os testes de validação, vai perceber que mesmo preenchendo os campos a mensagem ainda é preenchida, como ilustra a figura 6.6.

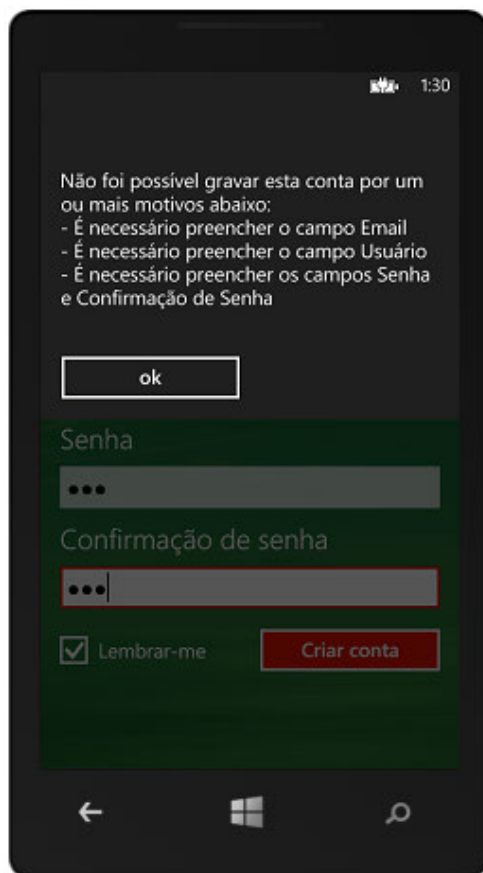


Figura 6.6: Problema com validação

Isso acontece porque o modo padrão do binding apenas vincula os dados no momento em que o contexto de dados é atribuído. Para que o objeto receba atualizações conforme o valor dos campos, precisamos de mais um detalhe em nossa interface. Após selecionarmos a propriedade `Path` do binding, também vamos atribuir a propriedade `Mode` com o valor `TwoWay`, conforme o exemplo:

```
<TextBox InputScope="EmailUserName"
  Text="{Binding Path=Email,Mode=TwoWay}"/>
```

Se fizermos um teste parecido, mas deixando apenas o campo “Usuário” em branco, teremos o resultado correto.

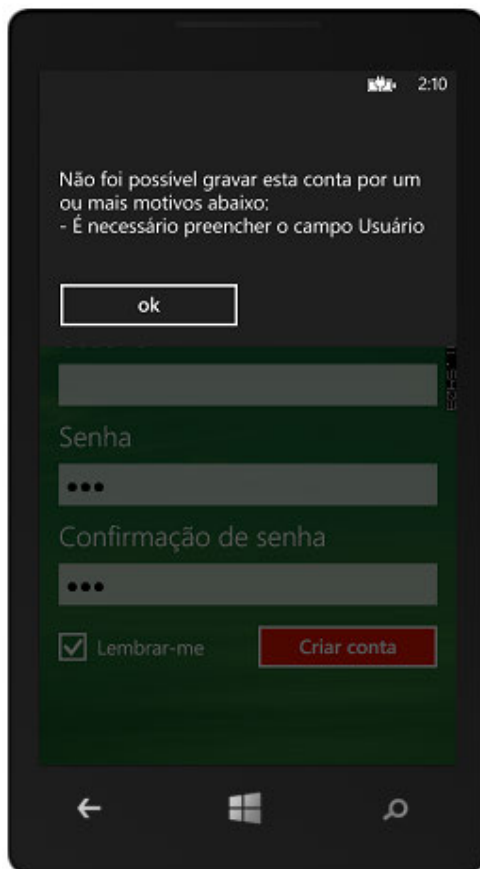


Figura 6.7: Validação correta

Agora vamos completar o método para armazenar o usuário criado e direcioná-lo para a página `ProdutosHub.xaml`. Criaremos um método chamado `GravarUsuario`, que deve extrair as informações do view model e armazená-las no banco de dados.

```
private void GravarUsuario()
{
    Usuario novoUsuario = new Usuario();
```

```
novoUsuario.Email = _usuarioVM.Email;
novoUsuario.NomeUsuario = _usuarioVM.Nome;
novoUsuario.Senha = _usuarioVM.Senha;
novoUsuario.EntrarAutomaticamente = _usuarioVM.EntrarAutomaticamente;

using (BancoDados bancoDados =
    new BancoDados(BancoDados.StringConexao))
{
    bancoDados.Usuarios.InsertOnSubmit(novoUsuario);
    try
    {
        bancoDados.SubmitChanges();
        NavigationService
            .Navigate(new Uri("/Paginas/ProdutosHub.xaml", UriKind.Relative));
    }
    catch
    {
        MessageBox.Show(
            "Houve um problema ao tentar criar sua conta,
            tente novamente mais tarde.");
    }
}
}
```

Utilizamos as instruções **try** e **catch** para exibir uma mensagem mais amigável ao usuário caso aconteça algum problema, mas você deve ter notado que estamos apenas passando as informações do view model para o modelo. Em uma aplicação real, a senha **jamais** deve ser armazenada desta forma — é extremamente importante que você utilize uma criptografia no momento de armazená-la.

Outra coisa que ainda está em aberto é o usuário autenticado. Apesar de fazermos a autenticação no momento da criação da conta, não estamos armazenando esta informação em nenhum lugar, o que nos impede de conhecer qual o usuário que está autenticado em nossa aplicação.

Utilizaremos um outro conceito para armazenar esta informação: trata-se do **ApplicationSettings**.

6.4 APPLICATIONSETTINGS E OS IMPACTOS EM NOSSO APLICATIVO

Este conceito é utilizado para armazenar informações simples em um modelo chave-valor. Nós o utilizaremos para armazenar qual usuário está autenticado no momento. Essas configurações são acessíveis através da propriedade `ApplicationSettings` da classe `IsolatedStorageSettings`.

Vamos simplesmente armazenar a propriedade `Id` do usuário autenticado. O mecanismo é tão simples que podemos fazer isso utilizando apenas uma linha de código, entretanto, é sempre necessário verificar se a chave já existe, pois o usuário pode cadastrar mais de uma conta no mesmo dispositivo.

Este método será criado dentro da classe `Usuario` e se chamará `Autenticar`, conforme o código:

```
public void Autenticar()
{
    IsolatedStorageSettings configuracoes =
        IsolatedStorageSettings.ApplicationSettings;

    if (configuracoes.Contains("usuarioId"))
    {
        configuracoes["usuarioId"] = this.Id;
    }
    else
    {
        configuracoes.Add("usuarioId", this.Id);
    }

    configuracoes.Save();
}
```

Teremos de fazer uma chamada para este método logo após armazenar o usuário no banco de dados, para garantir que teremos o identificador do usuário criado e autenticado.

Com isso, já conseguiremos detectar se há um usuário autenticado no aplicativo e agora precisamos cercar os pontos onde isso é importante. O primeiro passo é uma nova regra em nossa aplicação: caso o usuário esteja autenticado no aplicativo e pressione o botão `voltar` do dispositivo, temos de fechar o aplicativo, pois assumiremos que ele está saindo.

Atualmente, se criarmos um usuário, conectarmos na aplicação e pressionarmos o botão `voltar` do dispositivo, seremos redirecionado para a página de criação de usuário. Para alterar o comportamento padrão do botão `voltar` temos de ir até nossa página e criar um manipulador para o evento `BackKeyPress`.

Após criarmos o método para este evento, devemos validar a seguinte regra: caso o usuário esteja autenticado e pressione o botão voltar neste ponto, o aplicativo irá exibir uma mensagem confirmando a saída do aplicativo.

```
private void PhoneApplicationPage_BackKeyPress
(object sender, System.ComponentModel.CancelEventArgs e)
{
    IsolatedStorageSettings configuracoes =
        IsolatedStorageSettings.ApplicationSettings;

    if ( configuracoes.Contains("usuarioId") &&
        Convert.ToInt32(configuracoes["usuarioId"]) != 0)
    {
        MessageBoxResult resultado =
            MessageBox.Show("Deseja realmente sair do aplicativo?",
                           "Confirmação", MessageBoxButton.OKCancel);

        if (resultado == MessageBoxResult.OK)
        {
            Application.Current.Terminate();
        }
        else
        {
            e.Cancel = true;
        }
    }
}
```

Dessa forma evitamos o problema de o usuário retornar até a página de criação de contas ou mesmo até a página inicial. Ele terá de sair da conta atual para poder visualizar estas páginas novamente, e esta opção será criada na página “sua conta”. Mas antes disso, vamos fazer com que a marcação “Lembrar-me” funcione.

Para isso, teremos que voltar até nossa página inicial e sobrescrever o método `OnNavigatedTo`. Este método é disparado quando o usuário é enviado para a página, seja qual for o motivo, inclusive por ele abrir a aplicação. Vamos interceptar

este método e fazer com que o usuário vá para a página `ProdutosHub` quando a regra se cumprir.

```
protected override void
    OnNavigatedTo(NavigationEventArgs e)
{
    IsolatedStorageSettings configuracoes =
        IsolatedStorageSettings.ApplicationSettings;

    if ( configuracoes.Contains("usuarioId") &&
        Convert.ToInt32(configuracoes["usuarioId"]) != 0)
    {

        using (BancoDados dados = new BancoDados(BancoDados.StringConexao))
        {
            Usuario ultimoUsuario =
                dados.Usuarios
                    .FirstOrDefault(usuario =>
                        usuario.Id == Convert.ToInt32(configuracoes["usuarioId"]));

            if (ultimoUsuario.EntrarAutomaticamente)
            {
                NavigationService
                    .Navigate(new Uri("/Paginas/ProdutosHub.xaml", UriKind.Relative));
            }
            else
                configuracoes["usuarioId"] = 0;
        }
        base.OnNavigatedTo(e);
    }
}
```

Com esta implementação, caso o usuário tenha deixado a opção “Lembrar-me” marcada, sempre que ele entrar no aplicativo já será redirecionado diretamente para o hub de produtos.

Agora você já está apto a modificar a página `Entrar.xaml` para validar o usuário e senha e também armazenar a autenticação. Além disso, você deverá criar um novo item de *Pivot* na página `SuaConta.xaml` com o título de “Informações”.

Este item deve ser o primeiro da página e deverá mostrar os dados do usuário autenticado no momento, bem como a opção para sair, que deve cancelar a autenticação do usuário na aplicação. Para esta segunda opção, você deve utilizar a

`ApplicationBar`. É importante que você faça isso antes de seguir no livro, pois no próximo capítulo estas alterações já estarão implementadas.

Caso queira acessar o código-fonte do projeto até o fim deste capítulo, você pode encontrá-lo no github através do link: <http://bit.ly/GITcap6ParteIII>.

CAPÍTULO 7

Navegação e ciclo de vida

Até este capítulo, nossa aplicação já está boa parte implementada, mas ainda temos que criar a página para visualizar todos os produtos e para organizá-los por categorias. Entretanto, estas duas páginas podem ser apenas uma. Isso pode ser feito através de passagem de parâmetros entre as páginas.

7.1 NAVEGAÇÃO ENTRE PÁGINAS

Nós já utilizamos os princípios básicos de navegação nos capítulos anteriores. Agora vamos compreender melhor como isso funciona.

Cada aplicação Windows Phone possui uma instância de um **PhoneApplicationFrame**. Este componente torna possível a navegabilidade entre as páginas através de uma pilha de navegação, que faz com que o botão `voltar` do dispositivo direcione o usuário para a página correta.

Existem basicamente dois tipos de navegação: **forward** e **backward**. Nós já vimos que este conceito existia enquanto criávamos nossas animações. A navegação

forward é utilizada sempre que direcionamos o usuário para uma página através do `NavigationService` ou quando o usuário abre nosso aplicativo. Já a navegação **backward** é ativada quando o usuário pressiona o botão voltar de seu dispositivo.

Vamos criar uma página chamada `Produtos.xaml`. Aplique a ela o mesmo plano de fundo das páginas `Pivot` e `Panorama` e deixe apenas como título o texto “produtos”, como ilustra a figura 7.1.

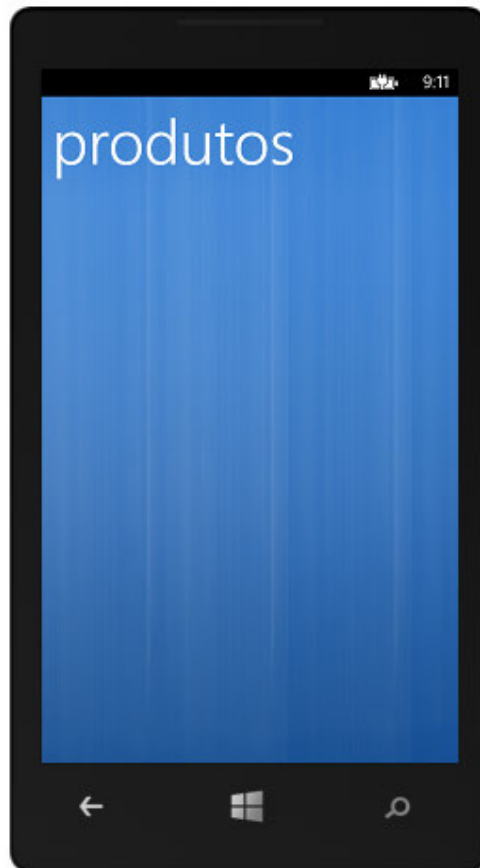


Figura 7.1: Nova página

Para listar os produtos, criaremos um template semelhante ao utilizado no item de panorama “produtos”, mas vamos fazer pequenas alterações na forma de exibir as informações, mostrando além do nome e do preço a avaliação média do produto. Veja o código.

```
<DataTemplate>
  <Grid Height="150" Background="#4C000000"
    VerticalAlignment="Top" Margin="0,5">

    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="110"/>
      <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>

    <Grid Background="White">
      <Grid.Resources>
        <converter:ImageSourceConverter x:Key="converter"/>
      </Grid.Resources>

      <Image Height="75" Source="{Binding Path=Icône,
        Converter={StaticResource converter}}"/>
    </Grid>

    <StackPanel Grid.Column="1" Margin="10">

      <TextBlock Text="{Binding Path=Descricao}"
        FontSize="24" TextWrapping="Wrap"/>

      <StackPanel Orientation="Horizontal">
        <TextBlock Text="por: " VerticalAlignment="Center"/>

        <TextBlock Text="{Binding Path=PrecoAPagar}"
          FontSize="24" VerticalAlignment="Center"/>

        <TextBlock Text=" R$" FontSize="24"
          VerticalAlignment="Center"/>

        <TextBlock Text="Avaliação média: "
          FontSize="24" VerticalAlignment="Center"
          Margin="10,0,0,0"
        />

        <TextBlock Text="{Binding Path=AvaliacaoMedia}"
          FontSize="24" VerticalAlignment="Center"/>
      </StackPanel>
    </StackPanel>
  </Grid>
</DataTemplate>
```

```
</StackPanel>

</Grid>
</DataTemplate>
```

Agora que já temos o design desta página criado vamos vincular os dados para exibir a listagem. Faremos isso sobrescrevendo o método `OnNavigatedTo`, como já fizemos na página `MainPage.xaml`.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    List<ProdutoVM> produtos = (from produto in Loja.Dados.Produtos
                                select new ProdutoVM
                                {
                                    Id = produto.Id,
                                    Descricao = produto.Descricao,
                                    Preco = produto.Preco,
                                    PrecoPromocao = produto.PrecoPromocao,
                                    AvaliacaoMedia = produto.AvaliacaoMedia,
                                    CategoriaId = produto.Categoria.Id,
                                    Icone = produto.Icone
                                })
                                .ToList();

    Listagem.ItemsSource = produtos;
}
```

Com esta implementação, já podemos ver a listagem de todos os produtos, como ilustra a figura [7.2](#)



Figura 7.2: Listagem dos produtos

Agora precisamos criar uma forma para reutilizar esta página com filtros por categoria. É a hora de utilizar passagem de parâmetro entre as páginas.

Primeiro, vamos voltar até o item `Categorias` na página `ProdutosHub.xaml`. Vamos adicionar ao template que lista a descrição das categorias um manipulador para o evento `Tap`, conforme o código.

```
<phone:LongListSelector x:Name="Categorias">
  <phone:LongListSelector.ItemTemplate>
    <DataTemplate>

      <TextBlock Text="{Binding Path=Descricao}" Tap="Categoria_Tap"
        Style="{StaticResource PhoneTextExtraLargeStyle}"/>
```

```

    </DataTemplate>
  </phone:LongListSelector.ItemTemplate>
</phone:LongListSelector>

```

Dentro deste método C#, teremos que obter todas as informações que vamos passar por parâmetro. Isso é feito através do parâmetro `sender`, que é enviado no método disparado pelo evento `Tap`. Vamos converter o objeto `sender` para seu tipo original (`TextBlock`) para então extrairmos as informações.

Para fazer a filtragem e alteração do título na página, precisaremos do `Id` da categoria. Entretanto, nosso componente `TextBlock` não possui esta informação. Existem diversas maneiras diferentes de fazer isso, vamos selecionar uma bastante simples.

Da mesma forma que armazenamos a descrição da categoria na propriedade `Text`, vamos armazenar seu identificador na propriedade `Tag` através do *binding*.

```

<TextBlock Text="{Binding Path=Descricao}" Tag="{Binding Path=Id}"
  Tap="Categoria_Tap"
  Style="{StaticResource PhoneTextExtraLargeStyle}"/>

```

Com isso, já podemos extrair as informações do componente e enviá-las. Para enviarmos o usuário para uma página, utilizamos um objeto do tipo `Uri` e, se você já trabalhou com desenvolvimento web em algum momento, você certamente já conhece a forma de passar parâmetros através dele.

Após o caminho, é necessário informar o caractere "?", seguido pelo nome e valor de cada parâmetro, conforme o código a seguir.

```

private void Categoria_Tap
(object sender, System.Windows.Input.GestureEventArgs e)
{
    TextBlock categoriaClicada = sender as TextBlock;

    string categoria = categoriaClicada.Text;
    string categoriaId = Convert.ToString(categoriaClicada.Tag);

    string parametros =
    string.Format("?categoria={0}&categoriaId={1}",
                  categoria, categoriaId);

    NavigationService.Navigate(new Uri

```

```
(string.Concat("/Paginas/Produtos.xaml", parametros),  
    UriKind.Relative));  
}
```

Precisamos voltar mais uma vez para o código da página `Produtos.xaml` para capturarmos os parâmetros enviados a ela. Os parâmetros são armazenados na propriedade `QueryString` do objeto `NavigationContext` na estrutura de chave e valor.

Existem duas formas de capturar um parâmetro na página. A primeira é simplesmente obtermos o valor através do nome do parâmetro na coleção, conforme o código de exemplo.

```
string valor = NavigationContext.QueryString["parametro"];
```

Esta técnica é bastante simples, mas possui um grande problema: caso o parâmetro que você informou na coleção não exista, você receberá uma exceção, que, se não tratada, pode até mesmo fechar seu aplicativo. Felizmente, há uma segunda forma que gerencia este problema e retorna o valor apenas quando ele existe, sem causar problemas quando não houver o parâmetro.

Esta segunda forma pode ser utilizada através do método `TryGetValue`. Apesar de ser útil para nossa situação, temos de ficar atento, pois caso o parâmetro seja obrigatório é mais aconselhável que sua aplicação lance uma exceção do que simplesmente a suprima.

Agora que já vimos as duas formas de obtermos o valor, vamos alterar o método `OnNavigatedTo` para pegarmos os parâmetros e utilizá-los para modificar a página, conforme o código.

```
protected override void OnNavigatedTo(NavigationEventArgs e)  
{  
    base.OnNavigatedTo(e);  
    List<ProdutoVM> produtos = (from produto in Loja.Dados.Produtos  
                                select new ProdutoVM  
                                {  
                                    Id = produto.Id,  
                                    Descricao = produto.Descricao,  
                                    Preco = produto.Preco,  
                                    PrecoPromocao = produto.PrecoPromocao,  
                                    AvaliacaoMedia = produto.AvaliacaoMedia,  
                                    CategoriaId = produto.Categoria.Id,  
                                })
```

```
        Icone = produto.Icone
    })
    .ToList();

string categoria, categoriaId;

NavigationContext.QueryString
    .TryGetValue("categoria", out categoria);

NavigationContext.QueryString
    .TryGetValue("categoriaId", out categoriaId);

if (!string.IsNullOrEmpty(categoria))
    Titulo.Text = categoria.ToLower();

if (!string.IsNullOrEmpty(categoriaId))
    produtos =
        produtos.Where(produto =>
            produto.CategoriaId == Convert.ToInt32(categoriaId))
            .ToList();

Listagem.ItemsSource = produtos;
}
```

Nossa página já está pronta para se ajustar conforme os parâmetros utilizados! A figura 7.3 ilustra as diferentes formas desta página.



Figura 7.3: Página recebendo parâmetros

Caso você queira acessar o código-fonte da aplicação até este momento, acesse meu github através do link: <http://bit.ly/GITcap7Partel>

7.2 CICLO DE VIDA DA APLICAÇÃO

É muito importante que você entenda o ciclo de vida de uma aplicação Windows Phone para que você possa manter uma experiência agradável para o usuário. Cada aplicativo possui quatro estados: **closed**, **running**, **dormant** e **tombstoned**.

Originalmente, todo aplicativo está em seu estado **closed**, ou seja, fechado. Neste estado, o aplicativo não consome memória e recursos do smartphone.

Quando o usuário abre o aplicativo, ele passa do estado **closed** para o estado **running**. Você pode injetar código C# durante esta transição criando um manipulador para o evento **Launching**, como já fizemos anteriormente para ler os dados do arquivo JSON.

No estado **running**, o aplicativo passa a consumir recursos do dispositivo ativamente, conforme sua demanda. Este estado permanece ativo enquanto o usuário utiliza o software. Caso o usuário pressione o botão voltar do dispositivo e feche seu aplicativo, ele voltará para o estado **closed** e o evento **Closing** é disparado.

Diferente do estado **closed**, o estado **running** possui mais de uma saída. Caso o usuário pressione o botão “windows” do dispositivo ou clique em algum link para outro aplicativo, a aplicação entrará no estado **dormant** e o evento **Deactivated** é disparado.

O estado **dormant** pode ser considerado um estado de pausa, ou seja, ele não está sendo executado, porém todo o estado atual permanece armazenado na memória. Dessa forma, caso o usuário pressione o botão voltar e retorne para seu aplicativo, ele é aberto rapidamente, como se nunca tivesse sido pausado.

Conforme o usuário navega entre diferentes aplicativos, mais aplicativos passam para o estado **dormant**, fazendo com que mais memória e recursos sejam consumidos. O Windows Phone irá fazer de tudo para que o aplicativo corrente receba todos os recursos necessários para que ele seja executado de forma fluída e isso inclui alterar seu estado para **tombstoned**.

O estado **tombstoned** pode ser considerado um estado final do aplicativo, entretanto a página na qual você estava, o estado da aplicação e o estado da navegação são armazenados. Caso seja aberto novamente estando nesse estado, é possível recuperar as informações do aplicativo na memória, passando a impressão ao usuário de que nunca foi fechado.

Ainda há um complicador a mais: o sistema pode manter apenas cinco aplicativos neste estado. Quando houver necessidade de memória e mais de cinco aplicativos estiverem no estado **tombstoned**, eles serão alterados para **closed** e nenhuma informação será mantida, a não ser que seja armazenada na memória permanente do dispositivo.

Para recuperar as informações, podemos criar um manipulador para o evento **Activated**, que é disparado quando o usuário retorna ao estado **running**, deixando de estar **dormant** ou **tombstoned**. Você deve ter notado que ao passar para o estado **tombstoned** nenhum evento é disparado, o que torna o evento **Deactivated** extremamente importante.

A figura 7.4 ilustra o fluxo descrito nos parágrafos anteriores.

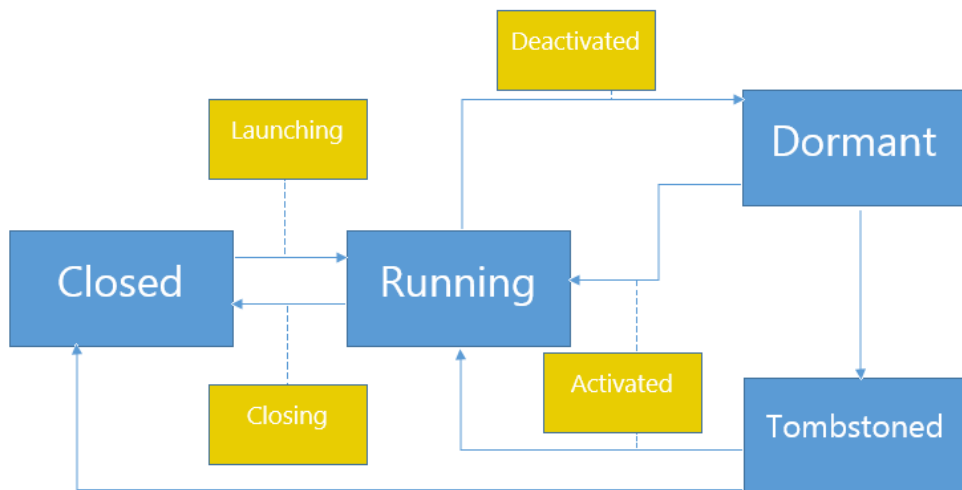


Figura 7.4: Ciclo de vida de um aplicativo

7.3 PAGESTATE

Neste ponto, você já aprendeu a teoria sobre o ciclo de vida do seu aplicativo. Agora está na hora de vermos o que podemos implementar para melhorar a experiência do usuário com o este novo conhecimento.

O **PageState** é um bom exemplo para utilizarmos estes conceitos. PageState refere-se ao estado atual da página, ou seja, as propriedades de cada campo. Quando uma aplicação entra no estado **dormant**, estas propriedades permanecem, mas quando ela entra no estado **tombstoned** perdemos estes valores, algo que pode causar frustração ao usuário final.

Vamos visualizar como isso funciona na página para criação de contas. Vá até ela em seu emulador, preencha os campos e pressione a tecla `pesquisar` do dispositivo. Depois disso, pressione o botão `voltar` e veja o que acontece.

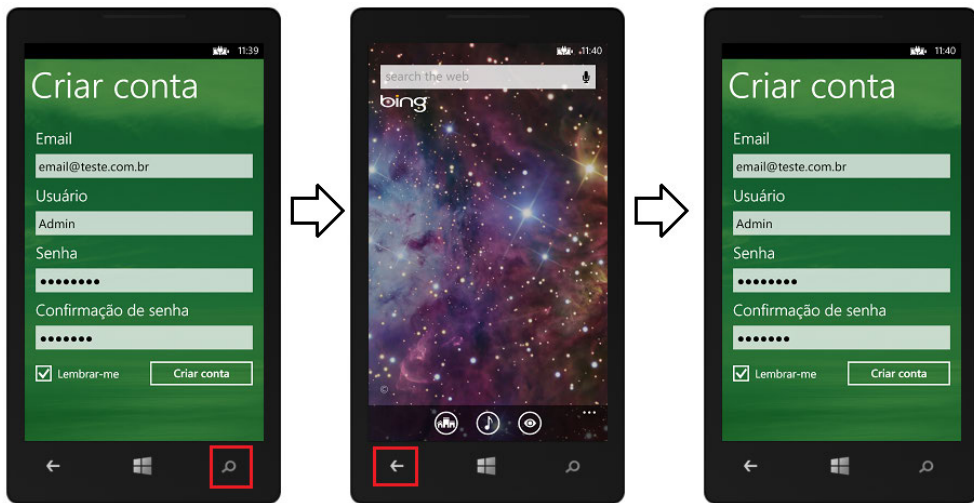


Figura 7.5: Aplicativo sendo restaurado do estado dormant

Como podemos ver na figura anterior, todo o estado da página continuava salvo, mesmo após sair do aplicativo. Isso aconteceu pois o aplicativo entrou apenas no modo **dormant**. Felizmente conseguiremos simular o aplicativo no modo **tombstoned** com a ajuda do Visual Studio.

Entre nas opções do seu projeto, aba `Debug`, e selecione a opção `Tombstone upon deactivation while debugging`, conforme ilustra a figura 7.6.

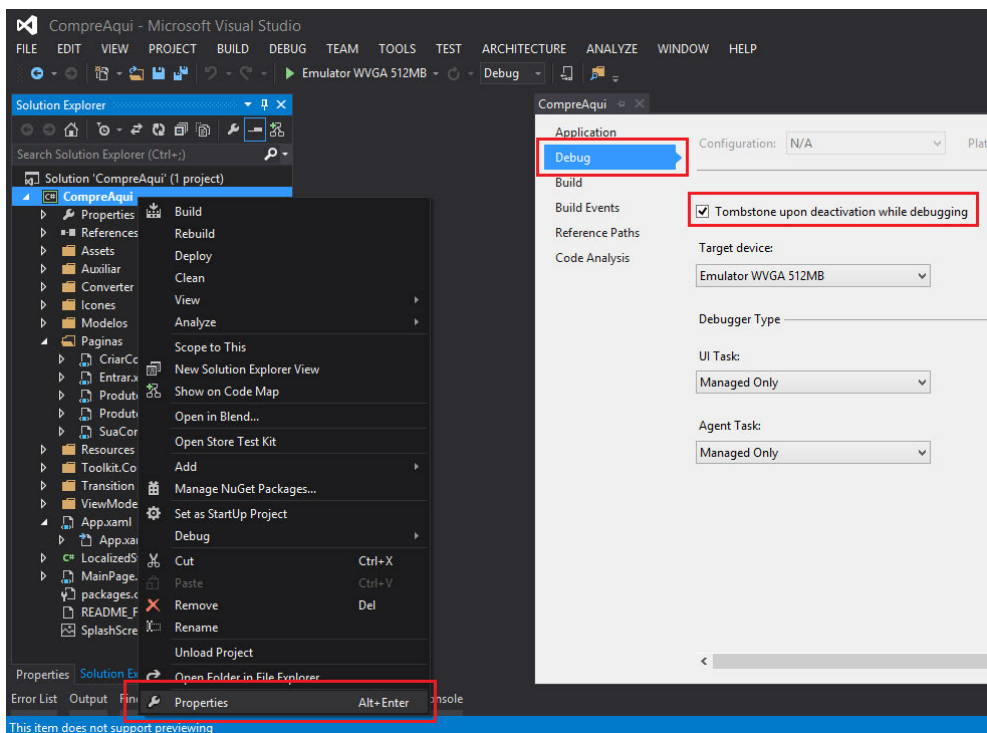


Figura 7.6: Colocando o modo tombostoned

Após fazer isso, refaça o teste anterior e você perceberá que, ao voltar, sua página ficará com os campos em branco novamente. Vamos resolver isso utilizando a propriedade `State`.

Anteriormente no livro nós já sobrescrevemos o método `OnNavigatedTo`, que ocorre quando o usuário é direcionada para esta página. Agora, além dele, também vamos sobrescrever o método `OnNavigatedFrom`, que é chamado quando o usuário sai da página, ou seja, vamos armazenar as informações quando o usuário sair e inseri-las novamente quando o usuário retornar para ela.

Primeiro, vamos implementar a saída do usuário desta página. Temos de nos certificar de que ele não está voltando para a página anterior, pois caso isso esteja sendo feito, a página está sendo fechada propositalmente, então não manteremos os dados. Tendo isto verificado, vamos armazenar as informações na propriedade `State`, conforme o código.

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
```

```

{
    if (e.NavigationMode != NavigationMode.Back)
    {
        State["email"] = _usuarioVM.Email;
        State["usuario"] = _usuarioVM.Nome;
        State["senha"] = _usuarioVM.Senha;
        State["confirmacao"] = _usuarioVM.ConfirmacaoSenha;
        State["lembrarme"] = _usuarioVM.EntrarAutomaticamente;
    }

    base.OnNavigatedFrom(e);
}

```

O código anterior obtém todos os dados através do *view model* da página. Da mesma forma, vamos preencher as informações do objeto quando o usuário voltar para a página. Anteriormente havíamos utilizado o evento `Loaded`, mas agora ele não será mais necessário, pois haverá esta implementação no método `OnNavigatedTo`.

```

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);

    if (_usuarioVM == null)
    {
        _usuarioVM = new UsuarioVM();

        if (State.ContainsKey("email"))
            _usuarioVM.Email = State["email"] as string;

        if (State.ContainsKey("usuario"))
            _usuarioVM.Nome = State["usuario"] as string;

        if (State.ContainsKey("senha"))
            _usuarioVM.Senha = State["senha"] as string;

        if (State.ContainsKey("confirmacao"))
            _usuarioVM.ConfirmacaoSenha = State["confirmacao"] as string;

        if (State.ContainsKey("lembrarme"))
            _usuarioVM.EntrarAutomaticamente =

```

```
        Convert.ToBoolean(State["lembrarme"]);
    }

    this.DataContext = _usuarioVM;
}
```

Agora, caso você refaça o teste proposto no começo desta seção, verá que os dados são recuperados assim que a página é reaberta, oferecendo uma sensação de que ela nunca havia sido fechada.

O código do aplicativo até este ponto pode ser encontrado em meu github através do link: <http://bit.ly/GITcap7ParteII>

7.4 APPLICATIONSTATE E INICIALIZAÇÃO RÁPIDA

Além do *PageState* que acabamos de conhecer também há o chamado *ApplicationState*. Este estado é algo que ultrapassa os limites de uma página, ou seja, um estado que deve permanecer durante toda a aplicação. Nós fizemos isso com os dados no capítulo anterior, mas agora vamos ver uma forma de melhorarmos o que já implementamos.

Nós utilizamos o evento `Application_Launching` para carregar os dados, o que pode se tornar um problema quando a conexão estiver lenta. Caso este evento demore tempo demais para executar, o sistema operacional irá fechar seu aplicativo e provavelmente ele não irá passar pelas baterias de testes antes de entrar oficialmente na loja para download.

Outro ponto interessante é que não precisamos carregar os dados do aplicativo até que o usuário precise deles, então vamos fazer com que os dados sejam carregados somente quando o usuário entrar na página `ProdutosHub.xaml`. Como já mostramos anteriormente, faremos isso através do método `OnNavigatedTo`.

Vamos criar um método chamado `CarregarDadosAsync` (na classe `Loja`), que irá fazer o que nós estávamos fazendo no evento de abertura da aplicação. Porém, já vamos prepará-lo para ser chamado de forma assíncrona, através das palavras chave `async` e `await`.

```
public async Task CarregarDadosAsync()
{
    string dados =
        LeitorArquivo.Ler("/CompreAqui;component/Resources/dados.txt");
}
```

```
Loja.Dados =  
    JsonConvert.DeserializeObject<Loja>(dados);  
  
await Task.Delay(3000);  
}
```

ASYNCHRONOUS AND AWAIT

No C#, podemos utilizar os comandos `async` e `await` para executar tarefas em paralelo. Esta é uma característica bastante singular da linguagem: ao criarmos um método que retorna uma `Task` como o código anterior, podemos defini-lo como método assíncrono através da palavra reservada `async`. Apesar de não ser regra, por convenção estes métodos possuem o sufixo **Async** em seu nome. Para entender melhor o funcionamento de processamento paralelo, acesse: <http://bit.ly/msdnAsyncAwait>

Note que no código anterior foi inserida uma espera de três segundos, apenas para que nosso carregamento fique um pouco mais lento e parecido com um cenário com maior quantidade de dados. Também vamos destruir o manipulador do evento `Loaded` da página, e todo o vínculo de dados feito nele será transferido para um método chamado `VincularDados`.

```
private void VincularDados()  
{  
    Categorias.ItemsSource = (from produtos in Loja.Dados.Produtos  
                              select new CategoriaVM  
                              {  
                                  Id = produtos.Categoria.Id,  
                                  Descricao = produtos.Categoria.Descricao  
                              }).Distinct().ToList();  
  
    Promocoes.ItemsSource = (from produtos in Loja.Dados.Produtos  
                              where produtos.PrecoPromocao != 0  
                              select new ProdutoVM  
                              {  
                                  Id = produtos.Id,  
                                  Descricao = produtos.Descricao,  
                              });  
}
```

```
        Preco = produtos.Preco,
        PrecoPromocao = produtos.PrecoPromocao,
        Icone = produtos.Icone
    })
    .OrderByDescending
    (produto => produto.Desconto)
    .ToList();

Produtos.ItemsSource = (from produtos in Loja.Dados.Produtos
    where produtos.Id == 3 || produtos.Id == 4
    select new ProdutoVM
    {
        Id = produtos.Id,
        Descricao = produtos.Descricao,
        Icone = produtos.Icone,
        Preco = produtos.Preco,
        PrecoPromocao = produtos.PrecoPromocao
    })
    .ToList();
}
```

Agora sim estamos pronto para sobrescrever o método `OnNavigatedTo`. Neste método, vamos verificar se os dados estão carregados. Caso não estejam, faremos com que eles sejam carregados de forma assíncrona, ou seja, em uma thread diferente da thread principal da aplicação. Dessa forma, não iremos bloquear nossa interface, evitando a impressão de que o aplicativo está travado.

Para fazer isso, vamos criar mais um método **Async**, chamado `CarregarDadosAsync`, mas desta vez dentro de nossa página `ProdutosHub.xaml`. Ele irá verificar se os dados já estão carregados e, caso não estejam, irá chamar o método assíncrono da classe `Loja` para carregá-los e então chamaremos o método `VincularDados` para fazer a ligação das informações. Veja o código:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    CarregarDadosAsync();
}

private async void CarregarDadosAsync()
```

```

{
    if (Loja.Dados.Produtos == null)
    {
        await Loja.Dados.CarregarDados();
        VincularDados();
    }
    else
    {
        VincularDados();
    }
}

```

Neste ponto você já poderá ver os dados sendo carregados quando abrir a página. Ainda não remova o delay inserido durante o carregamento, pois ele será útil para vermos a importância do armazenamento do estado da aplicação.

Se você já memorizou os eventos e estados em que a aplicação pode estar, você deve se lembrar de que o evento `Deactivated` é nosso último momento para armazenar o estado da aplicação. Não precisaremos implementar este método, mas caso em uma aplicação futura você precise, basta utilizar o objeto `PhoneApplicationService.Current.State`, que funciona da mesma maneira que o estado da página.

Ainda temos uma melhoria de usabilidade para fazer: enquanto os dados estiverem sendo carregados, o usuário não verá nenhuma mensagem avisando o motivo de os dados não estarem aparecendo. Vamos criar uma mensagem e uma barra de progresso em cada item do *Panorama* para notificar o usuário.

Faremos isso adicionando um `StackPanel` que irá conter um `TextBlock` e uma `ProgressBar`, que estarão invisíveis por padrão. Antes de executarmos o `await` para o carregamento dos dados, iremos torná-los visíveis e, após vincularmos os dados, faremos com que eles voltem a desaparecer.

Todos os itens do *Panorama* terão esta mesma estrutura antes da lista de seleção, conforme o código a seguir.

```

<phone:PanoramaItem Header="classificações">
    <ScrollView>
        <StackPanel Margin="10,0">

            <TextBlock x:Name="CampoMensagem1" Text="Carregando dados..."
                Visibility="Collapsed" TextWrapping="Wrap"
                Style="{StaticResource PhoneTextLargeStyle}" Margin="0,20"/>

```

```
<ProgressBar x:Name="BarraProgresso1" IsIndeterminate="True"
  Visibility="Collapsed" Foreground="Blue"/>

<Grid>
  <phone:LongListSelector x:Name="Categorias">
    <!--<Lista dos dados-->
  </phone:LongListSelector>
</Grid>
</StackPanel>
</ScrollView>

</phone:PanoramaItem>
```

Com isso, temos um resultado bastante agradável ao usuário, como ilustra a figura 7.7.



Figura 7.7: Carregando Dados

O código do aplicativo até este ponto se encontra em meu github através do link: <http://bit.ly/GITcap7ParteIII>.

CAPÍTULO 8

Integração com o sistema operacional

Neste capítulo vamos criar diversas formas de integrar nosso aplicativo com o sistema operacional para disponibilizar uma melhor experiência para o usuário. Mas antes de partirmos para este tópico, vamos finalizar as funcionalidades em aberto em nossa aplicação, isto é, vamos fazer uma nova página no aplicativo. Será uma página para detalhamento de produto, que deverá ser chamada quando o usuário selecionar um produto de qualquer lista. Além disso, faremos com que a pesquisa de produtos na página de hub passe a funcionar.

8.1 FINALIZANDO AS FUNCIONALIDADES JÁ CRIADAS

Vamos começar criando uma página para o detalhamento do produto. Ela deve conter todas as propriedades de um produto, e será utilizada sempre que o usuário selecionar um produto em uma das listas. É claro que criaremos apenas uma página e

buscaremos os seus dados de acordo com o parâmetro que recebermos ao abri-la.

Não entraremos em muitos detalhes, pois já fizemos todas estas implementações anteriormente. Vamos criar uma página que recebe como parâmetro o `Id` do produto e exibe suas propriedades, algo que você, neste ponto, já está apto a fazer. Tente fazê-la semelhante à ilustrada pela figura 8.1.



Figura 8.1: Detalhes de um produto

Também devemos criar uma página para que o usuário possa pesquisar os produtos. Nos capítulos anteriores, criamos um botão na *AppBar* no hub de produtos; agora criaremos a sua funcionalidade.

Neste caso, não é necessária uma página nova, já que temos a página que lista todos os produtos, ou mesmo os produtos por categoria. Vamos aproveitá-la também

para listar os resultados da pesquisa, então vamos prepará-la para receber mais um parâmetro, chamado `pesquisa`.

Este novo parâmetro deverá ser comparado com a descrição do produto. Para melhorar a experiência, faremos com que a pesquisa não seja sensível a maiúsculas, ou seja, vamos sempre comparar as informações em minúsculo (utilizando o método `ToLower`), tornando esta diferença irrelevante. Veja o código:

```
if (!string.IsNullOrEmpty(pesquisa))
    produtos = produtos.Where(produto =>
        produto.Descricao.ToLower()
            .Contains(pesquisa.ToLower()))
        .ToList();
```

Esta segunda parte será um pouco mais detalhada, pois ainda não fizemos nada parecido. Criaremos um painel com visibilidade dinâmica, ou seja, ela ficará invisível na maior parte do tempo, mas quando o usuário pressionar o botão `pesquisar` da barra da aplicação, ele se tornará visível e o usuário poderá digitar sua pesquisa em um campo.

A primeira coisa a fazer é circundar todo o componente de *Panorama* com um `Grid` dividido em duas linhas, uma para o *Panorama* já existente e outra para o painel de pesquisa. Vamos permitir que o *Panorama* ocupe as duas linhas, pois na maior parte do tempo o painel de pesquisa estará oculto. Isso é feito através da propriedade `Grid.RowSpan`.

Na segunda linha deste novo `Grid`, adicionaremos mais um `Grid` que ficará oculto por padrão. Neste componente, deve haver um `TextBox` que o usuário possa utilizar em suas pesquisas. Você também deve criar um manipulador para os eventos `KeyUp` e `LostFocus` deste `TextBox`, além, é claro, de criar um manipulador para o evento de `Click` do botão `pesquisar` da barra da aplicação.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="90" />
  </Grid.RowDefinitions>

  <phone:Panorama Title="CompreAqui" Grid.RowSpan="2">
    <!--<Componente Panorama já existente >-->
  </phone:Panorama>
```

```

<Grid Grid.Row="1" x:Name="PainelPesquisa"
      Background="#7F000000" Visibility="Collapsed">

    <TextBox Height="80" VerticalAlignment="Top" x:Name="CampoPesquisa"
      KeyUp="CampoPesquisa_KeyUp" LostFocus="CampoPesquisa_LostFocus" />

</Grid>

</Grid>

```

A parte visual desta página já está pronta. Agora precisamos implementar o código C#. Vamos começar criando dois métodos: `ExecutarPesquisa` e `DesaparecerPainelPesquisa`.

Ambos são bastante simples: no método `DesaparecerPainelPesquisa`, basta limparmos o campo de texto da pesquisa e voltarmos a visibilidade do painel para `Collapse`.

Já no método `ExecutarPesquisa`, devemos obter as informações do `TextBox`, direcionar o usuário para a página `Produtos.xaml` passando os parâmetros e, por fim, desaparecer com o painel de pesquisa utilizando o método citado anteriormente. Observe o código:

```

private void ExecutarPesquisa()
{
    string parametros =
        string.Format("?titulo={0}&pesquisa={1}",
            "resultados", CampoPesquisa.Text);

    NavigationService.Navigate(
        new Uri(string.Concat("/Paginas/Produtos.xaml", parametros),
            UriKind.Relative));

    DesaparecerPainelPesquisa();
}

private void DesaparecerPainelPesquisa()
{
    CampoPesquisa.Text = string.Empty;
    PainelPesquisa.Visibility = System.Windows.Visibility.Collapsed;
}

```

Com isso, já temos as duas principais funções para pesquisa implementadas;

agora devemos saber o momento de utilizá-las. Quando o usuário pressionar (evento `KeyUp`) a tecla `Enter` do teclado, devemos confirmar a pesquisa e, quando o usuário selecionar outro campo na página, fora o campo de texto da pesquisa (evento `LostFocus`), devemos fazer o painel desaparecer, conforme o código:

```
private void CampoPesquisa_LostFocus
(object sender, RoutedEventArgs e)
{
    DesaparecerPainelPesquisa();
}

private void CampoPesquisa_KeyUp
(object sender, System.Windows.Input.KeyEventArgs e)
{
    if (e.Key.Equals(Key.Enter))
    {
        ExecutarPesquisa();
    }
}
```

Agora, para finalizarmos a implementação nesta página, precisamos implementar o manipulador do evento `Click` do botão da barra da aplicação. Este método terá comportamentos distintos conforme a visibilidade do painel de pesquisa: caso ele esteja invisível, devemos mostrá-lo ao usuário; caso ele já esteja visível, devemos executar a pesquisa com as informações já inseridas no campo.

```
private void ApplicationBarIconButton_Click
(object sender, EventArgs e)
{
    if (PainelPesquisa.Visibility ==
        System.Windows.Visibility.Collapsed)
    {
        PainelPesquisa.Visibility = System.Windows.Visibility.Visible;
        CampoPesquisa.Focus();
    }
    else
    {
        ExecutarPesquisa();
    }
}
```

Agora já podemos fazer nossas pesquisas, como ilustra a figura 8.2.

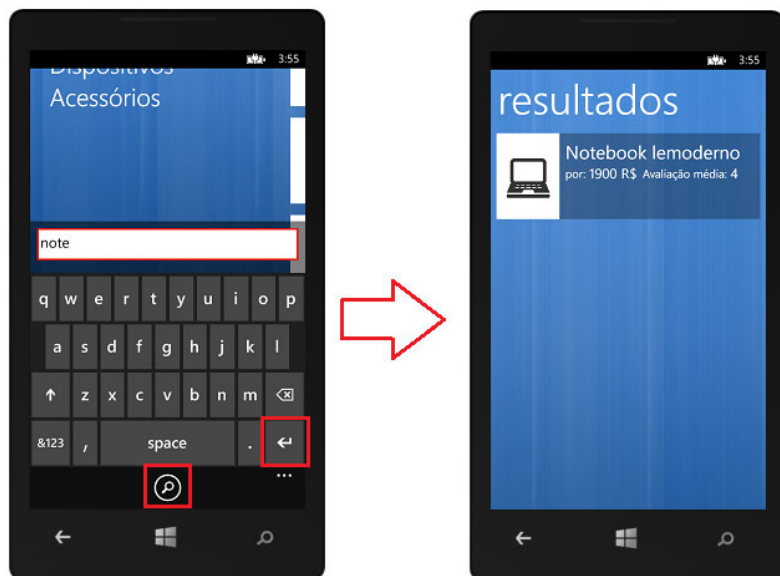


Figura 8.2: Pesquisando produtos

Mesmo com a funcionalidade pronta, ainda temos trabalho a fazer: quando o usuário executar uma pesquisa que não retorna nenhum dado, ele será enviado para esta página vazia. É sempre mais agradável ao usuário receber uma mensagem de que não foram encontrados dados com a pesquisa digitada — veja a diferença na figura 8.3.



Figura 8.3: Pesquisa sem dados

Esta será uma implementação muito simples, basta circundar a lista de produtos com um *StackPanel* e inserir o componente de texto com a mensagem. Este componente ficará invisível por padrão e, no momento de vincular os dados, basta fazer uma verificação. Caso não haja dados, tornamos o campo visível.

Agora que terminamos nossas implementações pendentes, voltaremos ao assunto principal deste capítulo. Caso você deseja obter ou verificar o código do projeto até este ponto, ele se encontra no meu github no link: <http://bit.ly/GITcap8ParteI>

8.2 ENTENDENDO OS LIVE TILES

Se você já está acostumado ou já conhece o Windows Phone, provavelmente já ouviu falar das *live tiles*. Tratam-se de retângulos que representam os ícones da aplicação de outros sistemas operacionais. Entretanto, os live tiles vão além do que só abrir o aplicativo: eles podem ser uma boa fonte de comunicação entre o aplicativo e o usuário, mesmo quando o usuário está com o aplicativo fechado.

Os live tiles podem receber atualizações de informações contínuas, mesmo com o aplicativo fechado. O usuário também pode escolher até três tamanhos diferentes para o tile. Como desenvolvedores, podemos escolher três diferentes tipos de tiles: **flip**, **iconic** e **cycle**, sobre os quais falaremos mais tarde.

Além disso, uma aplicação pode gerar mais de um tipo de live tile, ou seja, dentro da mesma aplicação você pode ter tiles secundários. Um bom exemplo de utilização deste recurso é o aplicativo pessoas, com o qual você pode gerar um tile diferente para cada contato do seu smartphone.

Flip é o template padrão das aplicações Windows Phone. Ele permite ao desenvolvedor configurar informações tanto na parte da frente quanto na parte de trás do tile. Isso ocorre porque, de tempos em tempos, o tile vai girar e mostrar outros tipos de informações. A figura 8.4 ilustra as propriedades que devem ser preenchidas para exibir informações neste tipo de template.

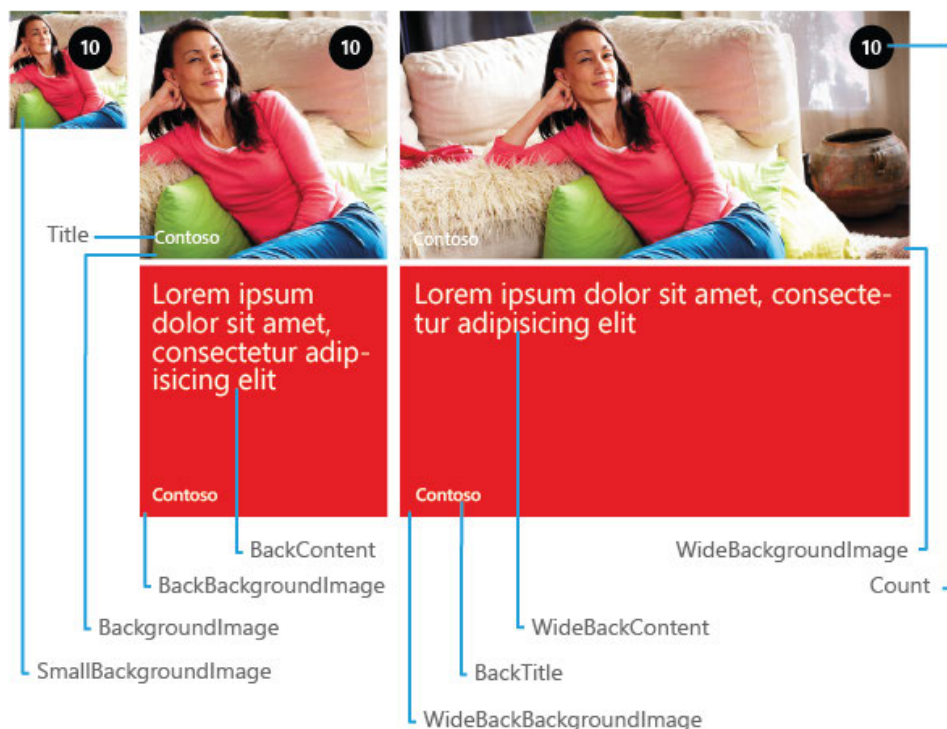


Figura 8.4: Tile utilizando o template Flip

Iconic é um template um pouco mais simples. Ele conta apenas com o ícone e um contador, quando está no tamanho pequeno. Serve para adicionar a propriedade **título** no tamanho médio e algumas linhas de texto no tamanho largo. A figura 8.5 ilustra as propriedades deste template.

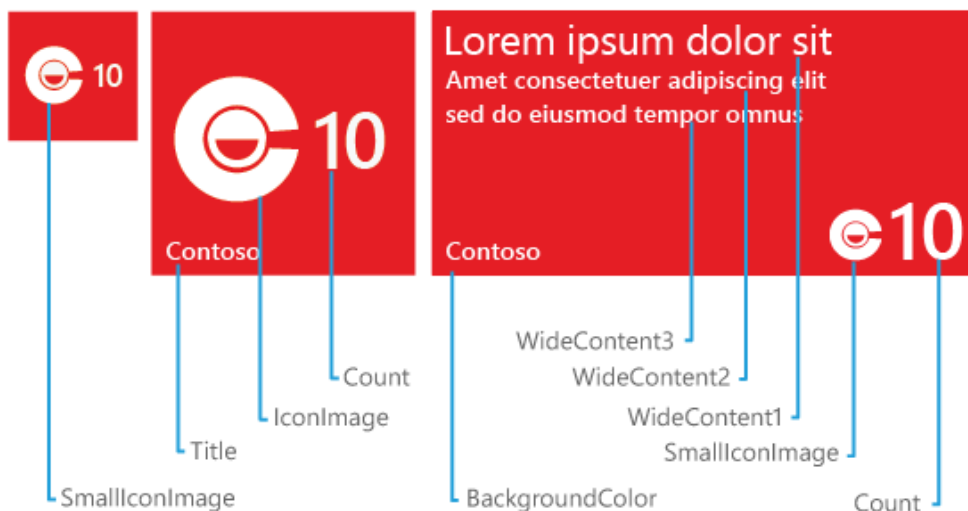


Figura 8.5: Tile utilizando o template Iconic

Cycle é um template com poucas informações textuais e bastante visual. Você pode criar um ciclo de até nove imagens diferentes nele e o sistema operacional tratará de alternar entre estas imagens regularmente. É um template bastante bonito visualmente, mas não se encaixa com uma gama muito alta de aplicativos. Nas figuras 8.6 e 8.7 você pode ver como o tile funciona e quais são suas propriedades, respectivamente.

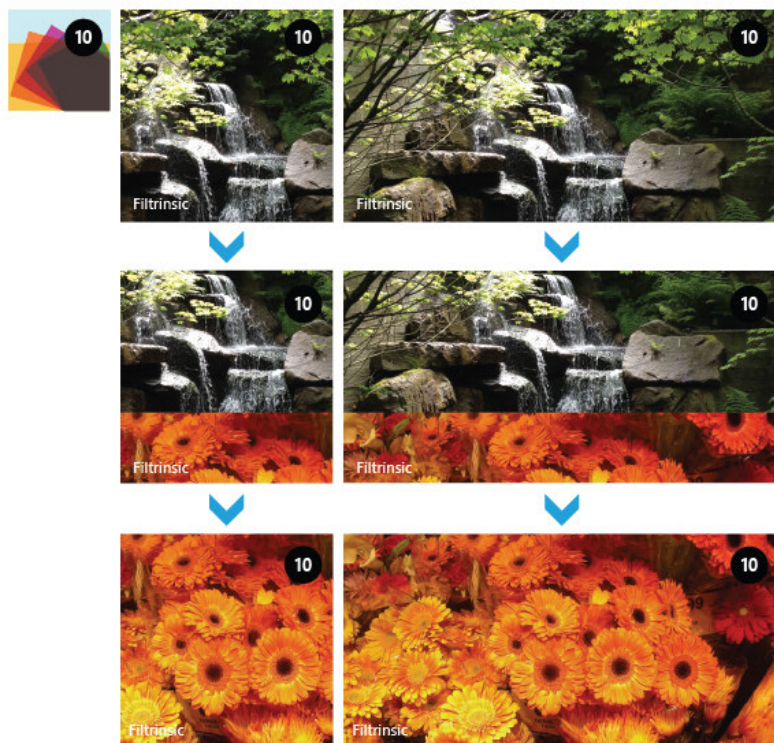


Figura 8.6: Modo de funcionamento do tile do tipo Cycle

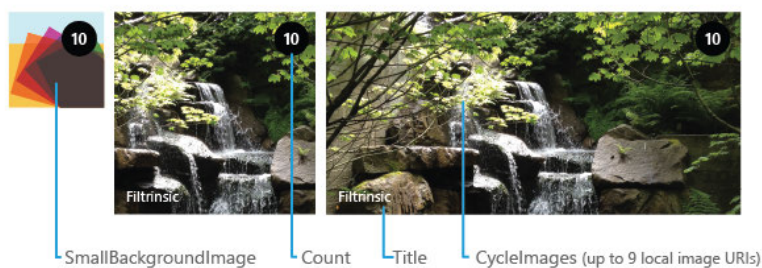


Figura 8.7: Tile utilizando o template Cycle

No exemplo do livro, vamos implementar o template **Iconic**, mas esteja à vontade caso queira testar as outras possibilidades. Isso é fácil de fazer, uma vez que você

conhece as propriedades referenciadas nas figuras anteriores.

Primeiro, vamos abrir o arquivo `WMAppManifest.xml` que já vimos anteriormente. Ele se encontra no caminho: `CompreAqui > Properties > WMAppManifest.xml`. Nesta janela, podemos ver diversas propriedades de nosso aplicativo, conforme a figura 8.8:

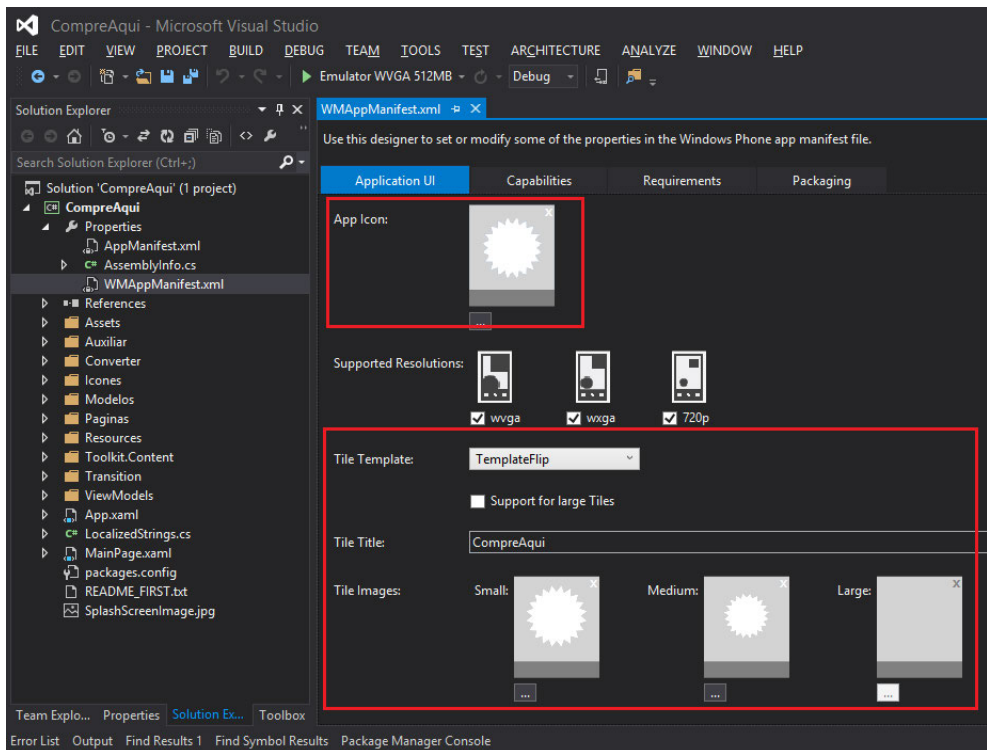


Figura 8.8: Arquivo Manifest

Toda a área destacada na figura 8.8 será alterada por nós. Primeiro, vamos escolher um novo ícone para nossa aplicação — é indicado utilizar uma imagem de tamanho 100x100 pixels para que ela não sofra nenhuma distorção —, então iremos aproveitar o ícone que já criamos para colocarmos na página de abertura e vamos diminuí-lo para 100x100.

Note também que, ao fazer com que o fundo da imagem seja transparente, o sistema faz com que o tile automaticamente receba a cor do tema do usuário.

Vamos selecionar o template **Iconic** e preencher os dois tamanhos de imagem

com nossos ícones. Depois, permitiremos que o tile fique no modo largo e colocaremos o nome da aplicação como descrição do tile.

Como já havia sido mencionado, se você desejar preencher as demais propriedades, alterar o ícone, ou fazer qualquer alteração no live tile principal, você terá de abrir o arquivo `manifest` através da opção `Open With...` (disponível ao pressionar o botão direito do mouse sobre o arquivo) e selecionar o editor de código XML. Você poderá ver as propriedades da seguinte forma:

```
<TemplateIconic>
  <SmallImageURI IsRelative="true" IsResource="false">
    Assets\Images\AppIcon.png
  </SmallImageURI>

  <Count>0</Count>
  <IconImageURI IsRelative="true" IsResource="false">
    Assets\Images\Icon.png
  </IconImageURI>

  <Title>
    CompreAqui
  </Title>

  <Message>
  </Message>
  <BackgroundColor>
  </BackgroundColor>
  <HasLarge>False</HasLarge>
  <LargeContent1>
  </LargeContent1>
  <LargeContent2>
  </LargeContent2>
  <LargeContent3>
  </LargeContent3>
  <DeviceLockImageURI IsRelative="true" IsResource="false">
  </DeviceLockImageURI>
</TemplateIconic>
```

Muito embora não iremos utilizar esta técnica, é importante que você conheça uma forma de atualizar os dados de seu aplicativo. Portanto, vamos criar um exemplo, apenas para fins de aprendizagem.

Vamos fazer com que, quando o usuário criar uma nova conta no aplicativo, ele possa ver o seu nome de usuário e senha no tile. Então, vamos voltar à página `CriarConta.xaml` e adicionar um novo método, chamado `AtualizarLiveTile`. Este método receberá o nome e o e-mail do usuário por parâmetro e fará suas atualizações.

Há um objeto chamado `ShellTile`, que possui uma propriedade estática nomeada `ActiveTiles`. Ela retorna todos os tiles ativos da aplicação, e sempre o primeiro tile desta coleção será o tile primário do aplicativo. Vamos selecionar este tile desta coleção e utilizar o método `Update` para atualizar as informações.

O método `Update` recebe um parâmetro do tipo `ShellTileData`, que na verdade é uma classe abstrata que representa a base de qualquer template de tiles. Você deve preencher um objeto `FlipTileData`, `CycleTileData` ou `IconicTileData` (em nosso caso) e passá-lo por parâmetro neste método, conforme o código:

```
private void AtualizarLiveTile(string nomeUsuario, string email)
{
    ShellTile appTile = ShellTile.ActiveTiles.FirstOrDefault();
    if (appTile != null)
    {
        IconicTileData dadosTile = new IconicTileData();
        dadosTile.WideContent1 = nomeUsuario;
        dadosTile.WideContent2 = email;

        appTile.Update(dadosTile);
    }
}
```

Agora colocaremos uma chamada a este método no método `GravarUsuário` após enviar o usuário para a página `ProdutosHub`. A figura 8.9 mostra a diferença entre o tile da aplicação antes e depois desta atualização.



Figura 8.9: Atualização do live tile

É importante ter em mente que esta atualização não seria bem-vinda em um aplicativo real, pois devem ser exibidas apenas informações relevantes ao usuário. Lembramos que esse exemplo foi criado apenas para aprendizagem.

8.3 CRIANDO TILES SECUNDÁRIOS

Já aprendemos a criar o tile principal de nosso aplicativo e a fazer atualizações neste tile. Agora vamos ver como criar tiles secundários, onde poderemos direcionar o usuário para uma página específica em nosso aplicativo.

Nós criaremos tiles secundários para as páginas `Produtos.xaml` e `ProdutoDetalhe.xaml`. Na página de listagem de produtos, o usuário poderá armazenar um atalho para verificar os produtos de uma categoria ou de um resultado de pesquisa. Já na página de detalhamento de produto, o usuário poderá criar um atalho para um único produto, para acompanhar, por exemplo, o momento em que ele entra em uma promoção.

Vamos começar com a página `Produtos.xaml`. Primeiro, habilitamos a `AppBar` desta página e inserimos um botão para fixar na tela inicial. Você também deve criar um manipulador para o evento `Click` deste botão.

Nós já utilizamos anteriormente o objeto `ShellTile`, cuja função é gerenciar o tile principal e também os tiles secundários de sua aplicação. Vamos criar o método `CriarDadosTile` no código desta página, que deve retornar um objeto do tipo `StandardTileData`.

Este é o tipo utilizado por tiles secundários; apesar de ser um objeto diferente, visualmente ele se comportará da mesma maneira que o template do tipo `Flip`. Então, podemos preencher propriedades que ficarão na parte da frente do tile e propriedades que ficarão na parte de trás, que serão mostradas quando o sistema operacional executar um `Flip` no tile.

Vamos preencher este tile com um contador para representar a quantidade de produtos desta lista, e seu título será o mesmo que o título desta página. Na parte de trás, preencheremos com informações a respeito do primeiro produto da lista, conforme o código:

```
private StandardTileData CriarDadosTile()
{
    StandardTileData data = new StandardTileData();
    data.Title = Titulo.Text;
    data.BackgroundImage =
        new Uri("/Assets/Images/tileBackground.png", UriKind.Relative);

    if (Listagem.ItemsSource.Count > 0)
    {
        List<ProdutoVM> produtos = (List<ProdutoVM>)Listagem.ItemsSource;
        ProdutoVM produto = produtos.FirstOrDefault();

        data.Count = produtos.Count;
        data.BackTitle = produto.Descricao;
        data.BackBackgroundImage =
            new Uri("/Assets/Images/tileBackground.png", UriKind.Relative);

        if (NavigationContext.QueryString.ContainsKey("pesquisa"))
            data.BackContent =
                string.Concat( "Pesquisa: ",
                               NavigationContext.QueryString["pesquisa"]);
        else
```

```

        data.BackContent = Titulo.Text;
    }

    return data;
}

```

Agora vamos completar o manipulador do evento de `Click`. Este método terá de montar a `Uri` desta página (incluindo seus parâmetros) e verificar se o tile já existe; caso ele ainda não exista, você deve utilizar o método `Create` do objeto `ShellTile`. Veja o código:

```

private void Fixar_Click(object sender, EventArgs e)
{
    StringBuilder parametros = null;

    foreach (string parametro in NavigationContext.QueryString.Keys)
    {
        if (parametros == null)
            parametros = new StringBuilder("?");
        else
            parametros.Append("&");

        parametros
            .AppendFormat("{0}={1}", parametro,
                NavigationContext.QueryString[parametro]);
    }

    if (parametros == null)
        url = "/Paginas/Produtos.xaml";
    else
        url =
            string.Concat("/Paginas/Produtos.xaml", parametros.ToString());

    if (ShellTile.ActiveTiles
        .Any(tiles =>
            tiles.NavigationUri.ToString() == url))
    {
        string mensagem =
            "Este atalho já está fixado em sua tela inicial";

        MessageBox.Show(string.Concat(

```



```
        "Não foi possível fixar o tile por um ou mais motivos abaixo:",  
        Environment.NewLine, mensagem));  
    }  
    else  
    {  
        ShellTile.Create( new Uri(url, UriKind.Relative),  
                          CriarDadosTile());  
    }  
}
```

Você já poderá fixar seus tiles secundários através da página de listagem de produtos através do botão da `AppBar`, conforme ilustra a figura 8.10.

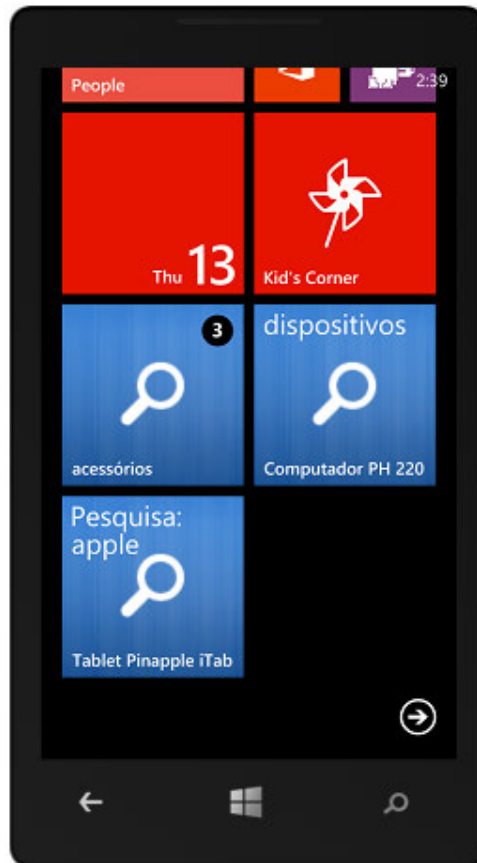


Figura 8.10: Arquivo Manifest

Apesar de a funcionalidade de criar um tile secundário ter sido feita com sucesso, ainda nos resta resolver um problema. A `Uri` informada na criação do tile secundário define a página à qual o usuário será redirecionado quando ele pressionar o tile, o que pode causar problemas quanto à questão de navegabilidade dentro da aplicação.

Agora, caso o usuário entre no aplicativo através de uma tile secundária, ele não irá conseguir acessar o resto da aplicação. Isso pode ser feito adicionando um item de menu para levar o usuário até a página inicial.

Este item de menu deve ser visível apenas quando a aplicação for aberta através do tile secundário. Vamos adicioná-lo e fazer com que o usuário seja direcionado para a página `MainPage.xaml`, lembrando que, se ele já estiver autenticado, ele será redirecionado para a página `ProdutosHub.xaml`.

Não é necessário me prolongar explicando como fazer este mesmo procedimento na página de detalhamento de produto, pois o procedimento será o mesmo. Apenas altere os dados do tile que você julgar importante e tente implementar isso sozinho. Se precisar, volte algumas páginas e releia o modo de criação. Lembre-se: programar se aprende praticando!

Caso você tenha mais experiência em programação, sugiro que todas estas funções repetidas sejam implementadas em um objeto. Neste livro, elas são implementadas separadamente para fins didáticos, mas em uma implementação real, o reaproveitamento do código é bastante importante.

Depois de já ter implementado essa funcionalidade, você pode conferir o código que se encontra em meu github através do link: <http://bit.ly/GITcap8ParteII>.

8.4 LAUNCHERS E CHOOSERS

Launchers e **choosers** são elementos muito importantes no desenvolvimento para a plataforma Windows Phone. Através deles, é possível reutilizar funções que já existem no dispositivo em outros aplicativos. Por exemplo, caso seja necessário enviarmos um e-mail através de nosso aplicativo, não precisaríamos implementar esta funcionalidade no próprio aplicativo, pois já existem aplicativos que tratam este problema.

Basicamente, reaproveitamos funcionalidades que outros aplicativos já fazem para que nosso aplicativo se preocupe somente em resolver o problema que ele se propõe a resolver. Dessa forma, não retiramos o foco do desenvolvimento.

Como já vimos no capítulo anterior, sobre navegação, sempre que uma nova aplicação é aberta, o estado da nossa aplicação é alterada. Portanto, tenha ciência de

que seu aplicativo pode entrar no estado **tombstoned** (apesar de não ser comum) e você precisará restaurar seu estado.

Os comportamentos supracitados são comuns para launchers e choosers, mas eles têm uma diferença bastante significativa entre si: uma aplicação executada através de um launcher não pode retornar informações para seu aplicativo, somente as tarefas disparadas através de um chooser poderão retornar dados.

Você pode acessar a lista com todos os launchers neste link: <http://bit.ly/msdnLaunchers> e a lista com todos os choosers neste link: <http://bit.ly/msdnChoosers>.

Para termos a experiência de criar um launcher, vamos explorar um exemplo bastante simples. Na página de detalhamento do produto, vamos inserir uma opção na `AppBar` para compartilharmos o produto em redes sociais. Você verá que isso pode ser feito de uma maneira incrivelmente simples.

Primeiro, vamos alterar a página para inserir o botão para compartilhar, conforme o código:

```
<shell:AppBarIconButton
    Text="Compartilhar"
    IconUri="/Assets/AppBar/share.png"
/>
```

No código C# da página, teremos de adicionar a diretiva de uso do namespace `Microsoft.Phone.Tasks`. Este namespace contém os objetos necessários para instanciarmos tanto os launchers quanto os choosers.

Para ativarmos o compartilhamento em redes sociais, teremos de utilizar o launcher `ShareStatusTask`. Ele possui a propriedade `Status`, que é o texto que será publicado na rede social que o usuário escolher. Veja o código a seguir.

```
private void Compartilhar_Click(object sender, EventArgs e)
{
    ProdutoVM dataContext = DataContext as ProdutoVM;
    ShareStatusTask launcherCompartilhar = new ShareStatusTask();
    launcherCompartilhar.Status =

        string.Concat( "Confiram o produto ", dataContext.Descricao,
                       " no aplicativo CompreAqui, está custando apenas ",
                       dataContext.PrecoAPagar, " R$." );

    launcherCompartilhar.Show();
}
```

Com este código-fonte, você já estará apto a compartilhar nas redes sociais às quais seu smartphone está conectado, como por exemplo, Facebook, Twitter e LinkedIn. Infelizmente, você precisará de um dispositivo físico para testar esta funcionalidade, já que o emulador não possui uma conta Microsoft vinculada e, por sua vez, não haverá autenticação em nenhuma rede social. A figura 8.11 ilustra essa funcionalidade em um dispositivo real.



Figura 8.11: Compartilhamento de status

O procedimento para criar um chooser é bastante parecido — a única diferença, além dos parâmetros, é que você precisará criar um manipulador para o evento `Completed`, que é disparado após o usuário obter as informações que estava buscando no outro aplicativo.

Você pode conferir o código que se encontra em meu github através do link: <http://bit.ly/GITcap8ParteIII>.

CAPÍTULO 9

Mapas e localização

Até este momento já criamos diversas funcionalidades em nosso aplicativo, como consulta de produtos, a criação de contas de usuário, compartilhamento informações em redes sociais etc. Entretanto, você deve ter percebido que, apesar de se tratar de um aplicativo de compra, ainda não fizemos a funcionalidade para o usuário poder efetuar sua compra.

9.1 CRIANDO A PÁGINA PARA FINALIZAR COMPRA

Vamos voltar a nossa página `ProdutoDetalhe.xaml`. Nela, o usuário consegue verificar as informações detalhadas de um determinado produto, bem como compartilhar informações e criar um live tile para entrar no aplicativo direto na página deste produto.

Agora vamos adicionar um último botão na `AppBar` desta página, que será utilizado para iniciar o processo de compra. Como o foco de nossa aplicação é meramente didático, vamos limitar a compra para apenas um produto, eliminando toda a criação de funcionalidades para gerenciar um carrinho de compras.

Quando o usuário pressionar este botão, ele será direcionado para a página onde irá informar o endereço para entrega da mercadoria, e já finalizará a compra. Claro que, antes de direcionarmos o usuário para esta página, precisamos verificar se ele está autenticado no aplicativo, pois usuários anônimos não podem realizar compras.

Já implementamos este tipo de validação antes: basta verificarmos se há a configuração `usuarioId` armazenada e se o `Id` é válido. Dependendo da resolução desta condição, vamos direcionar o usuário para a página de finalizar compras ou para a página de efetuar login, conforme o código.

```
private void Comprar_Click(object sender, EventArgs e)
{
    IsolatedStorageSettings configuracoes =
        IsolatedStorageSettings.ApplicationSettings;

    if ( configuracoes.Contains("usuarioId") &&
        Convert.ToInt32(configuracoes["usuarioId"]) != 0)
    {
        NavigationService.Navigate(
            new Uri("/Paginas/FinalizarCompra.xaml", UriKind.Relative));
    }
    else
    {
        MessageBox.Show("Ops, desculpe, mas você não pode efetuar uma
            compra sem estar autenticado no aplicativo.");

        NavigationService.Navigate(
            new Uri("/Paginas/Entrar.xaml", UriKind.Relative));
    }
}
```

Perceba que no trecho de código anterior nós estamos direcionando o usuário para a página `FinalizarCompra.xaml`. Como esta página ainda não existe, caso você tente executar esta ação será lançada uma exceção em seu aplicativo.

Agora vamos criá-la no mesmo diretório das demais. Além disso, você já deve executar algumas das tarefas que já aprendeu, como, por exemplo, preparar a página para abrir de forma animada e alterar o `Background` para manter a consistência da interface do aplicativo.

Esta página deve possuir os campos: logradouro e cidade, para que o usuário possa informar o endereço de entrega e, para fins didáticos, adicionaremos um mapa

que mostrará a posição atual do usuário e o endereço que ele digitar. Isso será feito utilizando o componente `Map`.

Como já foi explicado no começo livro, é necessário informar no arquivo `WMAppManifest.xml` todas as permissões para que seu aplicativo possa utilizar funcionalidades do sistema operacional. Para podermos utilizar o mapa e o gerenciamento de localização, precisamos ativar as **Capabilities**: `ID_CAP_MAP` e `ID_CAP_LOCATION`, ilustradas pela figura 9.1.

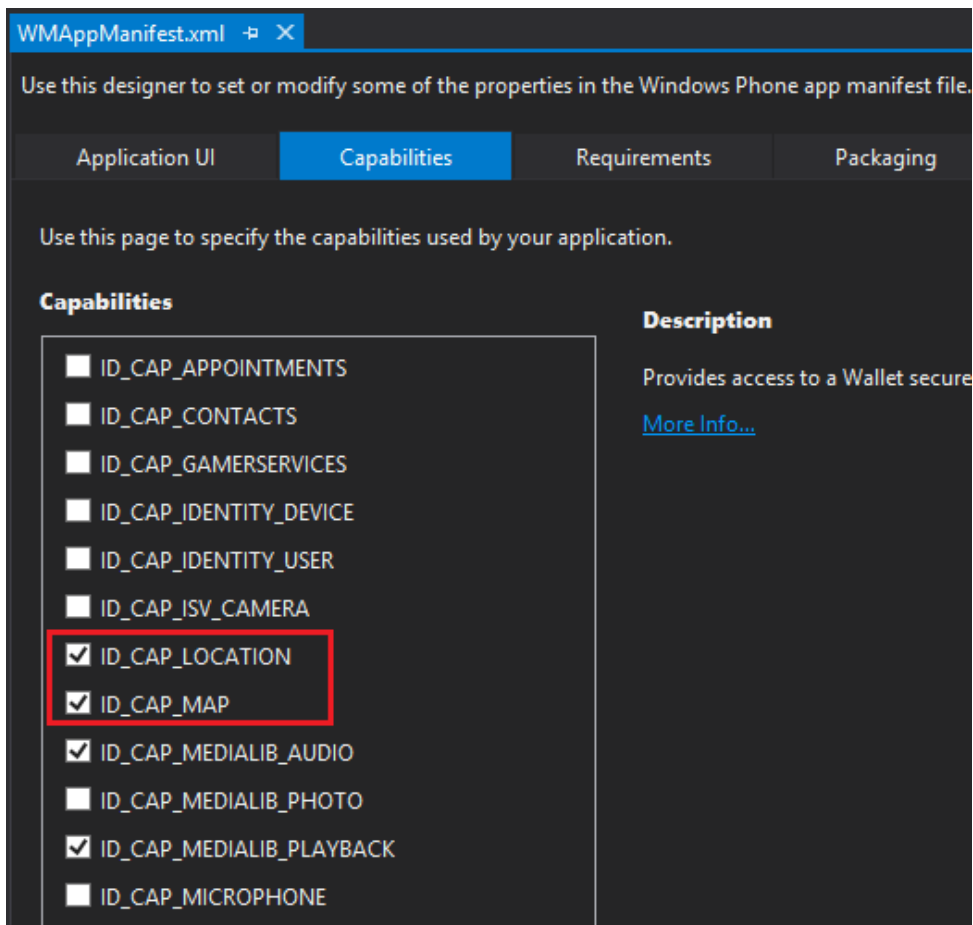


Figura 9.1: Capabilities para localização

Caso você não habilite estes capabilities, sua aplicação irá disparar uma exceção do tipo `Exception` com a mensagem indicando falta de permissão para utilizar o

recurso.

Com elas habilitadas, podemos voltar à nossa página e começar a criar os componentes. Vamos projetar o layout baseado em um `StackPanel`, envolvendo todos os componentes do conteúdo da página. Não se esqueça de circundar este painel com um `ScrollViewer` para evitar problemas de navegação.

A primeira parte da criação será bastante simples e similar ao que já fizemos até agora: vamos empilhar os campos para o usuário digitar o logradouro e a cidade e, depois destes campos, teremos o botão `Adicionar Endereço`, conforme o trecho a seguir.

```
<StackPanel>
    <TextBlock Style="{StaticResource PhoneTextLargeStyle}">
        Logradouro
    </TextBlock>

    <TextBox x:Name="Logradouro"/>

    <TextBlock Style="{StaticResource PhoneTextLargeStyle}">
        Cidade
    </TextBlock>

    <TextBox x:Name="Cidade"/>
    <Button>Adicionar Endereço</Button>

    ...
</StackPanel>
```

Com isso, os primeiros componentes da página já foram inseridos. Agora utilizaremos alguns componentes um pouco diferentes do que fizemos até agora, mas não será nada complicado. A próxima coisa a fazer é incluir o namespace para mapas, da mesma forma que fizemos para incluir o namespace do toolkit.

```
xmlns:maps="clr-namespace:Microsoft.Phone.Maps.Controls;
            assembly=Microsoft.Phone.Maps"
```

Feito isso, podemos utilizar o componente `Map` que foi citado anteriormente. Vamos incluir este mapa dentro de um painel do tipo `Grid`, que será dividido em duas colunas — mas nosso mapa ocupará todo o espaço disponível, conforme o código:

```
<Grid Margin="10">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="*/>
  </Grid.ColumnDefinitions>

  <maps:Map x:Name="Mapa"
    Height="400" Grid.ColumnSpan="2"/>

</Grid>
```

Apesar de o componente `Map` já possuir um recurso de zoom através do gesto de pinça, criaremos botões para facilitar o aumento e a diminuição do zoom, adicionando um `StackPanel` na primeira coluna do `Grid` recém-criado.

```
<Grid Margin="10">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="*/>
  </Grid.ColumnDefinitions>

  <maps:Map x:Name="Mapa"
    Height="400" Grid.ColumnSpan="2"/>

  <StackPanel>
    <Button Height="90" FontSize="32"> + </Button>
    <Button Height="90" FontSize="32"> - </Button>
  </StackPanel>

</Grid>
```

Para completar nossa página, vamos criar uma legenda para este mapa. No começo desta seção, foi dito que o mapa deveria mostrar ao usuário sua posição atual e a posição do endereço digitado por ele. Ambas as posições serão marcadas no mapa, cada uma com uma cor diferente: usaremos vermelho para posição do usuário e azul para endereço digitado.

Vamos identificar estas cores na legenda com o componente `Rectangle`, seguido de um texto simples que indique o que a cor representa. Por fim, teremos o botão para finalizar a compra.

```
<StackPanel>
  ...
```

```
<Grid Margin="10">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="*/>
  </Grid.ColumnDefinitions>

  <maps:Map x:Name="Mapa" Height="400"
    Grid.ColumnSpan="2"/>

  <StackPanel>
    <Button Height="90" FontSize="32"> + </Button>
    <Button Height="90" FontSize="32"> - </Button>
  </StackPanel>
</Grid>

<StackPanel Margin="10,0" Height="70" Orientation="Horizontal">
  <Rectangle Fill="Red" Width="50" Height="50"/>

  <TextBlock Style="{StaticResource PhoneTextLargeStyle}"
    VerticalAlignment="Center">
    Você
  </TextBlock>

  <Rectangle Fill="Blue" Margin="40,0,0,0" Width="50" Height="50"/>

  <TextBlock Style="{StaticResource PhoneTextLargeStyle}"
    VerticalAlignment="Center">
    Endereço
  </TextBlock>
</StackPanel>

<Button> Finalizar Compra </Button>
</StackPanel>
```

Com este layout definido, sua página deve estar bastante similar à ilustrada na figura 9.2.



Figura 9.2: Página para finalizar compra

9.2 ENTENDENDO O COMPONENTE MAP

Antes de seguirmos com a implementação de nossa página, é interessante que você conheça algumas das funcionalidades e comportamentos do componente `Map`.

A primeira mudança que podemos aplicar ao nosso mapa é a alteração de cor. Podemos escolher os valores `Dark` e `Light` (padrão) na propriedade `ColorMode`;

esta diferença pode ser notada através da figura 9.3, que exibe o modo `Light` e `Dark` respectivamente.



Figura 9.3: Mapas Light e Dark

Além do modo de cores, também podemos alterar o modo cartográfico do mapa, isto é, alterar as informações que ele exibe ao usuário. Neste caso, há quatro opções: `Road` (padrão), `Aerial`, `Hybrid` e `Terrain`. É importante ressaltar que as cores são aplicadas somente em mapas com o modo cartográfico `Road`. A seguir, a figura 9.4 ilustra os modos cartográficos citados respectivamente.

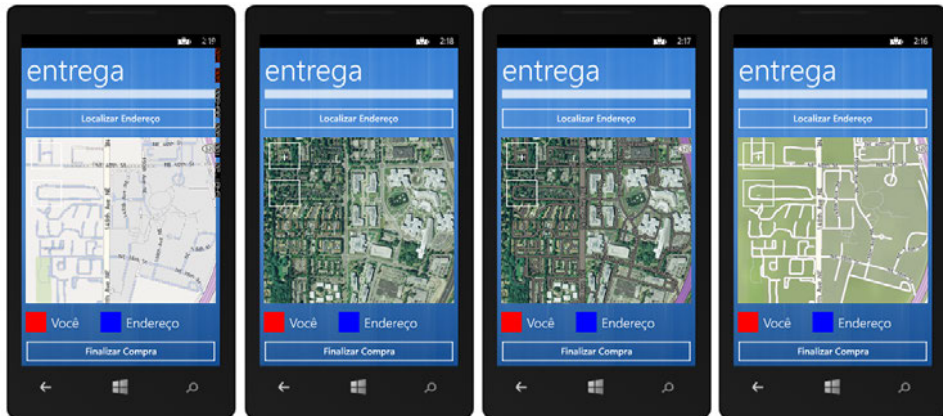


Figura 9.4: Mapas Road, Aerial, Hybrid e Terrain

Além destas opções também é possível alterar as propriedades `LandmarksEnabled` e `PedestrianFeaturesEnabled`, que exibem mais informações no mapa. A primeira propriedade faz com que o mapa exiba modelos 3D de lugares importantes; já a segunda habilita funcionalidades para pedestres, exibindo por exemplo, escadarias ou faixas de segurança para atravessar uma rua.

Além disso, é possível adicionar elementos no mapa via programação, que é o que faremos a seguir.

9.3 FINALIZANDO NOSSO APLICATIVO

Como citado anteriormente, é possível adicionar elementos no mapa em tempo de execução, devido ao conceito de camadas que existe no componente `Map`.

A visualização padrão do componente `Map` pode ser considerada a camada mais inferior de seu mapa (azul na figura 9.5) e programaticamente você pode criar uma nova camada de visualização e adicionar seus objetos nesta camada (amarela na figura 9.5). Após isso, você pode adicionar a sua camada no componente e ele passará a exibir a combinação de todas as camadas (verde na figura 9.5)

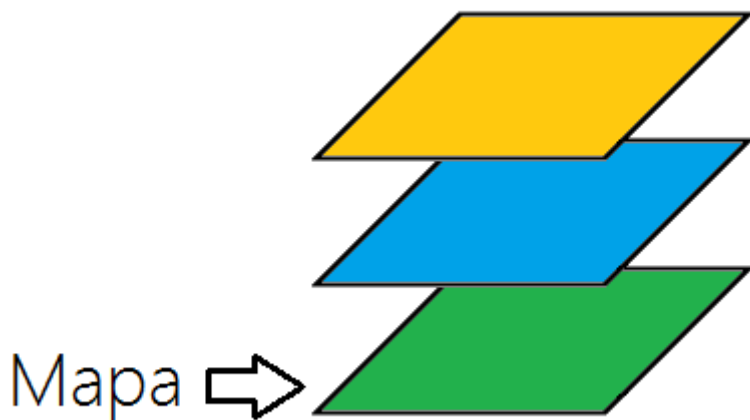


Figura 9.5: Camadas do mapa

Agora criaremos o método em nossa página `FinalizarCompra.xaml.cs` para marcar uma posição no mapa. Este método deverá receber por parâmetro três valores: latitude e longitude, para saber a posição na qual o objeto deve ser marcado, e a cor que o objeto terá. Veja o código:

```
private void MarcarPosicaoNoMapa
(double latitude, double longitude, Color cor)
{
    Ellipse marcacao = new Ellipse();
    marcacao.Fill = new SolidColorBrush(cor);
    marcacao.Height = 10;
    marcacao.Width = 10;

    MapLayer camada = new MapLayer();

    MapOverlay sobrecamada = new MapOverlay();
    sobrecamada.Content = marcacao;
    sobrecamada.GeoCoordinate =
        new GeoCoordinate(latitude, longitude);

    camada.Add(sobrecamada);
}
```



```
Mapa.Layers.Add(camada);  
}
```

Veja nesse código que o conceito de camadas é bastante explícito: primeiro criamos a nova camada (`MapLayer`), depois criamos um conteúdo para esta camada (`MapOverlay`), e por último adicionamos o conteúdo sobre a camada e a camada sobre o mapa.

Agora criaremos o método para alterar o zoom do mapa. Ele será utilizado no evento `Tap` dos botões “+” e “-”. Este método é bem simples: basta acumular um valor ao zoom atual do mapa e validar se o valor acumulado não ultrapassa os limites de zoom do mapa (1-20), conforme o código:

```
private void AlterarAcumulativoZoom(double zoomLevel)  
{  
    double zoomLevelLimite = Mapa.ZoomLevel + zoomLevel;  
  
    if (zoomLevel > 0)  
        zoomLevelLimite = Math.Min(20, zoomLevelLimite);  
    else  
        zoomLevelLimite = Math.Max(1, zoomLevelLimite);  
  
    Mapa.ZoomLevel = zoomLevelLimite;  
}
```

Note que sempre estamos somando o valor ao zoom atual e isso funcionará tanto para o aumento quanto para a diminuição do zoom — para fazer isso, basta informar um número negativo na soma. Agora vamos fazer com que os botões de auxílio do mapa funcionem. Para isso, você deve criar um método manipulador do evento `Tap` para cada um deles e fazer uma chamada a este método informando o aumento ou a diminuição que será aplicada.

```
private void AumentarZoom_Tap(object sender, RoutedEventArgs e)  
{  
    AlterarAcumulativoZoom(3);  
}  
  
private void DiminuirZoom_Tap(object sender, RoutedEventArgs e)  
{  
    AlterarAcumulativoZoom(-3);  
}
```

Na legenda de nosso mapa, temos indicativos para duas marcações: localização atual do usuário e endereço digitado. Primeiro vamos implementar a função para obter a localização do usuário, utilizando um objeto do tipo `Geolocator`. Este objeto nos permite buscar a posição do usuário em coordenadas, ou seja, latitude e longitude.

A precisão desta busca varia de acordo com a propriedade `DesiredAccuracy`. Ele poderá buscar a posição do usuário através de métodos diferentes, como IP ou triangulação de antenas. Para nosso exemplo, vamos deixar o valor `Default`.

O método para obtermos a localização se chama `GetGeopositionAsync`. Como já foi dito, por convenção os métodos assíncronos utilizam o sufixo `Async`, logo este é um método assíncrono. Depois de obter a posição geográfica, vamos utilizar o método para criar uma marcação vermelha no mapa — mas não se esqueça de circundar tudo isso com uma cláusula *try-catch*, pois o usuário pode estar com o recurso de localização desligado em seu dispositivo. O código a seguir ilustra este método:

```
private async void ObterPosicaoAtual()
{
    Geolocator localizador = new Geolocator();
    localizador.DesiredAccuracy = PositionAccuracy.Default;

    try
    {
        Geoposition posicaoAtual =
            await localizador.GetGeopositionAsync();

        MarcarPosicaoNoMapa(posicaoAtual.Coordinate.Latitude,
                           posicaoAtual.Coordinate.Longitude,
                           Colors.Red);

        Mapa.Center =
            new GeoCoordinate(posicaoAtual.Coordinate.Latitude,
                              posicaoAtual.Coordinate.Longitude);

        Mapa.ZoomLevel = 15.5;
    }
    catch
    {
        MessageBox.Show("Não foi possível encontrar sua localização.
        Por favor, verifique se suas configurações de
```

```
        localização estão habilitadas.");
    }
}
```

Note que o código também centraliza o mapa e o zoom em torno da posição atual do usuário. Devemos fazer uma chamada deste método no manipulador do evento `Loaded` desta página.

Implementaremos o botão `Adicionar Endereço` para que ele crie um ponto azul no mapa para o endereço que o usuário digitar. Para executarmos a pesquisa, utilizaremos o objeto `GeocodeQuery`, que busca por localizações utilizando termos de linguagem natural, conforme o código:

```
private void BuscarPorEndereco(string logradouro, string cidade)
{
    GeocodeQuery pesquisa = new GeocodeQuery();
    pesquisa.MaxResultCount = 1;
    pesquisa.SearchTerm = string.Concat(logradouro, ", ", cidade);
    pesquisa.GeoCoordinate = new GeoCoordinate(0, 0);
    pesquisa.QueryCompleted += pesquisa_QueryCompleted;
    pesquisa.QueryAsync();

    Logradouro.Text = string.Empty;
    Cidade.Text = string.Empty;
}

private void pesquisa_QueryCompleted
(object sender, QueryCompletedEventArgs<IList<MapLocation>> e)
{
    if( e.Result.Count > 0)
    {
        GeoCoordinate coordenadas = e.Result.First().GeoCoordinate;
        MarcarPosicaoNoMapa(coordenadas.Latitude,
                           coordenadas.Longitude,
                           Colors.Blue);
    }
    else
    {
        MessageBox.Show("Desculpe, mas o endereço não foi encontrado.");
    }
}
```

Agora basta que você adicione um manipulador ao evento `Tap` no botão e faça uma chamada para o método `BuscarPorEndereco`, passando por parâmetro os textos dos componentes da página, conforme ilustra a figura 9.6.

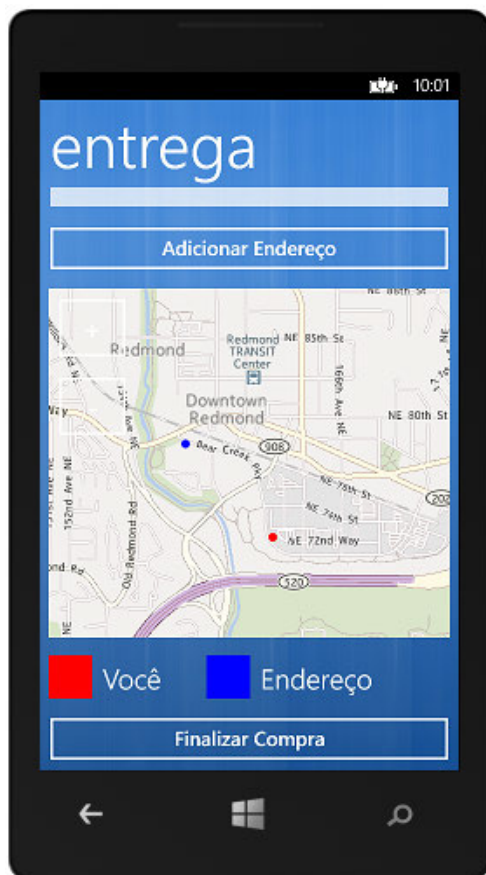


Figura 9.6: Endereço adicionado

Por fim, vamos fazer a última funcionalidade de nosso aplicativo: exibir uma mensagem quando o usuário pressionar o botão para finalizar uma compra.

```
private void FinalizarCompra_Tap
(object sender, System.Windows.Input.GestureEventArgs e)
{
    MessageBox.Show("Compra efetuada com sucesso!");
}
```

```
NavigationService.GoBack();  
}
```

O código completo do aplicativo que desenvolvemos ao longo deste livro se encontra em meu github no link: <http://bit.ly/GITversaoFinal>

CAPÍTULO 10

Dicas para publicação na loja do Windows Phone

Após completar seu aplicativo, você poderá publicá-lo na loja do Windows Phone! Este último capítulo é dedicado a explicar brevemente quais passos você deve executar para que esta publicação seja feita e algumas dicas e sugestões a respeito disso.

10.1 PASSOS PARA PUBLICAÇÃO

O primeiro passo para que você efetue sua publicação é ter uma conta Microsoft e cadastrá-la como uma conta de desenvolvedor. É necessário efetuar um pagamento anual, então crie uma conta somente quando estiver certo de que irá publicar algum aplicativo.

O segundo passo é você validar se o conteúdo de sua aplicação é aceito pelas políticas da Microsoft, ou seja, seu aplicativo não pode conter logos ou nomes de terceiros, nenhum conteúdo ofensivo, apologia a algum crime (sob a legislação do

país que o aplicativo está submetido) e vários outros problemas de conteúdo. Você pode verificar a lista completa no link: <http://bit.ly/WPvalidacaoConteudo>.

Você também terá de validar todos os aspectos técnicos de seu aplicativo. Todas as regras podem ser encontradas neste link: <http://bit.ly/WPregras>.

O terceiro passo é promover seu aplicativo da forma correta: dê um título que encaixe com sua funcionalidade e que seja chamativo, utilize a categorização dos aplicativos na loja para que ele seja encontrado com facilidade, dê as palavras-chave mais importantes para seu aplicativo e faça uso de uma versão gratuita para testes. Isso auxiliará a promover seu app.

O quarto passo é você pensar com cuidado a forma de receber retorno financeiro. Escolha entre propagandas, aplicativos pagos ou compras de funcionalidades dentro do aplicativo. Sempre tenha em mente o público que você quer atingir e se foque nisso.

Por fim, submeta seu aplicativo na loja do Windows Phone (<https://dev.windowsphone.com>) e faça correções tanto para que seu aplicativo seja aprovado na loja quanto para tornar a experiência do usuário cada vez melhor.

10.2 CONCLUSÃO

Neste livro, você passou pelas principais funcionalidades introdutórias necessárias para se criar um aplicativo para Windows Phone, focando sempre na didática e tentando trazer essas funcionalidades da forma mais simples possível.

Neste ponto você já deve estar apto a criar seus aplicativos sozinho, utilizando processamentos assíncronos, vínculo de dados, uma boa live tile e tudo mais. É importante ressaltar que vários padrões arquiteturais de aplicação foram ignorados com o objetivo de fazer com que o leitor se concentre apenas no conteúdo principal abordado.

Espero que este livro seja apenas um gatilho para sua busca por conhecimento para aplicativos para Windows Phone. Este material possui um foco bastante introdutório e não cobre de forma nenhuma todas as áreas possíveis do Windows Phone. Além disso, não se prenda a uma única plataforma, é uma boa dica você conhecer o máximo de plataformas que conseguir.

O desenvolvimento de aplicação para Windows Phone é bastante prático e divertido. Acredito que unir desenvolvimento com todas as noções de experiência de usuário seja um trabalho bastante rico e proveitoso para um desenvolvedor. Busque também por funcionalidades e códigos não mencionados neste livro, como a aborda-

gem de compras *in-app*, processamento em background, push notifications e várias outras.

Como autor, eu busquei compartilhar meu conhecimento da forma mais didática, simples e enxuta. Espero que este livro seja proveitoso para você e que, a partir de agora, você se torne oficialmente um desenvolvedor para a plataforma Windows Phone.