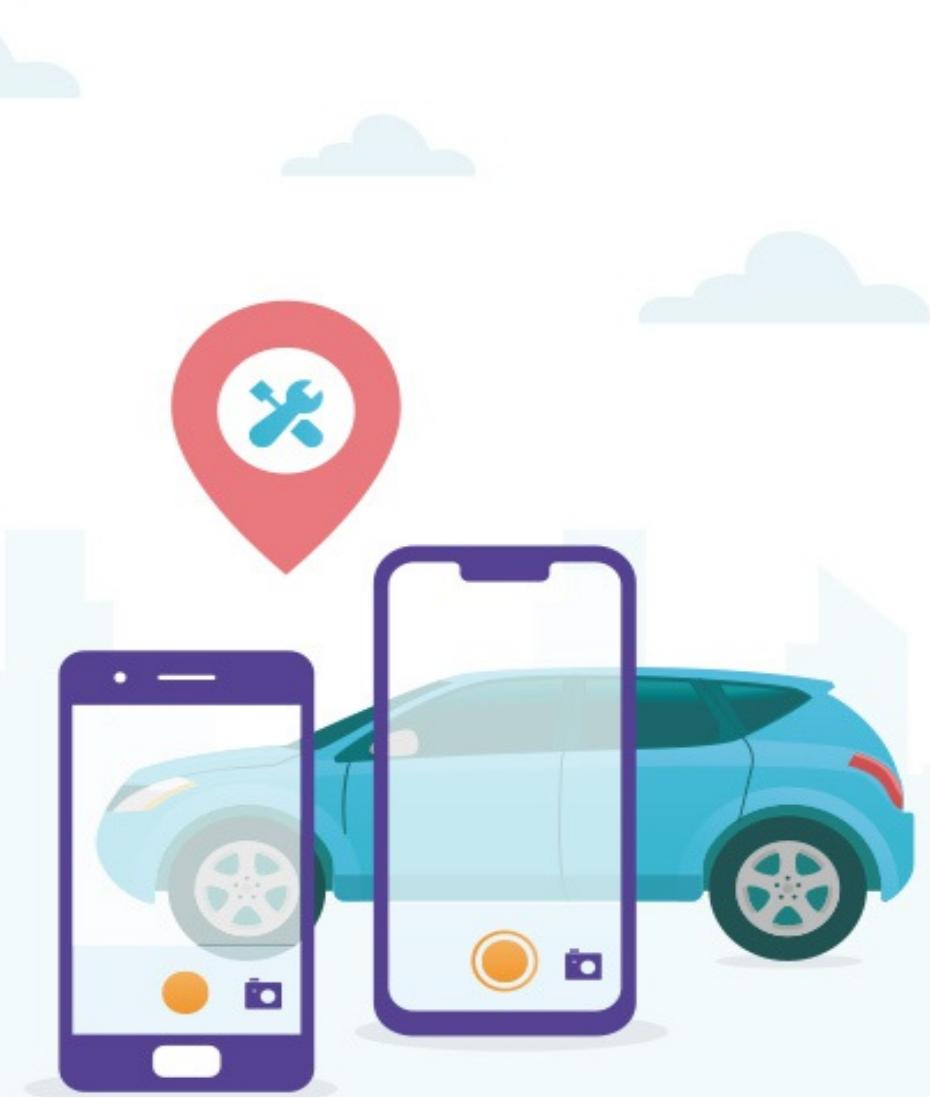


# Xamarin Forms e MVVM

Persistência local com Entity Framework Core



# Sumário

- [ISBN](#)
- [Agradecimentos](#)
- [Sobre o autor](#)
- [Prefácio](#)
- [Sobre o livro](#)
- [1. Dispositivos móveis, desenvolvimento cross-platform e o Xamarin](#)
- [2. Xamarin — Instalação e testes](#)
- [3. Tipos de páginas, layouts e alguns controles para interação com o usuário](#)
- [4. O padrão Model-View-ViewModel e o Messaging Center do Xamarin](#)
- [5. Execução no dispositivo físico, SQLite e Entity Framework Core](#)
- [6. Associações, Pesquisa, DatePicker, TimePicker e ActionSheet](#)
- [7. Associações com coleções](#)
- [8. Uso de câmera e álbum](#)
- [9. Listagem de fotos e manipulação de gestos](#)
- [10. Custom renderers, login de acesso e consumo de serviços REST](#)
- [11. O CRUD de peças com o consumo de serviços REST](#)
- [12. Os estudos não param por aqui](#)
- [13. Apêndice - Criação de serviços REST](#)

## **ISBN**

Impresso e PDF: 978-85-94188-98-4

EPUB: 978-85-94188-99-1

MOBI: 978-85-7254-000-1

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br> .

## Agradecimentos

Este é um espaço que considero muito importante em meus livros. É o momento de reconhecimento ao apoio direto ou indireto de pessoas importantes para mim, durante o processo de pesquisa e escrita do livro.

Não posso jamais deixar de agradecer ao apoio de meus filhos, pessoas chaves em minha vida. No momento em que me encontro, pessoal e profissionalmente, duas pessoas são muito importantes para os momentos de relaxamento: Maria Clara e Vicente Dirceu. Meus caçulas, minhas joias mais preciosas. Meu filho Gabriel e minha nora Camila, conversamos sempre à noite, em momentos em que me divirto muito com eles. Comecei a escrever este livro na casa deles. Obrigado, meus filhos.

Nos momentos finais do livro, quando já havia acabado a escrita básica, retomei contato com um aluno que tive, excelente pessoa e competentíssimo profissional: Leonan Fraga Leonardo, que mantém sempre seu LinkedIn atualizado. Procurei o Leonan para me auxiliar em um assunto que nem cheguei a inserir neste livro, mas quem sabe em um futuro trabalho, que era o uso de Push Messages. Depois, algumas dúvidas e problemas que surgiram, sempre levei para conversar com ele e as sugestões e experiências que ele trouxe foram primordiais para os capítulos 8 e 9. Sua ajuda foi tão importante que o convidei para fazer a revisão técnica deste livro e que bom que ele aceitou. Obrigado, Leonan. Te desejo um futuro brilhante.

Quando quis começar a trabalhar com a pesquisa e escrita para este livro, faltava-me alguma ideia para o tema a ser abordado nele, nas implementações. Foi quando, em um bate-papo com o Jorginho (Jorge Aikes Júnior), também um ex-aluno e hoje um importante colega de trabalho, ele sugeriu o trabalho com registro de atendimentos em uma oficina. Segui a ideia e comecei a pesquisa, implementação e escrita do livro. No meio do trabalho, convidei o Jorge para prefaciar o livro e ele, sem muita opção, aceitou. Obrigado, Jorge.

À Casa do Código, agradeço sempre à Vivian Matsui, um anjo que sempre me acompanha nos trabalhos que desenvolvo, sempre paciente e procurando tempo para ler e contribuir com as revisões dos livros. Obrigado, Vivian e este agradecimento se estende a toda a equipe da editora.

De maneira geral, agradeço sempre aos meus colegas de trabalho do Departamento de Computação da UTFPR, câmpus Medianeira, meu paraíso profissional, pelo coleguismo e pelos ouvidos.

## Sobre o autor

Everton Coimbra de Araújo atua na área de ensino de linguagens de programação e desenvolvimento. É tecnólogo em processamento de dados pelo Centro de Ensino Superior de Foz do Iguaçu, possui mestrado em Ciência da Computação pela UFSC e doutorado pela UNIOESTE em Engenharia Agrícola, na área de Estatística Espacial. É professor da Universidade Tecnológica Federal do Paraná (UTFPR), câmpus Medianeira, onde leciona disciplinas no Curso de Ciência da Computação e em mestrados. Já ministrou aulas de Algoritmos, Técnicas de Programação, Estrutura de Dados, Linguagens de Programação, Orientação a Objetos, Análise de Sistemas, UML, Java para Web, Java EE, Banco de Dados e .NET.

Possui experiência na área de Ciência da Computação, com ênfase em Análise e Desenvolvimento de Sistemas, atuando principalmente nos seguintes temas: Desenvolvimento Web com Java e .NET, Persistência de Objetos e agora em Dispositivos Móveis. O autor é palestrante em seminários de informática voltados para o meio acadêmico e empresarial.



# Prefácio

Mal sabia, há 13 anos, quando iniciava minha graduação e o professor Everton era um dos meus primeiros professores (desculpe, Everton, pelo spoiler da idade!), que teria a oportunidade, agora como colega professor na universidade, de escrever um prefácio para uma de suas obras. Livros estes, aliás, que não apenas usei durante minha graduação, mas que também indico aos meus alunos de nível universitário até hoje devido à linguagem clara, exemplos objetivos e simplicidade em passar conteúdos nada simples.

Esta simplicidade e riqueza de escrita, sabendo exatamente onde estarão as dúvidas e atacando-as antes mesmo que o leitor perceba, vêm de sua vasta experiência na área de treinamento. Seus livros são resultados de suas experiências dentro e fora das aulas, catalogadas e filtradas de maneira a validar a metodologia a ser utilizada na exposição do conteúdo antes mesmo da escrita do primeiro parágrafo da introdução. Isso será percebido por você, estimado leitor, ao notar em alguns trechos do livro que o autor adianta erros, falhas na documentação e bugs encontrados, e os traz para você, de maneira que você não sofra com problemas conhecidos e documentados, mas também com problemas vivenciados e ainda não tão documentados assim. Nesse sentido, esta nova obra segue exatamente as características das 16 anteriores, prezando a qualidade de conteúdo e a qualidade de apresentação.

Nesta nova obra você terá a oportunidade de se aventurar no mundo do desenvolvimento de aplicativos móveis “cross-plataform”. A maciça presença dos aplicativos móveis é inegável e a necessidade de conhecimento no desenvolvimento desses aplicativos é simplesmente primordial para qualquer desenvolvedor da atualidade. Infelizmente, a falta de padronização de sistemas operacionais e hardware torna esta tarefa extremamente desafiadora.

Com o objetivo de facilitar este processo, o framework Xamarin e Xamarin Forms vem ao nosso auxílio. Este livro os apresentará de maneira incremental e linear, presendo em introduzir os conteúdos com apenas a teoria necessária ao seu entendimento, indo direto para o “mão na massa”. Desse modo, os detalhes técnicos e teoria necessária são apresentados durante a construção dos elementos. Muitas vezes você se pegará aprendendo as técnicas, componentes e métodos sem se dar conta disso, devido à leve e objetiva leitura dos itens.

A jornada deste livro consiste na construção de um aplicativo para controle de oficinas mecânicas. Ela inicia com definições básicas sobre desenvolvimento para dispositivos móveis, apresentando as vantagens da utilização do Xamarin. Na sequência, serão apresentados, de maneira detalhada, os procedimentos para instalação e validação do funcionamento da plataforma. De maneira prática, serão explanados os conteúdos básicos necessários, como layouts e controles de interação do usuário, o modelo MVVM e como o utilizar com o Xamarin e diversos outros conteúdos intermediários como componentes do framework tais como DatePiker, ActionSheet e como os utilizar. A menina dos olhos dos dispositivos móveis, acesso e controle de câmeras e álbuns, não foram esquecidos e são utilizados dentro do contexto da aplicação de exemplo.

A integração com bases de dados é garantida com conceitos de SQLite e Entity Framework. Tudo isso será ainda trabalhado em contexto globalizado, isto é, internet, considerando o consumo de serviços REST com deploy em servidores reais.

A sequência da dificuldade dos itens apresentados garante que você aprenda os conteúdos conforme já domina os anteriores. Você verá que em pouco tempo saberá o processo completo do desenvolvimento móvel “cross-plataform”. E o mais importante: de maneira leve e divertida, então divirta-se.

---

Jorge Aikes Júnior

*Universidade Tecnológica Federal do Paraná - Campus Medianeira*

## Sobre o livro

Este livro traz, na prática, o desenvolvimento de aplicações *cross-platform* com o Xamarin e Xamarin Forms, frameworks para desenvolvimento de aplicativos para dispositivos móveis. O desenvolvimento de um aplicativo para ser publicado em dispositivos com plataformas diferentes (iOS e Android), de forma nativa, é uma tarefa muito tranquila com o Xamarin. É possível criar uma aplicação, utilizando a linguagem C#, e ela ser publicada para as duas plataformas.

O livro é desenvolvido em onze capítulos, além de conclusão e apêndice. O primeiro é apenas teórico, mas não menos importante, pois trago nele contextualizações sobre dispositivos móveis e as ferramentas usadas no livro. Já no segundo capítulo, apresento o processo de instalação e teste da plataforma e dos IDEs, fazendo uso de emuladores. Veremos também algumas ferramentas do Xamarin para pré-visualização das visões.

A aplicação a ser implementada durante o livro refere-se a uma oficina mecânica, com cadastro de Clientes, Serviços, Peças e Atendimentos. Persistiremos os dados em uma base de dados local e ao final em uma base web, centralizada. Ofereceremos ao usuário a possibilidade de capturar fotos no momento da entrada do veículo na oficina.

A prática começa bem legal no capítulo 3, com apresentações dos tipos de páginas com que uma aplicação Xamarin Forms pode trabalhar. São apresentados também alguns tipos de layout, gerenciadores de conteúdo, oferecidos pela ferramenta. Durante estas implementações, são apresentados controles visuais que podem ser utilizados em suas aplicações.

No quarto capítulo, teremos o tema MVVM, que é o modelo mais utilizado no desenvolvimento de aplicações mobile, no qual temos nosso modelo de negócio ligado a um modelo de visão, que por sua vez se liga, atualizando e sendo atualizado pela visão.

No capítulo 5, trabalharemos a execução de suas aplicações em dispositivos reais. Ficam muito mais fáceis os testes, principalmente para as aplicações Android, pois seus emuladores trazem consigo a sinal de serem pesados. Também implementaremos neste capítulo o acesso à base de dados, fazendo uso do Entity Framework Core, API da Microsoft para persistência de objetos em um paradigma relacional. É uma ferramenta poderosa, já utilizada amplamente, e consolidada, em aplicações .NET. Desenvolveremos neste capítulo toda uma estrutura para persistência.

No capítulo 6, começaremos a trabalhar associações entre objetos e sua persistência em uma base de dados SQLite, por meio do EF Core. Para as implementações das visões, continuaremos vendo novos controles e sempre evoluindo no MVVM. Veremos também técnicas para recuperação de objetos associados.

Dando sequência ao tema de associações, no capítulo 7, começaremos com multiplicidade um-para-muitos. Veremos neste capítulo controles nativos para seleção de data e hora.

Os capítulos 8 e 9 são, como todos, muito importantes e interessantes. Continuaremos com associações um-para-muitos, mas trabalhando com imagens e gestos. Capturaremos imagens diretamente da câmera e do álbum de fotos dos dispositivos. Teremos introduzidos aqui alguns dos novos recursos trazidos pelo Xamarin 3. Veremos também técnicas para remoção de objetos associados e trabalharemos com a captura de gestos do usuário sobre imagens.

O décimo capítulo e o décimo primeiro fecham o livro com chave de ouro. Implementaremos serviços REST em um servidor Web, um plus do livro. Consumiremos estes serviços em nossa aplicação Xamarin. Veremos técnicas para envio e recebimento de fotos do, e para o, dispositivo. Veremos nele também o acesso à aplicação por meio de um login e trabalharemos a sincronização de dados com uma base Web. São dois capítulos muito bons. No capítulo 12 concluímos nosso trabalho, e ainda há a disponibilização de um documento complementar no capítulo 13, um apêndice que o auxiliará nos estudos dos capítulos finais.

Certamente, este livro pode ser usado como ferramenta em disciplinas que trabalham o desenvolvimento de dispositivos móveis, quer seja por acadêmicos ou professores. Isso porque ele é o resultado da experiência que tenho em ministrar aulas dessa disciplina, então trago para cá anseios e dúvidas dos alunos que estudam comigo. Eu já utilizei este conteúdo com meus alunos e eles gostam muito.

É importante que o leitor tenha conhecimento de Orientação a Objetos e da linguagem C#, mas não é um fator impeditivo. Conhecimentos básicos sobre banco de dados também são interessantes. O repositório com todos os códigos-fontes usados no livro pode ser encontrado em: [https://github.com/evertontfoz/xamarin\\_mvvm\\_efcore/](https://github.com/evertontfoz/xamarin_mvvm_efcore/).

Que a leitura deste livro seja para você, tão prazerosa, como para mim foi escrevê-lo. Desfrute, sem moderação. Sucesso.

## C APÍTULO 1

# Dispositivos móveis, desenvolvimento cross-platform e o Xamarin

Olá! Seja bem-vindo ao primeiro capítulo deste livro. Ele será curto e conterá apenas teoria, mas nos demais compensarei com a prática. Nele buscarei trazer conteúdo sobre o panorama dos dispositivos móveis nos dias de hoje, tipos de dispositivos e suas características e opções para o desenvolvimento de aplicativos.

Ao ler este primeiro capítulo, se você leu meu primeiro livro sobre Xamarin, *Xamarin Forms: Desenvolvimento de aplicações móveis multiplataforma* (<https://www.casadocodigo.com.br/products/livro-xamarin-forms/>), pode pensar que é o mesmo material, mas não é. Aproveitei apenas a contextualização deste capítulo. A partir do segundo, você verá a diferença aparecer.

As ferramentas que serão usadas neste livro serão apresentadas conforme forem necessárias. É importante que você tenha conhecimento de Orientação a Objetos e de alguma linguagem de Programação Orientada a Objetos para um perfeito acompanhamento do desenvolvimento proposto para este livro.

A aplicação proposta para ser desenvolvida neste livro, a partir do quarto capítulo, refere-se ao atendimento oferecido por uma oficina mecânica. A aplicação subsidiará o cadastro de clientes, peças e serviços a serem contratados para veículos e registro de atendimento a clientes. Nestas funcionalidades, serão trabalhados tipos de telas para a aplicação, controles de entrada de dados, listagem de dados, templates para aparência da aplicação, acesso a banco de dados, consumo de serviços web e uso de câmera.

Durante o desenvolvimento da aplicação, diversas técnicas e recursos serão utilizados para subsidiar uma boa arquitetura para o desenvolvimento da aplicação.

## 1.1 Os dispositivos móveis na atualidade

Até anos atrás, era comum conhecer pessoas que adquiriam computadores apenas para navegar na internet, ler e-mails e acessar algumas aplicações para leitura de livros, artigos ou documentos diversos. No lado corporativo empresarial, notebooks eram fornecidos aos colaboradores para que desempenhassem algumas atividades, como registro de uma venda para um cliente, recebimento de uma conta, anotação de um pedido e agendamento de compromissos, dentre diversas outras atividades.

Com o surgimento dos dispositivos móveis, a venda de computadores pessoais tem sofrido constantes quedas. Isso começou lá atrás, de maneira modesta e quase despercebida, com *handhelds*, *palmtops* e PDAs (*Personal Digital Assistants*), mas que começou a ganhar destaque com a chegada do iPod e, depois, dos iPhones.

Lembro bem de quando Steve Jobs fez o lançamento do primeiro iPad e todos olharam para "aquilo" de maneira incrédula. Mal sabiam a revolução comportamental que aquele dispositivo traria. A massificação veio por meio do Google, com o desenvolvimento do sistema operacional Android para dispositivos mais **aceitáveis**, em relação aos produtos da Apple.

E os notebooks? Estão cada vez mais finos, mais leves e mais parecidos com tablets. E para aqueles que "só" queriam ler seus livros em um meio que não fosse no papel, surgiram os e-readers. Porém, estes livros também podem ser lidos em smartphones e tablets.

Embora um dispositivo móvel possa permitir acesso para necessidades pessoais e corporativas, é importante ressaltar que a maneira como este mercado se desenvolveu foi bem distinta. Tem-se o lado recreativo, em que é possível ter em seu dispositivo diversos jogos; o cultural, que permite o acesso a filmes e livros; a organização pessoal, com diversos recursos para gestão financeira e de compromissos, acesso a bancos, compras; e a parte empresarial, com aplicativos corporativos, de gestão, operacional das empresas e vendas.

Hoje, os smartphones são mais utilizados como ferramentas computacionais do que como aparelho telefônico e despertador.

Toda essa popularização e mudança comportamental resultaram no que é visto atualmente como um fenômeno BYOD (*Bring Your Own Device* — Traga seu Próprio Dispositivo). Ou seja, o colaborador está levando para seu ambiente de trabalho um dispositivo potente (o seu) e que pode, em alguns casos, ser substituído por um computador, quer seja desktop ou notebook.

Com isso e o acesso à rede corporativa liberado, todos podem ter no bolso as aplicações de seu local de trabalho. Isso pode gerar mais produtividade também - ou não :). Porém, no que diz respeito à segurança, traz novas ameaças e vulnerabilidades, mas isso é assunto para a equipe de infraestrutura.

A massificação da internet (que ainda está longe de ser atingida) foi um fator importante para que os dispositivos móveis fossem difundidos da maneira como se encontram. E ela ainda aumentará com o surgimento de relógios e óculos inteligentes, maiores televisores, geladeiras, micro-ondas e outros, que nem sabemos que surgirão.

Pense em empregos nos quais se exija atividade externa, quer seja em campo ou urbana, como a função de um engenheiro agrônomo ou um vendedor externo. Esses profissionais precisavam se dirigir à empresa, buscar blocos ou impressos específicos para coleta de dados, retornar aos seus escritórios e registrarem os dados coletados. Com um dispositivo móvel, ele não precisa nem ir ao seu escritório, pois seu equipamento já permite o registro de sua coleta ou venda no momento em que ela ocorre. Você também já reparou onde os atendentes em restaurantes registram nossos pedidos? Um dispositivo móvel.

E se o acesso à internet não for possível no momento da atividade, é possível uma sincronização com os servidores da empresa tão logo uma conexão seja possível. Os dispositivos também podem estar conectados apenas em uma rede interna, não sendo necessária a internet. Já viram na televisão, no jornalismo, que os repórteres se comunicam com suas centrais pelo Skype? Com transmissão ao vivo para os telespectadores? Isso é mobilidade.

As empresas buscam também tirar proveito dessa realidade, quer seja dando maior liberdade para seus colaboradores, ou economizando em despesas como energia, água, condomínio ou aluguel, dentre outras.

Com o advento da "internet para o usuário", surgiu o tema relacionado à Experiência do Usuário, pois, ao se desenvolver aplicações para o ambiente Web, não era desejado que nesta plataforma o usuário tivesse a mesma tela de aplicação que tinha em um ambiente desktop. Com o surgimento cada vez maior de aplicativos para dispositivos móveis, este tema reforçadamente é debatido. O usuário quer fazer uso dos recursos e características do dispositivo que usa e não ter nele a aplicação com a mesma aparência que ela tem, agora, na Web.

Neste livro, trabalharemos aplicações que possam ser executadas nos sistemas operacionais iOS (iPhones e iPads) e Android. Neste último, existem diversos dispositivos, nas mais diversas especificações de tamanho e resolução de tela, recursos e processamento. No primeiro livro sobre Xamarin eu apresentei também soluções para a plataforma Windows. Entretanto, as aplicações propostas nestes livros, em tese, podem ser executadas em emuladores e dispositivos Windows.

## 1.2 O desenvolvimento móvel cross-platform

Não é porque os dispositivos móveis estão em uma grande ascendência que os computadores pessoais se tornaram descartáveis. Existem ainda diversas atividades e aplicativos que precisam ser realizados e utilizados em um computador que possua um teclado (que não seja na tela), uma tela grande (às vezes mais do que uma) e um desempenho ainda não alcançado pelos dispositivos móveis atuais.

Com isso posto, é importante estar ciente de que as aplicações comerciais trazidas para o ambiente móvel precisam, nas interações com o usuário, ter ou solicitar dados e informações que possam caber "confortavelmente" na tela que será usada. Nada de encher de informações que são desnecessárias para o processo atual.

O desenvolvimento para dispositivos móveis, até a presente data, vem sendo focado nas plataformas iOS e Android — estas sendo as mais utilizadas.

Cada uma dessas plataformas possui suas próprias características e recursos. Com isso, você pensa em escolher uma

plataforma para que seu aplicativo funcione e limitar o uso a apenas esta?

Se for uma empresa que especifique a plataforma e o aplicativo deva funcionar apenas entre os colaboradores dela, isso é uma opção, sim. Mas se sua aplicação precisar ser utilizada em plataformas diferentes? Aí esta opção já não poderia ser escolhida.

O que vem ocorrendo no desenvolvimento para dispositivos móveis é que, ao desenvolver uma aplicação para um dispositivo e uma plataforma, esta aplicação possa funcionar em qualquer dispositivo, de qualquer plataforma.

Por exemplo, deseja-se que uma aplicação desenvolvida para o iPhone X possa funcionar da mesma maneira em um smartphone com Android. O que fazer para resolver este problema? Montar várias equipes de desenvolvimento, cada uma usando um ambiente de desenvolvimento diferente e focada em uma plataforma? Desenvolver diversas versões do mesmo aplicativo? Isso não é produtivo, e nem barato, concorda?

O objetivo é partir para uma ferramenta que possibilite o desenvolvimento *cross-platform*. Existem várias. Algumas geram o aplicativo para ser executado em um ambiente específico, como se fosse uma máquina virtual (conhecido como ambiente híbrido). Este tipo de aplicativo corre o risco de ter a "mesma cara" em dispositivos diferentes, não trazendo benefícios para a experiência do usuário.

Algumas ferramentas geram código para ser executado em um ambiente Web, como se fosse um site em um navegador.

Outras ferramentas, como é o caso do Xamarin, geram aplicativos nativos para as plataformas escolhidas, permitindo fazer uso das características e recursos oferecidos por estas plataformas e seus dispositivos. Outras tecnologias permitem também o uso de recursos físicos, mas não de maneira direta, nativa.

Ainda, no caso do Xamarin, podemos desenvolver uma aplicação que funcione nas duas plataformas que adotamos, fazendo uso da mesma ferramenta (o Visual Studio) e usando a mesma linguagem e os recursos oferecidos por ela, o C#.

Existem ainda aplicações desenvolvidas para a Web e que são utilizadas exclusivamente para dispositivos móveis. A melhor opção é o desenvolvimento nativo, e o Xamarin permite isso.

Um ponto importante é verificar na App Store (iOS) e no Google Play os padrões e as regras que devem ser respeitados para a publicação oficial de sua aplicação. Por exemplo, os ícones das aplicações são diferentes em cada plataforma, assim como a barra de navegação e o padrão de cores. São alguns dos pontos que você deve verificar e garantir que sua aplicação os respeite para que a publicação seja aceita. E para você distribuir aplicações pela App Store e Play Store, será necessária uma conta de desenvolvedor, que é paga.

## 1.3 O Xamarin

O Xamarin era uma plataforma proprietária, com custo para o desenvolvedor. Ele foi adquirido pela Microsoft, que eliminou este custo, deixando-o gratuito. Além disso, seguindo a nova filosofia da Microsoft, ela liberou o código-fonte do Xamarin. Ou seja, ele é *free* e *open source*.

Como uma plataforma para desenvolvimento de aplicativos cross-platform, o Xamarin tem como foco dispositivos móveis com o iOS, Android e outras plataformas que estão surgindo, mas que não serão focos neste livro.

Segundo a documentação, é possível a reutilização de 75% a 100% de código. Isso quer dizer que são poucas as situações em que será necessário escrever código específico para uma das plataformas citadas.

Aplicativos desenvolvidos por meio do Xamarin e do C# têm acesso nativo e total às plataformas que os executarão, podendo extrair ao máximo seus recursos.

Ao utilizar o Xamarin como plataforma de desenvolvimento, é possível o compartilhamento de praticamente toda a

lógica de negócio entre as plataformas alvo. Ou seja, você escreve o código uma única vez e o invoca em cada plataforma de execução.

Já o Xamarin Forms, que é uma plataforma direcionada para o desenvolvimento da camada de apresentação, permite o compartilhamento da interface com o usuário. Ou seja, você pode desenhar sua tela uma única vez e ela será renderizada, de maneira nativa, em cada plataforma móvel, usando seus controles nativos. Tudo isso, codificando em C#. As interfaces com o usuário podem ser codificadas fazendo uso de C# ou do XAML específico do Xamarin, que segue a mesma filosofia do XAML do WPF e Silverlight. Agora, em sua versão 3, o Xamarin Forms permite também o uso de CSS.

## 1.4 O foco prático deste livro com o MVVM e o Entity Framework Core

Em alguns momentos, neste capítulo, posso falar de meu primeiro livro sobre Xamarin, mas deixo claro para você que não há qualquer dependência de leitura dele para este livro, apenas comento para o caso de você ter curiosidade, pois o conteúdo dele é diferente do abordado neste que você está lendo.

É muito comum, ainda, ver situações em que programadores implementam suas soluções diretamente em métodos mapeados para eventos disparados pela visão na própria visão, ou, como chamamos em Xamarin, no *code-behind*. Isso é uma prática ruim, nada orientada a objetos e sem respeito aos princípios de coesão e acoplamento.

Neste livro, o foco central para a lógica da aplicação, no que diz respeito à interface com o usuário, está todo mapeado para utilizar o MVVM (*Model-View-ViewModel*) para tratar a lógica relacionada à interface com o usuário, a visão. O padrão de projeto MVVM busca propiciar a separação de responsabilidades, possibilitando tornar um aplicativo fácil de ser mantido. Imagine uma visão (janela ou página de um sistema), em que temos controles visuais que devem ter seus dados obtidos e fornecidos para objetos, mas não queremos ligar nossa camada de negócio diretamente à visão. É aí que entra o MVVM, fornecendo-nos um recurso chamado *View-Model*, ou seja, um modelo de negócio para a visão. E nosso modelo de negócio? Como é acessado? Por meio da View-Model.

Nossa começo de trabalho com MVVM começará no capítulo 4, após termos nossa plataforma de desenvolvimento toda instalada e testada e termos uma introdução com os tipos de páginas oferecidos pelo Xamarin Forms, seus Layouts e alguns controles. É um conhecimento necessário antes de chegarmos ao MVVM.

Outro ponto que vejo como importante neste livro é em relação à persistência dos dados. No primeiro livro utilizamos um plugin para o SQLite. Neste segundo livro faremos uso do Entity Framework, o ORM da Microsoft, que vem sendo utilizado cada vez mais em aplicações desenvolvidas para o .NET. Entretanto, para trabalhar a arquitetura que proporei, desenvolveremos também a persistência com outro mecanismo. Não trarei no livro detalhes sobre ele, porém, todo o código estará disponível e referenciado no GitHub. No livro, o foco será a persistência com o EF Core. Você verá que, facilmente, será possível selecionar entre uma e outra ferramenta para a persistência dos dados.

Para o uso de serviços Web, trago a implementação dos serviços REST em Java e com publicação em uma plataforma de serviço simples e de fácil uso. Nós implementaremos o serviço e o cliente. Trabalharemos também alguns recursos muito interessantes relacionados a fotos (imagens) e captura de gestos do usuário nos dispositivos.

## 1.5 Conclusão

Conforme prometido no início deste capítulo, ele foi breve. Eu poderia trazer muito mais teoria aqui, mas penso que você quer começar logo com a prática. Entretanto, precisava lhe apresentar o texto aqui trabalhado.

É importante ler um pouco sobre dispositivos móveis, cross-platform, o Xamarin e o Xamarin Forms. É claro que o que eu trouxe para você foi apenas uma introdução, mas, acredite, com ela você já está pronto para começar o desenvolvimento móvel e tem motivos para escolher o Xamarin. Isso porque você quer e precisa desenvolver para,

no mínimo, duas plataformas e posso assumir que não quer, ou não pode, ter uma equipe para cada uma.

Você precisa de produtividade e quer que sua aplicação seja desenvolvida em uma única plataforma, mas que seja executada nas duas mais usadas atualmente. Então, a solução que proponho é esta: Xamarin, Xamarin Forms e C#.

No próximo capítulo, instalaremos a ferramenta em um MacBook Pro e em um PC Windows, e testaremos seu funcionamento. Até lá.

## C APÍTULO 2

# Xamarin — Instalação e testes

Neste segundo capítulo, trabalharemos a instalação do Visual Studio. Apresentarei processos para manter sua versão sempre atualizada, criaremos um projeto de teste e o executaremos em emuladores. Também testaremos recursos para visualização de conteúdos no próprio ambiente de desenvolvimento e veremos orientações sobre o aplicativo `Xamarin Live`, uma ferramenta relativamente nova que possibilita o teste de sua aplicação em um dispositivo, inicialmente de uma maneira menos burocrática.

## 2.1 Download e instalação

A ferramenta que utilizaremos, tanto no Windows como no Mac, é o Visual Studio, em sua versão 2017 Community, que é gratuita. A que instalei em minha máquina estava atualizada com a versão 15.7.3 (Windows) e 7.52.40 (Mac). Você pode obter o instalador para ambas as plataformas em <https://www.visualstudio.com/downloads/>.

O processo de instalação destas ferramentas é praticamente o mesmo, sendo necessário apenas a seleção dos componentes que devem ser instalados. Como a plataforma .NET no Mac ainda é recente, os recursos disponíveis para instalação do Visual Studio nela serão todos necessários para nosso desenvolvimento neste livro. Sendo assim, recomendo selecionar todos que são disponibilizados. Já no Windows, onde a plataforma .NET tem diversos recursos que podem ser utilizados pelo Visual Studio, instalaremos apenas o que é realmente necessário para as implementações propostas para o livro e que estão expostas na figura a seguir.

Informo que o processo de instalação, principalmente no Windows, é bem demorado e depende de sua qualidade de acesso à internet.

Se você for utilizar emuladores da Microsoft, que façam uso do Hyper-V, é importante você saber que precisará do Windows Professional, mas, para os que utilizaremos no livro, isso não se fará necessário, pois eles fazem uso da tecnologia HAXM, da Intel (<https://software.intel.com/en-us/articles/intel-hardware-accelerated-execution-manager-intel-haxm>).

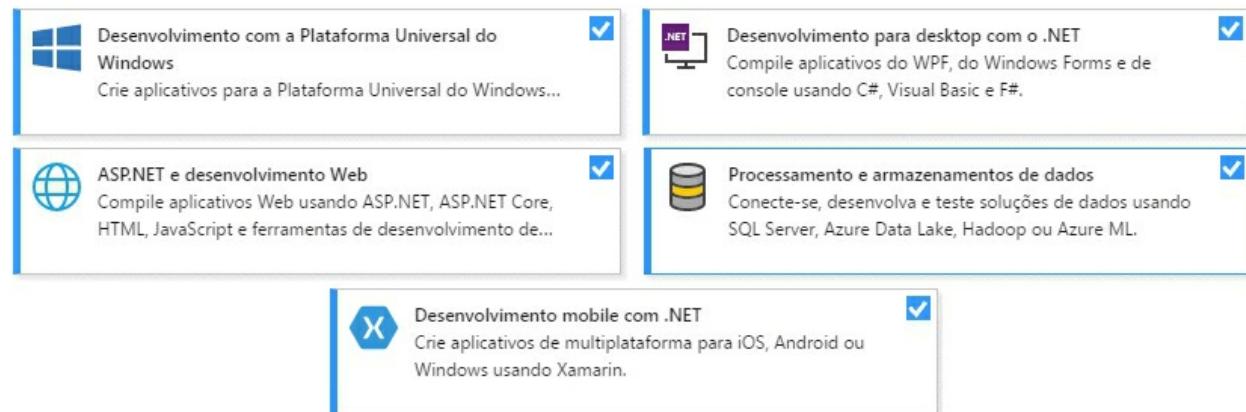


Figura 2.1: Componentes que devem ser instalados no Visual Studio para Windows

Para o Windows, é interessante que selezionemos os emuladores de Android da Google, para que sejam também instalados. Após marcar as opções da figura anterior, no topo da janela, clique em `Componentes Individuais`. Depois, nas opções que se apresentam, busque por `Emuladores` e marque os de nível de API 25 e 27, que eram os mais recentes no período de escrita deste livro. Você pode querer ter mais de um emulador em sua máquina, embora - já antecipo - após o **Capítulo 5**, onde veremos a implantação no dispositivo, você possa querer testar apenas nos dispositivos.

No Mac, ao executar o arquivo baixado, é apresentada uma tela referente à verificação de segurança na execução do arquivo obtido diretamente pela Web, e não pela App Store. Confirme a execução abrindo o arquivo.

É importante saber que a conexão com a internet deverá ser mantida por todo o processo (independente de ser Mac ou Windows), pois os pré-requisitos para a instalação terão seu download realizado neste momento. Outro ponto a se saber é que, durante a instalação no Mac, pode ser pedida a sua senha de administrador da máquina na qual se está executando a instalação. Ainda em relação ao Mac, você precisa ter o XCode atualizado. No momento em que escrevo este livro, a versão em minha máquina é a 9.4.

Em relação à versão do Xamarin Forms que utilizaremos neste livro, no momento de sua escrita, era a 3.0.0.550146 e a da .NET Standard Library era a 2.0.3. Você pode acompanhar as atualizações, no Windows, clicando com o botão direito do mouse sobre o nome da solução, ou projeto se preferir fazer isso individualmente, e então em Gerenciar Pacotes Nuget. No Mac você precisa escolher a opção update, que aparece quando clica com o botão direito do mouse sobre a pasta packages de cada projeto.

É muito importante que, caso você tenha o Visual Studio 2017 já instalado em sua máquina, ele esteja atualizado para a versão mais recente. Tive a situação em que alguns alunos não estavam com o IDE atualizado e ocorreram alguns erros relacionados a isso durante a criação dos projetos. As atualizações do Visual Studio, no Windows, podem ser verificadas e obtidas pela execução do visual Studio Installer e, no Mac, clicando no menu visual Studio Community e, então, em Check for Updates.

## 2.2 Teste da instalação realizada

Realizaremos agora a execução do Visual Studio nas duas plataformas e trabalharemos a criação de um projeto de teste para que possamos verificar se a instalação ocorreu de maneira correta. Nesta aplicação, não implementaremos nada, faremos uso dos templates oferecidos pelo IDE.

No Mac, com o Visual Studio aberto, clique no menu File->New Solution e, na categoria Multiplatform, escolha App. Do lado direito, em xamarin Forms, selecione Forms App e então clique no botão Next (figura a seguir).

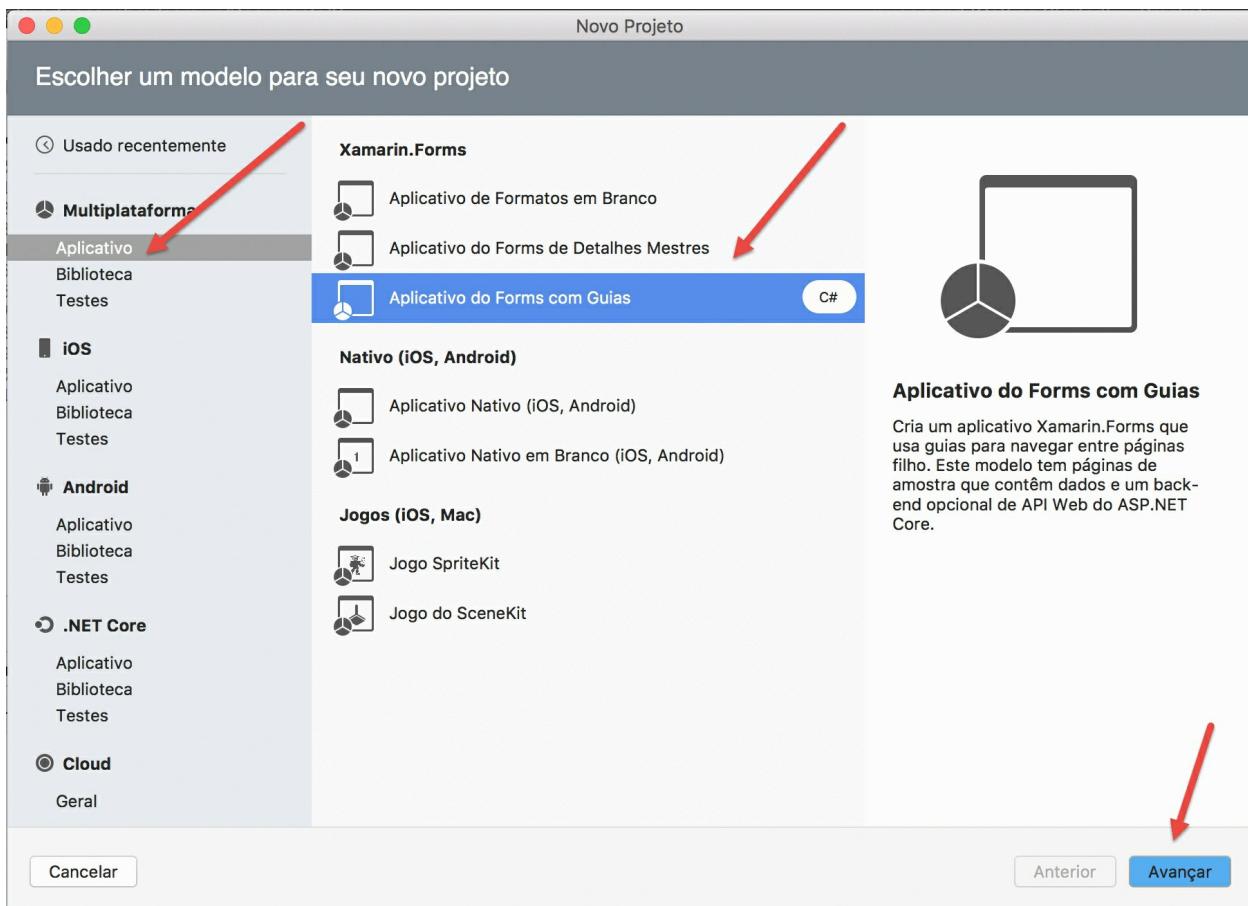


Figura 2.2: Criação do primeiro projeto Xamarin

Na janela que se abre, digite um nome para sua aplicação e o domínio que será utilizado para ela. Veja, na figura a seguir, meu preenchimento. Clique novamente em `Next`.

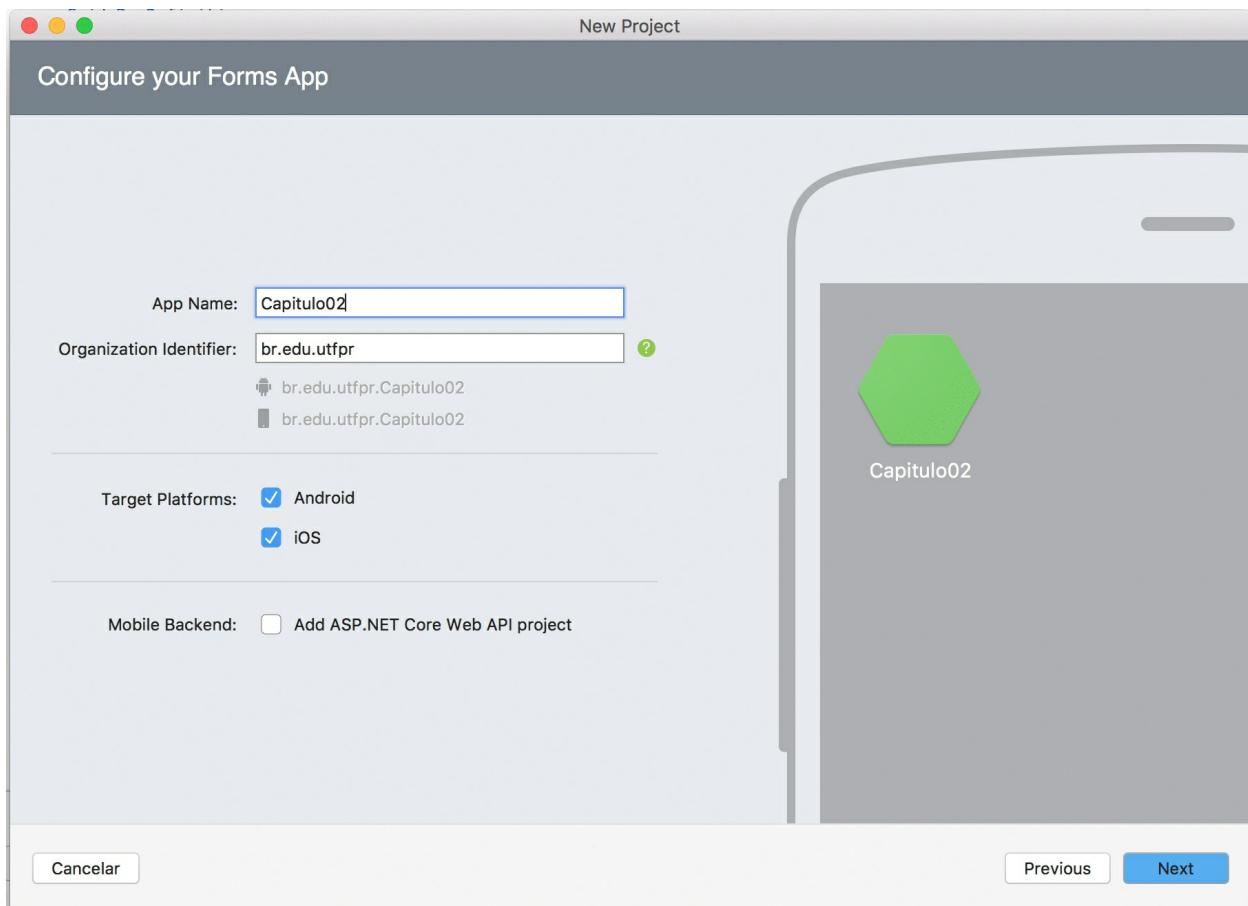


Figura 2.3: Definição do nome da aplicação

Para finalizar, uma página solicitando o nome para o projeto e o nome da aplicação é exibida. Eu optei por informar Capítulo02 no nome do projeto e xamarincc como nome da solução. Depois, clique no botão `create` para que o projeto seja criado. Você pode, também, escolher o local onde seu projeto será armazenado. O processo de criação do projeto, pelo Visual Studio, pode levar alguns segundos.

Após a criação do projeto, a janela `solution` deverá estar semelhante à apresentada na figura a seguir. Note que foram criados três projetos: o de biblioteca (*library*), que chamaremos sempre de `Projeto Xamarin Forms`, que é compartilhado com os projetos específicos para cada plataforma, um para iOS e um para Android. Tanto o projeto iOS como o Android fazem referência ao projeto compartilhado (biblioteca).

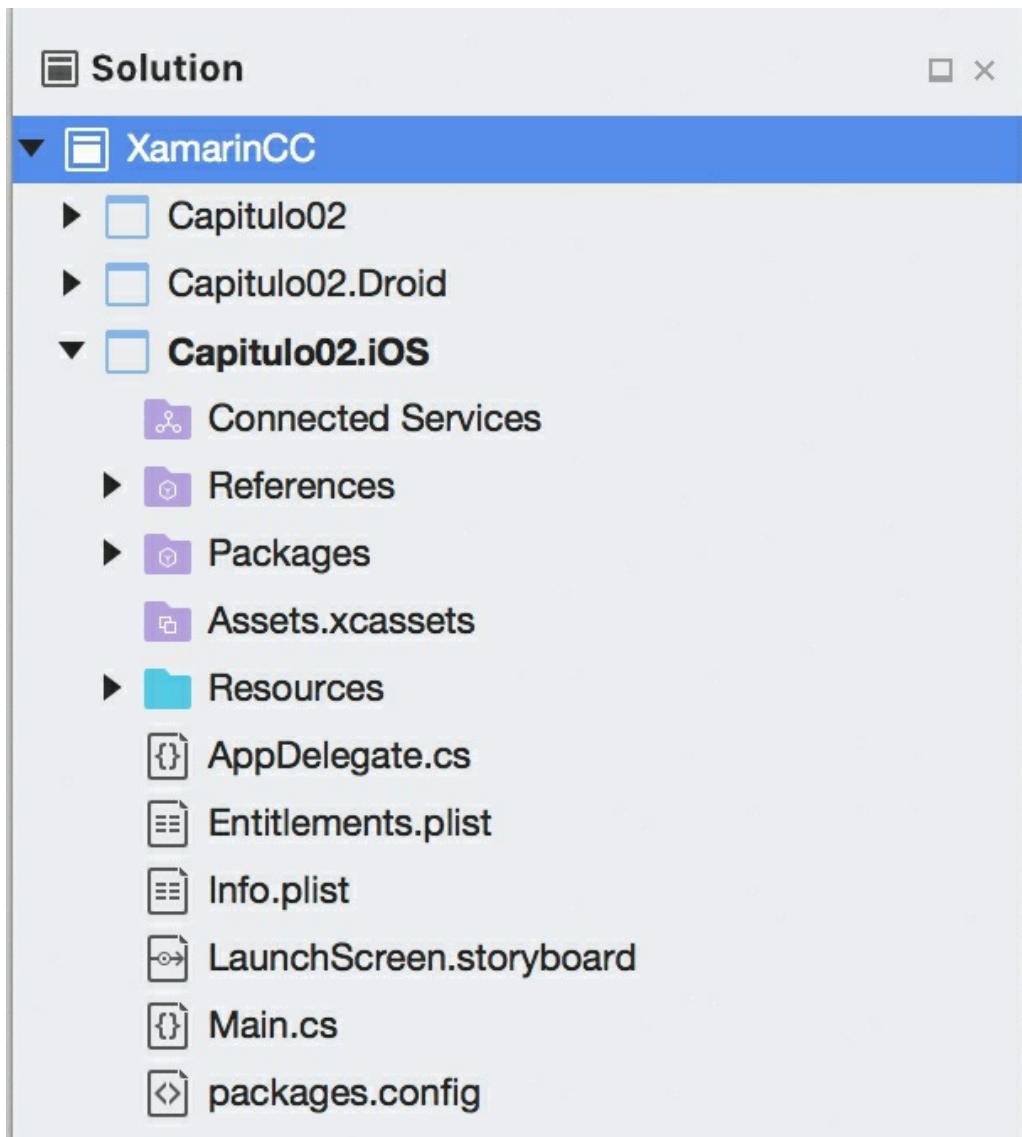


Figura 2.4: Janela da solução da aplicação

Aqui há um detalhe importante que requer uma especial atenção. Após a instalação da última atualização do Visual Studio no Mac, para a que estou usando no livro e foi comentada anteriormente, um erro surgiu.

Toda aplicação Xamarin Forms tem seus requisitos, como componentes e API, instalados nos projetos como pacotes Nuget e, sempre que uma solução é aberta ou criada, o Visual Studio procura recuperar estes componentes, que ficam disponibilizados na internet e são então atualizados localmente em seus projetos.

Em uma rede sem proxy, esta operação funciona muito bem. Entretanto, eu fiz um teste em uma rede com proxy, e os pacotes não puderam ser recuperados e um erro relativo a isso foi informado. Lembre-se de que estou falando do Mac, ok? No Windows não houve este problema. Detalhe, o proxy estava devidamente configurado nas preferências de rede.

A solução para este caso em específico foi acessar o terminal do Mac, atribuir o proxy como variável de ambiente, executando `export ALL_PROXY=proxy ip address :port number`. Depois, é importante lembrar de retirar essa configuração para uma rede sem proxy, também no terminal, executando `unset ALL_PROXY`. É interessante você fechar e abrir novamente o Visual Studio, mas você pode forçar a restauração clicando, com o botão direito do mouse, sobre a

pasta packages dos projetos e escolher a opção `Restore`. Este mesmo procedimento pode ser utilizado para atualizar os pacotes do projeto. Fica a dica se você se deparar com esse problema.

Quando eu comecei a escrever este livro, a API 27 (SDK 8.1) do Android ainda não estava disponível, mas esta disponibilização ocorreu durante o processo de escrita e quando atualizei o Visual Studio, alguns problemas aconteceram. Se você instalou agora o Visual Studio, já com o SDK 8.1, certamente não terá estes problemas.

Vou então relatar o que aconteceu comigo, para que você possa resolver o problema, caso ele também aconteça com você. Na última atualização do Visual Studio, os projetos que eu tinha em Android, deixaram de ser compilados, acusando a necessidade de um *runtime* para o SDK 8.1 do Android. Foi preciso acessar o menu `Tools->SDK Manager`, marcar o SDK 8.1 e clicar no botão para atualização. Este processo pode demorar um pouco, pois dependerá de seu acesso à internet e novamente o problema de proxy pode ocorrer.

Recomendo que nesta aplicação, na guia `Ferramentas` você realize as atualizações que sejam sugeridas, pois se o emulador para a versão 8.1 não tiver sido instalado automaticamente, você precisará destas atualizações para criá-lo. Ainda, na guia de ferramentas, após a atualização, marque a opção `Android Emulator` e realize a instalação. Depois, voltando para a guia `Plataformas`, marque, dentro do `Android 8.1 - Oreo` as opções `Google API Intel` e `ARM System Image` e as instale. A criação dos emuladores você poderá fazer pela opção `Tools->Google Emulator Manager` do Visual Studio.

Reforçando, se você instalou o Visual Studio agora, com o SDK 8.1, este problema não ocorrerá para você.

Voltemos ao projeto. Na figura anterior, verifique que o projeto para o iOS está com seu nome em negrito. Isso quer dizer que ele é o projeto padrão para execução (quando formos executar a aplicação pelo Visual Studio). Ele também está expandido, para que possa ser vista a estrutura criada para este tipo de projeto.

Na barra superior da janela do Visual Studio, existe, de acordo com o projeto padrão, a seleção do emulador, ou dispositivo, onde a aplicação deverá ser executada. Você pode alterar estas configurações direto nesta parte da janela. Veja a figura a seguir.

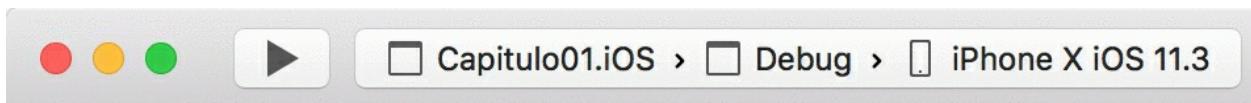


Figura 2.5: Escolhendo o emulador para a execução da aplicação

Na seleção do emulador, representada na figura anterior, eu escolhi o `iPhone X iOS 11.3`, por ser o modelo mais atual do dispositivo com a versão de seu sistema operacional também atualizada, em relação ao momento em que escrevo este livro. Clique no botão de execução para que seu projeto seja construído. Se a construção for bem-sucedida, o emulador será carregado (isso pode demorar um pouco).

Agora precisamos testar a aplicação em uma emuladora Android. Para isso, precisamos definir o projeto Android como `O Projeto Inicial`. Clique com o botão direito do mouse sobre o nome do projeto e clique na opção `Set as Startup Project`. Verifique, na seleção de emuladores, que agora são apresentados alguns específicos para o Android.

Para o teste, eu executei a aplicação nos dois emuladores Google para o Android disponibilizados pela minha instalação, o `Android_Accelerated_Nougat` e o `Android_Accelerated_Oreo`, sendo este último o mais atualizado. Faça sua seleção e execute a aplicação. Lembre-se de que a inicialização do emulador pode demorar um pouco. Veja o resultado da execução, no iOS e no Android, na figura a seguir.

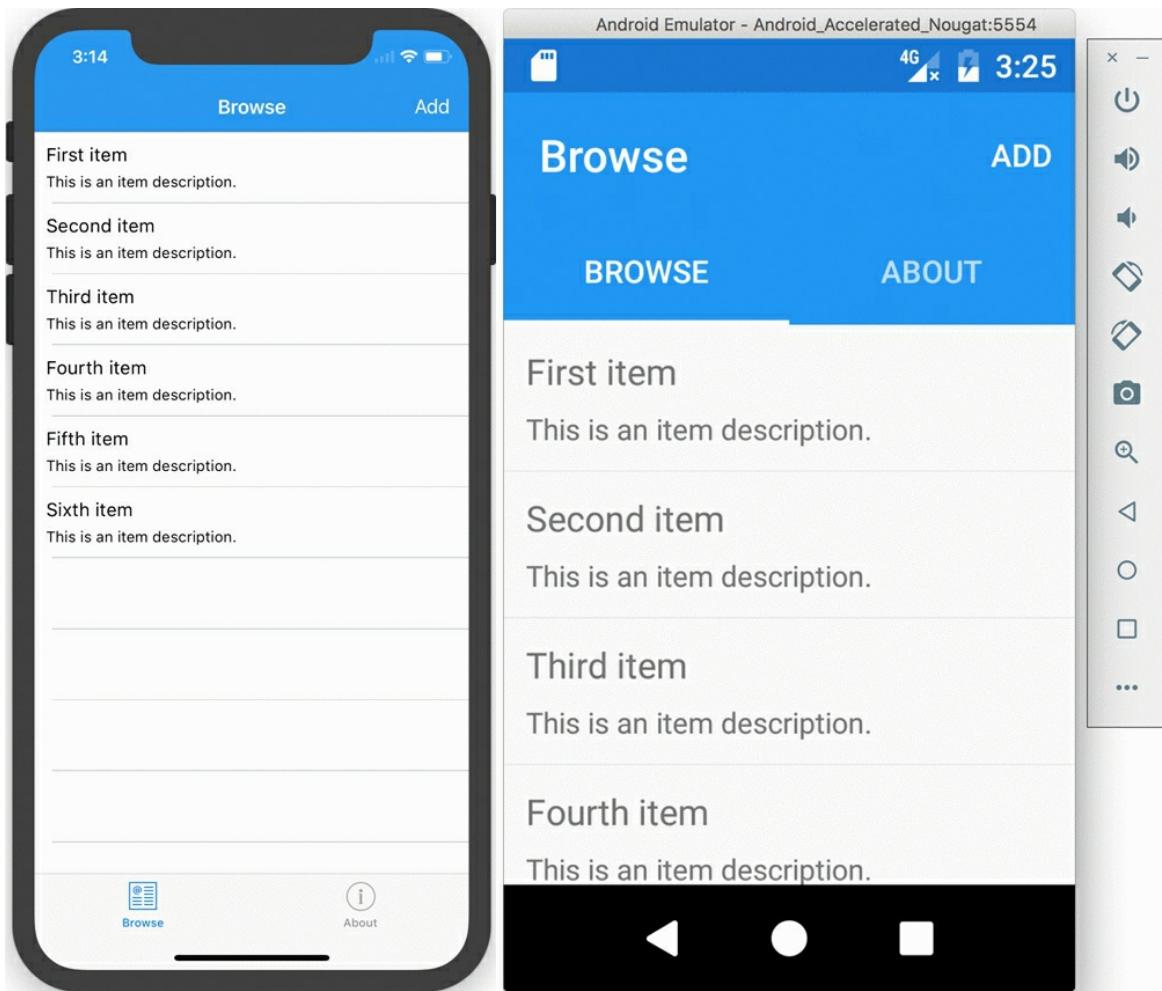


Figura 2.6: Aplicação funcionando em um emulador iOS e Android

É importante saber que, da mesma maneira que não é possível criar uma aplicação Windows em um Mac, não é possível em uma plataforma Windows criar uma aplicação iOS. Na realidade, é possível criar, mas realizar o build e executar esta aplicação é que não é possível. Estas situações são impostas pelas plataformas, que precisam fazer uso de recursos de seus sistemas operacionais para a construção de aplicações específicas a elas.

Entretanto, por meio de acesso remoto a uma máquina Mac, o trabalho de usar o Windows como plataforma para desenvolvimento se torna viável. Você precisará de um Mac conectado à mesma rede para que os testes em emuladores ou dispositivos iOS possam ocorrer. A situação de utilizar o Mac como plataforma para desenvolvimento de uma aplicação Windows não tem ainda recursos semelhantes.

Vamos realizar a configuração para desenvolvimento no Windows e configuração do Mac. Se você for desenvolver apenas para o Android, isso não se faz necessário. Ainda, se você for desenvolver no Mac, isso também não é necessário. Vamos lá. Em seu Mac, acesse as Preferências do sistema e, dentro delas, Compartilhamento , como pode ser visto na figura a seguir.

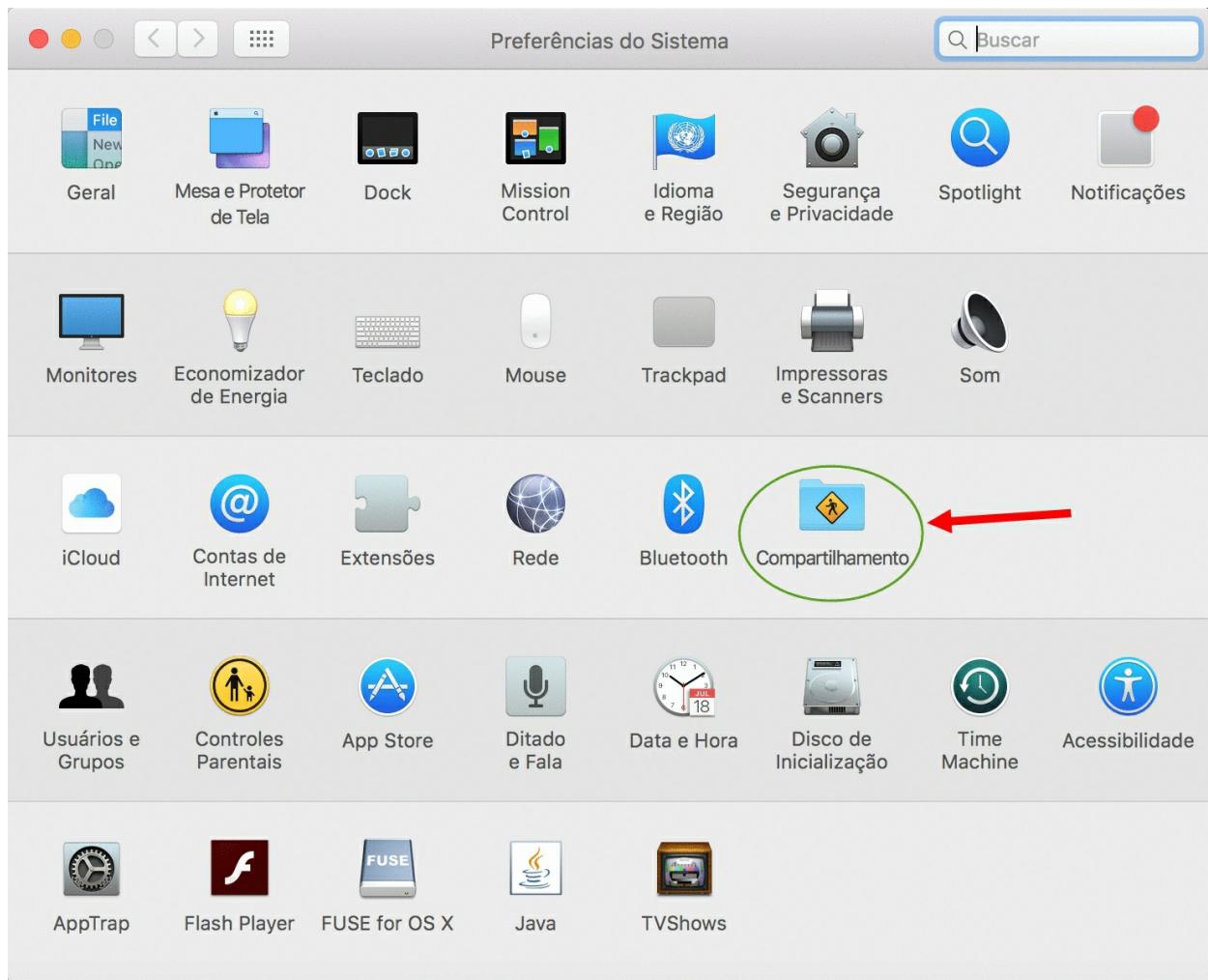


Figura 2.7: Acessando regras para compartilhamento em um Mac

Na janela que se abrir, ative o acesso remoto, tal qual é exibido na figura a seguir. Mantenha também a configuração para os usuários. Confirme essas alterações e vamos para a criação da aplicação.

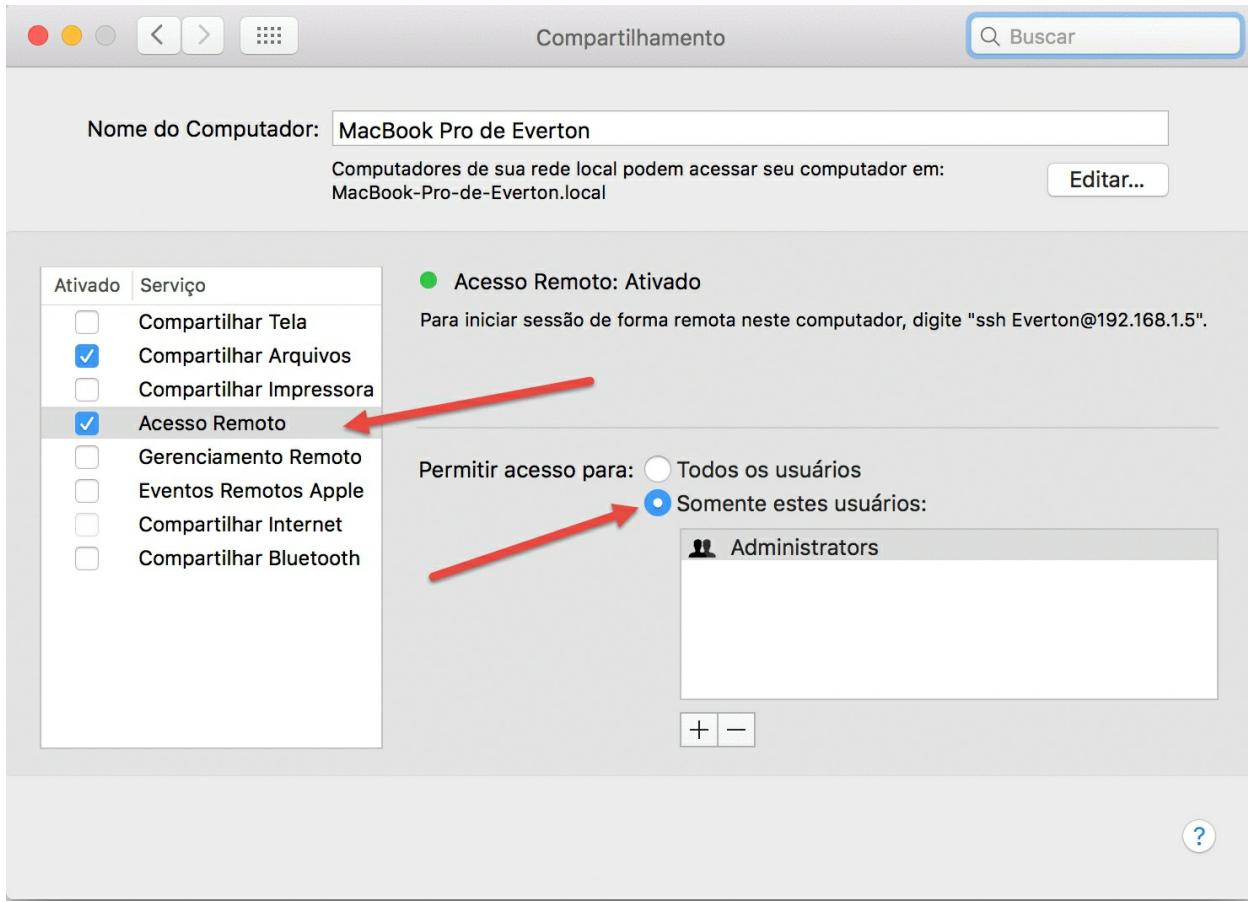


Figura 2.8: Configurando o acesso remoto

Abra agora o Visual Studio no Windows, pois já podemos criar nossa aplicação Mobile Cross-platform de teste, tal qual fizemos na plataforma Mac OS. Para isso, clique no menu Arquivo -> Novo -> Projeto . Na janela que se abre, escolha a linguagem Visual C# e, dentro dela, a categoria Cross-Platform.

Nos templates que são exibidos ao lado direito, selecione O Cross Platform App (Xamarin.Forms) , para que o Visual Studio crie uma aplicação com os requisitos mínimos que serão necessários para nosso teste. Agora eu usei, tanto para solução como projeto, o nome Capítulo02 . Importante: se você for criar seu projeto em uma estrutura de pastas com nomes compridos e níveis de pastas, pode ocorrer de, na compilação do projeto Android, ser acusado um problema relacionado a caminho muito longo. Isso ocorreu comigo, então fica a dica.

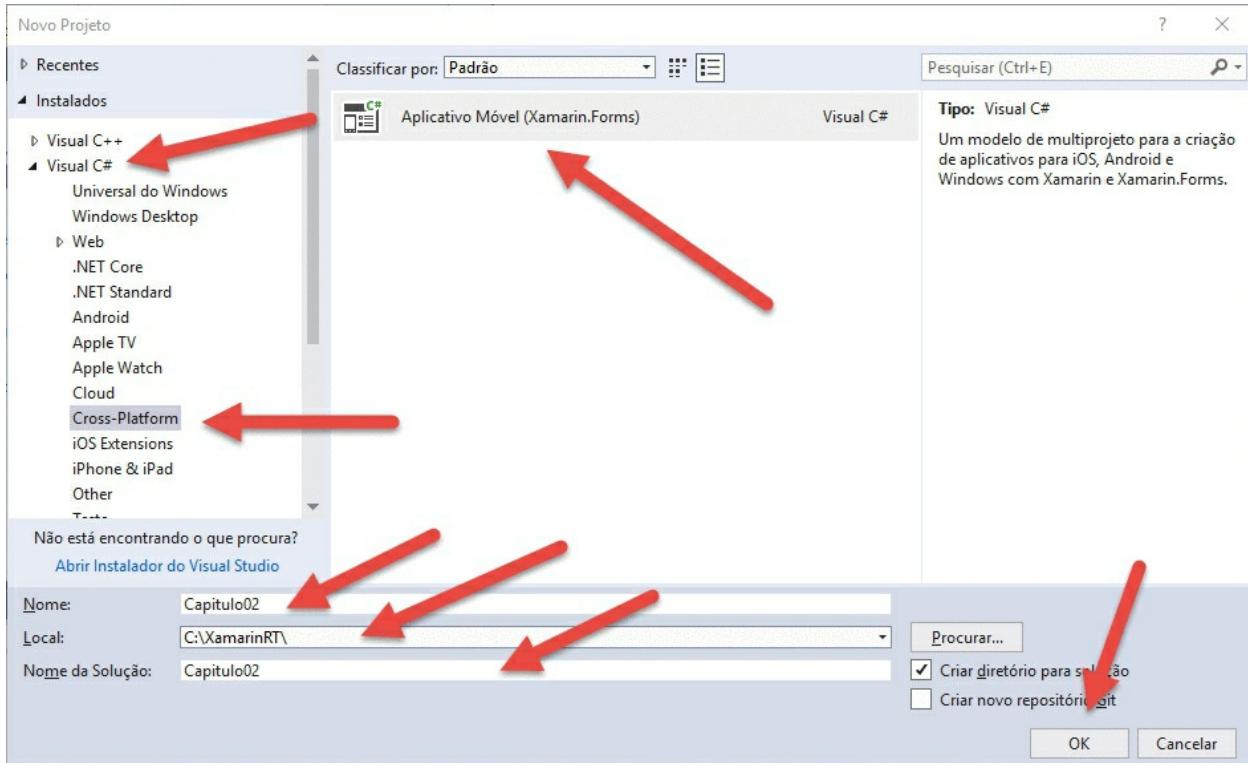


Figura 2.9: Criando um novo projeto pelo Visual Studio

Uma janela solicitando confirmações para a criação do projeto é exibida. Confirme as opções tal qual mostro na figura a seguir e clique em OK.

Escolhi a opção `Blank App` por ser mais simples para o propósito deste capítulo, que é uma navegação rápida pelo processo de desenvolvimento de uma aplicação. Observe, na figura, que a plataforma Windows (UWP) está desmarcada e a estratégia de compartilhamento de código é a .NET Standard. A PCL está em processo de depreciação, mas ainda é possível criar projetos com ela. Se você escolher o template `Master Detail`, uma opção para a criação de um servidor relacionado ao Azure é oferecida. Não trataremos esta opção no livro.

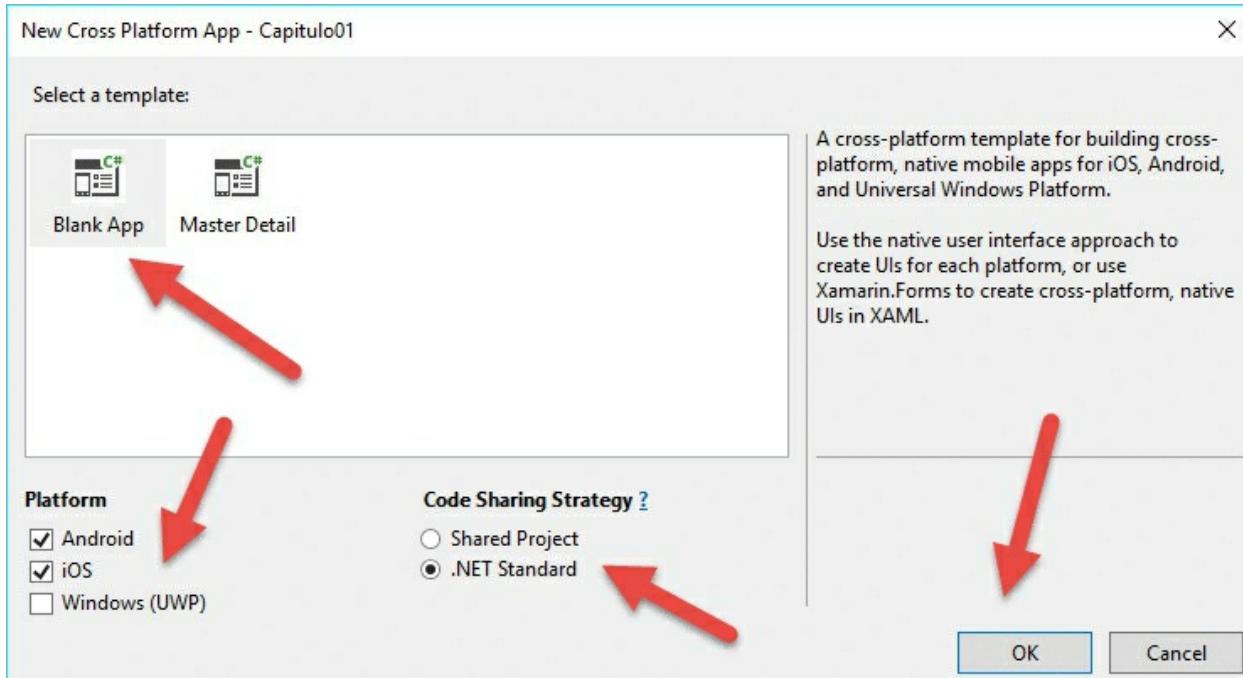


Figura 2.10: Confirmação do tipo de projeto

Na sequência, como será criado um projeto para a plataforma iOS, são apresentadas informações de como configurar a conexão do Visual Studio com seu Mac, mas isso só será necessário se você estiver desenvolvendo em Windows. Independentemente da plataforma utilizada para desenvolvimento, quando desejamos executar um projeto Xamarin Forms, precisamos escolher qual é a plataforma em que este projeto será executado. Podemos fazer isso clicando com o botão direito do mouse sobre o nome do projeto desejado para inicialização e, então, na opção Definir como projeto de inicialização. Lembre-se de que neste momento estamos com o foco no projeto iOS. Desta maneira, na sequência, na barra de tarefas, bem do lado direito, próximo ao final da janela, tem um botão para ativar o Xamarin Agent. Veja a figura a seguir. Clique neste botão após a definição do projeto iOS como inicialização.

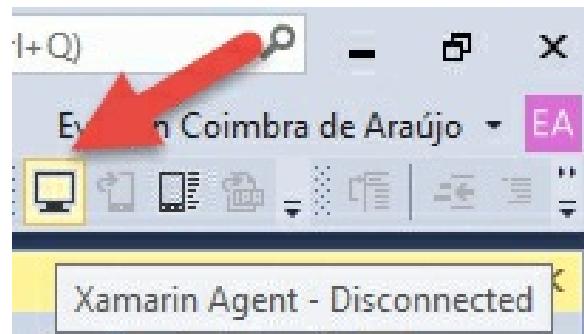


Figura 2.11: Orientações para conectar o Visual Studio com um Mac na rede

Uma nova janela fornecendo informações sobre o processo que será realizado é apresentada. Clique no botão Avançar. Se seu Mac estiver na mesma rede que sua máquina Windows, ele aparecerá na nova janela do Xamarin Agent. Clique no nome do computador e no botão conectar. Caso o Mac não apareça, você pode clicar no botão Adicionar Mac, na base esquerda da janela e digitar diretamente o IP dele.

Será solicitado a você o nome do usuário e senha (do Mac). Esta solicitação só será requisitada na primeira conexão com a máquina selecionada. O processo pode demorar um pouco, mas se tudo der certo, o computador aparecerá

como conectado e você pode clicar, então, no botão `Fechar`.

Caso o Xamarin.iOS do Visual Studio do Windows seja diferente do Xamarin.iOS do Visual Studio Mac, será preciso você atualizar as plataformas. No Windows, basta executar o instalador do Visual Studio e, no Mac, no menu Visual Studio Community, clicar no submenu `Check for Updates`. Este é um procedimento que recomendo que façam ao menos uma vez por mês, pois sempre há atualizações nos ambientes, mas fica a seu critério.

Se tudo estiver correto, com este processo realizado, um botão ao lado do que você clicou para a conexão é exibido. Ele executa o emulador para o iOS no seu Mac. A seleção de onde o projeto será executado é semelhante à apresentada para o Mac. Estamos usando a mesma ferramenta, apenas em plataformas diferentes. Veja a figura a seguir. O tempo levado para a inicialização do emulador no Mac e execução da aplicação é semelhante ao tempo levado no próprio Mac.



Figura 2.12: Escolhendo onde a aplicação será executada

Este processo foi mais demorado na máquina Windows do que no Mac, mas nada que assuste. Executar a aplicação no emulador no Mac, usando o Windows, é mais rápido que usar um emulador Android no Windows. Ao executar, você verá que a aplicação criada pelo template escolhido é mais simples do que a criada no Mac. Exibe apenas uma mensagem de boas-vindas. Uma aplicação semelhante à do Mac é criada quando se escolhe o template `Master-Detail`. O template `Master-Detail` do Visual Studio no Windows cria uma aplicação com algumas visões que podem ter navegação entre elas.

Muito bem, vamos agora falar um pouco sobre os testes no Android. Meu ambiente de testes e desenvolvimento em Windows é um i7, com 8GB de RAM. Fiz alguns testes e trago aqui os resultados e impressões sobre testar a aplicação criada pelo template em um emulador Android.

O primeiro passo foi definir o projeto Android como sendo o de inicialização. Lembra de como fazer? Botão direito do mouse no nome do projeto e, então, confirmar esta opção.

Para executar a aplicação no emulador do Android, eu selecionei o emulador `Android_Accelerated_Nougat (Android 7.1 - API 25)` e depois `Android_Accelerated_Oreo (Android 8.1 - API 27)`, que foram instalados e disponibilizados pelo Visual Studio. Em minha primeira execução, apenas cliquei no respectivo botão, na barra de ferramentas. O emulador da API 25 inicializou e deu um erro durante este processo, mas a aplicação chegou a ser implantada, entretanto, não foi executada no emulador, tendo uma mensagem de `timeout exibida`, ou seja, o processo demorou muito e foi identificado como travamento da aplicação. Isso levou 9 minutos. Executei novamente a aplicação, agora com o emulador já inicializado. A aplicação executou em 1 minuto. Já no emulador da API 27, o processo foi melhor e recomendo que você faça uso deste emulador daqui para a frente, ficando o anterior a ele como uma opção para testes, caso você assim deseje.

Com o processo narrado anteriormente, o que pude verificar foi que o Visual Studio levou um grande tempo para recuperação dos pacotes antes da compilação, inicialização do emulador, implantação e execução da aplicação nele. Isso fez com que a primeira execução não fosse bem-sucedida, pois, quando o emulador estava inicializado e a aplicação já havia sido implantada nele, o tempo foi relativamente bom.

Na barra de ferramentas do Visual Studio, existe também uma área relacionada aos emuladores Android, como pode ser visto na figura a seguir. O primeiro botão abrirá uma ferramenta do Android SDK, que é chamada Gerenciador de Emulador Android (AVD). Clique neste botão e a aplicação será aberta exibindo os emuladores que ela tem registrado em sua máquina.



Figura 2.13: Barra de ferramentas com destaque para o SDK Android

Em outro teste, com o emulador não inicializado, como se eu começasse novamente o processo para execução da aplicação Android, eu selecionei no AVD o emulador `Android_Accelerated_Nougat` (`Android 7.1 - API 25`), seguindo a mesma explicação dada anteriormente, e então cliquei no botão `start`. O emulador inicializou em 2 minutos. Depois, retornei ao Visual Studio e cliquei com o botão direito do mouse na opção `Implantar`. Este processo levou 5 minutos. Depois, apenas localizei a aplicação no emulador e a executei.

Você pode verificar, pelos comentários que fiz, que é custoso executar aplicações no emulador do Android. A recomendação sempre é deixar o emulador inicializado e, conforme for desenvolvendo, realizar a implantação e testá-las nele.

No Mac, executar uma aplicação no emulador Android não é tão custoso, mas demora mais do que o uso do emulador iOS. Mas, reforçando, a partir do **Capítulo 5** você aprenderá a testar aplicações diretamente em seu dispositivo e tenho certeza de que optará por isto, principalmente no caso do Android.

Vou repetir aqui um parágrafo que escrevi para o Visual Studio Mac anteriormente, pois o foco agora é Windows: na última atualização do Visual Studio, os projetos que eu tinha em Android deixaram de ser compilados, acusando a necessidade de um *runtime* para o SDK 8.1 do Android, que eu não tinha instalado com o Visual Studio. Foi preciso acessar o menu `Ferramentas->Android->Gerenciador do SDK do Android`, marcar o SDK 8.1 e clicar no botão para atualização. Este processo pode demorar um pouco, pois dependerá de seu acesso à internet. Se você instalou agora o Visual Studio, já com o SDK 8.1, certamente não terá estes problemas.

Após o download ter sido realizado, foi preciso realizar uma configuração no projeto Android. Desta maneira, clique com o botão direito sobre o projeto e depois na opção `Propriedades`. Ao lado esquerdo da janela que se abre, selecione a opção `Manifesto do Android` e, no lado direito, procure por `Versão do Android de Destino` e escolha `Android 8.1 (Nível de API 27 - Oreo)`. Para finalizar, foi preciso abrir externamente o arquivo do projeto e realizar esta mesma configuração, que, infelizmente não foi feita. Vamos lá então. Clique com o botão direito do mouse sobre o nome do projeto e, depois, na opção `Abrir pasta no Gerenciador de Arquivos`. Localize o arquivo `Capitulo02.Android.csproj` e, dentro dele, a tag `<TargetFrameworkVersion>`. Insira nela o valor `v8.1`. Isso funcionou em meu caso. Realize este procedimento apenas se o erro apontado anteriormente ocorreu para você. Este problema não ocorreu no Mac.

Reforçando, se você instalou o Visual Studio agora, com o SDK 8.1, este problema não ocorrerá para você.

## 2.3 Visualização das páginas diretamente no Visual Studio

Tanto o Visual Studio para Mac quanto o para Windows trazem consigo uma ferramenta de visualização das páginas XAML na própria IDE. Esta ferramenta viabiliza a visualização de suas visões, como uma prévia de como elas serão renderizadas nos emuladores e dispositivos.

Para isso, no Mac, no projeto chamado `Capitulo02` vamos expandir a pasta `views` e dar um duplo clique na visão `AboutPage.xaml`. Este projeto, que, como dito anteriormente, é nosso projeto Xamarin Forms, por ser um projeto do tipo `library` e não aplicação, como os das plataformas, pode ser referenciado por mim no livro simplesmente como `library`, ou projeto compartilhado, ok?

Com o arquivo aberto e exibido, no topo do editor de código, clique no botão `Preview`, escolha o dispositivo e a plataforma. Veja a figura a seguir.

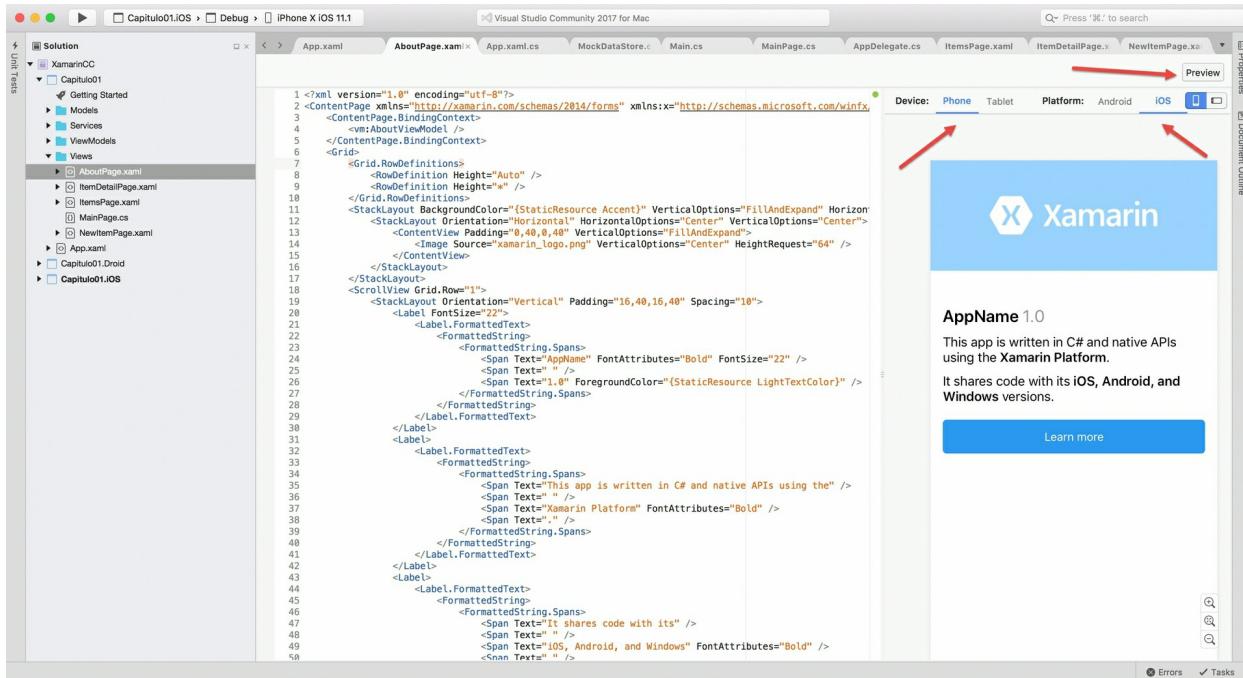


Figura 2.14: Xamarin Forms Preview - MAC

A visualização no Visual Studio Windows é um pouco diferente. É preciso abrir o visualizador via opção de menu. Selecione o menu **Exibir->Outras Janelas->Xamarin Forms Preview**. Para ter a visualização iOS, é preciso que haja a conexão com um Mac. Veja a figura a seguir.

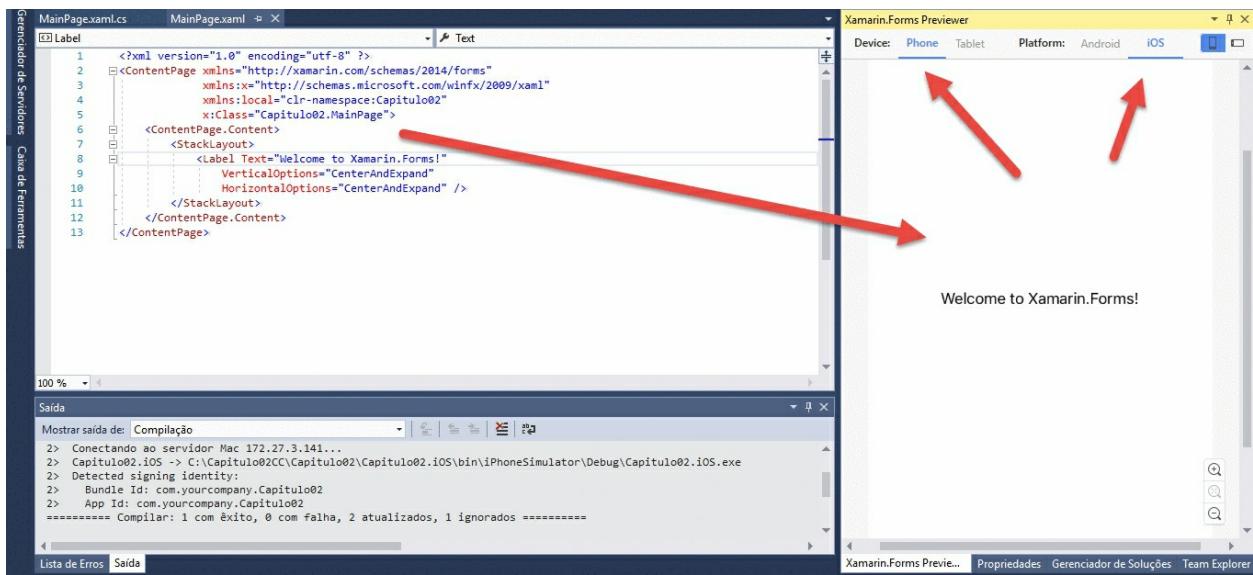


Figura 2.15: Xamarin Forms Preview - Windows

Caso a renderização não ocorra ou erros apareçam, pode ser necessário realizar o processo de limpeza da solução e a compilar novamente. Isso pode ser feito pelo menu de contexto da solução, que aparece quando clicarmos com o botão direito do mouse sobre ele.

## 2.4 Xamarin Live Player

A Microsoft trouxe para o Xamarin um player chamado `Xamarin Live Player`, que busca espelhar em seu dispositivo a aplicação em desenvolvimento no Visual Studio, auxiliando em um primeiro teste, antes de implantar a aplicação diretamente no dispositivo. No caso de dispositivos iOS, é possível que você execute sua aplicação sem a necessidade de realizar uma série de configurações para implantação, que veremos no [Capítulo 5](#).

O Live Player é um projeto ainda com status de laboratório, de testes. Não é um aplicativo concluído. Está sujeito a limitações e problemas, os quais ficam elencados no próprio aplicativo, quando ele é instalado em seu dispositivo. Eu o instalei a primeira vez em dezembro de 2017 e agora, na finalização do livro, junho de 2018, ele está mais robusto, com uma apresentação melhor e mais estável. Ou seja, é um projeto que vem crescendo e amadurecendo. Pode ser interessante você realizar alguns testes com suas aplicações.

A instalação do aplicativo em plataformas Android é bem simples, basta buscar por `Xamarin Live Player` na Google Store e instalá-lo. No iOS, até alguns dias era possível fazer este processo também pela App Store. Entretanto, como ele é um projeto ainda não distribuído em uma versão Release, ele foi tirado da loja da Apple, mas é possível instalá-lo como pré-release. Para isso, precisamos instalar antes, pela App Store, o `App TestFlight`, que é um aplicativo específico para estes tipos de projetos. Na sequência, veremos o procedimento para instalar o Live Player no iOS.

Com o `TestFlight` instalado, precisamos buscar um convite da Microsoft para podermos instalar o `Xamarin Live Player` dentro do `TestFlight`. Para isso, acesse <https://xamarinhq.wufoo.com/forms/live-player-alpha-program-agreement/>, preencha os formulários que forem exibidos e você receberá um e-mail com o convite. É preciso abrir este e-mail em seu dispositivo, para que ele reconheça e permita a instalação do player.

A primeira execução, tanto no iOS como Android, apresentará uma série de passos que você deve seguir para o correto funcionamento do aplicativo. Acho importante você ler todos. Depois da aplicação liberada, recomendo que dê uma navegada sobre as opções disponibilizadas, para se ambientar com o App.

Precisamos configurar agora nosso ambiente de desenvolvimento para reconhecer o `Xamarin Live Player` como um dispositivo, onde nossas aplicações poderão ser executadas. No Mac, dentro do Visual Studio, acesse o menu `Visual Studio Community` e então `Preferences`. Busque, ao lado esquerdo da janela, a opção `Xamarin Live Player (Preview)` e, do lado direito, marque o checkbox `Enable Xamarin Live Player`. No Windows, o procedimento é semelhante, mas devemos acessar o menu `Ferramentas->Opções` e, no lado esquerdo, buscar por `Xamarin->Outros`. Também é preciso marcar o checkbox de habilitação do `Xamarin Live Player`.

Para a execução de sua aplicação no Player, o dispositivo deve estar na mesma rede de sua máquina de desenvolvimento. Vamos ao teste. Acesse o App em seu dispositivo, seja iOS ou Android. No Visual Studio, tal qual escolhemos um dispositivo ou emulador para a execução de nossa aplicação, busque agora pela opção Live Player. Ela pode aparecer seguida do nome de seu dispositivo. Selecione a opção e teste seu aplicativo.

O Visual Studio solicitará um pareamento de sua máquina de desenvolvimento com o dispositivo. Este pareamento pode ser auxiliado por um QRCode, que é exibido pelo IDE. Funciona perfeitamente para desenvolvimento no Windows e Mac, tanto para iOS como para Android.

É para sua aplicação funcionar sem nenhum problema. Entretanto, se você estiver com um proxy rigoroso em sua rede, o pareamento pode não ocorrer. Um *workaround* se faz necessário e a Microsoft tem isso documentado em <https://docs.microsoft.com/pt-br/xamarin/tools/live-player/troubleshooting/>, que traz também explicações e soluções para outros tipos de erros que podem ocorrer. O App também possui limitações, que estão descritas em <https://docs.microsoft.com/pt-br/xamarin/tools/live-player/limitations?tabs=ios/>. É interessante que você faça uma leitura destes documentos.

A figura a seguir mostra, ao lado esquerdo, a execução do Xamarin Live no celular, e à direita a janela do Visual Studio. Basta clicar no link do celular e apontá-lo para o QRCode exibido na janela do Visual Studio, para que possa haver o pareamento.

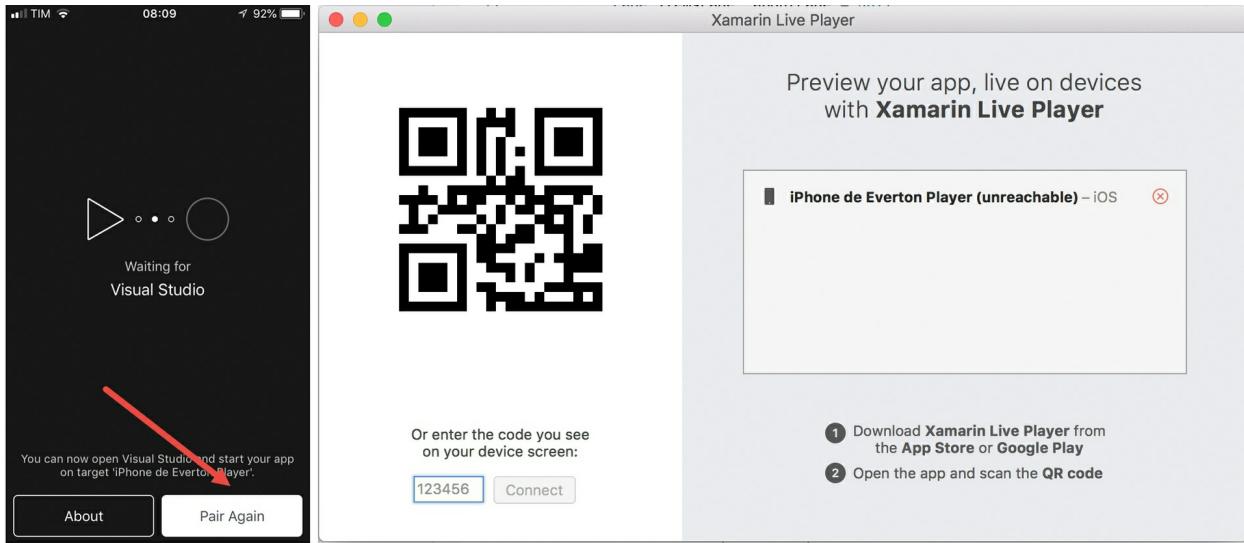


Figura 2.16: Xamarin Live

Na primeira experiência com este App eu não me senti motivado em utilizá-lo. Entretanto, sua evolução, nos 5 meses que me dediquei para a escrita deste livro, demonstra que ele se tornou uma ferramenta factível para testes, ainda que com limitações, as quais estão elencadas no link disponibilizado anteriormente. Para uma apresentação oficial a todas as características deste aplicativo, recomendo o acesso a <https://docs.microsoft.com/pt-br/xamarin/tools/live-player/>. Lembre-se de que o App ainda é *Preview*. Durante o desenvolvimento do livro, esta ferramenta não será utilizada, mas quis trazê-la para que você possa conhecê-la.

## 2.5 Conclusão

Chegamos ao final deste capítulo. Nele, apresentei como instalar o Visual Studio em sua máquina, seja Windows ou Mac. Criamos o projeto padrão para aplicações Xamarin Forms nas duas plataformas. Apresentei como visualizar páginas no próprio Visual Studio, sem necessidade de execução do projeto. Executamos a aplicação nos emuladores das duas plataformas, apresentei a visualização rápida das visões no próprio Visual Studio e introduzi o Xamarin Live Player.

O Xamarin Forms é um projeto em constante manutenção e evolução e está ficando mais estável. Desta maneira, é importante termos nossos projetos atualizados. Para isso, procure seguir as orientações que passei para manter seu ambiente de desenvolvimento e componentes atualizados.

No próximo capítulo, colocaremos a mão na massa. Criaremos nosso primeiro projeto, em que trabalharemos com os tipos de visões, layouts e alguns controles disponibilizados pelo Xamarin Forms. Será nossa introdução ao desenvolvimento com essa poderosa ferramenta.

## C APÍTULO 3

# Tipos de páginas, layouts e alguns controles para interação com o usuário

No capítulo anterior, vimos o processo de instalação e teste de uma aplicação básica criada pelo Visual Studio, tanto em Mac, quanto em Windows. Não entrei em detalhes sobre os recursos e controles utilizados na aplicação, pois eles serão tratados de maneira específica neste e nos próximos capítulos.

Criaremos uma aplicação que fará uso de alguns tipos de páginas disponibilizados pelo Xamarin e, dentro destas páginas, utilizaremos diferentes layouts para receber os controles visuais. Isso lhe permitirá identificar quando utilizar um tipo de página e/ou layout, sabendo qual o melhor para cada situação.

Os layouts são definidos para as páginas, que são as visões de uma aplicação. Dentro dos layouts é que são inseridos os controles visuais, que receberão e fornecerão dados e informações para o usuário da aplicação. Não usaremos o MVVM, por enquanto, pois o objetivo é uma ambientação aos recursos do Xamarin. A partir do próximo capítulo o MVVM estará em todas as nossas implementações. Desta maneira, capturaremos os eventos e atualizaremos a visão e modelo diretamente pelo *code-behind* da visão. Também trabalharemos a configuração visual de alguns controles, de acordo com o dispositivo em que a aplicação está sendo executada.

Antes de começarmos, precisamos ter uma noção sobre a anatomia de uma aplicação Xamarin Forms. É importante saber que todos os objetos Xamarin que aparecem em um dispositivo são chamados de elementos visuais, e são divididos em três categorias: `page` , `layout` e `view` .

Estes elementos não são abstratos. A API do Xamarin Forms define classes chamadas `visualElement` , `Page` , `Layout` e `view` . Estas classes e suas descendentes formam a espinha dorsal da interface com o usuário do Xamarin Forms. Um objeto `visualElement` é qualquer coisa que ocupa espaço na tela.

Uma aplicação Xamarin Forms consiste, normalmente, de uma ou mais páginas. Uma página geralmente ocupa toda a área da tela do dispositivo. Um tipo de página muito utilizada é a página de conteúdo, que é representada pela classe `ContentPage` .

Os componentes visuais são organizados, em cada página, em uma hierarquia pai-filho. O filho de uma página de conteúdo é, geralmente, um layout, que organiza os componentes visuais na tela. Alguns layouts possuem um único filho, mas existem layouts que podem possuir múltiplos filhos. Estes filhos podem ser outros layouts ou views.

O termo `view` para o Xamarin Forms refere-se a tipos de objetos de apresentação ou interação. São normalmente chamados de controles ou *widgets* em outros ambientes de programação.

## 3.1 ContentPage e o Stacklayout

Quando vamos criar uma aplicação, quer seja ela grande ou não, precisamos pensar em camadas. E quando pensamos em camadas, podemos pensar em alguns princípios, como coesão e acoplamento, que nos levam a separar nossa aplicação em projetos. Outra estratégia, e que vou utilizar, é ter estas camadas organizadas em pastas. É claro que a camada de modelo que vou implementar ficará atrelada a este projeto, mas para o que proponho no momento, isso não será problema.

Os exemplos criados deste capítulo em diante podem ser implementados tanto no Windows como no Mac, pois farei uso do Visual Studio, como foi visto no capítulo passado. Sendo assim, vamos lá. Crie um novo projeto e dê a ele um nome. Eu o nomeei de `capitulo03` . Vamos ao passo a passo?

Para criar o projeto no Windows, selecione Arquivo->Novo-Projeto e, na categoria `visual C#` , escolha `Cross-Platform` . Ao centro, escolha o template `Cross Platform App (Xamarin)` e, então, `Aplicativo móvel (Xamarin.Forms)` . Na janela que se abre, confirme `Aplicativo em Branco` e desmarque `Windows (UWP)` , deixando selecionada a opção `.NET Standard` como

estratégia para compartilhamento de código.

Se você estiver utilizando um Mac, selecione `File->New Solution`, depois `Multiplatform->App e Blank Form App em Xamarin.Forms`. Lembre-se de marcar o `.NET Standard` como opção de compartilhamento de código e deixe marcada a opção `Use XAML for user interface files`.

Meu objetivo aqui, nesta seção, é criar visões que me permitam apresentar algumas opções de layout, fazendo uso de páginas que têm como objetivo exibir conteúdos de maneira simplificada, que são tipificadas como `ContentPage`.

Vamos criar nossa visão (pode ser chamada de página ou formulário também). No projeto, crie uma nova pasta, chamada `views`. Com a pasta criada, clique com o botão direito sobre ela e então em `Adicionar->Novo item`. Selecione, na janela apresentada, a categoria `Xamarin.Forms` e depois o template `Content Page`. No Mac, ao clicar com o botão direito, selecione `Add->New File`, categoria `Forms` e o template `Forms ContentPage.xaml`. Eu nomeei meu arquivo de `ContentPageView.xaml`. Observe que estou adotando como sufixo para o nome das visões a palavra `view`.

Neste livro, eu optei em criar toda a interface com o usuário fazendo uso de arquivos XAML, que é uma linguagem específica para esta finalidade. É possível realizar a criação de todos os controles, expostos em elementos XAML, via código C#, pois cada elemento diz respeito a uma classe. As orientações que dei para a criação deste arquivo são específicas para o XAML. Entretanto, você deve ter verificado nos templates disponibilizados a possibilidade de criar páginas via código C#. Sempre que for preciso iterar, via código C#, com a camada de visão, destacarei no texto.

A listagem a seguir apresenta o arquivo criado pelo template. Observe que o elemento mais externo é o `<ContentPage>`, que define o tipo da página em questão. Se você verificar a classe por trás desta visão, que, como visto no capítulo anterior, chamamos de code-behind, ela estende de `ContentPage`. Você pode ir para o code-behind expandindo a visão e realizando duplo-clique no nome do arquivo `.cs`.

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

    x:Class = "Capitulo03.Views.ContentPageView" >

    <ContentPage.Content>
        <StackLayout>
            <Label Text = "Welcome to Xamarin.Forms!">
                VerticalOptions = "CenterAndExpand"
            </Label>
            <Label Text = "Welcome to Xamarin.Forms!">
                HorizontalOptions = "CenterAndExpand" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Uma página do tipo `ContentPage` , de acordo com a documentação do Xamarin, é uma página que exibe uma simples visão. Esta visão deve estar definida no elemento `Content` , declarado no código anterior como `<ContentPage.Content>` .

Os conteúdos de uma página precisam ser ajustados para que a aplicação possa expor e solicitar, de maneira organizada, as informações ao seu usuário. Esta organização pode ser realizada por meio de `Layouts` .

Ainda no código anterior, verifique o uso do elemento `<StackLayout>` . Sua característica é a de empilhar os componentes (ou controles) que estejam contidos dentro dele. O empilhamento pode ser vertical, no qual um elemento fica sobre o outro (de cima para baixo), ou horizontal, no qual um elemento fica ao lado do outro (da esquerda para a direita). O comportamento padrão é o vertical. Vamos desenhar nossa primeira visão. Ela deverá ficar tal qual a figura a seguir, sendo a primeira imagem no iOS, e a segunda, no Android.

Nesta visão eu inseri um cabeçalho para a página, que tem uma imagem com textos à sua direita, e abaixo deles, uma linha. Como corpo da página temos três campos que solicitam dados ao usuário. Antes de cada campo existe um texto que o descreve. Abaixo dos campos, um novo texto, orientativo e com uma palavra em negrito.

Finalizando a página, temos um botão. Nenhum comportamento foi implementado para este botão, pois o foco agora é o layout. Os layouts que eu apresentar sempre focarão, com mais precisão, no iOS, mas orientarei como configurar propriedades de elementos visuais de acordo com a plataforma em que a aplicação está executando. A escolha se deu com base na qualidade dos emuladores que tenho à disposição para executar os testes. Entretanto, a partir do capítulo 5 você aprenderá como executar as aplicações no dispositivo, então tudo ficará mais tranquilo. Infelizmente o emulador padrão do Android não é tão bom, mas você pode instalar e configurar emuladores específicos.

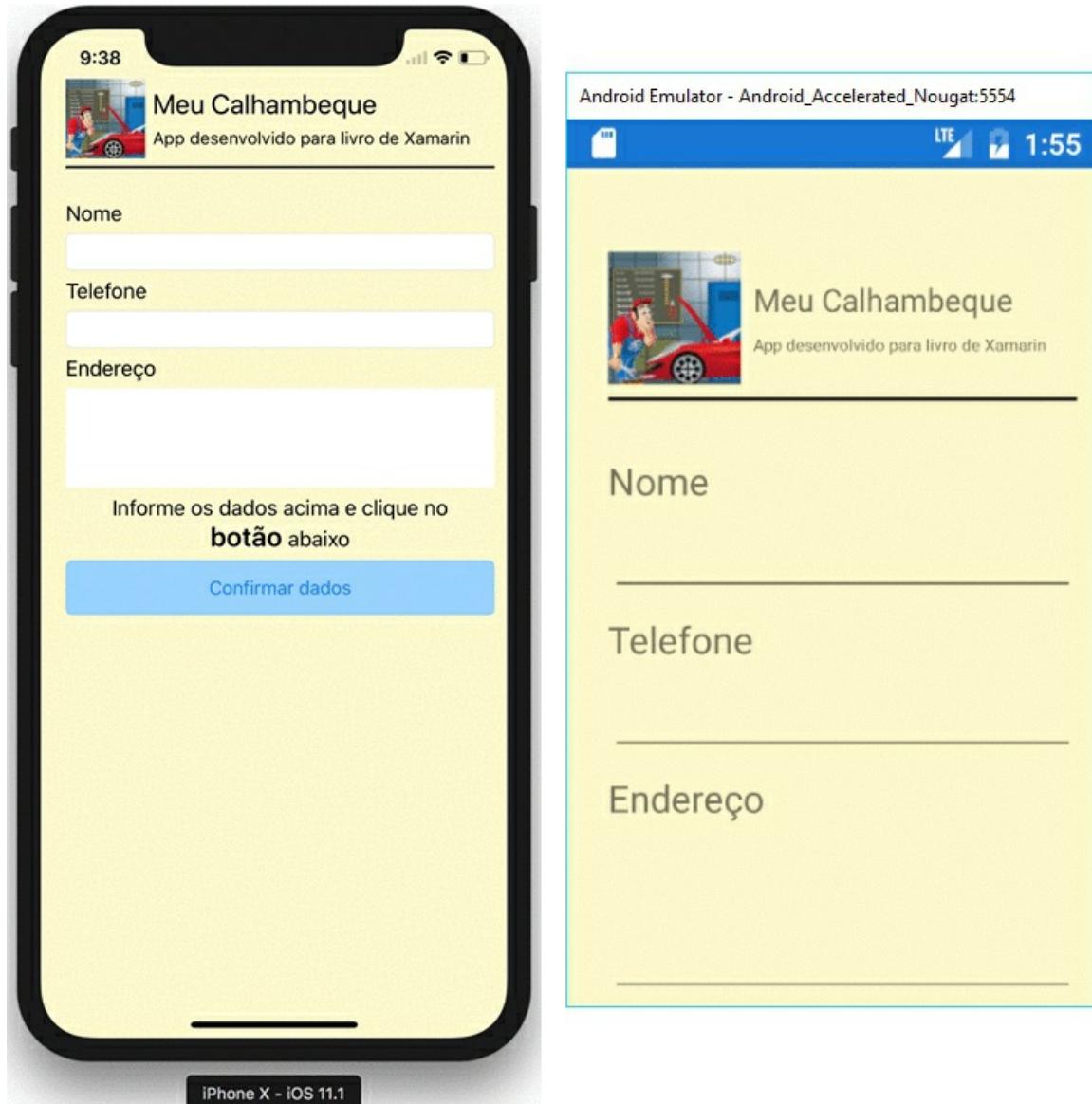


Figura 3.1: Visão fazendo uso do tipo ContentPage

Para desenhar a visão, fiz uso do `<StackLayout>` como organizador para os controles, do elemento `<Image>` para a figura, de vários `<Label>`s para os textos, de `<Entry>` para a entrada de dados simples, de `<Editor>` para a entrada do endereço e de `<Button>` para o botão ao final da página. Na sequência vamos analisar o código de maneira particionada.

O primeiro elemento dentro do `<ContentPage.Content>` é um `<ScrollView>`. Note na figura mostrada anteriormente como ficou a visão no simulador do Android. Não coube tudo em uma única janela do dispositivo. Tem controles abaixo, fora da área de exibição. Com `<ScrollView>`, ao manter pressionada a janela (com o dedo, ou mouse no emulador), o usuário pode fazer "correr" a visão, para baixo e para cima, exibindo os controles que estejam ocultos. Quando executarmos a aplicação em um dispositivo Android, a resolução será de acordo com a qualidade oferecida pelo seu dispositivo. Veremos isso no capítulo 5.

Dentro do `<ScrollView>` temos um `<StackLayout>` que será o contêiner de todos os controles visuais. O parâmetro `FillAndExpand` utilizado na propriedade `HorizontalOptions` define que este controle ocupará toda a largura da tela do dispositivo. Existem outras opções, como início (`Start`), final (`End`) e centro (`Center`). Todas têm a opção de

expansão (`AndExpand`) e são também oferecidas para a propriedade `VerticalOptions`, que trata da disposição do `<StackLayout>` em relação à ocupação da altura da tela do dispositivo. A propriedade `BackgroundColor` permite a definição da cor de fundo para o controle, que pode ser uma constante literal disponibilizada ou um valor em hexadecimal. A maioria das propriedades pode ter seus valores possíveis exibidos pelo pressionamento das teclas `CTRL+Espaço`. Finalizando este trecho de código, temos o `Padding`, que define o quanto o controle se distancia de seus elementos filhos (esquerda, topo, direita, base). Existe outra propriedade, chamada `Margin`, que trata a distância adjacente entre os controles.

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class = "Capitulo03.Views.ContentPageView"

    Title = "Clientes" >

    <ContentPage.Content>
        <ScrollView>

            <StackLayout Padding = "20, 50, 10, 10" BackgroundColor = "#FFFACD" HorizontalOptions = "FillAndExpand" >
```

Dentro do `<StackLayout>` principal, está definido um novo, que você pode abstrair como um retângulo, onde os objetos (controles) serão dispostos um ao lado do outro. Veja a propriedade `orientation` que está definida como `Horizontal`. Dentro do controle, temos um `<Image>` responsável por exibir a logomarca da oficina. O arquivo identificado nesta propriedade precisa estar na pasta de recursos dos projetos de cada plataforma, `Resources` no iOS e `Resources/drawable` no Android. Verticalmente a imagem ficará no topo do controle e usamos a `HeightRequest` para definir a altura desejada para o controle. Em seguida, ao lado direito da imagem, é inserido um novo `<StackLayout>` (um retângulo dentro do outro). Este novo elemento ficará centralizado verticalmente e dentro dele temos dois `<Labels>`. Verifique no código que os tamanhos das fontes são dependentes da plataforma em que o aplicativo está sendo executado.

```
        <StackLayout Orientation = "Horizontal" >

            <Image Source = "logo.png" VerticalOptions = "Start" HeightRequest = "64" />

            <StackLayout VerticalOptions = "Center" >

                <Label Text = "Meu Calhambeque" >

                    <Label.FontSize>

                        <OnPlatform x:TypeArguments = "x:Double" >

                            <On Platform = "iOS" > 30 </On>
```

```

<On Platform = "Android" > 15 </On>

</OnPlatform>

</Label.FontSize>

</Label>

<Label Text = "App desenvolvido para livro de Xamarin" >

<Label.FontSize>

<OnPlatform x:TypeArguments = "x:Double" >

<On Platform = "iOS" > 15 </On>

<On Platform = "Android" > 8 </On>

</OnPlatform>

</Label.FontSize>

</Label>

</StackLayout>

</StackLayout>

```

É importante trazer para nosso conhecimento uma informação sobre o uso de `<Image>`. Os arquivos de imagem possuem uma propriedade chamada `BuildAction`, que configura o comportamento que o processo de construção da aplicação deverá utilizar para cada arquivo. Clique com o botão direito sobre o nome do arquivo e em `Propriedades`, para acessá-la. Para projetos iOS, esta propriedade deve estar configurada como `BundleResource` e, para o Android, como `AndroidResource`. Desta maneira, se você referenciar alguma imagem e ela não for exibida em sua aplicação, você pode verificar inicialmente o valor desta propriedade para o arquivo em questão.

Voltemos ao código. A terceira e última parte do código começa inserindo um `<BoxView>` com a finalidade de traçar uma linha em preto, separando o cabeçalho do corpo da visão. Em seguida, um novo `<stackLayout>` é criado para abrigar os controles de interação com o usuário. Veja que deixei, para fins de exemplo, os labels com definição de tamanho de fonte na própria tag. O elemento `<Entry>` renderiza uma caixa simples para textos, como um campo. Observe, no `<Entry>` que solicita o telefone que estamos especificando um teclado para ele, por meio da propriedade `Keyboard`.

Os valores possíveis são: `Default`, `Text`, `Chat`, `Url`, `Email`, `Telephone` e `Numeric`. Para o endereço, como ele pode ter um valor que precise de mais de uma linha, eu fiz uso do controle `<Editor>`, que tem sua altura desejada especificada na já conhecida propriedade `HeightRequest`. No último `<Label>`, que antecede o `<Button>`, estamos configurando a palavra "botão" para ser exibida em negrito. Para isso, trabalhamos com subelementos do `<Label>`, os `Spans`, que permitem configurações particionadas para o texto que se deseja exibir na visão. Por fim, o `<Button>`, com uma cor específica para `BackgroundColor` e, por enquanto, sem comportamento.

```
<BoxView BackgroundColor = "Black" HorizontalOptions = "FillAndExpand" HeightRequest = "2" />
```

```

<StackLayout Padding = "0, 5, 0, 0" HorizontalOptions = "FillAndExpand" >

    <Label Text = "Nome" FontSize = "Medium" />

    <Entry FontSize = "Small" />

    <Label Text = "Telefone" FontSize = "Medium" />

    <Entry Keyboard = "Telephone" FontSize = "Small" />

    <Label Text = "Endereço" FontSize = "Medium" />

    <Editor FontSize = "Small" HeightRequest = "80" />

    <Label HorizontalTextAlignment = "Center" >

        <Label.FormattedText>

            <FormattedString>

                <FormattedString.Spans>

                    <Span Text = "Informe os dados acima e clique no " />

                    <Span Text = "botão" FontSize = "22" FontAttributes = "Bold" />

                    <Span Text = " abaixo" />

                </FormattedString.Spans>

            </FormattedString>

        </Label.FormattedText>

    </Label>

    <Button Text = "Confirmar dados" BackgroundColor = "#96d1ff" />

```

Apenas para fechamento do código, verifique na sequência as tags específicas.

```

        </StackLayout>

    </StackLayout>

```

```
</ScrollView>

</ContentPage.Content>

</ContentPage>
```

Um detalhe positivo para o Xamarin e o Visual Studio: usei o tempo todo a janela `xamarin Forms Preview` quando o projeto de inicialização era o iOS e funcionou muito bem. Em relação ao Android, não consegui usar com o mesmo sucesso, pois a renderização demorava muito. Mas você pode sempre pensar em fazer utilizar `xamarin Live Player`, apresentado no capítulo anterior.

Vamos testar sua aplicação? Precisamos alterar o código da classe `App`, na raiz do projeto, pois ela está executando a `MainPage`, que é criada por padrão, mesmo no template vazio. Desta maneira, no construtor da classe, ajuste o código da `MainPage` para `MainPage = new ContentPageView();`. Você pode excluir a visão `MainPage`, que está na raiz do projeto. Note a diferença do que utilizamos no projeto do capítulo anterior. Agora não teremos, por enquanto, uma navegação entre visões, então esta declaração basta para as duas plataformas.

A visão criada nesta seção fez uso de controles e recursos com o objetivo de introduzi-los a você. Não faz parte do escopo deste livro trabalhar design das visões que criaremos, mas veremos aqui alguns recursos possíveis. Podemos criar alguns estilos no arquivo `App.xaml` e reutilizá-los em visões de nossa aplicação. Dentro da tag `<Application.Resources>`, vamos inserir o código da sequência. Estamos definindo alguns tamanhos para fontes dos controles de nossa visão.

```
<ResourceDictionary>
```

```
<OnPlatform x:Key = "LabelFontSize" x:TypeArguments = "x:Double" iOS = "12" Android = "12" />

<OnPlatform x:Key = "SpacingStackLayout" x:TypeArguments = "x:Double" iOS = "6" Android = "2" />

<OnPlatform x:Key = "EntryFontSize" x:TypeArguments = "x:Double" iOS = "12" Android = "10" />

<OnPlatform x:Key = "HeightEditor" x:TypeArguments = "x:Double" iOS = "80" Android = "45" />

</ResourceDictionary>
```

Agora, faremos uso destes recursos em nossa visão `ContentPageView`. Na sequência, apresento as tags que precisamos mudar, localize-as em seu XAML. O `<StackLayout>` é o que está logo abaixo do que desenha a linha separatória do título da visão. Os recursos estão nas propriedades ligadas com `staticResource`. Teste sua aplicação novamente e veja a alteração em execução.

```
<StackLayout Spacing = "{StaticResource SpacingStackLayout}" Padding = "0, 5, 0, 0" HorizontalOptions = "FillAndExpand" >

<Label Text = "Nome" FontSize = "{StaticResource LabelFontSize}" />

<Entry FontSize = "{StaticResource EntryFontSize}" />

<Label Text = "Telefone" FontSize = "{StaticResource LabelFontSize}" />

<Entry Keyboard = "Telephone" FontSize = "{StaticResource EntryFontSize}" />
```

```

<Label Text = "Endereço" FontSize = "{StaticResource LabelFontSize}" />

<Editor FontSize = "{StaticResource EntryFontSize}" HeightRequest = "{StaticResource HeightEditor}" />

```

Para verificar como o layout da aplicação ficava no Android, testei o deploy em um S7 e a visão ficou bem parecida com a do iPhone, o que comprova o layout ruim dos emuladores padrões disponibilizados para o desenvolvimento para o Android. Reforço que no capítulo 5 aprenderemos a realizar o deploy diretamente nos dispositivos e, então, você poderá realizar todos os testes em seu aparelho. Novamente, fica a recomendação de tentar testar a execução no `Xamarin Live Player`.

### Especificidades do iOS 11 e do iPhone X

Vamos fazer um teste na aplicação iOS. Altere o `Padding` de seu `<StackLayout>` no exemplo anterior. Deixe apenas `Padding="20, 0, 10, 0"` e execute sua aplicação selecionando o iPhone X como emulador. Você verá que o cabeçalho aparece logo no topo da tela do emulador do dispositivo e, devido ao layout do iPhone X, parte do conteúdo fica abaixo do detalhe do aparelho. Se tivéssemos uma página completamente preenchida, o mesmo ocorreria na base, no componente de troca de aplicativos. Você pode desejar que sua página seja desenhada apenas onde ela possa ser completamente exibida. Para isso, veja o código apresentado na sequência para a classe `App`.

Observe os `usings`. Note a chamada ao método `On()`, que garante que a instrução seja executada apenas no iOS. Encadeado a este método, está sendo invocado o `SetUseSafeArea()` para garantir que a página ocupe apenas o espaço em que não há exibição ou componentes do dispositivo.

```

namespace Capitulo03
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();

            var page = new ContentPageView();
            page.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetUseSafeArea( true );
            MainPage = page;
        }
    }
}
// Código omitido

```

Teste sua aplicação e comprove a diferença. Fica a seu critério o uso ou não desta característica. Se você quiser, pode utilizar isso diretamente em seu XAML, tal qual apresento na sequência. Verifique as duas últimas linhas, que definem o namespace a ser utilizado e a habilitação da área segura para uso de sua página. Para testar, comente a instrução do método `On()`, apresentada na listagem anterior.

```

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

```

```

x:Class = "Capitulo03.Views.NavigationPageClientes"

xmlns:cabecalho = "clr-namespace:Capitulo03.Views.ContentViews"

Title = "Clientes"

xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"

ios:Page.UseSafeArea = "true" >

<!-- Código omitido -->

```

Tente mudar a orientação de seu dispositivo para vertical, e verá que o comportamento se mantém o mesmo. Sem este recurso, você precisa trabalhar com os `Paddings` sempre que ocorrer a mudança de orientação.

Podemos alterar a cor de fundo da página, pois com o uso de `SafeArea` temos uma parte da página fora do `<StackLayout>` e que deve estar em branco. No final do XAML anterior, referente à tag `<ContentPage>`, insira a atribuição `BackgroundColor="Red"`, podendo utilizar a cor que desejar, inclusive em hexadecimal. O ideal é a mesma cor do elemento que preenche toda a página. Se você quiser mudar a cor no código C# anteriormente apresentado, basta digitar `page.BackgroundColor = Color.Red;` .

**A TENÇÃO :** nesta última seção comentei sobre a cláusula `using` no início das classes C#. Elas trazem para o contexto do código do arquivo em edição os namespaces necessários na implementação. Se você optar por não importar os namespaces, será preciso utilizar o nome qualificado da classe, ou seja, com os namespaces precedendo o nome. O Visual Studio pode lhe auxiliar nesta importação, basta que, ao começar a digitação do nome da classe, você pressione `CTRL+.` e então escolha o namespace a ser importado. É preciso apenas ter bastante atenção para a importação correta do namespace, pois pode ocorrer de termos classes com mesmo nome, mas com namespaces diferentes. Você verá que, a partir do próximo capítulo, teremos essa situação. Com essa explicação dada, deixarei de exibir os `using` nos códigos.

## 3.2 TabbedPage e o Grid como layout

O exemplo que criamos no capítulo anterior fez uso do `TabbedPage` como tipo de páginas para as visões e o implementamos via código C#. Nesta seção veremos como criar este tipo de página fazendo uso do XAML. Também trabalharemos outro tipo de layout para organização dos controles visuais na página, o `Grid` .

O Visual Studio para Windows oferece um template para a criação de um `TabbedPage` e podemos utilizá-lo tal qual fizemos anteriormente (botão direito na pasta `Views` e depois em `Adicionar -> Novo Item -> Xamarin.Forms -> TabbedPage` ). Eu nomeei minha página de `TabPageView` e ela foi criada com o código XAML apresentado na sequência. O código é bem simples e de fácil compreensão. Observe que o primeiro elemento agora é `TabbedPage` e, verificando o code-behind do XAML, podemos ver que há a extensão da classe de mesmo nome. Se você estiver usando o Mac, precisará criar um `ContentPage` e mudar as tags para `TabbedPage` . Também será necessário alterar a classe estendida no code-behind.

```

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

<TabbedPage xmlns = "http://xamarin.com/schemas/2014/forms"

```

```
xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

x:Class = "Capitulo03.Views.TabbedPageView" >

<ContentPage Title = "Tab 1" />

<ContentPage Title = "Tab 2" />

<ContentPage Title = "Tab 3" />

</TabbedPage>
```

Uma visão do tipo `TabbedPage` cria abas que permitem visualizar informações categorizadas. Em nosso exemplo teremos duas guias, uma com a página de clientes, que criamos anteriormente e outra com o conteúdo relacionado a serviços ofertados pela oficina. A figura a seguir apresenta como deverá ficar sua visão depois de concluída.

Observe, no iOS, que as guias disponíveis podem ser acessadas por botões na base da tela do emulador, enquanto no Android esta barra aparece no topo da tela. No iOS, os ícones não aparecem coloridos, como ocorre no Android.

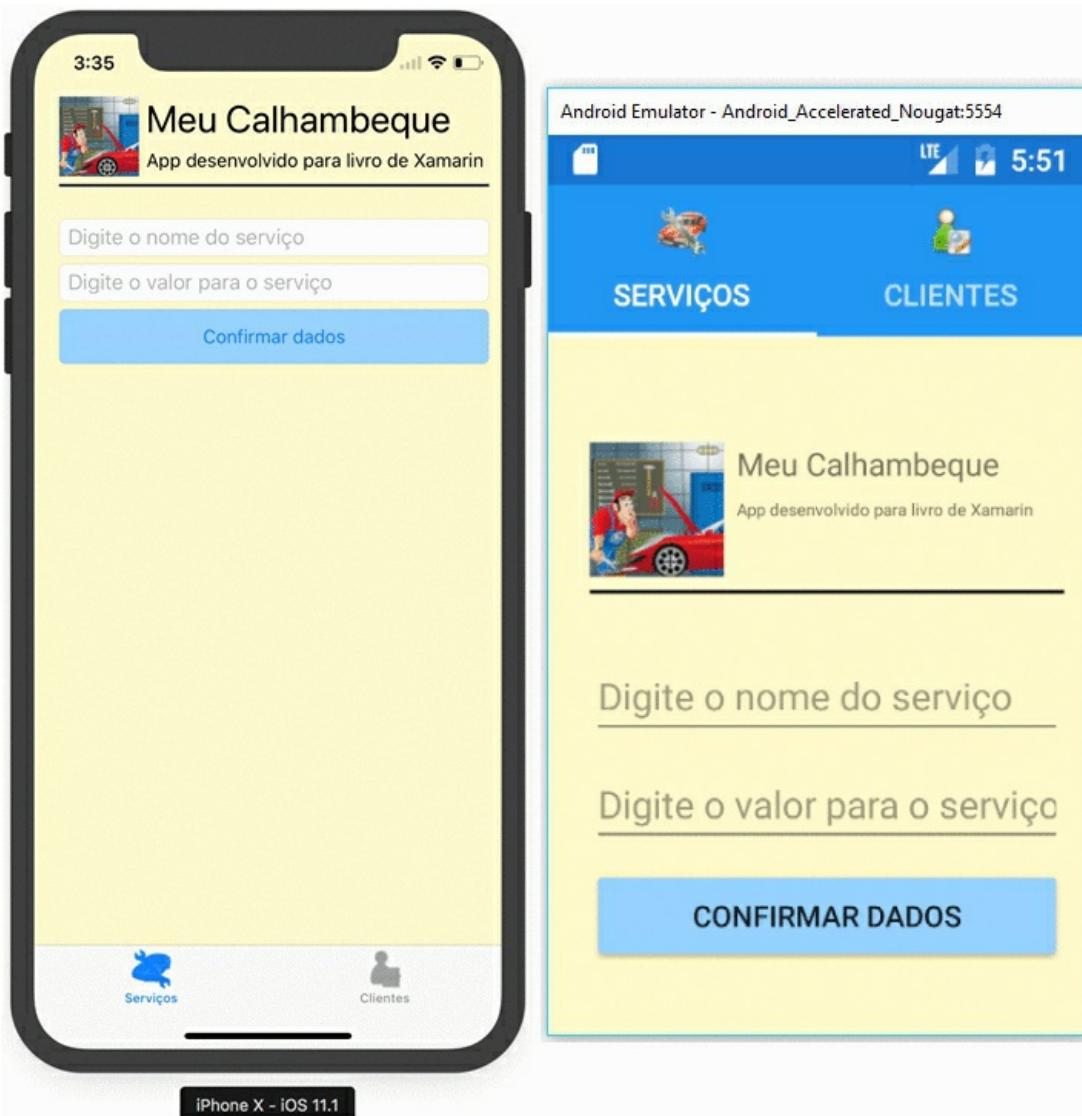


Figura 3.2: Visão fazendo uso do tipo TabbedPage

O código implementado em XAML para essa visão pode ser verificado na sequência. Seguindo o exemplo anterior, a apresentação do código será particionada, buscando facilitar a compreensão.

No final do elemento `<TabbedPage>`, verifique a declaração `<xm1ns:pages>`. Esta instrução traz para a visão atual todas as visões disponibilizadas no namespace `capítulo03.views`. Este código é necessário aqui para que possamos reutilizar a página que criamos para clientes nesta visão, sem ser necessário codificá-la novamente. Essa é uma ideia de componentização. Logo você verá como este reuso está implementado nessa visão.

Na sequência, está a definição de um `<ContentPage>` para `Serviços`, onde são configurados o título e ícone para o botão que será responsável pela primeira guia. É esta que será exibida por padrão quando a página for requisitada.

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

```
<TabbedPage xmlns = "http://xamarin.com/schemas/2014/forms"
```

```
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
```

```

x:Class = "Capitulo03.Views.TabbedPageView"

xmlns:pages = "clr-namespace:Capitulo03.Views;assembly=Capitulo03" >

<ContentPage Title = "Serviços" Icon = "tab_servicos" >

```

Como dito no início da seção, faremos uso de um novo controlador de layout, o Grid. Veja a definição dele no código a seguir. Um grid é composto por células, que são configuráveis como linhas e colunas. No código apresentado estas configurações estão sendo realizadas pelos elementos `<Grid.RowDefinitions>` e `<Grid.ColumnDefinitions>`. A priori, apenas a altura das linhas e larguras das colunas estão sendo trabalhadas pelas propriedades `Height` e `Width`. Os valores podem ser fixos, como a configuração da primeira coluna da primeira linha. As linhas e colunas começam em zero. As demais linhas têm sua altura definida com base na altura dos controles que serão inseridos nelas. A segunda coluna tem sua largura definida pelo asterisco (\*). Esta atribuição determina que deve ser utilizado todo o espaço restante do elemento, em nosso caso, a segunda coluna terá o espaço que restar na tela, tirando os 64 pontos definidos para a primeira coluna. O código a seguir deve estar após o código que implementamos anteriormente, na visão `TabbedPageView.xaml`.

```

<Grid Padding = "20, 50, 10, 20" BackgroundColor = "#FFFACD" >

<Grid.RowDefinitions>

<RowDefinition Height = "64" />

<RowDefinition Height = "Auto" />

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width = "64" />

<ColumnDefinition Width = "*" />

</Grid.ColumnDefinitions>

```

Com a estrutura do layout definida, precisamos agora inserir os controles visuais nas células do grid. Você deve ter notado pela figura apresentada anteriormente que estamos criando o mesmo cabeçalho utilizado na visão de clientes para esta nova visão, que se refere a serviços. Estamos tendo redundância de código aqui e isso não é uma boa prática. Mais à frente, ainda neste capítulo, componentizaremos este cabeçalho.

Veja, no código a seguir, a existência das propriedades `Grid.Column`, `Grid.Row` e `Grid.ColumnSpan`. Existe ainda a `Grid.RowSpan`, que não estamos utilizando no exemplo. As duas primeiras propriedades determinam em que coluna e linha o controle deverá ser inserido. Cada célula pode conter apenas um elemento. Temos, desta maneira, um `<Image>` e um `<StackLayout>` para os títulos. Note que, embora o título seja resultado de dois `<Label>`, estamos inserindo apenas 0 `<StackLayout>`.

Temos na sequência outro `<BoxView>`, que é o responsável por traçar a linha em preto, separadora do cabeçalho da página e dos controles de interação. Observe o uso da propriedade `Margin`, definindo uma base de 20 pontos, separando o `<StackLayout>` dos controles de interação.

```
<Image Source = "logo.png" VerticalOptions = "Start" Grid.Column = "0" Grid.Row = "0" />

<StackLayout Grid.Column = "1" Grid.Row = "0" >

<Label Text = "Meu Calhambeque" >

<Label.FontSize>

<OnPlatform x:TypeArguments = "x:Double" >

<On Platform = "iOS" > 30 </On>

<On Platform = "Android, UWP" > 15 </On>

</OnPlatform>

</Label.FontSize>

</Label>

<Label Text = "App desenvolvido para livro de Xamarin" Grid.Column = "1" Grid.Row = "1" >

<Label.FontSize>

<OnPlatform x:TypeArguments = "x:Double" >

<On Platform = "iOS" > 15 </On>

<On Platform = "Android, UWP" > 8 </On>

</OnPlatform>

</Label.FontSize>

</Label>

</StackLayout>
```

```
<BoxView BackgroundColor = "Black" HorizontalOptions = "FillAndExpand" HeightRequest = "2" Grid.Row = "1" Grid.Column = "0"
Grid.ColumnSpan = "2" Margin = "0, 0, 0, 20" />
```

Agora, concluindo a página da primeira guia, temos os controles de interação com o usuário. Os `<Entry>` e o `<Button>`, todos definindo em que célula serão visualizados. Observe a nova propriedade para o `<Entry>`, a `Placeholder`. Ela define um texto a ser exibido na caixa de texto antes de o usuário digitar a informação. Pode funcionar como uma ajuda, dispensando o `<Label>`.

Após o fechamento do `<ContentPage>` da primeira guia, temos o uso do elemento `pages` que criamos no início da visão. Neste elemento, fazemos uso de uma instância da classe `ContentPageView`. Esta será nossa segunda guia, que tem também a definição do ícone para ser utilizado no botão.

```
<Entry Placeholder = "Digite o nome do serviço" Grid.Row = "2"
Grid.ColumnSpan = "2" Grid.Column = "0" VerticalOptions = "Start" />

<Entry Placeholder = "Digite o valor para o serviço" Grid.Row = "3"
Grid.ColumnSpan = "2" Grid.Column = "0" VerticalOptions = "Start" />

<Button Text = "Confirmar dados" Grid.Row = "4"
Grid.ColumnSpan = "2" Grid.Column = "0"
BackgroundColor = "#96d1ff" />

</Grid>
</ContentPage>

<pages:ContentPageView Icon = "tab_clientes" />
</TabbedPage>
```

Para testar sua aplicação você precisa alterar o code-behind da classe `App` para que seja instanciada a nova visão. Em meu caso, o código `MainPage = new TabbedPageView();` precisou ser inserido no lugar do que instanciaava a página criada para o exemplo de `Content Page`.

### 3.3 Navegação entre páginas, ListView e ContentView

No exemplo implementado na seção anterior tivemos um tipo de navegação entre páginas, pelas guias de uma `TabbedPage`. Vimos que é possível criar `ContentPages` e utilizá-las como guias ou simplesmente como páginas individuais. Nesta seção, abordaremos a navegação entre páginas conhecida como "Navegação hierárquica". Neste tipo de navegação, as páginas vão se empilhando uma na outra (`push`), conforme formos navegando em um

processo e, quando retornamos para páginas anteriores, retiramos no topo da pilha a página atual (*pop*), tal qual uma pilha (estrutura de dados). Veja a figura a seguir, retirada da documentação do Xamarin.

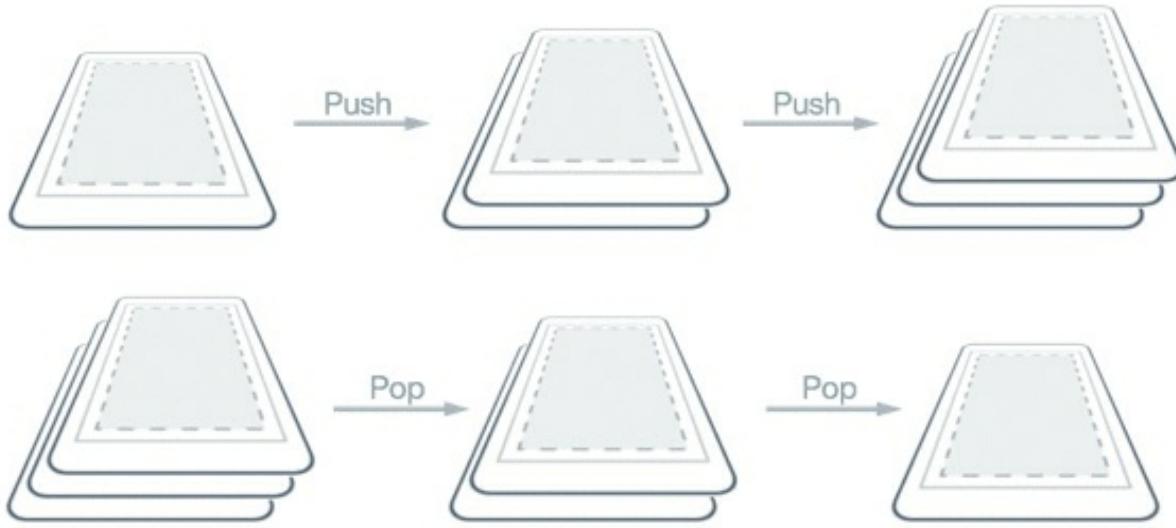


Figura 3.3: Processo de navegação hierárquica

Para a implementação de nosso exemplo da navegação hierárquica trabalharemos com dois `ContentPage`. O primeiro terá uma listagem dos clientes da oficina e o segundo a listagem dos veículos que o cliente tem, pois o cliente pode ser uma empresa, com vários carros. Implementaremos apenas o necessário para que você conheça este tipo de navegação. A implementação do code-behind começa no próximo capítulo, com o MVVM.

Começaremos então com a criação das classes de negócio. Tal qual fizemos para visão, vamos criar uma pasta específica para estas classes. Crie em seu projeto compartilhado uma pasta chamada `Models` e, dentro dela, duas classes, uma para `Cliente` e uma para `Veiculo`. Lembre da explicação do capítulo anterior sobre o que e qual é o projeto compartilhado? Veja os códigos na sequência.

Tal qual dito, um cliente pode ter vários veículos, desta maneira, a classe `Cliente` implementa esta associação. Para `Veiculo`, esta associação não existe, pois para o momento não é relevante quem é dono do veículo. Lembre-se de que omitirei as instruções `using` das listagens de códigos.

```
namespace Capitulo03.Models
{
    public class Cliente
    {
        public string Nome { get; set; }
        public string Endereco { get; set; }
        public string Telefone { get; set; }

        public IList<Veiculo> Veiculos { get; set; }

        public Cliente()
        {
            Veiculos = new List<Veiculo>();
        }
    }
}
```

```

namespace Capitulo03.Models
{
    public class Veiculo
    {
        public string Placa { get; set; }
        public string Marca { get; set; }
        public string Modelo { get; set; }
    }
}

```

Os dados que popularão as visões, no momento, serão dados baseados em coleções. Sendo assim, vamos criar uma pasta para hospedar estes dados e vamos chamá-la de `servicos`. Dentro dela, crie uma classe chamada `DadosParaTeste` e implemente tal qual o código a seguir.

Note que definimos três coleções na classe, e em seu construtor estas coleções recebem objetos de seu tipo declarado. Ao final do construtor, os veículos para cada cliente também são associados.

```

namespace Capitulo03.Servicos
{
    public class DadosParaTeste
    {
        public List<Veiculo> Veiculos;
        public List<Cliente> Clientes;

        public DadosParaTeste()
        {
            Veiculos = new List<Veiculo>()
            {
                new Veiculo() {Placa = "ABC-1234", Marca="Fiat", Modelo="147"},
                new Veiculo() {Placa = "DEF-5678", Marca="Chevrolet", Modelo="Monza"},
                new Veiculo() {Placa = "GHI-9012", Marca="Volkswagen", Modelo="Brasilia"},
                new Veiculo() {Placa = "JKL-3456", Marca="Ford", Modelo="Corcel"},
                new Veiculo() {Placa = "MNO-7890", Marca="Citroen", Modelo="C4"},
                new Veiculo() {Placa = "PQR-1234", Marca="Honda", Modelo="Civic"}
            };

            Clientes = new List<Cliente>()
            {
                new Cliente() { Nome="Gestrubindo", Endereco="Rua Sai debaixo", Telefone="12345-6789" },
                new Cliente() { Nome="Berssindrilio", Endereco="Rua Sobe, mas não desce", Telefone="01234-5678" },
                new Cliente() { Nome="Kestchbuncio", Endereco="Rua do beco sem fim", Telefone="90123-4567" }
            };

            Clientes[0].Veiculos.Add(Veiculos[0]);
            Clientes[0].Veiculos.Add(Veiculos[1]);
            Clientes[0].Veiculos.Add(Veiculos[2]);

            Clientes[1].Veiculos.Add(Veiculos[3]);
            Clientes[1].Veiculos.Add(Veiculos[4]);

            Clientes[2].Veiculos.Add(Veiculos[5]);
        }
    }
}

```

```
 }  
 }
```

Apenas com vistas de ilustrar o uso de `<ContentView>`, as duas páginas terão o cabeçalho utilizado nas implementações anteriores. Você pode verificar que este conteúdo será usado mais duas vezes, além das que fizemos até o momento, o que mais que caracteriza reutilização visual. Sendo assim, vamos implementar este conteúdo em um `<ContentView>` e então reutilizá-lo sempre que necessário. Existem literaturas que nomeiam estes conteúdos de controles do usuário (*User Control*).

Para criarmos um `ContentView` fazendo uso do Visual Studio Windows, é possível utilizar um template oferecido por ele. Vamos clicar com o botão direito do mouse sobre o nome da pasta `Views` e em `Adicionar->Nova pasta`. Eu utilizei o nome `ContentViews`. Nesta pasta, clicando com o botão direito nela, selecione `Adicionar->Novo Item` e na categoria `Xamarin.Forms`, escolha `Content View`. Eu nomeei este item de `CabecalhoView` e implementei o código apresentado na sequência.

Veja que o código começa com a declaração do elemento `<ContentView>`. Por definição, um `ContentView` só pode ter um elemento associado a ele, mas nada nos impede de utilizar um `StackLayout` ou `Grid` e, dentro destes gerenciadores, inserir os controles que forem necessários. E é isso que estamos fazendo no código apresentado.

```
<?xml version="1.0" encoding="UTF-8"?>  
  
<ContentView xmlns = "http://xamarin.com/schemas/2014/forms"  
  
xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"  
  
x:Class = "Capitulo03.Views.ContentViews.CabecalhoView" >  
  
<ContentView.Content>  
  
<StackLayout>  
  
<StackLayout Orientation = "Horizontal" >  
  
<Image Source = "logo.png" VerticalOptions = "Start" HeightRequest = "64" />  
  
<StackLayout VerticalOptions = "Center" >  
  
<Label Text = "Meu Calhambeque" >  
  
<Label.FontSize>  
  
<OnPlatform x:TypeArguments = "x:Double" >  
  
<On Platform = "iOS" > 30 </On>  
  
<On Platform = "Android, UWP" > 15 </On>  
  
</OnPlatform>
```

```

</Label.FontSize>

</Label>

<Label Text = "App desenvolvido para livro de Xamarin" >

<Label.FontSize>

<OnPlatform x:TypeArguments = "x:Double" >

<On Platform = "iOS" > 15 </On>

<On Platform = "Android, UWP" > 8 </On>

</OnPlatform>

</Label.FontSize>

</Label>

</StackLayout>

</StackLayout>

<BoxView BackgroundColor = "Black" HorizontalOptions = "FillAndExpand" HeightRequest = "2" />

</StackLayout>

</ContentView.Content>

</ContentView>

```

Já temos nosso modelo de negócio e componente visual comum entre as visões implementados. Seguiremos para a implementação das páginas que estarão em um sistema de navegação entre elas. Vamos criar a primeira página da navegação proposta, a de clientes. Adicione um `Content Page` à pasta `views` para isso. Eu o nomeei de `NavigationPageClientesViews`. Na sequência apresento o código para esta página. A explicação do código está após sua apresentação.

```

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"

xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

x:Class = "Capitulo03.Views.NavigationPageClientesView"

xmlns:cabecalho = "clr-namespace:Capitulo03.Views.ContentViews"

```

```
Title = "Clientes"

xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOS;assembly=Xamarin.Forms.Core"

ios:Page.UseSafeArea = "true"

BackgroundColor = "#FFFACD" >

<ContentPage.Content>

<StackLayout Padding = "10, 10, 10, 10" BackgroundColor = "#FFFACD" HorizontalOptions = "FillAndExpand" >

<cabecalho:CabecalhoView/>

<Label Text = "Selecione o CLIENTE" HorizontalOptions = "FillAndExpand" HorizontalTextAlignment = "Center" FontSize = "18" FontAttributes = "Bold" BackgroundColor = "Red" />

<ListView x:Name = "listViewClientes" VerticalOptions = "StartAndExpand" HasUnevenRows = "True" >

<ListView.ItemTemplate>

<DataTemplate>

<ViewCell>

<StackLayout Padding = "10" >

<Label Text = "{Binding Nome}" FontSize = "18" />

<Label Text = "{Binding Telefone}" FontSize = "11" />

</StackLayout>

</ViewCell>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

</StackLayout>

</ContentPage.Content>

</ContentPage>
```

Verifique a definição do `xmlns:cabecalho` antes de `<ContentPage.Content>`. Ele aponta para o namespace relativo à pasta que criamos para nossos componentes. Logo após a criação do primeiro `<StackLayout>` utilizamos este objeto para renderizar o cabeçalho. A ideia é a mesma do exemplo do `Tabbed Page` para reutilização de página, só que o `Content view` se refere a um único componente, comum a diversas páginas.

Logo após o cabeçalho da página existe um `<Label>` orientativo ao usuário, com algumas propriedades de configuração de sua aparência.

Estou trazendo também para esta nova página o uso do `ListView`. Este componente é responsável pela listagem de itens em uma visão. Não estamos fazendo uso de `Binding` para o `ListView`, pois está ligado ao MVVM, que veremos no próximo capítulo. Dei um nome ao componente, `listViewClientes`, pois vamos utilizá-lo no code-behind, então precisamos expô-lo. A propriedade `HasUnevenRows` determina se a listagem terá ou não linhas com tamanhos desiguais. Atribuindo `True` para esta propriedade, cada item ocupará a altura necessária para a exibição dos valores dela. É possível ainda definir a altura dos itens, atribuindo este valor à propriedade `RowHeight`.

Para que os dados sejam formatados no `ListView` é preciso definir seu padrão de exibição. Para isso, fazemos uso do `<ItemTemplate>`. É possível ainda configurar um template para o cabeçalho (`HeaderTemplate`) e para o rodapé do `ListView` (`FooterTemplate`). Como os itens que serão exibidos no `ListView` virão de um conjunto de dados, é preciso especificar o `DataTemplate`, responsável pela configuração visual dos itens e, dentro deste subelemento, é possível utilizar uma `<ViewCell>`, ou seja, uma célula com um componente. Em nosso exemplo estou fazendo uso de um `<StackLayout>` com dois `<Label>`s.

Estes `<Label>`s serão populados pelas propriedades especificadas como `Binding` na propriedade `Text` deles. Desta maneira, quando formos atribuir a fonte de dados para o `ListView`, ela deverá ter estas propriedades (`Nome` e `Telefone`) na estrutura de seus objetos. O `Binding` (ligação) dos controles dentro do `<DataTemplate>` se refere a cada objeto da coleção que popula o `ListView`. Esta ligação é conhecida como ligação de visão para visão, ou seja, um objeto da visão fornece dados para outro objeto da visão. É diferente da ligação com uma classe de modelo. Veremos mais sobre estes tipos de ligações a partir do próximo capítulo.

Com a visão da página definida, precisamos atribuir ao `ListView` a fonte de dados e faremos isso em nosso code-behind. Veja o código dele na sequência. Trabalhei apenas no construtor, fazendo uso da classe de dados de teste que criamos anteriormente. Quando começarmos a trabalhar com o MVVM, trabalharemos a ligação diretamente na visão, apontando na classe apenas o que será o contexto para as ligações que serão utilizadas, mas achei interessante e válido trazer esta experiência para este momento do livro.

```
namespace Capitulo03.Views
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class NavigationPageClientesView : ContentPage
    {
        public NavigationPageClientesView ()
        {
            InitializeComponent ();
            listViewClientes.ItemsSource = new DadosParaTeste().Clientes;
        }
    }
}
```

Com a implementação realizada até aqui, já podemos executar nossa aplicação e verificar o resultado dela, que é a exibição da lista de clientes, como pode ser visto na figura a seguir. Lembre-se de implementar a nova visão como `MainPage` na classe `App`.

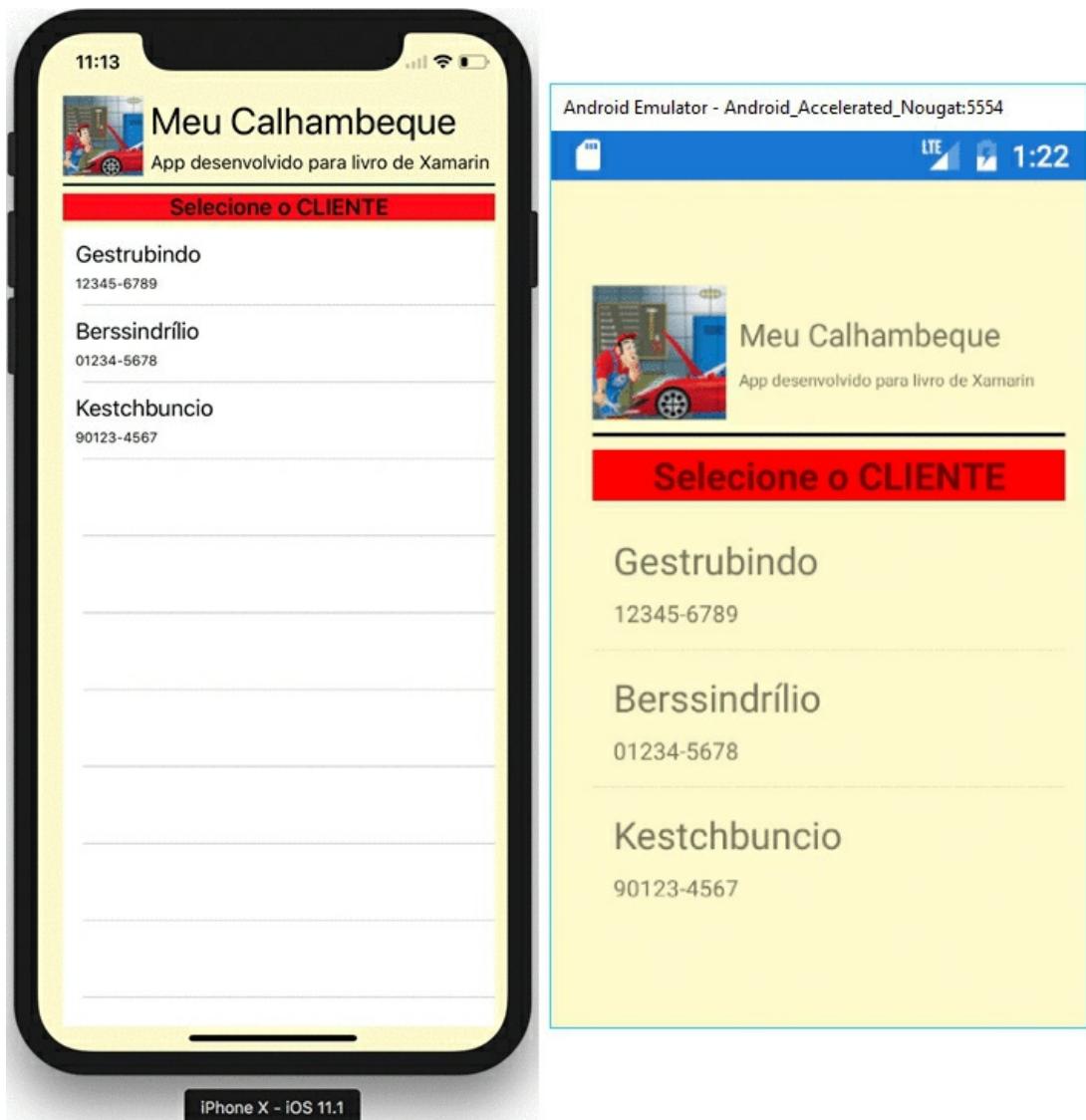


Figura 3.4: Visão com a listagem de clientes

Vamos agora implementar a visão de veículos. Inicialmente ela será semelhante ao que fizemos para clientes, exibirá uma listagem de veículos. O que precisaremos é selecionar apenas os veículos do cliente selecionado na visão anterior, pois queremos visualizar apenas os veículos dele. Mas vamos agora desenhar a página apenas. Portanto, crie uma nova Content Page , tal qual fizemos até agora no livro e implemente o código XAML apresentado na sequência. Eu nomeei a página de `NavigationViewVeiculosView` .

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"

xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

x:Class = "Capitulo03.Views.ContentViews.NavigationPageVeiculosView"
```

```
xmlns:cabecalho = "clr-namespace:Capitulo03.Views.ContentViews"

Title = "Veículos"

xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOS;assembly=Xamarin.Forms.Core"

ios:Page.UseSafeArea = "true"

BackgroundColor = "#FFFACD" >

<ContentPage.Content>

<StackLayout Padding = "10, 10, 10, 10" BackgroundColor = "#FFFACD" HorizontalOptions = "FillAndExpand" >

<cabecalho:CabecalhoView/>

<Label Text = "Selecione o VEÍCULO" HorizontalOptions = "FillAndExpand" HorizontalTextAlignment = "Center" FontSize = "18" FontAttributes = "Bold" BackgroundColor = "Red" />

<ListView x:Name = "listViewVeiculos" VerticalOptions = "StartAndExpand" HasUnevenRows = "True" >

<ListView.ItemTemplate>

<DataTemplate>

<ViewCell>

<StackLayout Padding = "10" >

<Label Text = "{Binding Modelo}" FontSize = "18" />

<Label Text = "{Binding Placa}" FontSize = "11" />

</StackLayout>

</ViewCell>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

</StackLayout>
```

```
</ContentPage.Content>

</ContentPage>
```

Muito bem, agora, precisamos dizer para a visão de veículos que os veículos que serão exibidos precisam ser apenas os que são do cliente selecionado na visão de clientes. Para isso, precisamos adaptar o nosso `ListView` de clientes para que capture o momento em que o usuário selecionar um determinado cliente. Adapte a declaração de seu `ListView` para que ele possua a declaração para o evento `ItemSelected`, tal qual mostra o código a seguir. Esta mudança deve ser realizada no `NavigationPageClientesView`.

No momento em que você for informar o nome do método que será disparado quando o evento ocorrer, o Visual Studio oferecerá a possibilidade de criar o método para você. Caso isso não apareça, pressione `CTRL+Espaço` para que o método possa ser criado no code-behind. Caso o auxílio para a criação do método não ocorra, você poderá criá-lo diretamente, conforme a listagem exibida mais adiante.

```
<ListView x:Name = "listViewClientes" VerticalOptions = "StartAndExpand" HasUnevenRows = "True" ItemSelected =
"listViewClientes_ItemSelected" >
```

Na sequência, precisamos implementar o método identificado no XAML anterior. Vá para o code-behind e verifique que, se você aceitou a sugestão do Visual Studio, o método já está criado, restando a você implementar seu comportamento, que tem seu código apresentado a seguir. Caso algum erro tenha impedido o VS de criar o método, você deverá fazê-lo, implementando, além do comportamento, a assinatura do método.

No início do método existe uma condição para o caso do item selecionado estar com valor nulo, o que dispararia uma exceção ao tentarmos recuperar este item. A possibilidade de o item selecionado ser nulo se encontra na última linha do método, que atribui nulo a ele. Esta implementação se fez necessária para quando o usuário for para página de veículos e voltar para a página de clientes. Com isso, limpamos a seleção realizada.

```
private async void listViewClientes_ItemSelected ( object sender, SelectedItemChangedEventArgs e )
{
    if (e.SelectedItem == null )
        return ;
    var cliente = e.SelectedItem as Cliente;
    await Navigation.PushAsync( new NavigationPageVeiculosView(cliente));
    ((ListView)sender).SelectedItem = null ;
}
```

Observe que o item selecionado (`SelectedItem`) que o método recebe por meio de seu argumento "`e`" é convertido para o tipo `Cliente` (`cast`) e então invocamos a página que exibirá os veículos do cliente selecionado, por meio da chamada ao método `PushAsync()` de `Navigation`. O cliente selecionado é enviado para o construtor da página de veículos. Como o método é assíncrono, precisamos prefixar a instrução com `await`, para que seja aguardada a execução do método. Note o uso da instrução `async` na declaração do método que captura o evento.

O código apresentado anteriormente ainda não funciona, pois precisamos criar o construtor que receba um cliente no code-behind da visão de veículos. Veja o código a seguir.

Observe a declaração de uma propriedade para `cliente`. Com esta declaração, estamos prevendo a situação de termos uma navegação para outra página e nela precisarmos deste dado, o que deveremos fornecer via construtor. Veja que o `set` para ela está privado, permitindo alteração apenas nessa classe. Estamos mantendo o construtor padrão para que seja possível utilizar o `Xamarin Forms Preview`. No construtor que recebe o cliente, estamos alterando o título da visão para exibir o nome do cliente selecionado, utilizando interpolação de strings, e então atribuímos aos itens do

`ListView` os veículos para este cliente. O código é finalizado com a atribuição do cliente recebido para o cliente do objeto atual da classe.

```
namespace Capitulo03.Views
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class NavigationPageVeiculosView : ContentPage
    {
        public Cliente Cliente { get; private set; }

        public NavigationPageVeiculosView ()
        {
            InitializeComponent ();
        }

        public NavigationPageVeiculosView (Cliente cliente)
        {
            InitializeComponent();
            Title = $"Veículos de {cliente.Nome}";
            listViewVeiculos.ItemsSource = cliente.Veiculos;
            this.Cliente = cliente;
        }
    }
}
```

Para testarmos nossa aplicação, precisamos adaptar nossa classe `App`. Veja o código a seguir. Observe que instanciamos uma `NavigationPage()` e passamos a instância de nossa página inicial para a navegação como argumento.

```
MainPage = new Xamarin.Forms.NavigationPage( new NavigationPageClientesView());
```

## Especificidades do iOS 11

O iOS 11 trouxe para as páginas de navegação a exibição do título dela em fonte maior do que era feito na versão anterior. Esta nova característica é nomeada de `Large Title` e vamos verificar agora o seu uso. Veja a implementação a seguir, que precisa ser feita no code-behind de `App`. Observe a invocação ao método `SetPrefersLargeTitles()`, enviando o valor `true`, o que habilitará esta funcionalidade. Utilizamos recursos dependentes da plataforma em que a aplicação está executando na invocação deste método.

Até a implementação do exemplo anterior, instanciávamos a página de exibição diretamente na linha da declaração de `MainPage`. Agora, como faremos uso de navegação, estamos instanciando a página de clientes (que é `ContentPage`) como `NavigationPage` em uma variável. Nada impediria de ter esta instanciação diretamente para o `MainPage`, como fizemos no código anterior. A escolha de se ter uma variável se dá pelo fato de querermos realizar as configurações de fonte, cores e dos títulos do iOS, antes da atribuição do objeto à página principal.

```
namespace Capitulo03
{
    public partial class App : Application
    {
        public App ()
    }
}
```

```

InitializeComponent();

var navigationPage = new Xamarin.Forms.NavigationPage( new NavigationPageClientesView() );
navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetPrefersLargeTitles( true );
navigationPage.BarBackgroundColor = Color.Red;
navigationPage.BarTextColor = Color.White;

MainPage = navigationPage;
}

// Demais códigos omitidos

```

Com esta implementação realizada, podemos testar novamente a aplicação. A figura a seguir apresenta a página de veículos renderizada após a escolha de um cliente. Observe que no iOS toda a barra de título aceita a configuração de cores e que uma opção de retornar para a página anterior é exibida no topo da página em ambas plataformas.

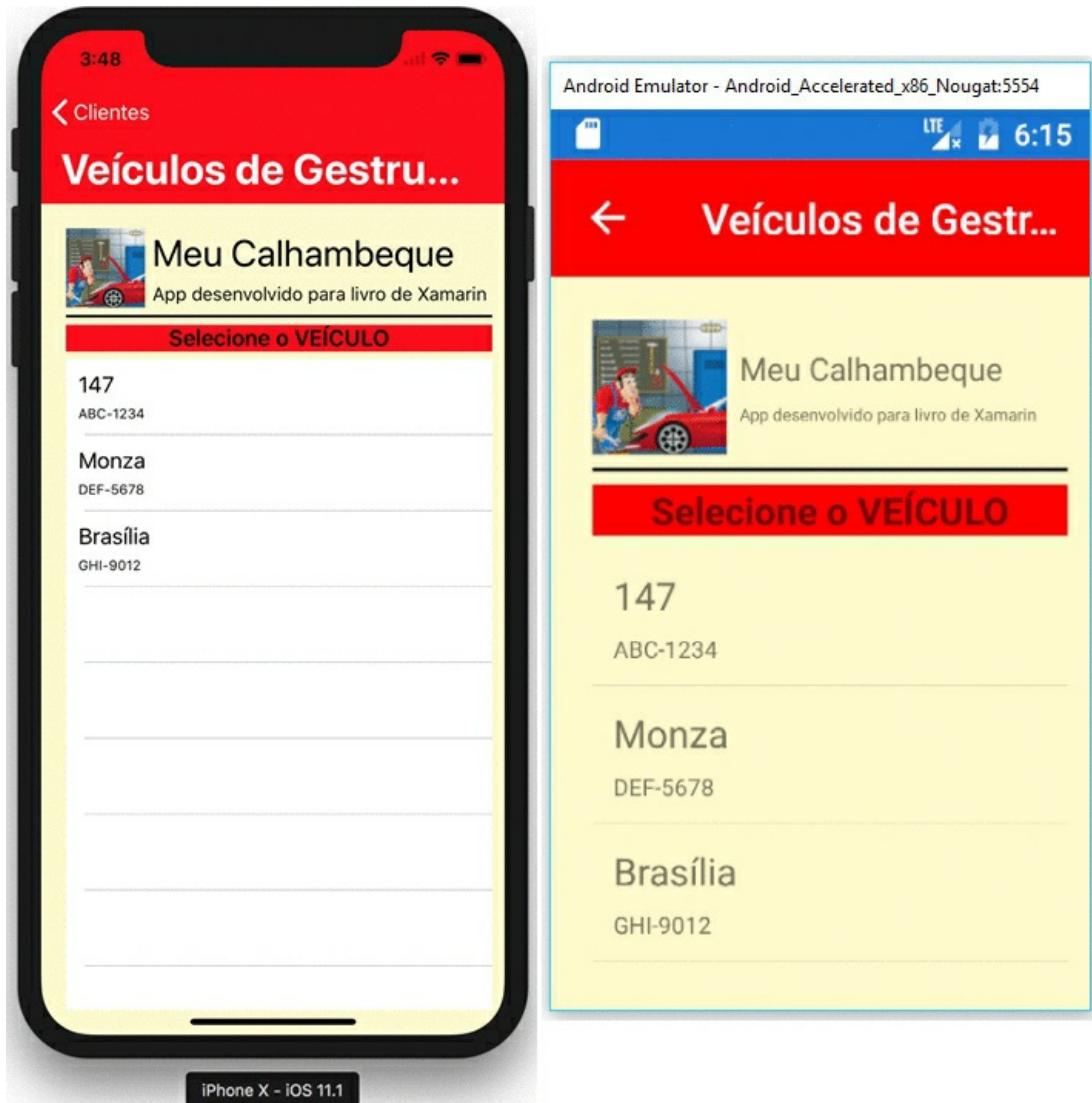


Figura 3.5: Navegação entre páginas

O uso da figura e nome da aplicação no topo de cada página foi apenas ilustrativo, para verificação de controles e técnicas de reutilização. Não é comum deixar este tipo de dados em aplicações. Fica a dica.

### Especificidades do Android

No Android, a barra que está azul na figura anterior é chamada de `status Bar` e a configuração para ela precisou ser feita no projeto `Android`. Na classe `MainActivity`, no final do método `onCreate()`, insira a instrução `Window.SetStatusBarColor(Android.Graphics.Color.Argb(255, 255, 0, 0));`, cujos três últimos argumentos representam as cores Vermelho, Verde e Azul (RGB). O primeiro argumento, chamado de `Alpha`, está relacionado à transparência da cor. Teste isso e execute novamente sua aplicação. Veremos no capítulo 10 a criação de renderizadores personalizados para controles, que é um recurso muito bom oferecido pelo Xamarin e que pode ser utilizado para customizar estas particularidades de plataforma.

## 3.4 Master Detail Pages e Hamburger Menu

Para finalizar os exemplos de páginas que trabalharemos no livro, veremos aqui nesta seção o tipo de página conhecido como `Master Detail`, que consiste em duas páginas ligadas entre si e que podem ser visualizadas pela rolagem horizontal. O uso deste tipo de página está mais ligado à possibilidade de disponibilizar ao usuário um menu vertical de opções de navegação, conhecido como `Hamburger Menu`.

O Visual Studio (Windows) oferece um template para criar a estrutura necessária para uma página do tipo `Master Detail`, simulando especificamente o menu vertical de opções de acesso. Este recurso é oferecido por meio da adição de um novo item e escolhendo a categoria `Xamarin.Forms`. Infelizmente no Mac não existe este template. É preciso criar os arquivos individualmente, mas não é tão trabalhoso. Se após criar a visão ocorrer algum erro dizendo não identificar os componentes criados, realize a limpeza da solução e a compile novamente.

O template criará quatro arquivos: um que representa uma classe de negócio para a exibição de dados, que para o template representará cada item do menu de opções. Esta classe terá em seu nome o sufixo `MenuItem`; uma página XAML para a parte `Master` e outra para a parte `Detail`, ambas do tipo `ContentPage` e com sufixos `Master` e `Detail`, respectivamente; e a quarta, que terá o nome que você informou na criação e que será do tipo `MasterDetailPage`.

Embora seja possível fazer uso do template, nada impede você de criar estas classes comentadas de maneira individual. Fica a seu critério. Eu optei por fazer uso do template e nomeei a página como `MasterDetailPageView`, mas por enquanto retirarei dele o comportamento relacionado ao MVVM. Lembre-se de que a criação deve ser feita na pasta `Views`.

Veja no código a seguir a implementação da classe `MasterDetailPageViewMenuItem`. Retirei o código referente ao construtor, pois no template ele se baseava em apenas uma página. A propriedade `TargetType` conterá o tipo da classe de visão que será invocada quando as opções forem selecionadas. Como esta classe é restrita às visões, eu a mantive no mesmo namespace (pasta). Se você estiver no Mac, precisa criar esta classe.

```
namespace Capitulo03.Views
{
    public class MasterDetailPageViewMenuItem
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string IconSource { get; set; }
        public Type TargetType { get; set; }
    }
}
```

Vamos agora criar a visão que representará a parte principal (`master`) da `MasterPage`. O arquivo já está criado se você

está no Windows, caso esteja no Mac, crie uma página `contentPage` . O código a seguir representa a visão `MasterPageView` .

Observe a tag principal da visão definindo-a como `MasterDetailPage` e ao final dela a declaração do `xmlns` para o namespace de nossas visões. Dentro da tag principal temos duas outras, específicas para o tipo de página que estamos criando, são elas: `<MasterDetailPage.Master>` e `<MasterDetailPage.Detail>` .

Na tag `Master` apontamos a página `MasterDetailViewMaster` , criada pelo template do Visual Studio, como conteúdo a ser exibido como página principal. Damos também um nome ao objeto, para que possa ser manipulado pelo code-behind (`masterPage` ). Para a página de detalhes fazemos uso da tag que define a página a ser utilizada como `NavigationPage` , pois será uma página de navegação e queremos que a barra de navegação seja exibida. Como argumento para o construtor, enviamos uma instância de `ContentPageView` , a primeira visão que implementamos neste capítulo.

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

<MasterDetailPage xmlns = "http://xamarin.com/schemas/2014/forms"

    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

    x:Class = "Capitulo03.Views.MasterDetailView"

    xmlns:pages = "clr-namespace:Capitulo03.Views" >

    <MasterDetailPage.Master>

        <pages:MasterDetailViewMaster x:Name = "masterPage" />

    </MasterDetailPage.Master>

    <MasterDetailPage.Detail>

        <NavigationPage>

            <x:Arguments>

                <pages:ContentPageView />

            </x:Arguments>

        </NavigationPage>

    </MasterDetailPage.Detail>

</MasterDetailPage>
```

Precisaremos implementar alguns códigos em C# para que possamos testar nossa visão, mas primeiro apresentarei a visão para a página principal, pois realizei modificações em relação ao criado pelo template.

Para dar um efeito visual legal na aplicação, faremos uso de um componente disponibilizado para o Xamarin Forms: o `ImageCircle` (<https://github.com/jamesmontemagno/ImageCirclePlugin> ). O autor, James Montemagno, tem vários

componentes interessantes e é um profissional chave da equipe do Xamarin, vale a pena você visitar seu repositório.

Clique com o botão direito sobre o nome da solução e em Gerenciar pacotes Nuget da solução . Na janela que se abre, em Procurar , digite ImageCircle . Clique no elemento que aparecerá. A versão que utilizei no momento da escrita deste livro foi a 3.0.0.5 . Certifique-se de que o componente é do autor que mencionei. Marque todos os projetos como destino do componente e clique em Instalar . Veja a figura a seguir.

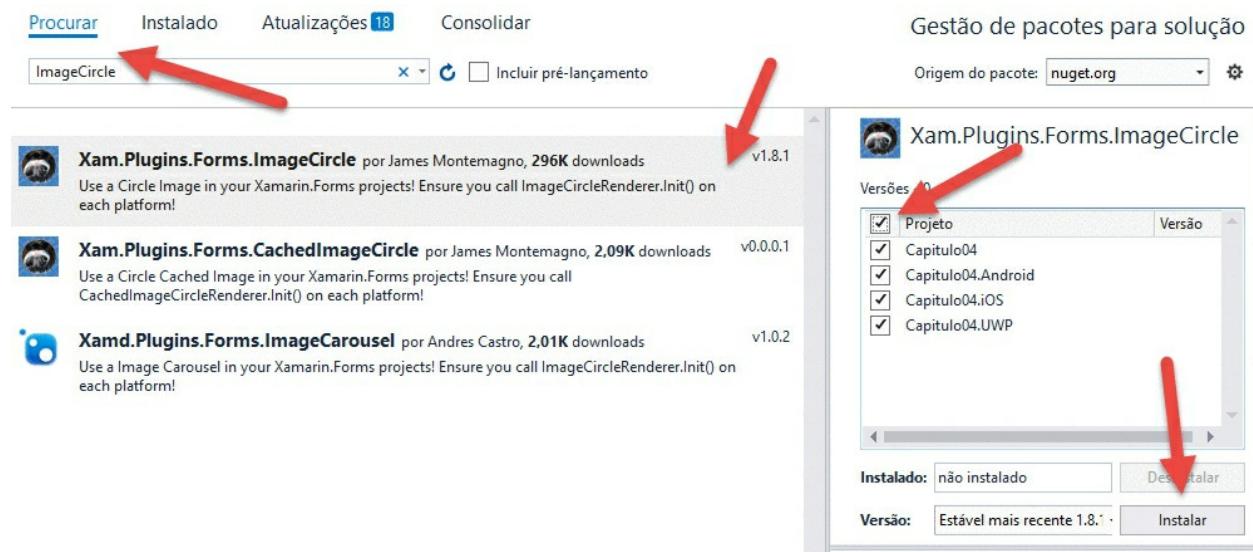


Figura 3.6: Instalação do componente ImageCircle

Para utilizar este novo componente em nosso projeto compartilhado, precisamos inicializá-lo nos projetos específicos para cada plataforma. No iOS, na classe AppDelegate , no método FinishedLaunching() , logo após a chamada do Init() , você deve inserir ImageCircleRenderer.Init(); . Lembre-se dos usings . No Android, esta mudança deve ser realizada na classe MainActivity , no método OnCreate() . Sem estas instruções o componente não estará disponível na execução.

Vamos usar este novo componente na página principal, como cabeçalho da lista de opções disponibilizadas como um menu para a aplicação. Veja o código desta visão na sequência. Verifique nele a declaração do namespace controls , que aponta para O ImageCircle .

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"

    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

    x:Class = "Capítulo03.Views.MasterDetailPageViewMaster"

    xmlns:controls = "clr-namespace:ImageCircle.Forms.Plugin.Abstractions;assembly=ImageCircle.Forms.Plugin"

    Title = "Master" >

<StackLayout>
```

```

<ListView x:Name = "itensMenuListView" SeparatorVisibility = "None" HasUnevenRows = "true" ItemsSource = "{Binding
OpcoesMenu}" >

<ListView.Header>

<Grid BackgroundColor = "#03A9F4" >

<Grid.ColumnDefinitions>

<ColumnDefinition Width = "10" />

<ColumnDefinition Width = "64" />

<ColumnDefinition Width = "*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height = "30" />

<RowDefinition Height = "64" />

<RowDefinition Height = "Auto" />

<RowDefinition Height = "10" />

</Grid.RowDefinitions>

```

Esta visão exibirá uma lista de opções em um `ListView`, abstraindo um menu de opções que, ao serem clicadas, exibirão como página detalhe a página associada à opção selecionada. Estas opções são objetos da classe `MasterDetailPageMenuItem` que implementamos anteriormente e têm as páginas atribuídas a cada objeto. Associe a propriedade `ItemsSource` com uma propriedade que implementaremos no code-behind chamada `opcoesMenu`. Como precisaremos manipular o `ListView`, também no code-behind, foi dado a ele um nome (`itensMenuListView`).

Uma nova seção de configuração do `ListView` é apresentada neste exemplo, a `Header`, que representa o cabeçalho para as opções que serão exibidas para o usuário. Dentro do cabeçalho está implementado um `Grid`, com 3 colunas e 4 linhas. A primeira coluna e a primeira e última linha, são utilizadas como separadores de conteúdo, um tipo de `padding`. Veja este trecho de código apresentado a seguir.

A segunda parte da visão, apresentada na sequência, define como o controle de imagem em forma de círculo será apresentado na página e em que célula do grid também. Observe as configurações diferentes para cada plataforma. Na sequência, um `StackLayout` que conterá os dois `Labels` também é configurado para uma célula do grid.

```

<controls:CircleImage Source = "logo.png" Aspect = "AspectFit" Grid.Row = "1" Grid.Column = "1" >

<controls:CircleImage.WidthRequest>

```

```

<OnPlatform x:TypeArguments = "x:Double" >

<On Platform = "Android, iOS" > 55 </On>

<On Platform = "UWP" > 75 </On>

</OnPlatform>

</controls:CircleImage.WidthRequest>

<controls:CircleImage.HeightRequest>

<OnPlatform x:TypeArguments = "x:Double" >

<On Platform = "Android, iOS" > 55 </On>

<On Platform = "UWP" > 75 </On>

</OnPlatform>

</controls:CircleImage.HeightRequest>

</controls:CircleImage>

<StackLayout Grid.Row = "1" Grid.Column = "2" VerticalOptions = "Center" >

<Label Text = "Meu Calhambeque" TextColor = "White" FontAttributes = "Bold" FontSize = "20" />

<Label Text = "Livro Xamarin - CC" TextColor = "White" FontSize = "Small" />

</StackLayout>

</Grid>

</ListView.Header>

```

Finalizando o XAML da visão, temos o `ItemTemplate` configurado para exibição dos dados que serão responsáveis pela apresentação das opções de acesso ao usuário.

```

<ListView.ItemTemplate>

<DataTemplate>

<ViewCell>

<Grid>

```

```

<Grid.ColumnDefinitions>

    <ColumnDefinition Width = "10" />

    <ColumnDefinition Width = "32" />

    <ColumnDefinition Width = "*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

    <RowDefinition Height = "5" />

    <RowDefinition Height = "32" />

</Grid.RowDefinitions>

<Image Source = "{Binding IconSource}" HeightRequest = "32" Grid.Column = "1" Grid.Row = "1" />

<Label Text = "{Binding Title}" FontSize = "Default" Grid.Column = "2" Grid.Row = "1" VerticalTextAlignment = "Center" />

</Grid>

</ViewCell>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

</StackLayout>

</ContentPage>

```

Vamos agora para a implementação do code-behind desta nossa última visão, a principal. Veja o código na sequência.

Temos duas propriedades definidas nesta classe: `opcoesMenu` e `ListView`. A primeira é uma matriz que é populada no construtor e será ligada com o controle `ListView` que definimos no XAML anterior, por meio do `Binding` da propriedade `ItemsSource`. A segunda propriedade tem o mesmo nome do controle e armazenará o `ListView` das opções nela. O objeto precisa dessa informação, pois a utilizaremos no code-behind da `MasterDetailPage`.

No construtor também está definida a imagem que será utilizada como ícone quando as páginas de detalhes forem exibidas, para que represente o acesso ao menu. Esta imagem é a do `Hamburger Menu`. O construtor termina com a definição do contexto de ligação para a visão atual, que é ela mesma. Isso quer dizer que, no XAML, quando utilizarmos o `{Binding}`, é no objeto que as propriedades serão buscadas para a ligação.

```

namespace Capitulo03.Views
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class MasterDetailPageViewMaster : ContentPage
    {
        public MasterDetailPageViewMenuItem[] OpcoesMenu { get; set; }
        public ListView ListView { get; set; }

        public MasterDetailPageViewMaster()
        {
            Icon = "menu.png";
            InitializeComponent();
            OpcoesMenu = new[]
            {
                new MasterDetailPageViewMenuItem { Id = 0, Title = "Clientes", TargetType =
typeof(ContentPageView), IconSource="tab_clientes.png"},

                new MasterDetailPageViewMenuItem { Id = 0, Title = "Serviços", TargetType =
typeof(TabbedPageView), IconSource="tab_servicos.png"}
            };
            ListView = itensMenuListView;
            BindingContext = this;
        }
    }
}

```

Precisamos agora voltar para a `MasterDetailPage` e trabalhar no code-behind dela. Veja na sequência a implementação feita para ela.

O construtor da classe inicia com a atribuição de um método ao evento `ItemSelected` da propriedade `ListView` da página principal que definimos anteriormente. Para relembrar, o objeto `masterPage` está definido no XAML da `MasterDetailPage`. Finalizando o construtor, é atribuído o valor lógico `true` à propriedade `IsPresented` da visão, que determina que a página principal deve ser exibida ao usuário no início da renderização da `MasterDetailPage`. Se não informarmos, por padrão é exibida a página detalhe.

Na sequência, temos a implementação do método disparado na captura do evento de seleção de item no `ListView`. A lógica é a mesma que trabalhamos no exemplo de navegação. A diferença está na instanciação da página que deverá ser exibida, que ficou dependente do item selecionado. Também manipulamos a `IsPresented` para ocultar parcialmente a página principal.

```

namespace Capitulo03.Views
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class MasterDetailPageView : MasterDetailPage
    {
        public MasterDetailPageView()
        {
            InitializeComponent();
            masterPage.ListView.ItemSelected += ListView_ItemSelected;
            IsPresented = true;
        }

        private void ListView_ItemSelected(object sender, SelectedItemChangedEventArgs e)

```

```

{
    var item = e.SelectedItem as MasterDetailPageViewMenuItem;
    if (item == null)
        return;

    var page = (Page)Activator.CreateInstance(item.TargetType);
    page.Title = item.Title;

    Detail = new NavigationPage(page);
    IsPresented = false;

    masterPage.ListView.SelectedItem = null;
}
}
}
}

```

Você pode pensar em configurar, no método `ListView_ItemSelected()`, para que toda página selecionada já realize a configuração da área segura e do tamanho especial para os títulos das páginas, características relativas ao iOS 11, substituindo a instrução `Detail = new NavigationPage(page);`, pelas que estão na sequência.

```

var navigationPage = new Xamarin.Forms.NavigationPage(page);

navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetPrefersLargeTitles( true );
navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetUseSafeArea( true );
Detail = navigationPage;

```

Com esta implementação realizada concluímos a aplicação com uso do tipo de página `MasterDetailPage`. Você pode apagar o arquivo criado para a página de detalhe pelo template. Nós não o utilizamos. Para testar sua aplicação, na classe `App` altere a atribuição da `MainPage` para a página `MasterDetailPageView`. Veja a figura a seguir que traz a aplicação em execução, mostrando parte da página em detalhe. Você pode acessar a página de detalhe pressionando-a e arrastando para a esquerda. O movimento para a direita retorna ao detalhe o foco da aplicação, isso, é claro, sem contar a opção de acesso pelo menu que criamos.

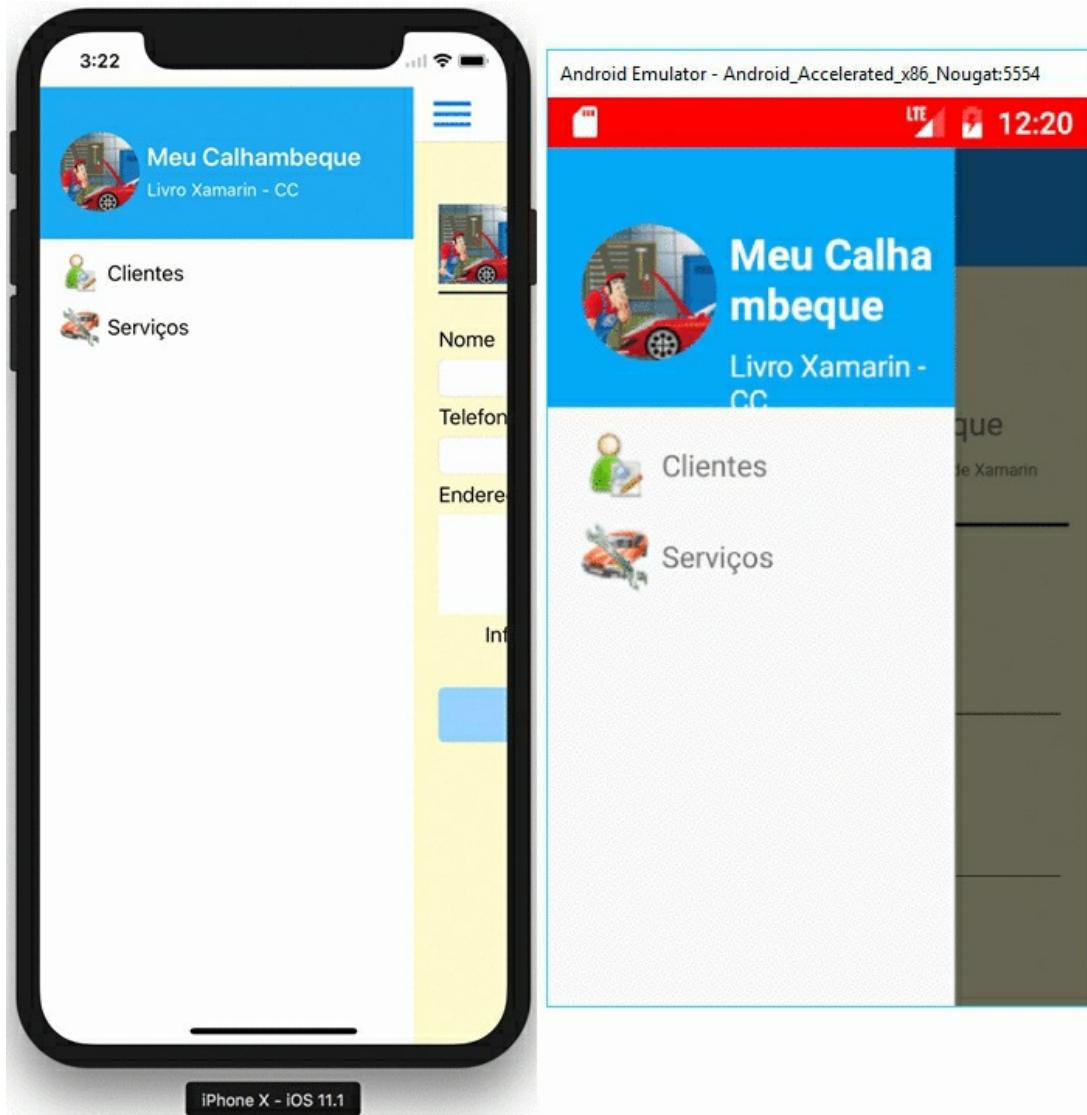


Figura 3.7: Execução da aplicação com MasterDetailPage

### 3.5 Conclusão

Chegamos ao final do capítulo. Nele apresentei diferentes tipos de páginas, layouts, controles e alguns comportamentos para interação com o usuário. Com este conhecimento aprendido é possível que você consiga desenhar páginas para sua aplicação. Toda implementação feita neste capítulo, em relação à interface com o usuário foi feita com base no XAML. Vimos também como configurar estilos visuais de controles, de acordo ao dispositivo em que a aplicação esteja sendo executada. Fizemos também uso de um componente de terceiro, que possibilitou a renderização de uma imagem em forma de círculo.

No próximo capítulo introduzirei o MVVM e teremos nossas páginas ligadas a objetos que estarão disponíveis para persistência e recuperação. Ainda faremos uso de coleções, mas será um capítulo muito bom. Até lá.

## C APÍTULO 4

# O padrão Model-ViewModel e o Messaging Center do Xamarin

O capítulo anterior trouxe algumas páginas para interação com o usuário, mas os controles visuais não estavam ligados a classes do modelo de negócio que permitiriam o manuseio dos dados informados. Foi apresentado como expor os objetos XAML no code-behind pela definição da propriedade `x:Name`, o que possibilita a você obter os dados dos controles e então trabalhar com eles, apesar de não ser uma maneira elegante e coesa. Este capítulo tem como objetivo introduzir o MVVM e por meio dele nos fornecer os controles ligados ao nosso modelo de negócio. Durante a implementação proposta aqui, com MVVM, veremos alguns controles novos.

O padrão de projeto MVVM, criado por John Goshmann, busca propiciar a separação de responsabilidades, possibilitando tornar um aplicativo fácil de ser mantido. Inicialmente, este padrão foi usado em aplicações WPF e Silverlight, mas sua essência não muda para o uso em aplicações Xamarin Forms. Em algumas situações, ele se assemelha ao MVC (*Model-View-Controller*) e ao MVP (*Model-View-Presenter*).

### MVC — M ODEL- V IEW- C ONTROLLER OU M ODELO- V ISÃO- C ONTROLADOR

O padrão MVC busca dividir a aplicação em responsabilidades relativas à definição de sua sigla. A parte do `Modelo` trata as regras de negócio, o domínio do problema; a `visão` busca levar ao usuário final informações acerca do modelo, ou solicitar dados para registros; e o `controlador` realiza a comunicação entre as camadas.

### MVP — Model-View-Presenter ou Modelo-Visão-Apresentação

O MVP é uma derivação do MVC, usado para construir principalmente interfaces gráficas. É projetado para facilitar os testes unitários automatizados e melhorar a separação de interesses em lógica de apresentação. A Apresentação atua sobre `Modelo` e `visão`, recuperando os dados dos repositórios e formatando-os para exibir na `visão`.

O MVVM mantém uma espécie de `Façade` (outro padrão de projeto) entre o modelo de negócios e a visão, que é a interface com o usuário. Um `Façade`, de maneira simples, é uma camada que recebe requisições e as delega para os responsáveis. Veja a figura a seguir. Note que há uma ligação (*binding*) entre a View e a ViewModel, em que a View não tem acesso direto à camada de negócio, ficando isolada. Na camada ViewModel, ainda é possível implementar `Commands`, que são também ligados e reagem a interações do usuário, o que chamamos de lógica da camada de apresentação.

Você pode ver, também na figura a seguir, que existe o envio de notificações que pode ocorrer entre as camadas. Este processo, em aplicações Xamarin Forms, pode ocorrer por meio do serviço de mensagens, que também veremos neste capítulo.

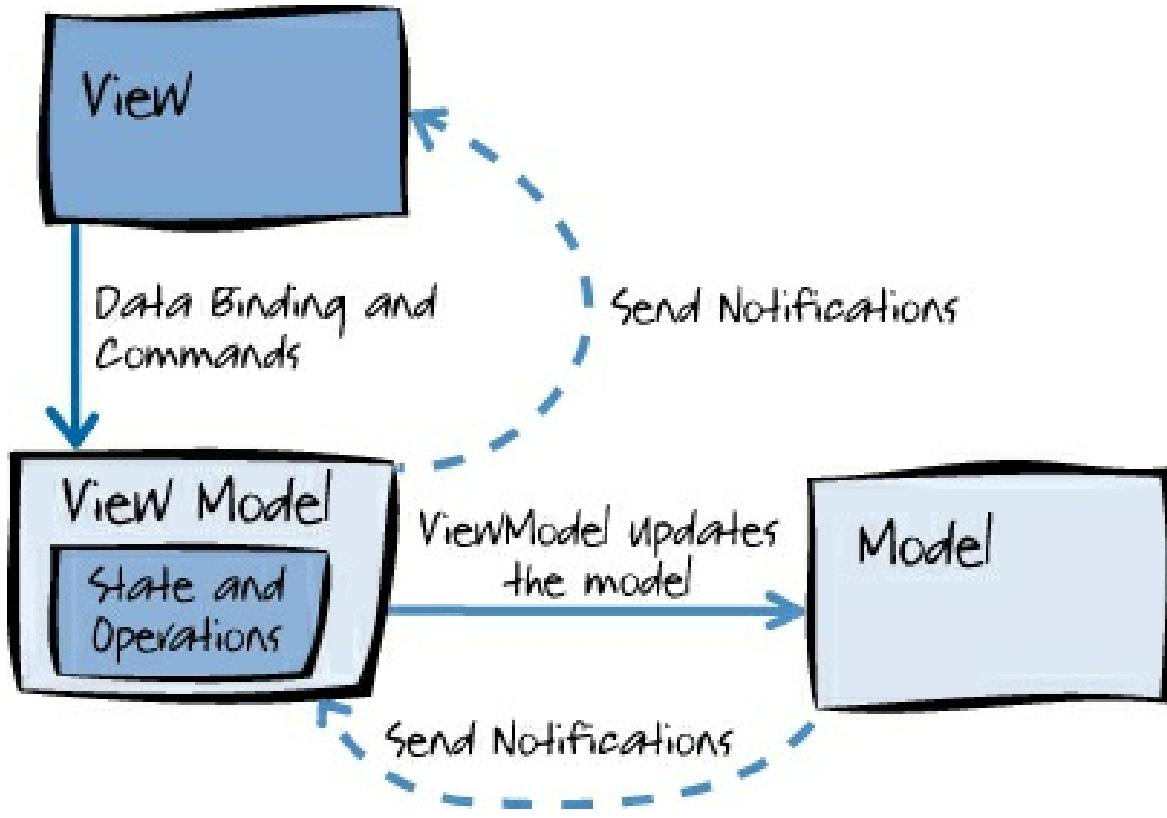


Figura 4.1: Representação do MVVM — <https://i-msdn.sec.s-msft.com/dynimg/IC564167.png/>

## 4.1 A página de serviços oferecidos

Nossa oficina oferecerá serviços que serão prestados aos clientes. Desta maneira, precisamos ter estes serviços disponíveis para seleção quando o cliente for registrar a solicitação de atendimento. As operações CRUD para os tipos de serviços serão implementadas nesta seção, com o uso do MVVM.

Trabalharemos o MVVM inicialmente em conjunto com páginas navegáveis (hierarquicamente). A primeira página deste conjunto será responsável por exibir uma relação dos serviços já registrados, por meio de um `ListView`.

Veremos um controle para disponibilizarmos uma barra de tarefas (`ToolBar`), com a opção de o usuário navegar para uma página responsável por solicitar os dados de um novo tipo de serviço.

No `ListView`, ofereceremos a possibilidade de o usuário selecionar um serviço, que pode ser alterado ou, então, excluído da listagem. Estas opções estarão disponíveis a partir do `ListView`.

Começaremos nossa aplicação criando um novo projeto para ela, chamado de `Capítulo04`. Lembre-se de selecionar o template de projeto vazio e a .NET Standard como estratégia de compartilhamento de código.

Após a criação do projeto, precisamos criar uma pasta que será responsável pelos modelos de negócio de nossa aplicação neste capítulo. Seguindo nosso padrão, eu a chamei de `Models`. Nesta seção teremos apenas a classe `Servico`, com o código apresentado na sequência. Nele temos a definição de quatro propriedades, sendo a última delas (`valorFormatado`) uma especialização em relação ao valor de cada serviço, para que seja possível obtermos o valor formatado, que é o que exibiremos na listagem de serviços.

```
namespace Capítulo04.Models
```

```

{
    public class Servico
    {
        public long? ServicoID { get; set; }
        public string Nome { get; set; }
        public double Valor { get; set; }

        public string ValorFormatado { get { return string.Format("R$ {0:N2}", this.Valor); } }
    }
}

```

Como dito anteriormente, a primeira página que o usuário visualizará será a de listagem dos serviços registrados. Como estamos implementando esta funcionalidade agora, e não temos nenhum serviço Web ou base de dados local que poderiam nos fornecer os dados de que precisamos para testes, faremos uso de dados em coleções. Para isso, seguiremos a estrutura que o Visual Studio cria como exemplo, que vimos no capítulo 2. Então, vamos criar uma pasta chamada `Services`, no mesmo nível em que criamos a `Models`. A partir do próximo capítulo trabalharemos o acesso a dados e, nos capítulos 10 e 11, criaremos e consumiremos serviços REST.

Dentro desta nova pasta, precisamos criar a interface para os métodos que precisaremos implementar para trabalhar com os dados de nossa aplicação. Crie, na pasta, uma nova interface, que nomeei aqui de `IDataStore`. Veja o código dela na sequência. Não há novidades, ela é genérica, pois poderemos implementá-la para diversos tipos de dados de nosso modelo.

```

namespace Capitulo04.Services
{
    public interface IDataStore<T>
    {
        void Add(T item);
        void Update(T item);
        void Delete(T item);
        T GetById(long? id);
        IEnumerable<T> GetAll();
    }
}

```

Agora precisamos criar a classe para trabalhar com os dados específicos de `Servico` e esta classe implementará a interface que criamos anteriormente. A princípio, implementaremos apenas o objeto que conterá os itens que usaremos como teste para nossa aplicação, com sua inicialização de valores para serem utilizados. Observe que o campo é definido como `static` para que possa ser compartilhado entre as instâncias da classe `ServicoDataStore`. Como a classe implementa `IDataStore`, ela é obrigada a implementar o comportamento de todos os métodos definidos na interface, se não ocorrerá erro de compilação. Para evitar este erro, os métodos serão criados, mas sem comportamento, apenas para termos o código compilável.

Vamos criar esta classe na pasta `services`, eu a nomeei de `ServicoDataStore`. Veja o código na sequência, mas observe que temos um comportamento para o método `GetAll()`, logo no início, que retorna todos os serviços, classificados pela propriedade `Nome`.

```

namespace Capitulo04.Services
{
    public class ServicoDataStore : IDataStore<Servico>
    {
        private static List<Servico> servicos = new List<Servico>()
        {

```

```

        new Servico() { ServicoID = 1, Nome = "Freios", Valor = 100},
        new Servico() { ServicoID = 2, Nome = "Suspensão", Valor = 200},
        new Servico() { ServicoID = 3, Nome = "Caixa de direção", Valor = 300},
        new Servico() { ServicoID = 4, Nome = "Caixa de câmbio", Valor = 400},
        new Servico() { ServicoID = 5, Nome = "Alinhamento/Balanceamento", Valor = 25}
    };

    public IEnumerable<Servico> GetAll()
    {
        return servicos;
    }

    public void Add(Servico servico)
    {
        throw new System.NotImplementedException();
    }

    public void Delete(Servico servico)
    {
        throw new System.NotImplementedException();
    }

    public Servico GetById(long? id)
    {
        throw new System.NotImplementedException();
    }

    public void Update(Servico servico)
    {
        throw new System.NotImplementedException();
    }
}
}

```

Se fôssemos seguir o que vimos nos exemplos desenvolvidos até este ponto do livro, partiríamos para a implementação da visão. Entretanto, como o foco deste capítulo está no MVVM, precisamos implementar a classe responsável pelo modelo a ser utilizado na visão. Esta classe é a que representa o ViewModel, da sigla MVVM. Na prática deste padrão, implementamos o mínimo possível de código no code-behind da visão. Fazemos uso da classe ViewModel para isso.

Vamos criar a estrutura de pastas para hospedar nossa nova classe. Crie uma pasta no projeto Xamarin Forms chamada `ViewModels`. Dentro dela, crie outra, chamada `servicos` e, dentro desta última pasta, crie uma classe chamada `ListagemViewModel` e implemente nela o código apresentado na sequência.

Observe que na classe criamos nossa fonte de dados, por meio do campo chamado `DataStore`. Em seguida, temos uma propriedade do tipo `ObservableCollection`, que será responsável por fornecer os dados de serviços para a visão. No construtor, instanciamos esta propriedade e a populamos com o retorno do método `DataStore.GetAll()`.

```

namespace Capitulo04.ViewModels.Servicos
{
    public class ListagemViewModel
    {

```

```

    public IDataStore<Servico> DataStore = new ServicoDataStore();
    public ObservableCollection<Servico> Servicos { get; set; }

    public ListagemViewModel()
    {
        Servicos = new ObservableCollection<Servico>(DataStore.GetAll());
    }
}

```

Com o modelo de negócio, o modelo para a visão e a fonte de dados implementados, precisamos definir, enfim, nossa visão para `Servico`. Como realizamos anteriormente, vamos criar uma pasta específica para as visões, com nome `Views`, e dentro dela, outra pasta, chamada `Servicos`. Dentro desta pasta criei uma `Content Page` chamada `ListagemView`. Veja o código dela na sequência. Após a listagem estão as explicações.

```

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"

    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

    xmlns:viewModel = "clr-namespace:Capitulo04.ViewModels.Servicos"

    x:Class = "Capitulo04.Views.Servicos.ListagemView"

    xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"

    ios:Page.UseSafeArea = "true"

    Title = "Serviços" >

    <ContentPage.BindingContext>

        <viewModel:ListagemViewModel/>

    </ContentPage.BindingContext>

    <ContentPage.ToolbarItems>

        <ToolbarItem Icon = "plus.png" />

    </ContentPage.ToolbarItems>

    <ContentPage.Content>

        <StackLayout Padding = "10, 0, 0, 0" VerticalOptions = "FillAndExpand" >

```

```

<ListView x:Name = "listView" HasUnevenRows = "True" ItemsSource = "{Binding Servicos}" >

<ListView.ItemTemplate>

<DataTemplate>

<ViewCell>

<StackLayout Padding = "10" >

<Label Text = "{Binding Nome}" FontSize = "18" FontAttributes = "Bold" />

<Label Text = "{Binding ValorFormatado}" FontSize = "14" />

</StackLayout>

</ViewCell>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

</StackLayout>

</ContentPage.Content>

</ContentPage>

```

Precisamos especificar que o contexto que fornecerá e receberá dados para a visão e responderá a eventos ocorridos nela será nossa classe ViewModel e não a classe da visão que estamos implementando. Desta maneira, para que possamos referenciar a ViewModel na visão, precisamos trazê-la para a página, e isso é feito definindo uma variável de namespace, chamada `viewModel`, dentro da tag `<ContentPage>`. Também trazemos para a página o namespace `ios` para que possamos configurar o uso de área segura no iPhone.

A primeira tag filha de `<ContentPage>` é a `<ContentPage.BindingContext>`, que faz uso do objeto `viewModel` para definir a classe `ListagemViewModel` como fornecedora de um objeto que será o responsável pelo fornecimento de dados para a visão. Após o fechamento da tag de definição de contexto, há um `StackLayout` que contém um `ListView`.

Trazemos para esta visão o uso de barra de ferramentas, ou `Toolbar`, como é chamado na XAML, que vimos em funcionamento no capítulo 2. Cada elemento `<ToolbarItem>` pode conter uma imagem (`Icon`) ou texto a exibir (`Text`). Caso sejam informados os dois, o ícone prevalece e o texto não é exibido. Podemos ter dois níveis de itens, definidos pela propriedade `order`, que podem ser `Primary` ou `Secondary`. O primeiro valor faz com que o item apareça na barra de navegação no iOS e o segundo exibe uma barra abaixo da navegação para o item ser apresentado. O Android mostra um botão na barra de navegação que exibe as opções se a propriedade `order for Secondary`. É preciso aqui avaliar a necessidade. Eu achei mais elegante usar o `default`, que é o `Primary`. Uma última propriedade é a `Priority`, que define a ordem em que os itens são exibidos na barra de tarefas. Estes valores são números inteiros, entre aspas, começando em "0". Como no exemplo apenas um item é exibido e optei por fazer uso de uma imagem, estas propriedades não foram necessárias. Lembre-se de que você precisa ter nos projetos das plataformas a imagem `plus.png`, ou outra que queira utilizar.

Em relação à `<ListView>`, alerto para a propriedade `x:Name`, pois, como precisaremos manipular o elemento via code-behind, é preciso expô-lo por meio de um nome.

Com toda a implementação anterior realizada, resta-nos apenas implementar a classe `App` para invocar nossa página. Veja o código a seguir para o construtor da classe. Note que estou definindo o uso de títulos grandes e protegendo a área segura para o iPhone. Removi de meu projeto a visão `MainPage`, pois ela se tornou desnecessária. Após implementar o código apresentado, execute sua aplicação.

```
public App ()  
{  
    InitializeComponent();  
  
    var navigationPage = new Xamarin.Forms.NavigationPage( new Views.Servicos.ListagemView());  
    navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetPrefersLargeTitles( true );  
  
    MainPage = navigationPage;  
}
```

Para que você possa comparar sua implementação com a que fizemos aqui no livro, veja na figura a seguir a aplicação em execução.

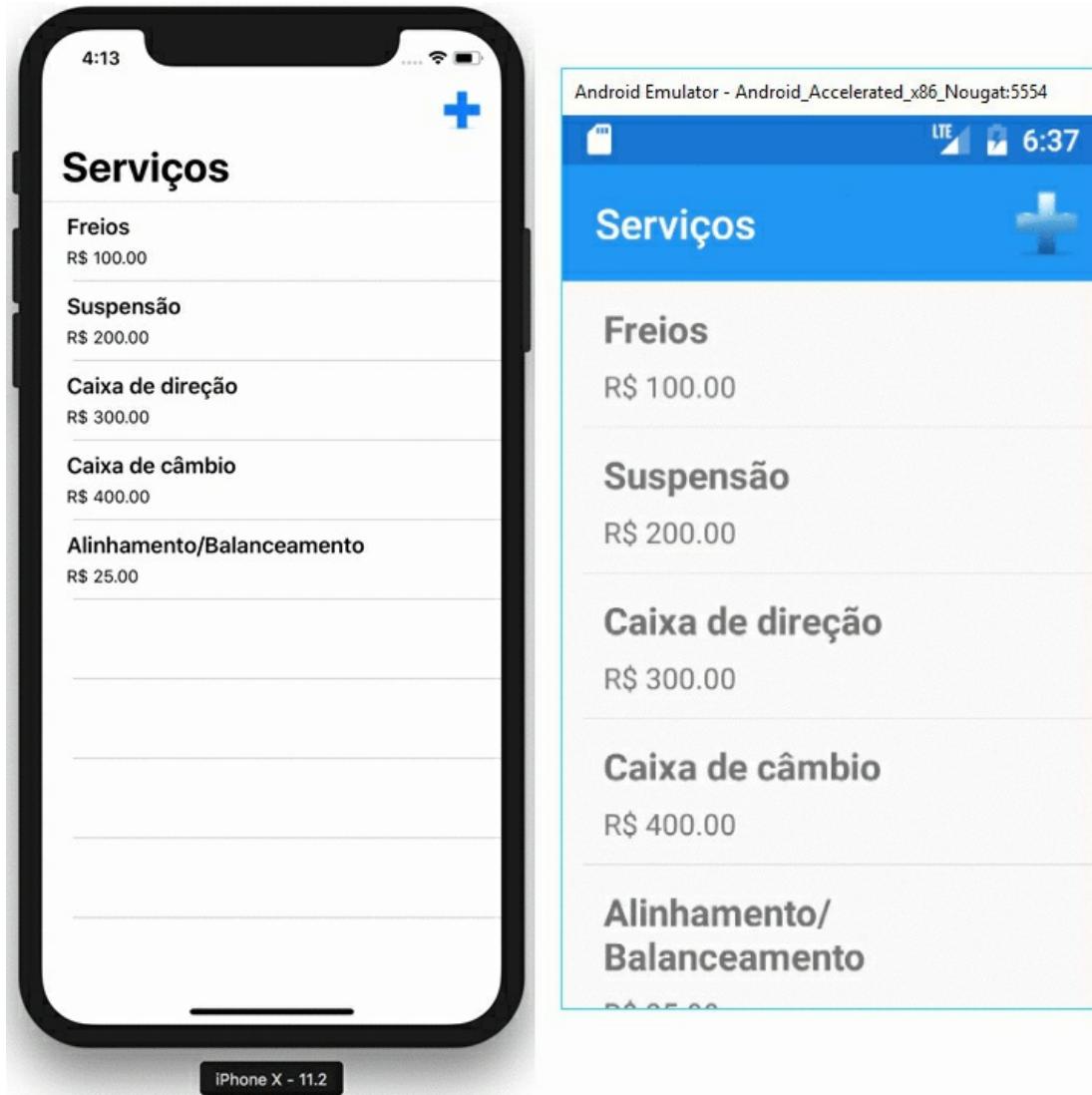


Figura 4.2: Listagem de Serviços

## 4.2 Alteração de um serviço

Com a listagem de serviços implementada, precisamos agora pensar na possibilidade de o usuário selecionar um deles no `Listview`, para visualizar todos os dados de tal serviço e poder realizar alterações nele. Para isso, implementaremos uma nova visão com o `ViewModel` para ela e manipularemos os dados por meio de nosso `Datastore`. Faremos uso do MVVM com troca de mensagens pelo `Messaging Center` do Xamarin e veremos um novo controle do Xamarin Forms, o `Table View`.

Na pasta `Servicos`, dentro de `Views`, crie uma nova `Content Page` e atribua a ela o código apresentado na sequência. Note que tiramos da tag `<ContentPage>` a propriedade `Title`. Ela será atribuída pelo code-behind. Eu dei a esta nova visão o nome de `CRUDview`.

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

```
<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
```

```

xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

x:Class = "Capitulo04.Views.Servicos.CRUDView"

xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOS;assembly=Xamarin.Forms.Core"

ios:Page.UseSafeArea = "true" >

<ContentPage.Content>

<StackLayout>

<TableView Intent = "Form" >

<TableRoot>

<TableSection Title = "Dados do serviço" >

<EntryCell Label = "Nome:" Text = "{Binding Nome}" ></EntryCell>

<EntryCell Label = "Valor:" Text = "{Binding Valor}" Keyboard = "Numeric" HorizontalTextAlignment = "End" ></EntryCell>

<ViewCell>

<StackLayout>

<Button Text = "Gravar Alterações" FontAttributes = "Bold" Command = "{Binding GravarCommand}" />

</StackLayout>

</ViewCell>

</TableSection>

</TableRoot>

</TableView>

</StackLayout>

</ContentPage.Content>

</ContentPage>

```

No código apresentado, podemos notar o novo controle do Xamarin Forms, o `Table View`. Este controle é um componente pré-configurado para alguns usos comuns em aplicativos móveis, como, em nosso caso, a entrada de

dados. A propriedade `Intent` deste controle é quem determina o objetivo a ser abstraído, que na nossa implementação é o `Form`, que representa visualmente um formulário de coleta de dados. Outros valores são permitidos: `Data`, `Menu` e `Settings`. O que acha de depois testar estas variações?

Dentro da tag `<TableView>` existe outra tag, que representa a raiz da tabela (`<TableRoot>`), onde seções (`<TableSection>`) podem ser definidas para agrupamento de dados comuns. Em nosso exemplo teremos apenas uma seção.

Dentro das `<TableSection>`, é possível inserir apenas componentes que sejam derivados de `viewCell`, ou seja, não é qualquer controle que pode ser inserido dentro de seções, mas podemos resolver este problema com o elemento `<viewCell>`, que logo utilizaremos. Em nosso código estamos fazendo uso de `<EntryCell>`, que é semelhante ao `<Entry>`, mas específico para o `<TableView>`. Nestes controles ligamos à propriedade `<Text>` com valores que estarão no `Binding Context`, ou seja, estarão ligados com nossa `View-Model`. Veja que, para o valor, o teclado será o `Numeric` e o texto será alinhado à direita (`end`).

Após os campos de entrada, será exibido um `button`, que você pode ver dentro de um `stackLayout` que está dentro de um `viewCell`. A ação que deverá ocorrer quando o usuário clicar no botão está ligada (`Binding`) à propriedade `command` do `<Button>`. O comum para capturar eventos de seleção de um botão seria a implementação de um método que capturasse o evento `clicked` dele. Entretanto, estamos fazendo uso de MVVM e Data Bindings, o que nos remete a `Commands`. Pense nos `Commands` como recursos para manipularmos eventos que ocorram em uma visão, em sua respectiva `ViewModel`.

Com a visão implementada, precisamos agora codificar a chamada a ela e isso deverá ocorrer quando o usuário selecionar um item no `Listview` de serviços. Desta maneira, precisaremos realizar algumas modificações no código implementado na seção anterior. A primeira mudança está relacionada à tag `<ListView>`, que precisará definir uma ligação à propriedade `ItemSelected`, que apontará para uma propriedade na `ViewModel` da listagem. Veja o novo código na sequência. Lembre-se de que ele está na `ListagemView` de serviços.

```
<ListView x:Name = "listView" HasUnevenRows = "True" ItemsSource = "{Binding Servicos}" SelectedItem = "{Binding ServicoSelecionado}" >
```

Apenas relembrando, no capítulo anterior, manipulamos o item selecionado por meio de um método implementado no code-behind da visão. Agora, estamos ligando este evento a uma propriedade na `ViewModel`. Veja o código dela na sequência. Lembre-se de que este código é para a classe `ListagemViewModel` de serviços.

Observe que a definição da propriedade está de maneira completa, com definição de um campo privado e implementação específica para os métodos `get()` e `set()` para este campo. Veja a chamada ao método `Send()` de `MessagingCenter`. Por meio desta invocação, enviamos uma mensagem a uma lista de mensagens registradas na aplicação. A mensagem enviada procura por uma lista registrada chamada `Mostrar` e que receba um argumento do tipo `Servico`. Este argumento é enviado à mensagem como primeiro parâmetro do método `Send()`. Atente para a verificação de não nulidade do valor recebido, pois caso o objeto seja nulo, a aplicação travará, como já vimos anteriormente.

```
private Servico servicoSelecionado;  
  
public Servico ServicoSelecionado  
{  
    get { return servicoSelecionado; }  
  
    set  
    {  
        if ( value != null )  
        {  
            servicoSelecionado = value ;  
            MessagingCenter.Send<Servico>(servicoSelecionado, "Mostrar" );  
        }  
    }  
}
```

```
    }
}
}
```

## MESSAGING CENTER

O `MessagingCenter` é uma ferramenta que permite uma comunicação entre a camada de visão e de modelo de visão quando se utiliza o padrão MVVM. Consiste de três operações, sendo elas: 1) realizar o registro da visão no serviço de mensageria (o `MessagingCenter`); 2) enviar mensagens para a lista registrada; e 3) cancelar a assinatura da lista no serviço de mensageria. Quando se realiza o registro em uma lista de mensagens (`subscribe`), ela fica "escutando" e esperando mensagens (`send`) até que a assinatura seja cancelada (`unsubscribe`).

Agora, precisamos implementar a inscrição de algum objeto "ouvintor" que ficará à espera de mensagens de nome "Mostrar", que recebe um `Servico`. E este objeto é nossa visão de listagem de serviços. Veja o código a seguir, que você deverá inserir dentro da classe `ListagemView`.

```
protected override void OnAppearing () {
{
    base.OnAppearing();

    if (listView.SelectedItem != null)
        listView.SelectedItem = null;

    MessagingCenter.Subscribe<Servico>( this , "Mostrar" , async (servicoSelecionado) => { await Navigation.PushAsync( new
CRUDView(servicoSelecionado));
});
}
```

A verificação de não nulidade do item selecionado se dá pelo fato de que, ao selecionarmos um item no `ListView`, este item continua marcado como selecionado, mesmo que o controle tenha sido repopulado. Então, quando selecionamos um item, temos a visão de CRUD e retornamos para a listagem. Desta maneira, não poderíamos selecionar o mesmo item, pois isso não seria visto pelo controle como uma nova seleção.

Ao atribuirmos `null` ao `SelectedItem`, causamos o disparo da propriedade `ServicoSelecionado` do nosso `ViewModel`, e é por isso que lá (na propriedade) é preciso verificar se o valor recebido é nulo, antes de enviar a mensagem para o serviço de mensageria. Se não tivermos este tratamento, a aplicação trava.

Observe que há a sobreescrita (`override`) do método `OnAppearing()`, que será invocado sempre que a visão for exibida. Neste método, adotaremos o registro de mensagens para o `MessagingCenter`. Este registro se dá pela invocação do método `Subscribe()` da classe `MessagingCenter`.

O tipo do dado para a lista registrada está definido entre `<>`. O primeiro argumento para o método é o "ouvintor" responsável por receber as mensagens, em nosso caso, o objeto da classe `ListagemView`. O segundo argumento refere-se ao nome da lista que se está registrando e o terceiro argumento define o comportamento que deve ocorrer quando uma mensagem for enviada. Este comportamento se dá por uma função anônima (ou *arrow function*), que recebe uma mensagem, com o tipo de dado `Servico`, sendo utilizado na instanciação da visão `CRUDView`, que é enviada para a pilha de navegação de visões.

No código anterior, observe que a mensagem receberá um objeto da classe de serviço, que deixei em uma variável chamada `servicoSelecionado`. Como a visão `CRUDView` precisará deste dado para popular os controles visuais, ele é enviado via construtor para ela, entretanto, nós ainda não temos este construtor implementado, pois, em relação a

essa visão, só vimos o XAML. Desta maneira, no code-behind de `CRUDView`, implemente o código a seguir. Observe que mantemos o construtor sem argumento, pois o Xamarin Preview instancia as classes de visão por este construtor, para poder gerar suas renderizações.

```
namespace Capitulo04.Views.Servicos
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class CRUDView : ContentPage
    {
        public CRUDView()
        {
            InitializeComponent();
        }

        public CRUDView(Servico servico) : this()
        {
        }
    }
}
```

Da mesma maneira que realizamos inscrição no serviço de mensageria, precisamos cancelar esta inscrição quando ela não for mais necessária. Em nosso caso, a necessidade desta inscrição se dá apenas quando a visão de listagem estiver sendo exibida. Para cancelar a inscrição, precisamos sobreescriver o método `OnDisappearing()` na `ListagemView`, tal qual na listagem a seguir. Observe a chamada ao método `Unsubscribe()`, que recebe dois argumentos, que têm a mesma explicação dos dois argumentos do método `Subscribe()`.

```
protected override void OnDisappearing()
{
    base.OnDisappearing();
    MessagingCenter.Unsubscribe<Servico>( this , "Mostrar" );
}
```

Como informado no início desta seção, esta nova visão dará ao usuário a possibilidade de visualizar os dados de um serviço e realizar alterações nele, para que então sejam atualizados em nossa coleção na classe `ServicoDataStore`, que está na pasta `services`. Desta maneira, precisamos pensar na implementação do método de atualização desta classe. Veja-o na sequência. Lembre-se de que ele já foi implementado, porém sem o comportamento.

A primeira instrução do método recupera o objeto na coleção `servicos` que possui identificador igual ao recebido pelo método. Na sequência, este objeto recuperado é removido da coleção e o recebido é inserido. Como estamos trabalhando com coleções, e não acesso remoto ou em uma base de dados local, não temos a possibilidade de ocorrência de exceções, mas é importante você já saber que trabalhar isso é uma boa prática. Você precisará inserir a instrução `using System.Linq;` no início da classe, pois estamos fazendo uso do LINQ - recurso oferecido pelo .NET para consultas em coleções.

```
public void Update (Servico servico)
{
    var _servico = servicos.Where((Servico s) => s.ServicioID == servico.ServicioID).FirstOrDefault();
    servicos.Remove(_servico);
    servicos.Add(servico);
}
```

Como estamos utilizando o MVVM, nosso próximo passo é criar a ViewModel para a visão `CRUDView` de Serviço. Na pasta `Servicos`, dentro de `ViewModels`, vamos criar uma classe chamada `CRUDViewModel`. Veja a primeira parte na sequência, após algumas explicações.

Logo no início da classe definimos a fonte de dados (`DataStore`); uma propriedade chamada `Servico`, que conterá o objeto recebido por meio do construtor, que está implementado no mesmo código; e um `Command`, que está ligado à propriedade `command` do botão que implementamos na visão `CRUDView`.

O construtor recebe o serviço que se deseja exibir e/ou cujos dados se quer atualizar e então invoca o método `RegistrarCommands()`, que está implementado também no código da sequência. Este método, que adotaremos como responsável por registrar os `Commands` que serão disparados pela visão, instancia um `Command`, que tem como argumentos duas funções anônimas. A primeira invoca o método `Gravar`, implementado também na listagem, e envia uma mensagem para o serviço de mensageria. A segunda função anônima (*arrow function*) é responsável por definir se o botão estará habilitado ou não, e a regra que estipulamos aqui é que as propriedades que representam os controles visuais possuam valores informados para que o botão esteja habilitado. Os comportamentos para as funções anônimas podem ser substituídos por métodos. Não usaremos essa estratégia aqui, ainda, pois temos pouco código e a leitura não fica prejudicada, mas veja que a responsabilidade de gravar o serviço é delegada para um método.

```
namespace Capitulo04.ViewModels.Servicos
{
    public class CRUDViewModel
    {
        private IDataStore<Servico> DataStore = new ServicoDataStore();
        private Servico Servico { get; set; }
        public ICommand GravarCommand { get; set; }

        public CRUDViewModel(Servico servico)
        {
            this.Servico = servico;
            RegistrarCommands();
        }

        private void RegistrarCommands()
        {
            GravarCommand = new Command(() =>
            {
                Gravar();
                MessagingCenter.Send<string>("Atualização realizada com sucesso.", "InformacaoCRUD");
            }, () =>
            {
                return !string.IsNullOrEmpty(this.Servico.Nome) && this.Servico.Valor > 0;
            });
        }

        private void Gravar()
        {
            DataStore.Update(this.Servico);
        }
    }
}
```

Quando executarmos nossa aplicação e solicitarmos os dados de um serviço em específico, o botão estará habilitado para ser clicado, pois escolhemos um serviço que já possui valores válidos para a condição de habilitação do controle relativo ao `Command` configurado. Mas e se o usuário alterar os valores válidos, deixando o `Nome` vazio e o `Valor` em zero? O botão deve mudar seu estado, e é isso que faremos na sequência.

Note que, no parágrafo anterior, foi comentado que poderemos alterar o valor de uma propriedade que está em nosso `ViewModel` e que esta alteração precisa ser refletida na visão, e isso não ocorre de maneira automática. Precisamos intervir neste comportamento, que é uma característica muito importante do MVVM.

A ideia é básica: quando algum dado que está ligado à visão sofrer alteração, precisamos "notificá-la" de que houve tal mudança. Poderíamos implementar este comportamento em cada classe que venhamos a implementar como `ViewModel`, mas, se isso ocorre com frequência, por que não generalizar? Vamos então criar uma superclasse para esta funcionalidade. Dentro da pasta `ViewModels` do nosso projeto Xamarin Forms, crie uma classe chamada `BaseViewModel`, com o código apresentado na sequência. Note a definição de herança por interface para `IPropertyChanged`. Isso faz com que implementemos o código definido na listagem, que é o responsável por notificar a visão que possui a ligação com a `ViewModel` que estenderá esta classe. Após a implementação da classe, adicione, na `ListagemViewModel` o trecho : `BaseViewModel`, para a tornarmos subclasse da que estamos implementando.

```
namespace Capitulo04.ViewModels
{
    public class BaseViewModel : IPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        public void OnPropertyChanged([CallerMemberName]string name = "")
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
        }
    }
}
```

Precisamos agora, em nossa `CRUDviewModel`, declarar que ela estenderá nossa classe `BaseViewModel`. Para isso, mude a declaração da classe para `public class CRUDViewModel : BaseViewModel`.

Muito bem, com o problema contextualizado e a solução para ele implementada, precisamos codificar as propriedades que estarão ligadas à nossa visão. Se você verificar o código XAML que implementamos, notará que elas se chamam `Nome` e `Valor`. Veja o código a seguir para estas propriedades.

```
public string Nome
{
    get { return this.Servico.Nome; }

    set {
        this.Servico.Nome = value;
        ((Command)GravarCommand).ChangeCanExecute();
    }
}

private string valor;

public string Valor
{
```

```

    get { return valor; }

    set
    {
        this.valor = value;

        this.Servico.Valor = string.IsNullOrEmpty( value ) ? 0 : Convert.ToDouble(valor);
        OnPropertyChanged();
        ((Command)GravarCommand).ChangeCanExecute();
    }
}

```

Em relação ao valor, precisaremos de uma variável de nível de objeto que represente o valor informado pelo usuário na aplicação. Ele será retornado sempre pelo método `get()` da propriedade e o método `set()`, que, além de atualizar esta variável, precisará convertê-la para `double` e atribuir o valor convertido para a propriedade `valor` do objeto `Serviço`. Veja a implementação na sequência. Observe, no `set()` de `valor`, a invocação ao método `OnPropertyChanged()` após a alteração do valor da propriedade. Estas simples instruções realizam a notificação à visão sobre a alteração, para que ela possa exibir o novo valor. A invocação ao método `ChangeCanExecute()` causará a execução da segunda função anônima que inserimos no `GravarCommand` anteriormente, ou seja, a cada alteração de valores, será verificado se o botão associado ao `Command` pode ser habilitado ou não.

A conversão do valor da visão para o tipo `double` apresentada na listagem anterior não será bem-sucedida se seu dispositivo estiver em português. Isso porque a interface mostra o separador de decimais como vírgula, porém, quando a conversão da string ocorre, ele usa o padrão para a aplicação, que é definido como o ponto. Uma maneira simples para resolvemos isso é, ao iniciarmos a aplicação, definirmos a cultura (idioma e demais configurações relativas) para ela como português brasileiro. Sendo assim, no construtor da classe `App`, antes da invocação à nossa `MainPage`, insira as instruções a seguir. Você precisará inserir o namespace `System.Globalization` aos *usings*.

```

CultureInfo.DefaultThreadCurrentCulture = new CultureInfo( "pt-BR" );
CultureInfo.DefaultThreadCurrentUICulture = new CultureInfo( "pt-BR" );

```

Para finalizar e garantir que o valor atribuído a um serviço seja apresentado corretamente quando a visão de CRUD for exibida, implemente a seguinte instrução ao final do construtor parametrizado da classe `CRUDViewModel` de serviços.

```
this.valor = string.Format( "{0:N}" , servico.Valor);
```

Vamos implementar agora o código necessário no code-behind da visão `CRUDView`. Veja este código na sequência. Definimos um objeto chamado `crudViewModel`, pois neste exemplo aplicaremos O Binding Context via code-behind, e não no XAML, como fizemos na seção anterior. O construtor que recebe o serviço selecionado invoca o construtor sem argumentos, para inicializar os componentes visuais, depois instancia a `view Model` e a declara como fonte de dados para a visão. O código do construtor conclui com a definição do título para a visão `CRUDView`. Para terminar a classe, implementamos a sobreescrita dos métodos `OnAppearing()` e `OnDisappearing()`, que assinam e cancelam assinatura na lista chamada `Informacao`.

```

namespace Capitulo04.Views.Servicos
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class CRUDView : ContentPage
    {
        private CRUDViewModel crudViewModel;

        public CRUDView ()
        {

```

```

        InitializeComponent ();
    }

    public CRUDView(Servico servico) : this()
    {
        this.crudViewModel = new CRUDViewModel(servico);
        this.BindingContext = this.crudViewModel;
        this.Title = "Consulta Serviço";
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();
        MessagingCenter.Subscribe<string>(this, "InformacaoCRUD", async (msg) => { await DisplayAlert("Informação",
msg, "ok"); });
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();
        MessagingCenter.Unsubscribe<string>(this, "InformacaoCRUD");
    }
}

```

Uma vez que o usuário pode alterar os dados de serviços apresentados na `ListView`, é importante que os dados deste controle sejam atualizados quando se retornar da `CRUDView` para a `ListView`. Desta maneira, precisamos realizar algumas alterações em nossas classes. Vamos ponderar a situação. Na classe `ListagemViewModel`, temos uma propriedade `observableCollection` chamada `Servicos`. Este tipo de dado, quando possui atualizações, ele mesmo se encarrega de avisar o `ListView` sobre isso e o controle é automaticamente atualizado com o novo dado ou a alteração realizada. Já sabemos que as alterações e inserções ocorrerão na funcionalidade de CRUD, então, podemos "injetar" a propriedade `Servicos` na nossa visão e, consequentemente, para nosso `ViewModel` de CRUD.

Vamos então realizar as mudanças em nosso `CRUDViewModel`. Veja a implementação da sequência. A primeira linha, que declara o objeto privado, deve ser implementada antes do construtor, a segunda é a nova assinatura para o construtor parametrizado, e a terceira deve ser inserida dentro do construtor.

```

// Antes do construtor

private ObservableCollection<Servico> Servicos;

// Nova assinatura construtor parametrizado

public CRUDViewModel (Servico servico, ObservableCollection<Servico> servicos)

// No final do construtor

this.Servicos = servicos;

```

Precisamos mudar também o construtor de `CRUDView` para receber `Servicos` e enviá-lo para o `CRUDViewModel`. Veja as instruções a seguir. A primeira linha é a nova assinatura do construtor parametrizado para a visão e a segunda é a nova instânciação de `CRUDViewModel`, dentro do construtor. Não guardamos na visão o objeto recebido, mas o

```

encaminhamos direto, pois é a ViewModel que precisará dele. A terceira instrução é a nova invocação para a visão
CRUDView , no OnAppearing() de ListagemView .

// Nova assinatura do construtor parametrizado

public CRUDView (Servico servico, ObservableCollection<Servico> servicos) : this ()

// No final do construtor

this.crudViewModel = new CRUDViewModel(servico, servicos);

// No OnAppearing() de ListagemView

await Navigation.PushAsync( new CRUDView(servicoSelecionado, viewModel.Servicos));

```

Precisamos agora, no método `Gravar()`, realizar operações de atualização do objeto `servicos`, para que, ao retornar para a listagem, os dados exibidos pela visão estejam atualizados. Como a atualização desta propriedade é uma responsabilidade diferente de gravar o registro, vamos criar um método que assumirá esta responsabilidade. Veja o código dele na sequência. Você precisará invocá-lo antes da finalização do método `Gravar()`.

```

private void AtualizarPropriedadesParaVisao ()

{
    var servico = this .Servicos.FirstOrDefault(s => s.ServicoID == Servico.ServicoID);

    int indexOfServico = this .Servicos.IndexOf(servico);

    this .Servicos[indexOfServico] = servico;
}

```

Só com estas alterações já teremos nosso `Listview` exibindo o dado atualizado, entretanto, se a alteração ocorrer na propriedade `Nome`, que classifica nossa listagem, o item não é repositionado. Para contornarmos este problema, vamos criar um método em nossa classe `ListagemViewModel`, como é apresentado na sequência. Veja que instanciamos um novo `observableCollection`, com a lista de serviços classificada, depois, tal qual fizemos para o `CRUD`, notificamos a visão da mudança na propriedade. Lembre-se de que, para isso, precisamos estender a classe `BaseViewModel`. Com essa nova implementação, podemos tirar do construtor a responsabilidade de popular a propriedade `servicos`, sendo assim, remova dele a instrução que faz isso.

```

public void AtualizarServicos ()

{
    if (Servicos == null )

        Servicos = new ObservableCollection<Servico>(DataStore.GetAll().OrderBy(s => s.Nome));

    else

        Servicos = new ObservableCollection<Servico>(Servicos.OrderBy(s => s.Nome));

    OnPropertyChanged(nameof(Servicos));
}

```

Agora, precisamos alterar nosso método `OnAppearing()` da classe `ListagemView` para que invoque nosso novo método `AtualizarServicos()`. Como ele está na `ViewModel` de listagem, precisamos trazer para a visão a declaração de um campo para acessá-la. Desta maneira, no início da classe, declare-a como mostra a primeira linha da listagem a seguir. Precisamos inicializar agora este objeto e o faremos no construtor, por meio da instrução que está na segunda

linha da listagem. A terceira linha deve ser inserida no `onAppearing()`, após `base.OnAppearing()`.

```
// Início da classe

private ListagemViewModel viewModel;

// No final do construtor

BindingContext = viewModel = new ListagemViewModel();

// No OnAppearing()
viewModel.AtualizarServicos();
```

Com estas implementações, sabemos que os serviços serão exibidos na `Listview` quando a página for exibida pela primeira vez e quando o usuário retornar do `CRUDView`, garantindo sua atualização. Note, no método `AtualizarServicos()`, que recorremos à nossa fonte de dados apenas na instanciação da `ListagemViewModel`, verificando a quantidade de elementos em `servicos`. Como nossa aplicação ainda não sofre processo de sincronização, a fonte de dados não mudará entre as navegações das visões. Execute sua aplicação e confira com a imagem a seguir. Experimente alterar o nome de um serviço para que a ordem de exibição na listagem possa ser alterada.

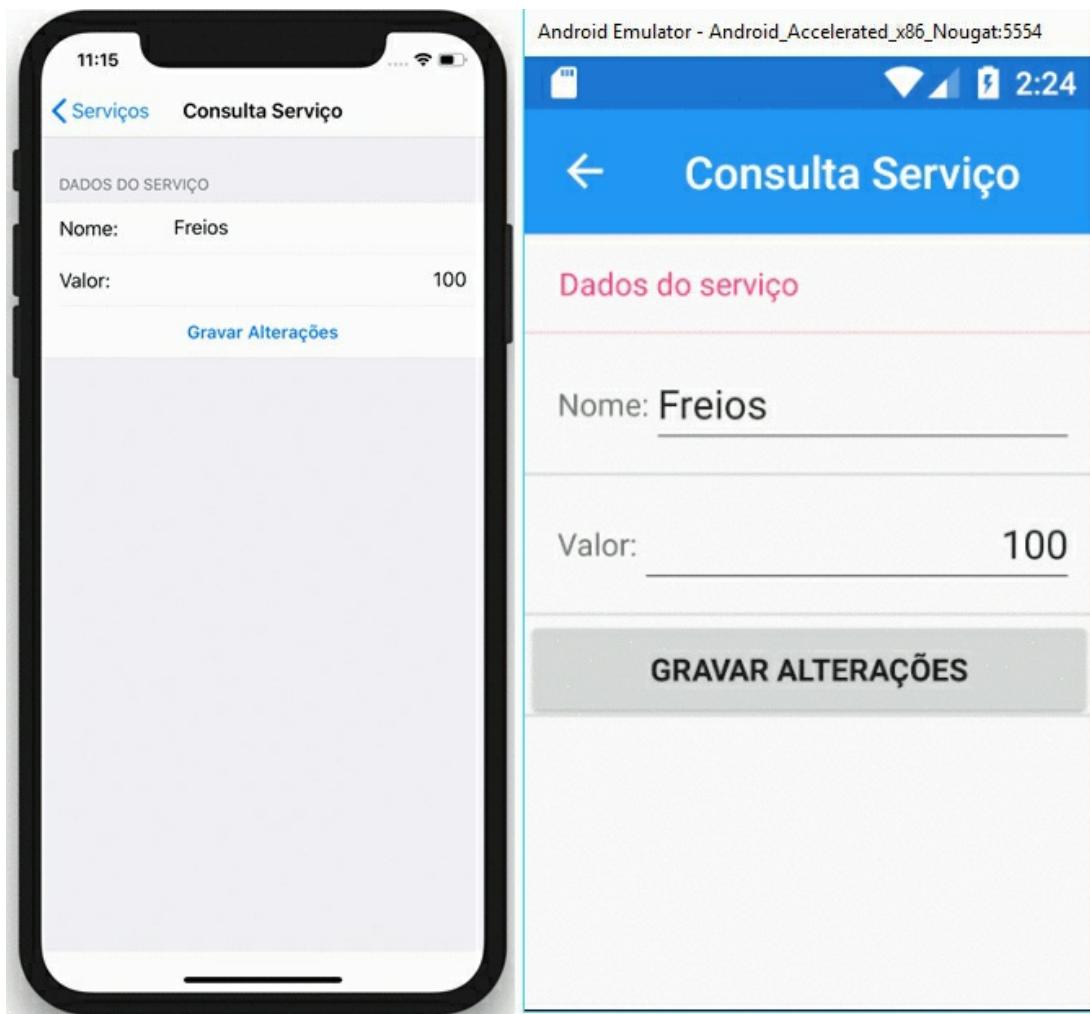


Figura 4.3: Alteração de Serviços

### 4.3 Inserção de um novo serviço

Para que um novo serviço possa ser inserido em nossa coleção, precisamos configurar o nosso `ToolbarItem` para que seja ligado a um `command` que cumprirá esta finalidade. Desta maneira, altere a linha de declaração dele no XAML de sua `ListagemView`, tal qual é mostrado no código a seguir.

```
<ToolbarItem Icon="plus.png" Command="{Binding NovoCommand}" />
```

Na sequência, precisamos implementar o `Command NovoCommand` na `ListagemViewModel` e isso será feito em três partes. A primeira é a declaração da propriedade logo no início, antes do construtor, e o código que deve ser inserido é o que está na sequência. Fique atento aos `usings`.

```
public ICommand NovoCommand { get ; set ; }
```

A segunda etapa será a responsável por registrar o comportamento que o `NovoCommand` deverá ter quando o `ToolbarItem` for pressionado. Para isso, vamos criar um método que será responsável por criar todos os `commands` que forem necessários no `ListModel`. Veja o código na sequência. Observe que mandamos uma mensagem para a mesma lista que utilizamos para a alteração de um serviço já existente. A diferença é que agora estamos enviando um objeto novo da classe `Servico`. É fácil perceber a reutilização da `Content Page CRUDView`. Invoque este novo método ao final de seu construtor.

```
private void RegistrarCommands ()  
{  
    NovoCommand = new Command( () =>  
    {  
        MessagingCenter.Send<Servico>( new Servico(), "Mostrar" );  
    });  
}
```

Para finalizar, precisamos realizar algumas alterações em nosso método `Gravar()`, pois temos agora a possibilidade de inserção de um novo objeto. Veja a primeira instrução do código a seguir, que deve ser inserida antes da invocação ao `Update()`. Ela tem uma variável que indicará quando o objeto a ser inserido é novo ou não. Note que utilizamos o operador ternário `? :`. Depois desta instrução, temos a nova implementação para o método `AtualizarPropriedadesParaVisao()`, que agora recebe o valor que declaramos em `Gravar()`. Veja que o comportamento é diferente quando estivermos inserindo um novo serviço. Com esta mudança, no método `Gravar()`, quando formos invocar o `AtualizarPropriedadesParaVisao()`, a nova variável `ehNovoServico` deverá ser enviada.

Nossa ideia é que, após a inserção de um novo serviço, a visão fique preparada para a inserção de novos dados, sendo possível inserir mais de um serviço antes de retornar à página de listagem. Para isso, é importante que a página não exiba os valores já utilizados, por isso a instanciação de um novo objeto e atribuição de valores às propriedades ligadas.

```
// No método Gravar(), antes do Update()  
  
var ehNovoServico = ( this .Servico.ServicoID == null ? true : false );  
  
// Novo método, tirar o anterior de mesmo nome e sem parâmetro  
  
private void AtualizarPropriedadesParaVisao ( bool ehNovoServico )
```

```

{
    if (ehNovoServico)
    {
        this .Servicos.Add(Servico);
        this .Servico = new Servico();
        this .Nome = string .Empty;
        this .Valor = string .Empty;
    }
    else
    {
        var servico = this .Servicos.FirstOrDefault(s => s.ServicoID == Servico.ServicoID);
        int indexOfServico = this .Servicos.IndexOf(servico);
        this .Servicos[indexOfServico] = servico;
    }
}

```

Agora você pode executar sua aplicação e pressionar o botão de "Novo serviço" que implementamos na barra de tarefas. É para aparecer a já conhecida visão de informação de dados de serviço, porém, sem dados. Insira um nome e valor para o novo serviço, grave e retorne para a listagem. Deve aparecer nela seu novo serviço.

Porém, uma questão precisa ser apontada. Não temos o valor da propriedade `ServicoID`, que seria o valor de identidade para o objeto registrado. Veja a nossa nova implementação para o método `update()` da classe `ServicoDataStore` na sequência.

Observe que agora, antes de realizar a recuperação do serviço, verificamos se o objeto recebido possui valor na propriedade `servicoID` e, se retornar verdadeiro, é realizada a lógica para atualização (em caso de alteração). Caso o valor da propriedade seja nulo, é obtido o valor máximo da propriedade e incrementado em um, para definir o valor para o novo objeto.

```

public void Update (Servico servico)

{
    if (servico.ServicoID != null )
    {
        var _servico = servicos.Where((Servico s) => s.ServicoID == servico.ServicoID).FirstOrDefault();
        servicos.Remove(_servico);

    } else
    {
        servico.ServicoID=servicos.Max(s => s.ServicoID)+1;
    }
    Add(servico);
}
}

```

Para finalizar, o método (do próprio `Datastore`) `Add()` agora é invocado para realizar a inserção, em vez de fazermos direto na coleção, como estávamos fazendo. Este método faz parte da interface e nós já o tínhamos implementado,

mas sem comportamento. Agora, com a tela de inserção habilitada, precisamos também liberar a funcionalidade na classe. Logo trabalharemos a situação do `Remove()`, que ainda elimina diretamente na coleção. Veja o código do método atualizado na sequência.

```
public void Add (Servico servico)
{
    servicos.Add(servico);
}
```

Execute agora sua aplicação e insira um serviço como teste. Após a gravação dos dados, veja que o nome que você digitou continua aparecendo, mas que o valor foi limpo e nada aparece. No método `AtualizarPropriedadesParaVisao()`, quando se identifica que é para preparar a visão para a entrada de novos dados, nós estamos atribuindo novos valores para as propriedades ligadas à visão. Então, por que apenas o valor foi limpo? É que, no `set()` de valor, nós já implementamos a notificação de alteração da propriedade na ViewModel, mas não temos isso ainda para o `Nome`. Vamos resolver isso. Insira a instrução a seguir após a atribuição do novo valor de nome, no método `set()`, da propriedade `Nome`. Teste novamente sua aplicação e veja-a funcionando agora. Consegue visualizar o funcionamento do MVVM?

```
OnPropertyChanged();
```

## 4.4 Remoção de um serviço da coleção

Precisamos agora implementar a funcionalidade de remoção de um serviço da lista de serviços cadastrados. Adotaremos a estratégia de o usuário selecionar na listagem de serviços qual é o que ele desejará remover. Faremos isso oferecendo um menu que se tornará visível quando o usuário arrastar um serviço da lista para a esquerda, no iOS. Para o Android, basta manter pressionado o item da lista por alguns instantes. A inserção destes menus, que são conhecidos como `ContextActions`, deve ser implementada logo no início da tag `<ViewCell>`, em nosso `ListagemView.xaml`, tal qual é apresentado no código a seguir.

```
<ViewCell>

<ViewCell.ContextActions>

<MenuItem Command = "{Binding Path=BindingContext.EliminarCommand, Source={x:Reference listView}}" CommandParameter = "
{Binding .}" Text = "Remover" IsDestructive = "True" />

</ViewCell.ContextActions>
```

Cada elemento `<MenuItem>` refere-se a uma opção que será exibida para o item da `ListView` selecionado. Observe como estamos fazendo o `Binding` para este componente na propriedade `Command`. Estamos dizendo que ligamos o `Command` do componente ao `BindingContext` de nosso `ListView`, ao qual demos, anteriormente, o nome de `listView`.

Essa declaração, não tão simplificada, se deve ao fato de que, se utilizarmos apenas o `Binding` e o nome do `Command`, o contexto atual para o `Command` seria a classe `Servico`, pois o `MenuItem` está sendo utilizado na definição da apresentação de um item da coleção definida em `ItemsSource` do `ListView` (`<ListView.ItemTemplate>`). Com isso, estamos fazendo o que é conhecido como `Binding View-to-View`, ou seja, estamos ligando um controle da visão a outro controle da própria visão.

A propriedade `CommandParameter` está ligando a ela o item que está em exibição no momento em que o botão for pressionado. Ele será, então, enviado como parâmetro para o `Command EliminarCommand`. Já a propriedade `IsDestructive`, quando verdadeira, apresenta o botão em cor vermelha no iOS. No Android, este menu é apresentado no topo da `ListView`.

Agora precisamos implementar em nossa `ListagemViewModel` o Command `EliminarCommand` e você pode obtê-lo pela listagem a seguir. A segunda declaração da listagem deve ser implementada ao final do método `RegistrarCommands()`, que já existe em sua classe, pois o implementamos anteriormente.

```
public ICommand EliminarCommand { get; set; }

EliminarCommand = new Command<Servico>((servico) =>
{
    MessagingCenter.Send<Servico>(servico, "Confirmação");
});
```

Precisamos também, na `ListagemViewModel`, implementar o método que realizará a remoção do serviço. O código para este método está na sequência. Veja que, após a remoção do `Datastore`, o removemos também da coleção, o que fará com que nossa `ListView` seja atualizada.

```
public void EliminarServico (Servico servico)
{
    DataStore.Delete(servico);
    this.Servicos.Remove(servico);
}
```

Como pode ser verificado no código anterior, invocamos um método de nosso `Datastore`, desta maneira, precisamos implementá-lo como na sequência.

```
public void Delete (Servico servico)
{
    var _servico = servicos.Where((Servico s) => s.ServicoID == servico.ServicoID).FirstOrDefault();
    servicos.Remove(_servico);
}
```

Agora, precisamos implementar a mensagem "Confirmação", que será a responsável por interagir com o usuário quando ele remover um serviço. O código a seguir deve ser implementado no método `OnAppearing()` da `ListagemView`. Observe que usamos o `DisplayAlert()` para realizar uma pergunta ao usuário e, a partir da resposta dele, as ações são tomadas, que em nosso caso referem-se à remoção do serviço e à atualização dos dados que populam a `ListView`.

```
MessagingCenter.Subscribe<Servico> ( this , "Confirmação" , async (servico) =>
{
    if ( await DisplayAlert ( "Confirmação" ,
        $ "Confirma remoção de {servico.Nome.ToUpper()}?" , "Yes" , "No" ) )
    {
        this.viewModel.EliminarServico(servico);
        await DisplayAlert ( "Informação" , "Serviço removido com sucesso" , "Ok" );
    }
});
```

Note que o registro da mensagem, que faz parte do método `OnAppearing()`, começou a ter muitas instruções. Isso pode deixar o código do método poluído. Uma boa prática é termos um método em nossa classe e o referenciarmos no

registro da mensagem. Veja, no código a seguir, a definição de um método para a mensagem e, na sequência, como o registro da mensagem deve ficar no `OnAppearing()`. Desta maneira, deixamos o `OnAppearing()` mais claro em relação a suas funcionalidades.

```
private async void RemoverServico (Servico servico)
{
    if ( await DisplayAlert ( "Confirmação" ,
        $ "Confirma remoção de {servico.Nome.ToUpper()}?" , "Yes" , "No" ) )
    {
        this .viewModel.EliminarServico(servico);
        await DisplayAlert ( "Informação" , "Serviço removido com sucesso" , "Ok" );
    }
}

// No OnAppearing

MessagingCenter.Subscribe<Servico>( this , "Confirmação" , async (servico) => await RemoverServico (servico) );
```

Para finalizar, precisamos cancelar a assinatura na lista anterior e fazemos isso no método `OnDisappearing()`, tal qual segue o código.

```
MessagingCenter.Unsubscribe<Servico>( this , "Confirmação" );
```

Agora é hora de testarmos a aplicação e vermos estas novas funcionalidades em execução. Com a aplicação no emulador, pressione um item do `ListView` e o arraste para a esquerda (no iOS) ou mantenha-o pressionado (no Android). Sua tela deverá estar semelhante ao apresentado na figura a seguir. O que acha de tentar executar também no `Xamarin Live Player`?

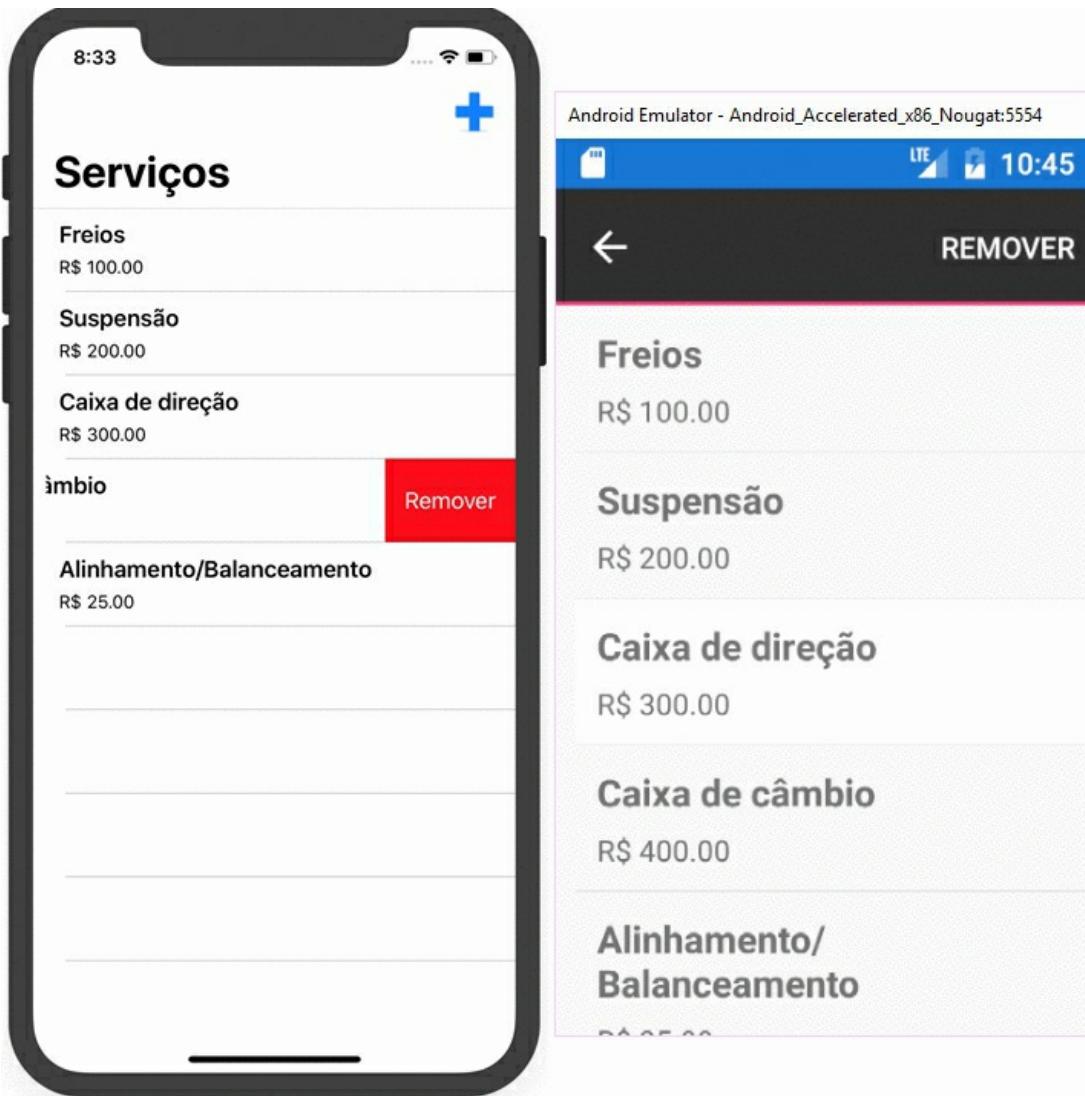


Figura 4.4: Menus nos itens da ListView com ContextActions

**A TENÇÃO :** fica convencionado, a partir deste momento, que a inscrição (`Subscribe`) e cancelamento de inscrição (`Unsubscribe`) para mensagens do Messaging Center serão sempre realizadas nos métodos `OnAppearing()` e `OnDisappearing()`, respectivamente, não se fazendo necessário comentar sobre isso nos próximos códigos.

## 4.5 Conclusão

Neste capítulo, começamos a trabalhar com o MVVM e a dar comportamento para nossas visões. Introduzimos nossos trabalhos com o serviço de mensageria (Messaging Center) oferecido pelo Xamarin Forms e vimos alguns novos recursos oferecidos, como `commands` e `ContextActions`, além de algumas técnicas para atualização de dados em uma coleção e também nas visões. Vimos como habilitar controles ligados a `Commands` com base em condições que possam ser avaliadas a cada interação do usuário com os dados e visão.

Com o trabalho que desenvolvemos neste capítulo, você já tem conhecimento para implementar a camada de apresentação, a classe para o modelo de negócios para suas aplicações e também a classe que representa a ViewModel. No próximo capítulo introduziremos a persistência de objetos em uma base de dados e a execução de nossas aplicações em dispositivos reais, por meio de implantação direta no dispositivo, sem usar o `Xamarin Live Player`.

## C APÍTULO 5

# Execução no dispositivo físico, SQLite e Entity Framework Core

No capítulo anterior, implementamos as funcionalidades relacionadas ao CRUD para o modelo de negócio de Serviços, utilizando o MVVM. Neste capítulo, aplicaremos o que aprendemos nos capítulos 3 e 4, executando nossa aplicação diretamente no dispositivo físico.

A funcionalidade principal que buscarmos implementar até o final do livro está relacionada à entrada de um veículo para manutenção na oficina. Entretanto, em nossa aplicação, precisamos ter os dados que comporão a entrada do veículo já persistidos.

Até o capítulo anterior, fizemos uso de coleções como fonte de dados. Aqui, faremos uso de uma base de dados para persistir e recuperar os dados para a aplicação. A implementação proposta para este capítulo servirá como base para os próximos. Implementaremos as funcionalidades para registro e manutenção em dados de clientes e tipos de serviços prestados.

A execução da aplicação a ser desenvolvida poderá ocorrer, já com a persistência dos dados, tanto em emuladores, quanto em dispositivos reais. Vamos trabalhar uma arquitetura que permitirá o uso do EF Core ou qualquer outro mecanismo de persistência de sua preferência, pois teremos as responsabilidades divididas em camadas, em forma de projetos.

Trabalharemos o uso de `Dependency Service` no projeto Xamarin Forms para a injeção de objetos que estejam definidos nos projetos específicos para cada plataforma, o que nos possibilitará utilizar os recursos de cada uma.

Neste começo da implementação das funcionalidades para acesso à base de dados, é necessário um pouco de paciência com a estratégia que adotaremos, pois teremos um pouco mais de código e explicações para a primeira implementação de um CRUD com acesso a dados. Após esta primeira etapa, as implementações serão mais tranquilas, uma vez que já teremos o conhecimento e a técnica absorvidos.

**A TENÇÃO :** é importante que você comece um novo projeto para as implementações que realizaremos neste capítulo.

## 5.1 Publicação da aplicação para um dispositivo iOS

Para que você se torne um desenvolvedor certificado pela Apple e possa fazer parte do `Apple Developer Program`, existe uma série de tramitações que devem ser seguidas e que estão muito bem documentadas em [https://developer.xamarin.com/guides/ios/getting\\_started/installation/device\\_provisioning/](https://developer.xamarin.com/guides/ios/getting_started/installation/device_provisioning/). Entretanto, para fazer um deploy como um desenvolvedor "simples" e sem custo, existe um processo mais simples de ser realizado, e é ele que veremos aqui.

O primeiro requisito é ter um Id Apple. Se não tem, obtenha um em <https://appleid.apple.com/account#!&page=create>. Em seguida, em seu Mac, acesse o XCode.

Com o XCode ativo, acesse o `Xcode Menu -> Preferences`. Na categoria `Accounts`, clique na base esquerda no botão `+`, para adicionar seu Id Apple. Sua senha será solicitada neste momento e a janela a seguir é exibida.

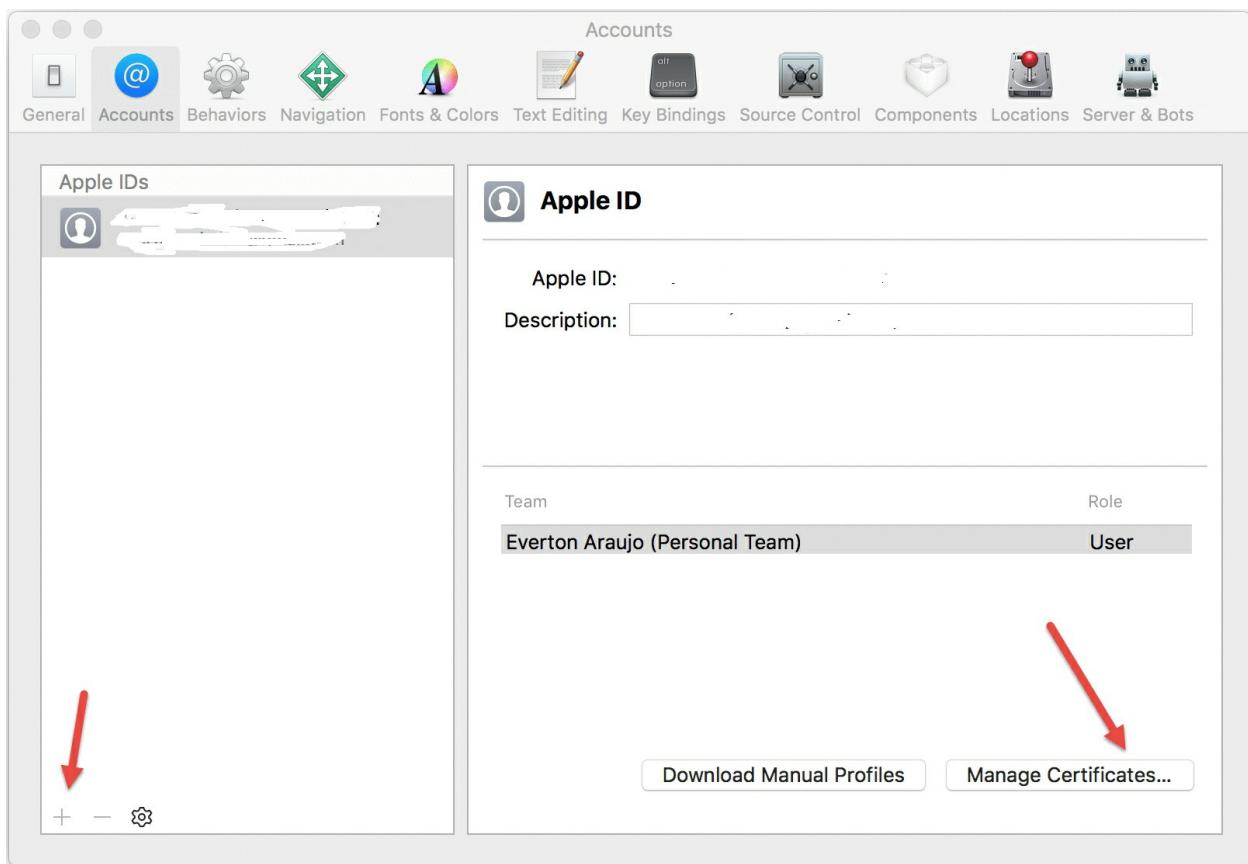


Figura 5.1: Criando uma conta no XCode

Com a conta criada, precisaremos criar uma signing Identity . Para isso, clique no botão Manage Certificates... e a janela apresentada na figura a seguir é exibida. Clique no botão create e selecione ios Development .

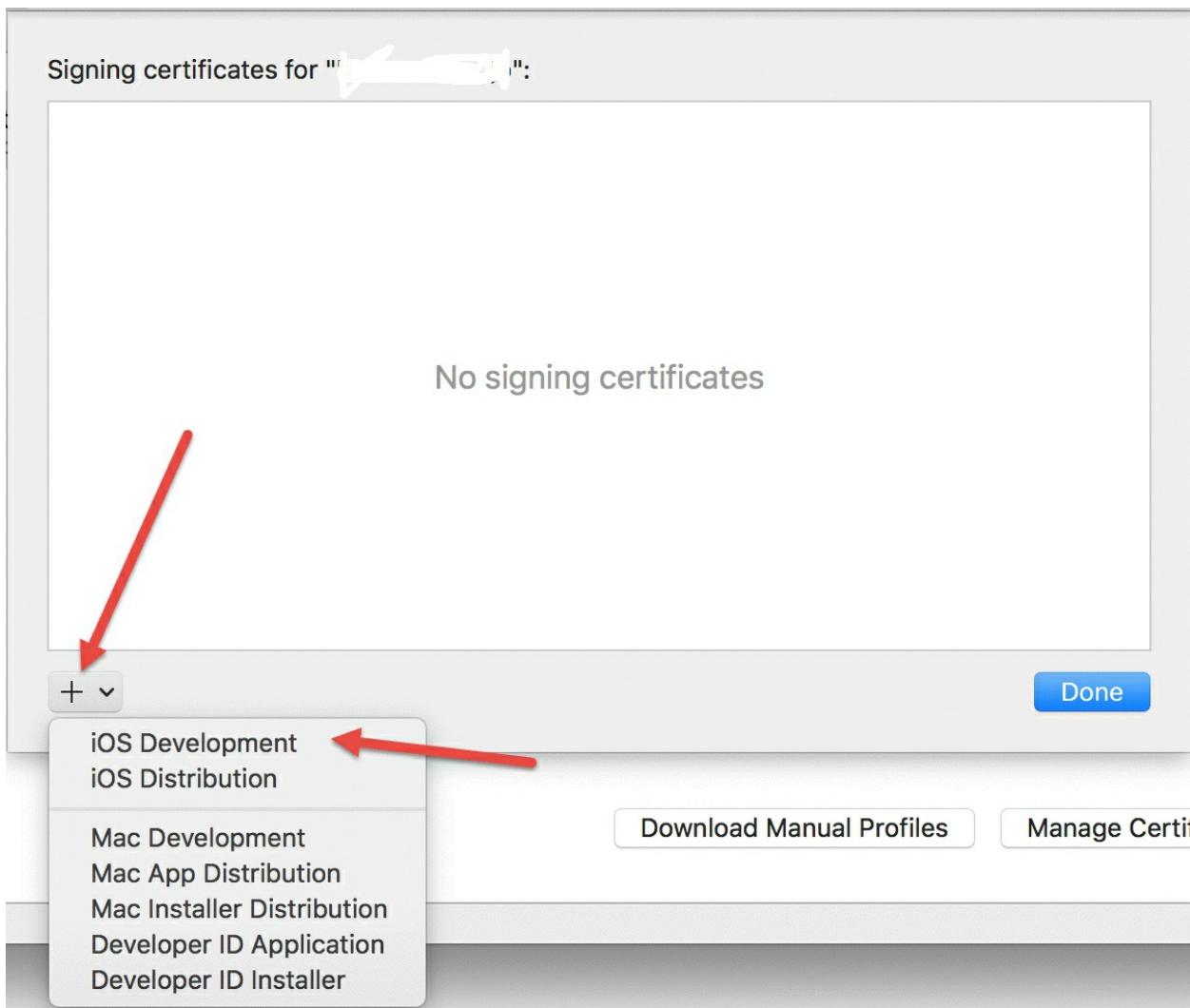


Figura 5.2: Criando uma Signing Identity

Precisamos criar um projeto no XCode e instalá-lo no dispositivo que usaremos para teste. Sendo assim, conecte seu dispositivo iOS em seu Mac. Em meu caso, estou usando um iPhone SE com o iOS 11.4. No XCode, selecione o menu `File -> New -> Project` e, na categoria `iOS -> Application`, escolha `Single View Application` e clique no botão `Next`.

Na janela que se abre, precisamos fornecer algumas informações. A mais importante é a `Organization identifier`, que deverá ser a mesma na aplicação Xamarin. Muita atenção neste item. Veja a figura a seguir. Após registrar as informações, clique em `Next`. Selecione a página para salvar e clique no botão `Create` na nova janela que se abre.

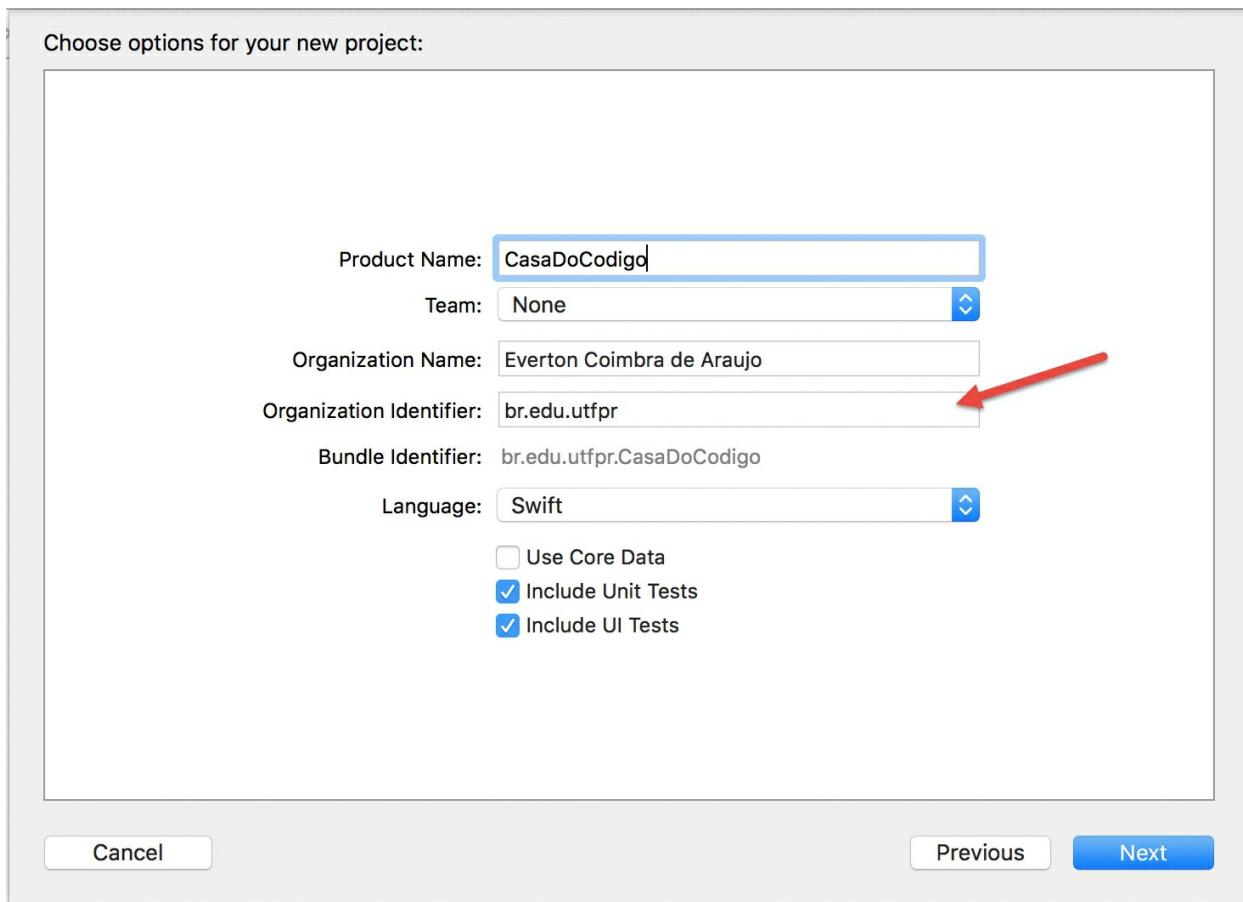


Figura 5.3: Criando uma aplicação no XCode

Vamos agora fazer a configuração do `signing identity` e a versão mínima para uso de sua aplicação. Quando você clicou no botão `Create`, uma janela com diversas guias apareceu. Em `Team`, selecione sua conta criada anteriormente, e informe a versão do iOS que deseja como mínima. Você pode escolher qualquer uma, mas lembre-se de que as versões estão relacionadas a recursos oferecidos para sua aplicação. Quanto menor for a versão, menos recursos atualizados ela terá.

É preciso também configurar a execução para seu dispositivo no topo da janela, como fizemos para executar no emulador, no capítulo 2. Selecione seu dispositivo conectado ao Mac e execute a aplicação. Se neste momento uma janela de erro aparecer em seu dispositivo, informando-o para verificar configurações de confiança, não se preocupe, isso ocorre na primeira execução de uma aplicação pelo mecanismo que estamos utilizando, o de depuração. Realize a configuração indicada pelo dispositivo, que é bem simples, e execute novamente sua aplicação. Em meu iPhone, eu acessei `Ajustes->Geral->Gerenciamento de Dispositivo->Selecionei meu ID Apple->Confirmar confiança`. Se tudo deu certo, sem exibir nada, uma aplicação será executada em seu dispositivo. Lembre-se de encerrar a execução da aplicação, no Xcode, ou no próprio iPhone.

Vamos agora criar uma aplicação no Visual Studio (Mac) para executá-la no dispositivo. Desta maneira, crie um `Blank Solution` e, no nome da organização e projeto, utilize o mesmo que foi registrado no XCode, no campo `Bundle Identifier`. Tente executar sua aplicação. Caso ela não seja executada, clique com o botão direito do mouse sobre o nome do projeto iOS e então em `Options`, e execute a configuração dos campos destacados de acordo aos seus dados. Caso você queira, pode alterar estes valores no arquivo `Info.plist` no projeto iOS.

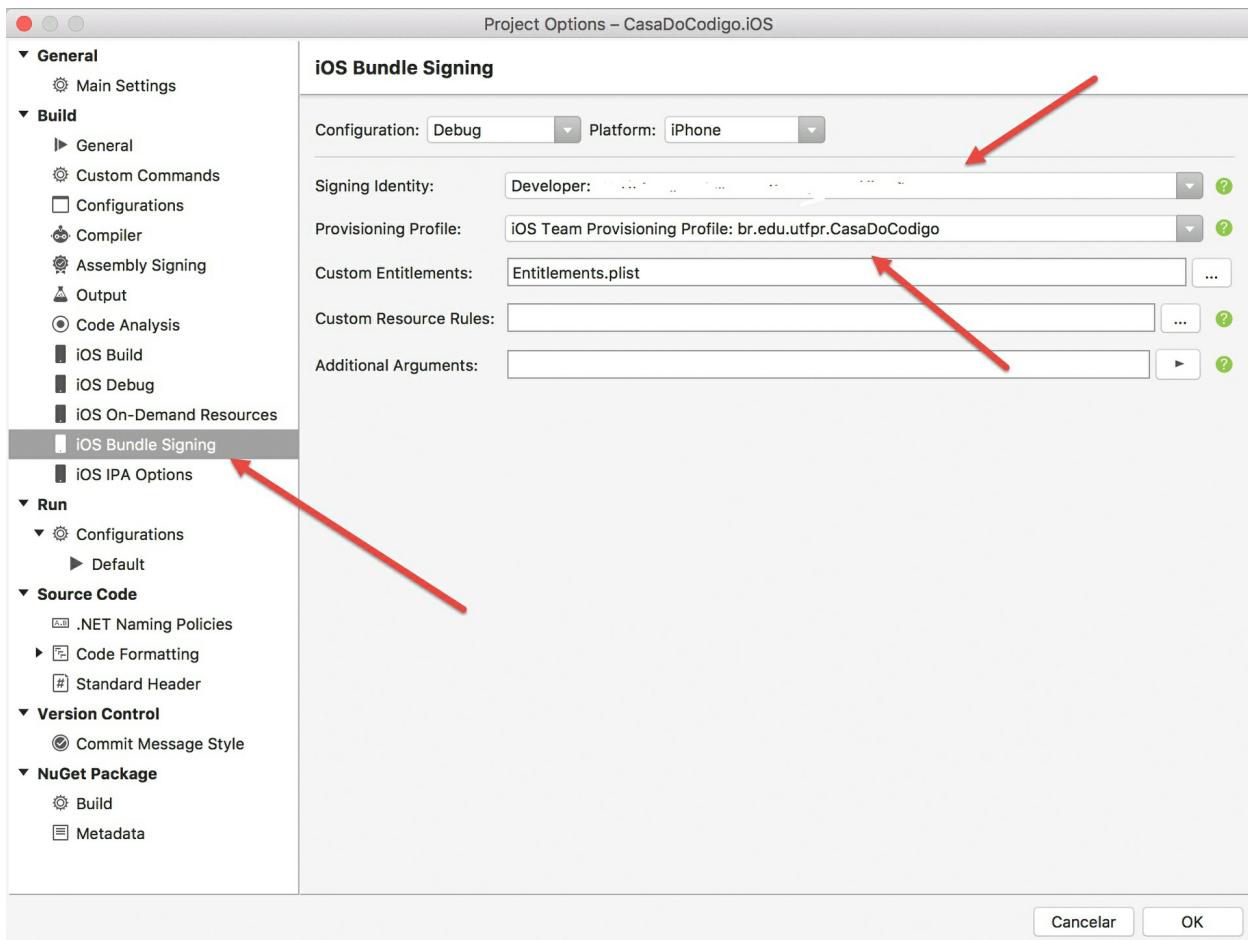


Figura 5.4: Configuração da aplicação no Visual Studio para execução no dispositivo

Vamos agora fazer com que uma aplicação criada no Visual Studio para o Windows execute também no dispositivo iOS. Conecte seu dispositivo ao seu Mac e crie seu projeto da mesma maneira que estamos fazendo desde o início do livro e, após a criação, defina o projeto iOS como de inicialização. Para configurarmos nosso projeto no Windows para executar no Mac, precisamos conectar nosso Visual Studio do Windows ao Mac, tal qual fizemos no capítulo 2.

Com a conexão estabelecida, vamos às configurações necessárias ao projeto. Clique com o botão direito do mouse sobre o nome do projeto iOS e então em `Propriedades`. Ao lado direito, na opção `Assinatura de Pacote iOS`, informe a `Assinatura de Pacote`, tal qual apresento na figura a seguir. O processo é semelhante ao que fizemos no Mac. Depois, precisamos abrir o arquivo `Info.plist` do projeto iOS e informar o `Identificador de Pacote`, igual ao que informamos no XCode, pois ele se baseará nas configurações da aplicação instalada no dispositivo. Agora basta selecionar seu dispositivo na barra de ferramentas e executar a aplicação.

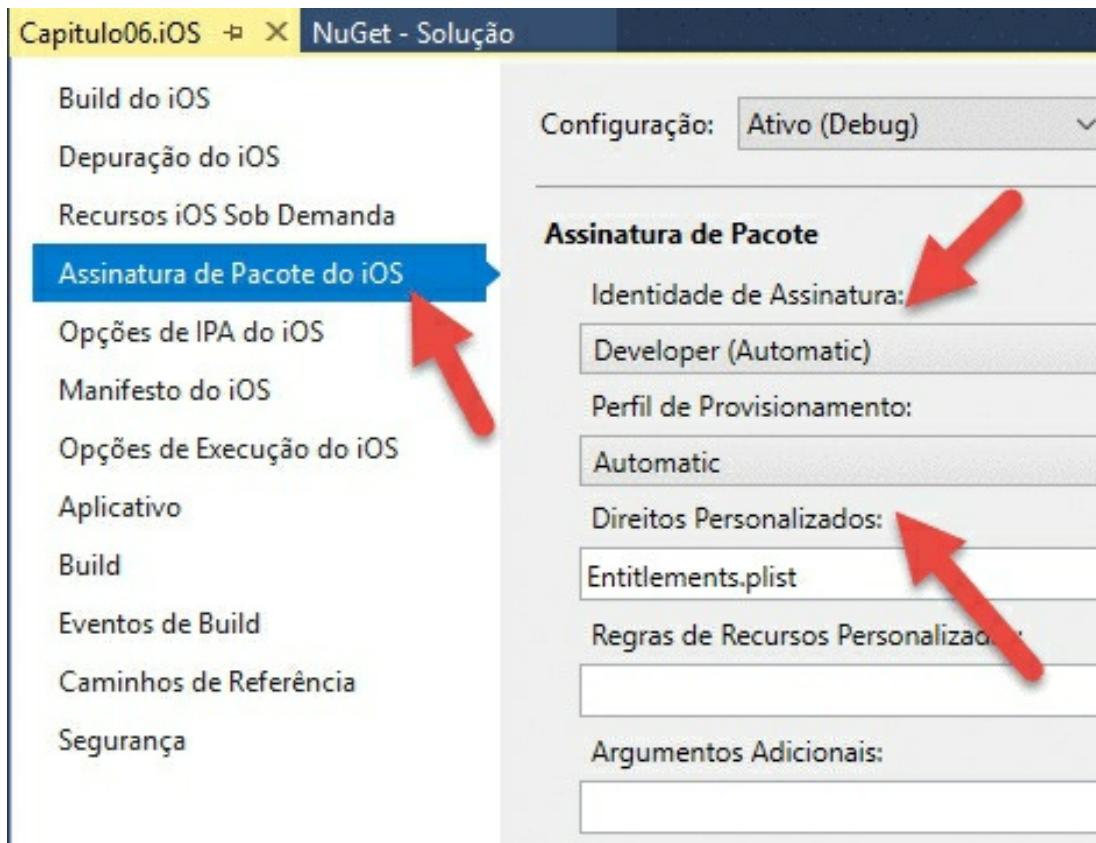


Figura 5.5: Configuração da aplicação no Visual Studio para execução no dispositivo

Pode ocorrer a necessidade de repetição deste processo caso as configurações se percam.

## 5.2 Publicação da aplicação para um dispositivo Android

A distribuição e execução de uma aplicação em um dispositivo Android é um processo mais simples do que o que vimos para um iOS. Apenas para registrar, executei testes das aplicações do livro em dispositivos Samsung e Motorola com Android 7 e 7.1.

O primeiro passo que precisamos realizar é habilitar as opções para o desenvolvedor (*Developer Options*). Para isso, no dispositivo, vá em *Configurações* -> *Sobre o dispositivo* -> *Informações de Software* e clique 7 vezes sobre o Número de compilação (*build number*). Após isso, retorne para *Configurações* e clique em opções do Desenvolvedor (*Developer Options*) e habilite a opção *usb debugging*. Se estas opções não aparecerem para seu dispositivo, você precisará buscar na documentação dele como acessá-las. Normalmente, nos sites de pesquisa, ao digitar as palavras-chaves *android ativar modo desenvolvedor <fabricante-e-modelo-dispositivo>* já são trazidas as orientações necessárias.

Com o modo desenvolvedor habilitado, conecte seu dispositivo em sua máquina. Com esta operação realizada ele aparecerá na listagem de dispositivos para execução da aplicação, no Visual Studio, se seu projeto de inicialização for o Android. Execute sua aplicação e veja-a rodando em seu dispositivo.

Em minhas aulas ocorreram casos de alguns dispositivos não aparecerem no Visual Studio. Foi preciso buscar no site do fabricante o drive específico para ser instalado na máquina. Outra situação que ocorreu foi a de cabos USB que não se comunicavam com o dispositivo.

## 5.3 Persistência de dados de maneira física com SQLite

Quando trabalhamos com aplicações complexas, robustas, seja para o ambiente Web ou desktop, temos sempre um servidor de banco de dados funcionando na persistência dos dados do sistema. Pode ser o Oracle, o SQL Server, Postgre, MySQL ou ainda soluções NoSQL, como Firebase, Mongo e tantos outros. São produtos mundialmente conhecidos.

Quando o desenvolvimento de uma aplicação tem como plataforma alvo um dispositivo móvel, fica difícil imaginar uma solução dessa. Quem sabe no futuro isso seja possível, mas, atualmente, o que se utiliza na persistência de dados, localmente, em dispositivos móveis, tem sido em sua maioria o SQLite. Em alguns casos em que não haja a necessidade de recursos relativos a uma base de dados e a quantidade de dados não seja elevada, é possível utilizar a API específica de `Settings` OU `Properties` do Xamarin. Existe um plugin interessante para isso em <https://github.com/jamesmontemagno/SettingsPlugin>. Fica como referência para visualização futura.

O SQLite possui uma licença simples de uso e recomendo que você a leia no site do produto (<https://sqlite.org/>). Ele não requer nenhuma configuração no dispositivo em que será usado.

Como dito na introdução do capítulo, trabalhamos nos capítulos anteriores a implementação com dados de clientes e tipos de serviços. Entretanto, os exemplos não realizaram persistência física e tinham como foco o aprendizado de técnicas e recursos. Nesta seção, implementaremos estes modelos de maneira mais voltada para o objetivo da aplicação.

Existem algumas versões de plugins para o SQLite disponíveis para serem instaladas por meio do NuGet. Durante o livro, faremos uso do framework de mapeamento de objetos para o modelo relacional (ORM) da Microsoft, o Entity Framework Core.

Para a manipulação e acesso de dados na base SQLite por meio do EF Core, será necessária a criação de um projeto para o seu uso, pois nele configuraremos suas especificidades. Isso nos dará o acoplamento e coesão necessários para nosso desenvolvimento. Este projeto será uma biblioteca .NET Standard e, para criá-lo, clique com o botão direito do mouse sobre o nome da solução, escolha `Adicionar -> Novo Projeto`. Nas categorias, escolha `.NET Standard` e, no centro da janela `Biblioteca de classes (.NET Standard)`. Eu nomeei o projeto de `SQLiteEF`. Remova o arquivo `Class1.cs` que é criado pelo template.

Vamos agora instalar os plugins específicos para nosso projeto SQLiteEF. Para isso, basta clicarmos com o botão direito do mouse sobre o nome do projeto, e escolhermos a opção `Gerenciar Pacotes do Nuget`. Na guia `Procurar`, digite `Microsoft.EntityFrameworkCore` e quando ele aparecer, clique no botão de instalar. A versão que estava disponível na escrita do livro é a 2.1.1. Como utilizaremos o EF Core para persistir no SQLite, precisamos instalar outro componente, o `Microsoft.EntityFrameworkCore.Sqlite`. Realize o mesmo processo para a instalação dele. A versão também é a 2.1.1. Você precisará instalá-lo também nos projetos iOS e Android.

Precisamos compreender que o arquivo que representará nossa base de dados ficará hospedado no emulador, ou dispositivo, e neles o projeto que é executado é o projeto específico da plataforma alvo. Desta maneira, precisamos obter o caminho deste arquivo e o utilizar em nosso projeto de biblioteca responsável pelo acesso a dados. Isso será possível por meio do uso de `Serviços de Dependência`, ou `Dependency Service`. Vamos trabalhar com isso agora.

## 5.4 Dependency Service para utilizar a base de dados

Para que o serviço de dependência possa ocorrer na comunicação entre projetos específicos de plataformas e o projeto do Xamarin Forms, faremos uso de interfaces, que é a maneira adotada pelo Xamarin para a injeção de dependência deste serviço. Vamos criar um novo projeto .NET Standard Library para hospedar as interfaces que utilizaremos e que precisam ser visíveis entre os projetos. O processo é o mesmo que vimos para o `SQLiteEF`, mas dê agora o nome de `Interfaces`.

Na raiz deste novo projeto crie uma pasta chamada `DataAccess`. Dentro dela, uma outra, chamada `Interfaces` e dentro

desta, crie uma interface chamada `IDBPath`, com o código apresentado na sequência. Atente para o namespace, pois ele é muito importante para o funcionamento de nossa arquitetura.

```
namespace CasaDoCodigo.DataAccess.Interfaces
{
    public interface IDBPath
    {
        string GetDbPath();
    }
}
```

Vamos começar pelo iOS e precisamos adicionar a referência ao projeto de interfaces que criamos. Para isso, clique com o botão direito do mouse sobre o nome do projeto e, então, em `Adicionar -> Referência`. Na janela que se abre, clique em `Projetos` e marque o projeto `Interfaces`. Agora, ainda no projeto iOS, crie uma pasta chamada também `DataAccess`, tal qual fizemos para a interface anteriormente, e, dentro dela, uma classe chamada `DatabasePath`, que implementa `IDBPath`, como pode ser verificado no código a seguir.

```
[assembly: Xamarin.Forms.Dependency(typeof(DatabasePath))]
namespace Capitulo05.iOS.DataAccess
{
    public class DatabasePath : IDBPath
    {
        public string GetDbPath()
        {
            var caminhoCompleto = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments), "...",
"Library", "Oficina.db");
            return caminhoCompleto;
        }
    }
}
```

O código do método implementado retornará o caminho físico no dispositivo ou emulador, para o arquivo `oficina.db`, que será a base de dados para o SQLite. Observe o uso de `Environment.SpecialFolder.MyDocuments` e `Library` para obter o caminho completo. Esta é a orientação da documentação do SQLite para obter acesso a um arquivo de base de dados no iOS. Antes do namespace, note a declaração que faz com que esta classe seja fornecida como serviço de dependência. Esta classe precisará também ser implementada em seu projeto Android. Para reforçar, o código a seguir traz a implementação no Android. Veja, além do namespace diferente, a maneira como o caminho é recuperado. Em Android, fazemos uso de `Environment.SpecialFolder.Personal` e não precisamos do `Library`.

```
[assembly: Xamarin.Forms.Dependency(typeof(DatabasePath))]
namespace Capitulo05.Droid.DataAccess
{
    public class DatabasePath : IDBPath
    {
        public string GetDbPath()
        {
            return Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.Personal), "Oficina.db");
        }
    }
}
```

## 5.5 A classe de modelo

Quando implementamos uma classe, precisamos levar em consideração que todo objeto precisa de uma propriedade que o identifique perante os demais e essa característica é mapeada para um campo conhecido como chave primária na tabela que será criada na base de dados (*primary key*). O EF Core identifica esta propriedade quando o nome dela possui o nome da classe e o sufixo `ID` e, por padrão, quando um objeto é persistido sem o valor para a propriedade que seja a identificadora (chave primária), o valor para ela é definido durante a inserção, de maneira automática.

Da mesma forma que criamos, anteriormente, um projeto específico para o acesso a dados pelo EF Core (SQLiteEF), precisamos também criar um projeto que implementará propriedades que sejam específicas para ele, como é o caso das propriedades identificadoras de objetos, ou *primary keys*. Vamos então criar, na solução, um projeto chamado `IDPropertiesEF`, do tipo `.NET Standard Library`.

Com o projeto criado, implementaremos uma classe base, que terá a declaração da propriedade de identidade para os objetos. Vamos lá! Clique no nome do projeto, `IDPropertiesEF`, e adicione uma pasta chamada `Models` e, nela, uma classe chamada `ClienteIDProperty`, com o código apresentado na sequência. Pelo nome da classe você já pode imaginar que começaremos o trabalho com clientes. Observe também que a classe está declarada como abstrata, para não ser instanciada.

```
namespace CasaDoCodigo.Models
{
    public abstract class ClienteIDProperty
    {
        public long? ClienteID { get; set; }
    }
}
```

Muito bem, vamos para mais uma etapa da burocracia inicial. Precisamos agora criar a classe de modelo, em nosso caso, a classe `Cliente`. Mais uma vez, buscando acoplamento e coesão, princípios da Orientação a Objetos, criaremos nossos modelos em um projeto isolado da aplicação. Sendo assim, vamos criar, na solução, um novo projeto `.NET Standard`, que chamaremos de `oficinaModels`. Nesse projeto, crie uma pasta chamada `cadastros` e, dentro dela, uma classe chamada `Cliente`.

Precisaremos acessar nossa classe base, que define a propriedade de identidade. Vamos adicionar a referência ao projeto `IDPropertiesEF`, lembrando que, para isso, basta clicar com o botão direito do mouse sobre o nome do projeto e em `Adicionar -> Referência`. Basta selecionar o projeto na listagem de projetos da solução. Com a referência adicionada, implemente o código a seguir na classe. Veja o namespace para a classe e a extensão de `ClienteIDProperty`. Veja que o namespace da classe `cliente` é o mesmo que definimos para a classe `ClienteIDProperty`. É preciso seguir esta organização lógica se formos utilizar diferentes plugins.

```
namespace CasaDoCodigo.Models
{
    public class Cliente : ClienteIDProperty
    {
        public string Nome { get; set; }
        public string Telefone { get; set; }
        public string EMail { get; set; }
    }
}
```

## 5.6 A classe de contexto

O EF Core trabalha com o padrão de ter uma classe que representa seu contexto com a base de dados, onde ocorrerá o mapeamento de objetos para o modelo relacional e vice-versa. Desta maneira, na pasta `DataAccess` do projeto `SQLiteEF`, crie uma classe chamada `DatabaseContext`, que estenderá `DbContext`, que é uma classe do EF Core, tal qual pode ser verificado no código a seguir. Observe que estamos criando um objeto privado, que será utilizado na configuração de acesso à base de dados.

```
namespace CasaDoCodigo.DataAccess
{
    public class DatabaseContext : DbContext
    {
        private static string DbPath;
    }
}
```

Com a classe criada, precisamos declarar um conjunto tipificado para a classe `Cliente`. Este conjunto será mapeado para uma tabela com o nome da propriedade, em nosso caso `Cientes`, na base de dados que será utilizada por nossa aplicação. Para a declaração desta propriedade, precisamos adicionar ao projeto `SQLiteEF` a referência para o projeto `OficinaModels`, tal qual fizemos anteriormente para adicionar a referência para `IDPropertiesEF`. Com a referência adicionada, insira na classe anterior, abaixo do `private`, a propriedade a seguir.

```
public DbSet<Cliente> Clientes { get; set; }
```

Precisamos agora sobreescriver um método que existe na classe `DbContext`, que estamos estendendo. Este método, `OnConfiguring()`, é responsável por realizar a configuração do acesso à base de dados. Veja seu código na sequência. Veja a instrução que informa o uso do `sqlite`, enviando o caminho obtido para o objeto `optionBuilder`. Este caminho será recebido por injeção de dependência no construtor, que já implementaremos. Observe que o caminho é informado pelo objeto que declaramos na listagem anterior.

```
protected override void OnConfiguring (DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlite($"Filename={DbPath}");
}
```

Para finalizar esta classe, precisamos, no construtor, garantir que a base de dados seja criada e isso é feito por meio da invocação ao método `EnsureCreated()`. Caso mudanças tenham sido realizadas no modelo de negócio que está associado às propriedades de conjuntos definidas nessa classe de contexto, você pode receber mensagens de erro apontando não correspondência entre o modelo de negócio e a base de dados. A princípio, para corrigirmos isso, invocaremos um método que remova a base de dados antes de criá-la. Este método é o `EnsureDeleted()`. Na listagem a seguir estou deixando a invocação ao método comentada. Note que o construtor é privado, o que impedirá instânciação da classe. Nós faremos uso de um método estático, que retornará sempre uma instância desta classe para nossas atividades em base de dados. Este método está após o construtor. Observe que ele recebe o caminho da base e a atualiza na variável do objeto.

```
private DatabaseContext ()
{
    //this.Database.EnsureDeleted();

    this.Database.EnsureCreated();
}

public static DatabaseContext GetContext ( string dbPath)
```

```

{
    DbPath = dbPath;
    return new DatabaseContext ();
}

```

Agora, no projeto iOS, precisamos habilitar o uso para o SQLite. Desta maneira, no método `FinishedLaunching()`, da classe `AppDelegate` é preciso inserir, logo no início, a instrução `SQLitePCL.Batteries.Init()`. Veremos mais à frente como utilizar o Migrations para que as atualizações na base de dados possam ocorrer sem termos que eliminar a base de dados e recriá-la novamente. Para o Android não há necessidade desta instrução.

Em testes realizados em dispositivos físicos iOS, foi preciso adicionar a instrução a seguir antes da declaração do namespace na classe `Main` do projeto iOS. Pelo resultado que obtive em pesquisas sobre este problema, é algo relacionado à reflexão para obtenção e mapeamento dos dados. Parece ser um bug, que pode ser resolvido em versões futuras.

```
[assembly: Preserve( typeof (System.Linq.Queryable), AllMembers = true )]
```

## 5.7 A classe de manipulação de dados

Com o código gerado anteriormente, já possuímos o acesso à base de dados SQLite pelo EF Core. Agora, precisamos criar a classe DAL (*Data Access Layer*). É por meio desta classe que implementaremos as funcionalidades de acesso e manutenção aos dados persistidos na base de dados e os mapearemos a objetos de nosso modelo de negócio. Na estratégia que vamos utilizar, teremos uma classe DAL para cada classe do modelo de negócio e precisamos garantir que cada uma implemente um conjunto básico de funcionalidades.

Para garantir isso, faremos uso de interfaces e polimorfismo. Vamos voltar ao nosso projeto `Interfaces` e criar, na pasta `DataAccess`, uma interface chamada `IDAL`, com o código apresentado na sequência. Observe o namespase declarado.

```

namespace CasaDoCodigo.DataAccess.Interfaces
{
    public interface IDAL<T> where T : class
    {
        Task<List<T>> GetAllAsync(string campoClassificacao = null);
        Task<T> UpdateAsync(T item, long? itemID);
        Task DeleteAsync(T item);
        Task<T> GetByIdAsync(long? id);
        Task<IEnumerable<T>> GetStartsWithByFieldAsync(string field, string value);
    }
}

```

A princípio, estamos definindo apenas 5 métodos como contrato para as classes que implementarem esta interface. Os nomes dos métodos explicam semanticamente suas funcionalidades. Usamos o sufixo `Async`, pois é interessante que estes métodos sejam implementados de maneira assíncrona, mas lembre-se de que apenas o sufixo não garante isso.

Implementaremos uma classe chamada `DALBase` em uma pasta chamada `DAL` no projeto `SQLiteEF`, que você deverá criar. Será preciso também adicionar a referência ao projeto `Interfaces` no `SQLiteEF`. O código inicial para a classe é apresentado a seguir. Veja que definimos um objeto para `DatabaseContext` e o construtor que receberá o caminho para a base de dados. Estamos definindo os objetos como `protected`, para que possamos usá-los diretamente nas subclasses. Atenção ao namespase. Nossa objetivo é ter um DAL que possa ser utilizado de maneira genérica para as operações que definimos na interface anteriormente implementada, deixando para uma classe DAL especializada

as especificidades que possam surgir. Veja a definição da classe como abstrata, o que impossibilitará sua instanciação. Receber o caminho para a base de dados pelo construtor é um processo de injeção de dependências, que provê um bom isolamento do código.

```
namespace CasaDoCodigo.DAL
{
    public abstract class DALBase<T> : IDAL<T> where T : class
    {
        protected DatabaseContext context;
        protected string dbPath;

        public DALBase(string dbPath)
        {
            this.dbPath = dbPath;
        }
    }
}
```

O código, como está apresentado na listagem anterior, está apontando um erro, pois declaramos que implementaremos a interface `IDAL`, mas não implementamos o comportamento para os métodos definidos nela. Precisamos fazer isso. Clique na lâmpada que aparece no editor ou pressione `CTRL+.` com o cursor sobre o nome da interface. Confirme o desejo de implementar a interface no menu popup que aparecerá. Veja que todos os métodos são implementados, mas com uma exceção `NotImplementedException()` disparada. Não se preocupe com isso por enquanto.

Vamos implementar, inicialmente, o método do DAL que retorna todos os clientes registrados na base de dados. Com este método implementado poderemos criar uma visão e testar nossa aplicação, afinal, estamos implementando toda uma burocracia de código para isso e estamos curiosos para ver o resultado, não é verdade?

É importante também verificar no código a instrução `using()`, para manipularmos nosso contexto. Ela garante que o objeto `context` seja destruído após a execução das instruções em seu bloco. É também uma forte recomendação do EF Core que um contexto seja obtido e removido logo após a execução das instruções em foco. Será preciso adicionar `using Microsoft.EntityFrameworkCore;` aos usings. Observe a adição da instrução `virtual` na assinatura do método. Isso caracteriza que ele pode ser sobreescrito por subclasses. Insira esta instrução em todos os métodos implementados da interface. Veja na sequência.

Note que buscamos um conjunto no contexto com base no tipo de dado utilizado na criação do DAL. Aplicamos ao conjunto a característica de não rastreamento aos objetos recuperados, por meio do método `AsNoTracking()`. Esta invocação faz com que os objetos recuperados não permaneçam no contexto do EF Core, que não sejam gerenciáveis. Isso aumenta a performance de nossa aplicação e, para nosso objetivo, manter os objetos no contexto não é relevante. Depois, em caso de ter sido recebida alguma propriedade para ser utilizada como classificadora dos dados, fazemos o uso de reflexão para a criação de uma expressão lambda, para que ela possa ser utilizada no método `OrderBy()`. Ao final, invocamos o `ToListAsync()` para a recuperação dos dados e retorno ao chamador.

```
public async virtual Task<List<T>> GetAllAsync( string campoClassificacao = null )
{
    using ( var context = DatabaseContext.GetContext( this .dbPath) )
    {
        var query = context.Set<T>().AsNoTracking();

        if ( ! string .IsNullOrEmpty(campoClassificacao) )
        {
```

```

        var parameter = Expression.Parameter( typeof( T ));

        var sortExpression = Expression.Lambda<Func<T, object>>( Expression.Property( parameter,
campoClassificacao), parameter);

        query = query.OrderBy( sortExpression);

    }

}

return await query.ToListAsync();
}
}

```

Nosso método anteriormente implementado é uma solução genérica para recuperar os dados de uma coleção, classificados por uma determinada propriedade. Como esta propriedade pode variar de uma classe para outra e cada classe pode ter uma propriedade padrão para esta funcionalidade, vamos já especializar nosso DAL para `clientes`. Desta maneira, na pasta `DAL`, do mesmo projeto, crie a classe `ClienteDAL`, com o código que se apresenta. A classe é simples, estende uma superclasse, implementa o construtor que será utilizado para a instanciação, invoca o construtor da classe base e tem o método `GetAllAsync()` sobrescrito, ao qual, caso não seja passado um argumento para classificação, o padrão é atribuído. Adotaremos sempre esta técnica nos DALs especializados. Observe sempre o namespace das classes que estamos criando.

```

namespace CasaDoCodigo.DAL
{
    public class ClienteDAL : DALBase<Cliente>
    {
        public ClienteDAL( string dbPath ) : base( dbPath )
        {

        }

        public async override Task<List<Cliente>> GetAllAsync( string campoClassificacao )
        {
            campoClassificacao = string.IsNullOrEmpty( campoClassificacao ) ? nameof( Cliente.Nome ) : campoClassificacao;
            return await base.GetAllAsync( campoClassificacao );
        }
    }
}

```

## 5.8 A visão que lista todos os clientes

No capítulo 4 implementamos o CRUD para Serviços, mas trazendo os dados de uma coleção. O CRUD para Clientes será semelhante, a diferença está no mecanismo de acesso e manipulação de dados, mas praticamente já temos toda esta infraestrutura implementada. Desta maneira, serei direto em alguns códigos nesta seção. Vamos lá!

No projeto `capitulo05`, que é nosso projeto Xamarin Forms, crie uma pasta chamada `views`. Dentro dela, uma chamada `clientes` e, dentro desta última pasta, um `Content Page` chamado `ListagemView`, com o código apresentado na sequência. Já temos definidos na listagem todos os `Commands` e propriedades que estarão ligadas à visão.

```

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"

xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

```

```
x:Class = "Capitulo05.Views.Clientes.ListagemView"

xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOS;assembly=Xamarin.Forms.Core"

ios:Page.UseSafeArea = "true"

Title = "Clientes" >

<ContentPage.ToolbarItems>

<ToolbarItem Icon = "plus.png" Command = "{Binding NovoCommand}" />

</ContentPage.ToolbarItems>

<ContentPage.Content>

<StackLayout Padding = "10, 0, 0, 0" VerticalOptions = "FillAndExpand" >

<ListView x:Name = "listView" HasUnevenRows = "True" ItemsSource = "{Binding Clientes}"

SelectedItem = "{Binding ClienteSelecionado}" >

<ListView.ItemTemplate>

<DataTemplate>

<ViewCell>

<ViewCell.ContextActions>

<MenuItem Command = "{Binding Path=BindingContext.EliminarCommand, Source={x:Reference listView}}" CommandParameter = "{Binding .}" Text = "Remover" IsDestructive = "True" />

</ViewCell.ContextActions>

<StackLayout Padding = "10" >

<Label Text = "{Binding Nome}" FontSize = "18" FontAttributes = "Bold" />

<Label Text = "{Binding Telefone}" FontSize = "14" />

</StackLayout>

</ViewCell>
```

```

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

</StackLayout>

</ContentPage.Content>

</ContentPage>

```

Como vimos no capítulo anterior, precisamos de uma classe que será a ViewModel para a visão que criamos. Vamos implementar o código inicial para esta classe. No projeto `Capítulo05`, crie uma pasta chamada `ViewModels`, dentro dela, outra pasta chamada `Cientes` e, dentro desta pasta, uma classe chamada `ListagemViewModel`, com o código apresentado na sequência. Lembre-se de que precisamos adicionar referências ao projeto `Capítulo05`, essas referências são os projetos: `Interfaces`, `OficinaModels` e `SQLiteEF`. Veja o uso de `Dependency Service` para a declaração do DAL. Estamos, neste momento, invocando o método `GetPath()`, implementado na plataforma que estará executando a aplicação, e estamos injetando seu retorno na instanciação de `DatabaseContext`. No código também já temos implementada a propriedade que populará o `ListView`, a `Cientes`. Definimos um objeto que receberá qual cliente da listagem está selecionado e declaramos os `commands` que deveremos implementar para a inserção e remoção de um cliente.

```

namespace Capítulo05.ViewModels.Cientes
{
    public class ListagemViewModel
    {
        private IDAL<Cliente> ClientesDAL = new ClienteDAL(DependencyService.Get<IDBPath>().GetDbPath());
        private Cliente ClienteSelecionado;
        public ObservableCollection<Cliente> Clientes { get; set; }
        public ICommand NovoCommand { get; set; }
        public ICommand EliminarCommand { get; set; }
    }
}

```

Usando a mesma estratégia que utilizamos no capítulo 4 para a criação dos `commands`, vamos implementar na classe da listagem anterior o método `RegistrarCommands()`, tal qual a listagem que se segue.

```

private void RegistrarCommands ()
{
    NovoCommand = new Command(() =>
    {
        MessagingCenter.Send<Cliente>( new Cliente(), "Mostrar" );
    });

    EliminarCommand = new Command<Cliente>((cliente) =>
    {
        MessagingCenter.Send<Cliente>(cliente, "Confirmação" );
    });
}

```

Observe que na listagem do método `RegistrarCommands()` estamos enviando mensagens para o `Messaging Center` na execução dos `Commands`. Desta maneira, relembrando, precisamos nos registrar e cancelar a assinatura das mensagens na visão, em nossa `ListagemView`, sobrescrevendo os métodos `OnAppearing()` e `OnDisappearing()`. A princípio deixaremos as mensagens sem comportamento.

```
protected override void OnAppearing ()  
{  
    base.OnAppearing();  
  
    MessagingCenter.Subscribe<Cliente>( this , "Mostrar" , async (cliente) => { });  
    MessagingCenter.Subscribe<Cliente>( this , "Confirmação" , async (cliente) => { });  
}  
  
protected override void OnDisappearing ()  
{  
    base.OnDisappearing();  
    MessagingCenter.Unsubscribe<Cliente>( this , "Mostrar" );  
    MessagingCenter.Unsubscribe<Cliente>( this , "Confirmação" );  
}
```

Precisamos implementar na `ViewModel` um método responsável pela atualização da propriedade `clientes`. Vamos chamar este método de `AtualizarClientesAsync()` com o código da listagem seguinte. Você precisará inserir no `using` a instrução `using System.Threading.Tasks;`.

```
public async Task AtualizarClientesAsync ()  
{  
    var clientes = await clientesDAL.GetAllAsync();  
}
```

Note que o método apenas obtém os dados da base de dados, não atualiza a propriedade que populará o `ListView`. Vamos discutir um pouco sobre esta funcionalidade. Quando obtemos a lista da base de dados, poderíamos fazer tal qual fizemos no capítulo anterior, que trabalhava com coleções e de maneira síncrona, na qual inicializava a propriedade ligada ao `ListView` com os dados recuperados da coleção. Isso ocorria na primeira visualização da visão, ou com dados recuperados da própria propriedade, quando retornávamos de uma inserção ou alteração. <!--Everton, por favor confira a última frase. A parte "Isso ocorria na primeira visualização da visão, ou recuperados da própria propriedade" ficou estranha, pois "ou recuperados" não dá para entender do que você está falando.

OK-->

Para esta situação da inserção e alteração, nós envíavamos a coleção ligada para o CRUD e então a atualizávamos na `ViewModel` do CRUD e, como tudo é objeto, ao retornar para a listagem, o `ListView` era atualizado. Com o EF Core e as operações assíncronas, isso não funciona, pois os itens inseridos não são trazidos para a reinicialização da propriedade ligada. Precisamos de outra estratégia.

Infelizmente, para mantermos os dados classificados, sempre precisamos recuperá-los da base, percorrer a coleção ligada ao `ListView` e atualizar elemento a elemento. Se a classificação não precisar ser feita, a estratégia anterior funciona. Você precisará avaliar os requisitos de sua aplicação. Esta funcionalidade que implementaremos será comum para todas as visões em que tenhamos listagens, o que nos leva à redundância de código. Poderíamos pensar

em uma classe auxiliadora e nela implementar um método estático, mas o C# nos traz um recurso chamado `ExtensionMethods`, que permite a criação dinâmica de um método que pode ser adicionado a um tipo de dados qualquer. Vamos a ele.

No nosso Xamarin Forms, crie uma nova pasta, chamada `ExtensionMethods` e, nela, crie uma nova classe, chamada `ObservableExtensionMethods`, que deverá ser estática, pois é a sintaxe para declaração de métodos de extensão no C#. Veja que, como sintaxe, o método de extensão também precisa ser estático e ter, em seu primeiro argumento, a referência `this` para o tipo de dado em que se deseja implementar o método.

O segundo argumento é o que será recebido pelo método. Veja que, no método, percorremos a coleção recebida e verificamos se cada item deve ser alterado ou inserido em nossa coleção de origem, a chamadora do *extension method*. Ao final, se existem itens na lista de origem que não estejam mais no destino, precisamos remover estes elementos da lista destino, para atualizá-la de acordo a nova origem.

Os itens que já existem no destino são sempre atualizados pela origem, pois se um item for alterado, sua mudança não é refletida no `Listview`.

```
namespace Capitulo05.ExtensionMethods
{
    public static class ObservableExtensionMethods
    {
        public static void SincronizarColecoes<T>(this ObservableCollection<T> destino, List<T> origem)
        {
            for (int i = 0; i < origem.Count; i++)
            {
                if (destino.Count <= i)
                    destino.Add(origem[i]);
                else if (!destino[i].Equals(origem[i]))
                    destino[i] = origem[i];
            }
            for (int i = origem.Count; i < destino.Count; i++)
            {
                destino.RemoveAt(i);
            }
        }
    }
}
```

Estamos utilizando o método `Equals()` dos objetos de origem para comparar com os objetos de destino. Isso nos remete à necessidade de implementarmos a sobreescrita para este método em nossa classe de modelo `Cliente`, que está no projeto `OficinaModels`. Quando sobreescrivemos o método `Equals()`, a recomendação é também de sobreescrermos o método `GetHashCode()`. Sendo assim, ao final da classe `Cliente`, implemente os métodos a seguir. Eles buscam implementar a identidade de um objeto, que em nosso caso está ligada à propriedade `clienteID`, que será a chave primária de nossa tabela para clientes.

```
public override bool Equals ( object obj )
{
    return ClienteID.Equals((obj as Cliente).ClienteID);
}

public override int GetHashCode ()
```

```

{
    var hashCode = -1711974838;

    hashCode = hashCode * -1521134295 + EqualityComparer<string>.Default.GetHashCode(ClienteID.ToString());

    return hashCode;
}

```

Agora, ao final do método `AtualizarClientesAsync()`, invoque nosso método de extensão, como é feito na sequência. Será preciso inserir o `using` para nossa classe `ObservableExtensionMethods`.

```
Clientes.SincronizarColecoes(clientes);
```

Precisamos realizar as últimas implementações para podermos testar nossa aplicação e termos a visão de listagem de clientes. A primeira é a declaração de nosso construtor para a classe `ListagemViewModel`, pois nela instanciaremos nossa propriedade de clientes e invocaremos o método `RegistrarCommands()`. Podemos ver isso na listagem que se segue.

```

public ListagemViewModel ()
{
    Clientes = new ObservableCollection<Cliente>();
    RegistrarCommands();
}

```

A segunda implementação é referente a disponibilizar a `ViewModel` como um objeto em nossa visão. Fazemos isso no código apresentado na sequência, que já implementa também o construtor para a `ListagemView`.

```

private ListagemViewModel viewModel;

public ListagemView ()
{
    InitializeComponent ();

    viewModel = new ListagemViewModel();
    BindingContext = viewModel;
}

```

A terceira implementação que nos falta é invocar o método `AtualizarClientesAsync()` no método `OnAppearing()`, logo após a chamada ao método da classe base, mas adotaremos uma estratégia específica para que esta atualização possa ocorrer. Veja o código a seguir, que pertence ao `OnAppearing()` da `CRUDView` de `clientes`. A documentação do Xamarin recomenda que, sempre que formos realizar uma atualização de dados em controles visuais que dependam de operações assíncronas, é preciso fazê-lo por meio `BeginInvokeOnMainThread()`, que passa a responsabilidade de execução da instrução assíncrona para a thread principal da aplicação.

```

Device.BeginInvokeOnMainThread( async () =>
{
    await viewModel.AtualizarClientesAsync();
});

```

Após o bloco, precisamos implementar o trecho a seguir. Ele cancela a seleção do último item selecionado, que remete à execução para a `CRUDView` com o dado deste item.

```
if (listView.SelectedItem != null )  
    listView.SelectedItem = null ;
```

Por fim, a quinta e última implementação necessária para vermos a listagem em execução deverá ser feita em nossa classe `App`. Pois é, precisamos configurar nossa aplicação para exibir a visão `ListagemView`. Altere a implementação do construtor para o código a seguir.

```
var navigationPage = new Xamarin.Forms.NavigationPage( new Views.Clientes.ListagemView());  
navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetPrefersLargeTitles( true );  
MainPage = navigationPage;
```

Ufa, finalmente. Agora podemos testar nossa aplicação, inicialmente no dispositivo iOS. Entretanto, se ocorrer um erro com a mensagem `xamarin can't resolve the reference 'System.ReadOnlySpan...'`, precisaremos realizar o *downgrade* do EF Core. Este erro ocorreu logo no início da distribuição da versão 2.1.0 e 2.1.1 dos plugins relacionados ao EF Core. Em meu caso, precisei atualizar para a 2.0.3. Isso foi feito no projeto `SQLiteEF` e nos projetos de plataforma. A execução no emulador não sofreu nenhum problema com a nova versão do EF Core. Outra opção para manter a versão atualizada do EF Core é instalar, nos projetos, o nuget `System.Buffers`.

Para executarmos a aplicação no Android, precisaremos realizar um *workaround*. Grave toda sua solução. Clique com o botão direito do mouse sobre o nome do projeto Android e na opção `Abrir pasta no gerenciador de arquivos`. Abra o arquivo `Capitulo05.Android.csproj` para edição. Busque pelo elemento `<AndroidLinkMode>` e entre as tags informe `SdkOnly`. Em minha configuração estava `none`. No início do arquivo, logo após a tag `<Project>`, insira o trecho a seguir. Este problema também está relacionado ao EF Core, que certamente terá uma correção em versões futuras.

```
<PropertyGroup>  
  
<NoWarn> $(NoWarn);NU1605 </NoWarn>  
  
</PropertyGroup>
```

Precisamos de mais uma mudança para o Android no arquivo aberto anteriormente. Procure pelo `<ItemGroup>` que declare o pacote para `Microsoft.EntityFrameworkCore.Sqlite`. Ao final deste grupo, insira `<PackageReference Include="System.Runtime.CompilerServices.Unsafe" Version="4.3.0" />` como último item. Grave o arquivo. Pode ser que o Nuget solicite atualização deste componente, mas não a faça. Eu testei com a 4.5.1 e não funcionou. Novamente, são bugs que podem estar corrigidos no momento de sua leitura do livro.

Ao retornar para o Visual Studio, será identificada a alteração externa no arquivo e você precisará recarregar seu projeto. O link <https://docs.microsoft.com/en-us/xamarin/android/deploy-test/building-apps/build-process/> traz diversas explicações sobre configurações do processo de build de uma aplicação Xamarin Android, talvez você ache interessante dar uma lida. Infelizmente, até que seja resolvido este problema do EF Core com o Android, estes ajustes são necessários. Teste agora sua aplicação Android. A figura a seguir apresenta a execução da aplicação nas duas plataformas. A listagem está vazia ainda, pois não temos nada persistido.

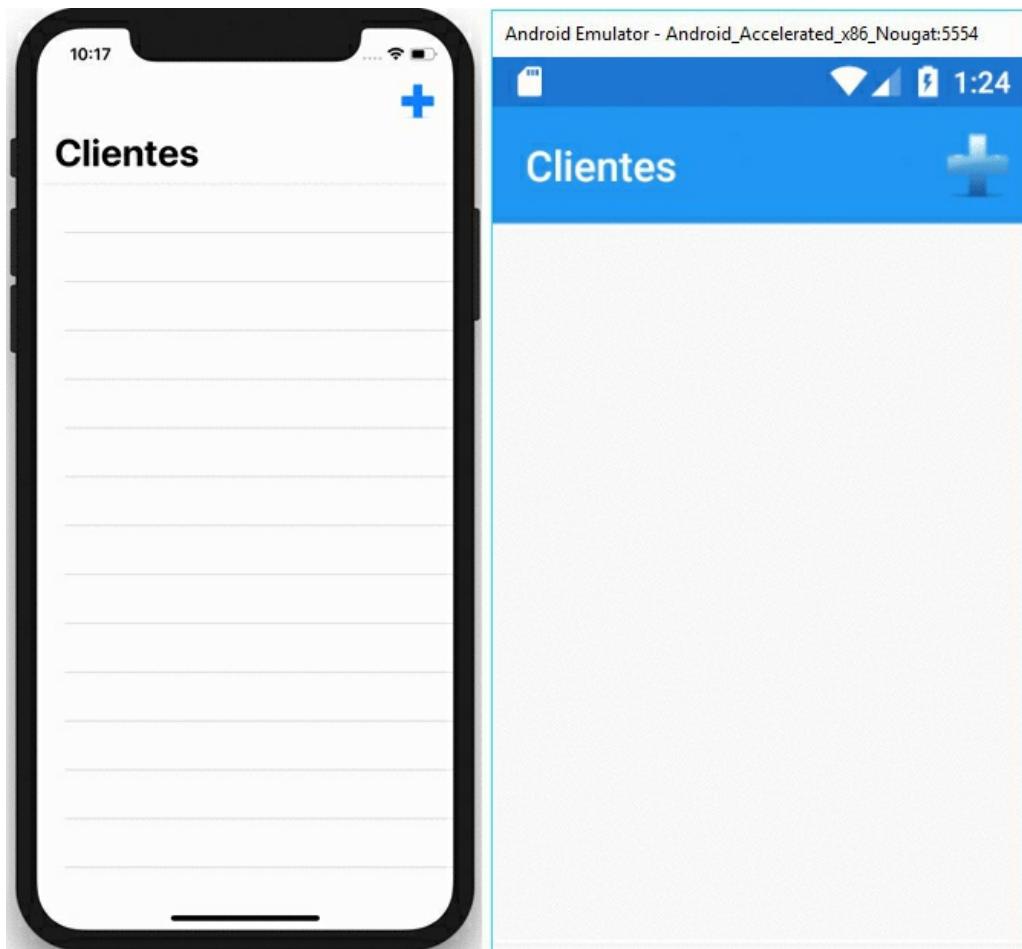


Figura 5.6: Listagem de clientes persistidos no SQLite

## 5.9 Implementação do Hamburger Menu

Com as implementações realizadas até aqui e as instalações das dependências necessárias para o uso do EF Core com o SQLite, já podemos começar a implementar os itens necessários para a interação com o usuário. Vamos começar com a implementação de nossa estrutura de navegação, que será a `Master Detail`, sendo a `master` nossa página de menu de opções, e a `detail` a que conterá a página com cujos dados estivermos interagindo. Em nosso primeiro exemplo, será a página de listagem de clientes.

Como vimos no capítulo 3, na criação de um `Hamburger Menu`, uma classe foi criada para fornecer os itens que comporão o menu de opções que deverá ser disponibilizado para o usuário da aplicação. Faremos o mesmo aqui. Crie uma pasta chamada `Models` na raiz do projeto Xamarin Forms e, dentro dela, uma classe chamada `MenuItem`, com o código apresentado na sequência.

```
namespace Capitulo05.Models
{
    public class MenuItem
    {
        public int Id { get; set; }
        public string Title { get; set; }
```

```

    public string IconSource { get; set; }
    public Type TargetType { get; set; }
}
}

```

Com a classe responsável pelas opções de acesso à aplicação criada, vamos partir para a criação da página que apresentará o menu com estas opções. Na pasta `Views`, crie um `Content Page` chamado `MasterPageView` e o codifique tal qual o XAML a seguir. Peço que você dê uma lida no código para comprovar que nada de novo é trabalhado nele. No código, temos um `ListView` com cabeçalho populado por uma propriedade chamada `OpcoesMenu`, os itens são organizados também dentro de um `Grid`. As imagens apontadas no XAML precisam estar fisicamente em seus projetos. Observe que estou fazendo uso do componente `Image Circle`, que usamos no capítulo 3, então, instale o Nuget para ele nos projetos Xamarin Forms, Android e iOS, lembrando da invocação do método `ImageCircleRenderer.Init();` na `MainActivity` e `AppDelegate`, ok?

```

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
              xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
              x:Class = "Capitulo05.Views.MasterPageView"
              xmlns:controls = "clr-namespace:ImageCircle.Forms.Plugin.Abstractions;assembly=ImageCircle.Forms.Plugin"
              Title = "Menu" >

    <ContentPage.Content>
        <StackLayout>

            <ListView x:Name = "itensMenuListView" SeparatorVisibility = "None" HasUnevenRows = "true" ItemsSource = "{Binding OpcoesMenu}" >
                <ListView.Header>
                    <Grid BackgroundColor = "#03A9F4" >
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width = "10" />
                            <ColumnDefinition Width = "64" />
                            <ColumnDefinition Width = "*" />
                        </Grid.ColumnDefinitions>
                
```

```
<Grid.RowDefinitions>

    <RowDefinition Height = "30" />

    <RowDefinition Height = "64" />

    <RowDefinition Height = "Auto" />

    <RowDefinition Height = "10" />

</Grid.RowDefinitions>

<controls:CircleImage Source = "logo.png" Aspect = "AspectFit" Grid.Row = "1" Grid.Column = "1" >

    <controls:CircleImage.WidthRequest>

        <OnPlatform x:TypeArguments = "x:Double" >

            <On Platform = "Android,iOS" > 55 </On>

            <On Platform = "UWP" > 75 </On>

        </OnPlatform>

    </controls:CircleImage.WidthRequest>

    <controls:CircleImage.HeightRequest>

        <OnPlatform x:TypeArguments = "x:Double" >

            <On Platform = "Android,iOS" > 55 </On>

            <On Platform = "UWP" > 75 </On>

        </OnPlatform>

    </controls:CircleImage.HeightRequest>

</controls:CircleImage>

<StackLayout Grid.Row = "1" Grid.Column = "2" VerticalOptions = "Center" >
```

```

<Label Text = "Meu Calhambeque" TextColor = "White" FontAttributes = "Bold" FontSize = "20" />

<Label Text = "Livro Xamarin - CC" TextColor = "White" FontSize = "Small" />

</StackLayout>

</Grid>

</ListView.Header>

<ListView.ItemTemplate>

<DataTemplate>

<ViewCell>

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width = "10" />

<ColumnDefinition Width = "32" />

<ColumnDefinition Width = "*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height = "5" />

<RowDefinition Height = "32" />

</Grid.RowDefinitions>

<Image Source = "{Binding IconSource}" HeightRequest = "32" Grid.Column = "1" Grid.Row = "1" />

<Label Text = "{Binding Title}" FontSize = "Default" Grid.Column = "2" Grid.Row = "1" VerticalTextAlignment = "Center" />

</Grid>

</ViewCell>

</DataTemplate>

```

```

</ListView.ItemTemplate>

</ListView>

</StackLayout>

</ContentPage.Content>

</ContentPage>

```

Após uma rápida leitura no XAML anterior, que não tem nada que já não tenhamos visto, vamos para o code-behind desta visão, que está apresentado na sequência. Veja a declaração da propriedade de opções que serão exibidas ao usuário e sua instanciação no construtor da classe. Aqui também é importante reforçar que as figuras utilizadas devem fazer parte da estrutura física de seus projetos. Atente para as imagens que você precisará em seus projetos específicos de plataforma, ok? Veja que o `TargetType` do primeiro item de menu já aponta para nossa listagem.

```

namespace Capitulo05.Views
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class MasterPageView : ContentPage
    {
        public Models.MenuItem[] OpcoesMenu { get; set; }
        public ListView ListView { get; set; }

        public MasterPageView()
        {
            Icon = "menu.png";

            InitializeComponent();
            OpcoesMenu = new[]
            {
                new Models.MenuItem { Id = 0, Title = "Clientes", TargetType = typeof(Clientes.ListagemView),
IconSource="tab_clientes.png"},

                new Models.MenuItem { Id = 1, Title = "Serviços", TargetType = typeof(ContentPage),
IconSource="tab_servicos.png"}
            };
            ListView = itensMenuListView;
            BindingContext = this;
        }
    }
}

```

Com nossas páginas que representarão O Hamburger Menu (Master e Details ) implementadas, vamos criar a nossa página que implementa esta associação Master Detail entre as visões. Lembre-se de que nosso primeiro details é a listagem de clientes. Não seguiremos o template do Visual Studio agora, criaremos uma Content Page na pasta views e realizaremos as alterações nela. Minha sugestão de nome para esta página é de `MainPageView` e ela terá, em seu XAML, o código apresentado na sequência.

Observe que o primeiro elemento foi alterado para `<MasterDetailPage>` e que temos dois namespaces para o XML (`xmlns`), o `masterPage` e o `detailPage`. O `masterPage` é utilizado na definição do elemento `Master` da estrutura, já o elemento `Detail`, para que possa conhecer a técnica, será resolvido via code-behind.

```
<MasterDetailPage xmlns = "http://xamarin.com/schemas/2014/forms"
```

```

xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

x:Class = "Capitulo05.Views.MainPageView"

xmlns:masterPage = "clr-namespace:Capitulo05.Views"

xmlns:detailPage = "clr-namespace:Capitulo05.Views.Clientes" >

<MasterDetailPage.Master>

<masterPage:MasterPageView x:Name = "masterPage" />

</MasterDetailPage.Master>

<MasterDetailPage.Detail>

</MasterDetailPage.Detail>

</MasterDetailPage>

```

Vamos para a implementação do code-behind para esta página, veja o código na sequência. Observe que a classe estende `MasterDetailPage`. No código do construtor, é atribuído um método para o evento `ItemSelected` do `ListView`, e a propriedade `IsPresented` é inicializada com o valor `true` para que a página master seja exibida. Em seguida, é instanciada e configurada a página que será o `Detail` inicial, que é nossa listagem de clientes.

No método `ListView_ItemSelected`, são tratadas a preparação e instanciação de qual página será renderizada, de acordo com a configuração do item associado ao evento, informado pela seleção no `ListView`. Neste método também é atribuído `false` para o `IsPresented` para que a página `Detail` possa ser exibida.

```

namespace Capitulo05.Views
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class MainPageView : MasterDetailPage
    {
        public MainPageView()
        {
            InitializeComponent();
            masterPage.ListView.ItemSelected += ListView_ItemSelected;
            IsPresented = true;

            var navigationPage = new Xamarin.Forms.NavigationPage(new Views.Clientes.ListagemView());
            navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetPrefersLargeTitles(true);

            this.Detail = navigationPage;
        }

        private void ListView_ItemSelected(object sender, SelectedItemChangedEventArgs e)
        {
            var item = e.SelectedItem as Models.MenuItem;
        }
    }
}

```

```
        if (item == null)
            return;

        var page = (Xamarin.Forms.Page)Activator.CreateInstance(item.TargetType);
        page.Title = item.Title;

        Detail = new Xamarin.Forms.NavigationPage(page);
        (Detail as Xamarin.Forms.NavigationPage).On<Xamarin.Forms.PlatformConfiguration.iOS>
().SetPrefersLargeTitles(true);
        IsPresented = false;

        masterPage.ListView.SelectedItem = null;
    }
}
}
```

Precisamos alterar nossa classe `App` para utilizar a nossa nova visão inicial. Altere a declaração de `MainPage` para `MainPage = new MainPageView();`. Será preciso inserir o respectivo `using` para a classe. Execute sua aplicação, teste-a também em seus dispositivos. Você deverá ter acesso a algo semelhante à figura apresentada na sequência. Lembre-se de que já vimos como configurar propriedades de exibição de acordo com a plataforma, e que a resolução do emulador não é das melhores para o Android, mas você agora já sabe fazer o deploy para seu dispositivo, que certamente tem uma maior qualidade.

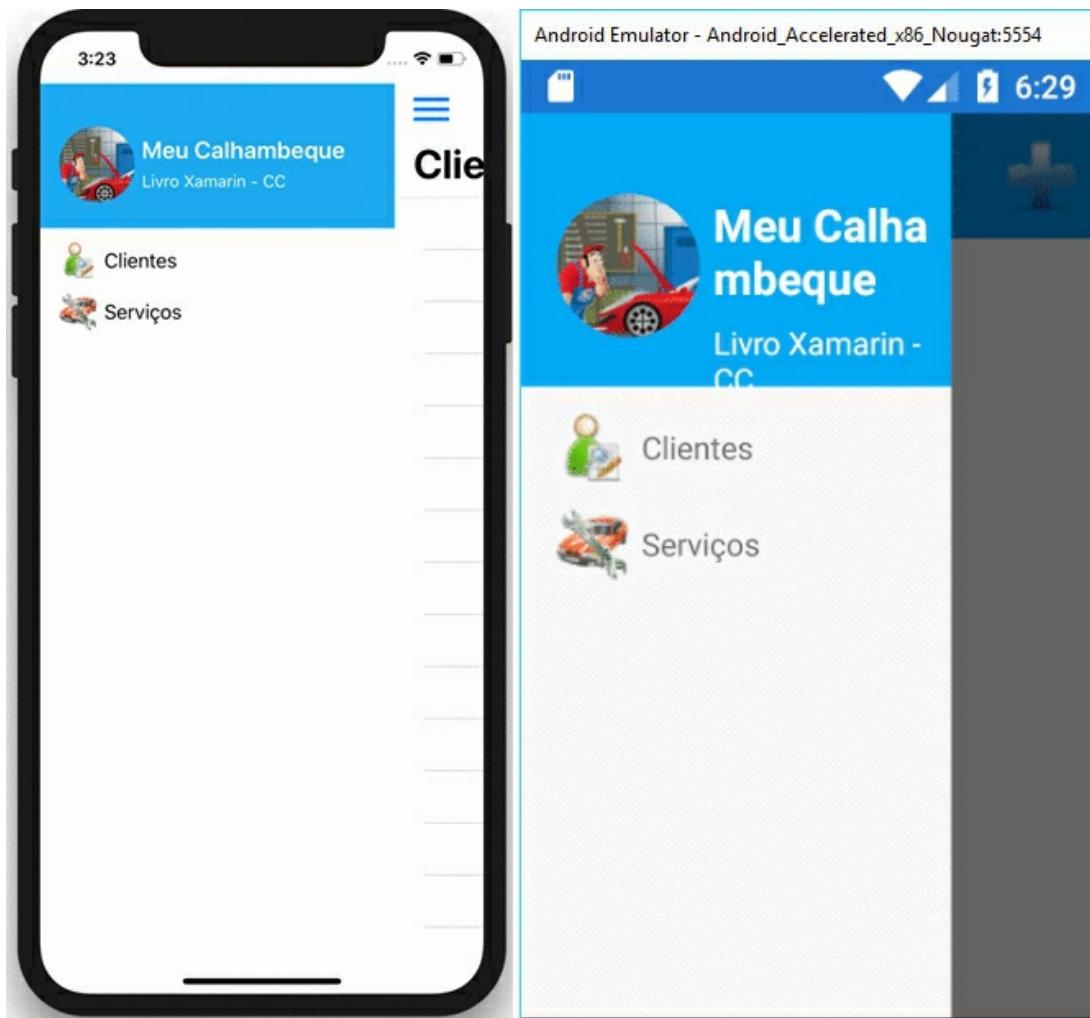


Figura 5.7: Visão Master Detail para o Hamburger Menu e a Listagem de Clientes

## 5.10 A página responsável pela inserção e alteração de um cliente

Para que possamos implementar nosso código de maneira gradativa, seguindo as funcionalidades para clientes, vamos trabalhar agora a visão responsável pela inserção e alteração deles. Vimos que uma única visão pode ser responsável por estas duas operações e ela será invocada a partir da janela de listagem, tal qual fizemos no capítulo anterior. Portanto, vamos criar nosso `Content Page` na pasta `Clientes`, dentro de `Views`. Implemente seu código para ficar semelhante ao apresentado na sequência. Veja que o nome para a visão é `CRUDView`. Leia o código e veja que não temos nada de novo nele, estamos apenas utilizando nosso conhecimento adquirido nos capítulos anteriores.

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"

xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

x:Class = "Capitulo05.Views.Clientes.CRUDView"
```

```

xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOS;assembly=Xamarin.Forms.Core"

ios:Page.UseSafeArea = "true" >

<ContentPage.Content>

<StackLayout>

<TableView Intent = "Form" HasUnevenRows = "True" >

<TableRoot>

<TableSection Title = "Dados do cliente" >

<EntryCell Label = "Nome:" Text = "{Binding Nome}" ></EntryCell>

<EntryCell Label = "Telefone:" Text = "{Binding Telefone}" Keyboard = "Telephone" ></EntryCell>

<EntryCell Label = "e-Mail:" Text = "{Binding Email}" Keyboard = "Email" ></EntryCell>

<ViewCell>

<Button Text = "Gravar Alterações" FontAttributes = "Bold" VerticalOptions = "End" Command = "{Binding GravarCommand}" />

</ViewCell>

</TableSection>

</TableRoot>

</TableView>

</StackLayout>

</ContentPage.Content>

</ContentPage>

```

Precisamos implementar o databinding para nossa visão e, já que estamos seguindo o MVVM, você já sabe que precisamos de uma classe com o papel de ViewModel. Relembrando o capítulo 4, onde introduzimos o MVVM, fizemos uso de uma classe básica para os ViewModels. Vamos trazê-la para nossa aplicação agora, pois ela tem funcionalidades comuns que utilizaremos em nossas ViewModels. Na pasta `viewModels`, crie uma classe chamada `BaseViewModel`, tal qual o seguinte código.

```

namespace Capitulo05.ViewModels
{
    public class BaseViewModel : INotifyPropertyChanged
    {

```

```

private bool isBusy = false;
public bool IsBusy
{
    get { return isBusy; }
    set { isBusy = value; OnPropertyChanged(); }
}

public event PropertyChangedEventHandler PropertyChanged;
protected void OnPropertyChanged([CallerMemberName] string propertyName = "")
{
    var changed = PropertyChanged;
    if (changed == null)
        return;

    changed.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
}
}

```

Muito bem, agora que temos nossa classe base para as ViewModels já implementada, vamos criar o básico de nossa classe ViewModel para registrarmos a ligação à nossa visão de inserção e alteração de clientes. Na pasta `viewModels`, dentro de `clientes`, crie a classe `CRUDViewModel`, com o código apresentado na sequência. Observe a declaração para o DAL, que é inicializado no construtor. Veja que o construtor recebe um objeto `Cliente`, que terá suas propriedades ligadas a propriedades da ViewModel.

```

namespace Capitulo05.ViewModels.Clientes
{
    public class CRUDViewModel : BaseViewModel
    {
        private IDAL<Cliente> clientesDAL;
        private Cliente Cliente { get; set; }
        public ICommand GravarCommand { get; set; }

        public CRUDViewModel(Cliente cliente)
        {
            clientesDAL = new ClienteDAL(DependencyService.Get<IDBPath>().GetDbPath());
            this.Cliente = cliente;
        }
    }
}

```

Vamos agora para o code-behind da visão `crudView`, pois já podemos realizar a ligação dela com o seu respectivo ViewModel. Veja o código na sequência. Observe a declaração do objeto `crudViewModel` e sua inicialização no construtor parametrizado, que, para que possa inicializar corretamente a ViewModel, recebe um objeto de Cliente e uma string, que será o título de nossa visão, dependendo do que a visão estará realizando com o objeto recebido.

```

namespace Capitulo05.Views.Clientes
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class CRUDView : ContentPage
    {
        private CRUDViewModel crudViewModel;

```

```

public CRUDView()
{
    InitializeComponent();
}

public CRUDView(Cliente cliente, string title) : this()
{
    this.crudViewModel = new CRUDViewModel(cliente);
    this.BindingContext = this.crudViewModel;
    this.Title = title;
}
}
}

```

Muito bem, você viu no XAML da visão que estamos trabalhando para que os controles de entrada de dados estejam todos ligados a propriedades, mas não temos a implementação para elas ainda. Vamos realizar isso. Só reforçando, a ligação toda fica em nossa classe `CRUDViewModel`. No código a seguir, veja o que comentei anteriormente para a ligação das propriedades. Optei por trabalhar diretamente com o objeto em foco, pois o receberemos pelo construtor.

```

public string Nome
{
    get { return this.Cliente.Nome; }

    set
    {
        this.Cliente.Nome = value;
        ((Command)GravarCommand).ChangeCanExecute();
        OnPropertyChanged();
    }
}

public string Telefone
{
    get { return this.Cliente.Telefone; }

    set
    {
        this.Cliente.Telefone = value;
        ((Command)GravarCommand).ChangeCanExecute();
        OnPropertyChanged();
    }
}

public string EMail
{
    get { return this.Cliente.EMail; }

    set

```

```

{
    this.Cliente.EMail = value;
    ((Command)GravarCommand).ChangeCanExecute();
    OnPropertyChanged();
}
}

```

Com a implementação das propriedades, já podemos abstrair que temos os dados informados na visão. O que precisamos fazer agora? Precisamos providenciar o método responsável pela persistência na base de dados, dos dados informados para o cliente da visão/ViewModel. Isso nos remete à classe `DAL`, que está no projeto `SQLiteEF`. Veja, na sequência, a implementação do método responsável por persistir os dados de um cliente. Esta implementação deve ser realizada na classe `DALBase`. O EF Core não oferece um método assíncrono para `update()`.

```

public async virtual Task<T> UpdateAsync (T item, long ? itemID)

{
    using ( var context = DatabaseContext.GetContext(dbPath))
    {
        if (itemID != null )
        {
            context.Update(item);
        }
        else
        {
            await context.AddAsync(item);
        }
        await context.SaveChangesAsync();
    }
    return item;
}

```

Muito bem, agora precisamos providenciar o método, em nossa ViewModel, que consumirá este método do DAL. Vamos criá-lo na classe `CRUDViewModel`, tal qual se segue. Veja o flag criado para indicar que o objeto que está sendo gravado representa um novo cliente. Estamos utilizando isso para possibilitar ao usuário a inserção de vários clientes, um após o outro. Seguindo o que vimos no capítulo anterior, fazemos uso de um método específico para a atualização dos controles visuais para o novo objeto. Este método (`AtualizarPropriedadesParaVisao()`) está também na sequência.

```

private async Task GravarAsync ()

{
    var ehNovoCliente = (Cliente.ClienteID == null ? true : false );
    await clientesDAL.UpdateAsync(Cliente, Cliente.ClienteID);
    AtualizarPropriedadesParaVisao(ehNovoCliente);
}

private void AtualizarPropriedadesParaVisao ( bool ehNovoObjeto)

```

```

{
    if (ehNovoObjeto)
    {
        this .Nome = string .Empty;
        this .EMail = string .Empty;
        this .Telefone = string .Empty;
        this .Cliente = new Cliente();
    }
}

```

É preciso agora registrar nosso `Command` e, novamente, seguindo nosso padrão, faremos isso implementando o método `RegistrarCommands()`, que está na sequência. Lembre-se de que você precisa invocar este método em seu construtor. Para isso, basta digitar ao término do construtor a instrução `RegistrarCommands();`. Veja a invocação ao método `GravarAsync()` e a declaração da lógica para habilitar o botão de gravação.

```

private void RegistrarCommands ()
{
    GravarCommand = new Command( async () =>
    {
        await GravarAsync();

        MessagingCenter.Send< string >( "Atualização realizada com sucesso." , "InformacaoCRUD" );
    }, () =>
    {
        return ! string .IsNullOrEmpty( this .Cliente.Nome) && ! string .IsNullOrEmpty( this .Cliente.Telefone) && ! string
        .IsNullOrEmpty( this .Cliente.EMail);
    });
}

```

A seguir, estão as implementações que devem ser feitas na classe da visão, em relação à mensagem ao *Messaging Center* do código anterior.

```

// OnAppearing

MessagingCenter.Subscribe< string >( this , "InformacaoCRUD" , async (msg) => { await DisplayAlert ( "Informação" , msg, "ok"
});

// OnDisappering

MessagingCenter.Unsubscribe< string >( this , "InformacaoCRUD" );

```

Estamos com quase tudo pronto para testarmos a inserção de um cliente. Vamos lembrar onde está o acesso à inserção de um novo cliente? Na `Toolbar` da listagem. Lá temos a ligação com o `NovoCommand`, já implementado em nossa `ListagemViewModel`. O que nos falta? Implementar na `ListagemView` a invocação de nossa visão de inserção, pois temos as mensagens configuradas lá, mas não o comportamento para elas. Vamos fazer isso. Adapte seu código da mensagem para o exposto na sequência. Observe que utilizamos um operador ternário para identificar o título para a nova visão e em seguida a invocamos. Nós já implementamos anteriormente o `Unsubscribe()`.

```
// OnAppearing
```

```

MessagingCenter.Subscribe<Cliente>( this , "Mostrar" ,
    async (cliente) => { await Navigation.PushAsync( new CRUDView(cliente, viewModel.Clientes, (cliente.ClienteID == null )
    "Novo Cliente" : "Alterar Cliente" )); });

```

Muito bem! Com o que temos já podemos testar nossa aplicação e verificar a inserção de um novo Cliente. Veja a seguinte figura, que traz a visão de inserção de clientes.

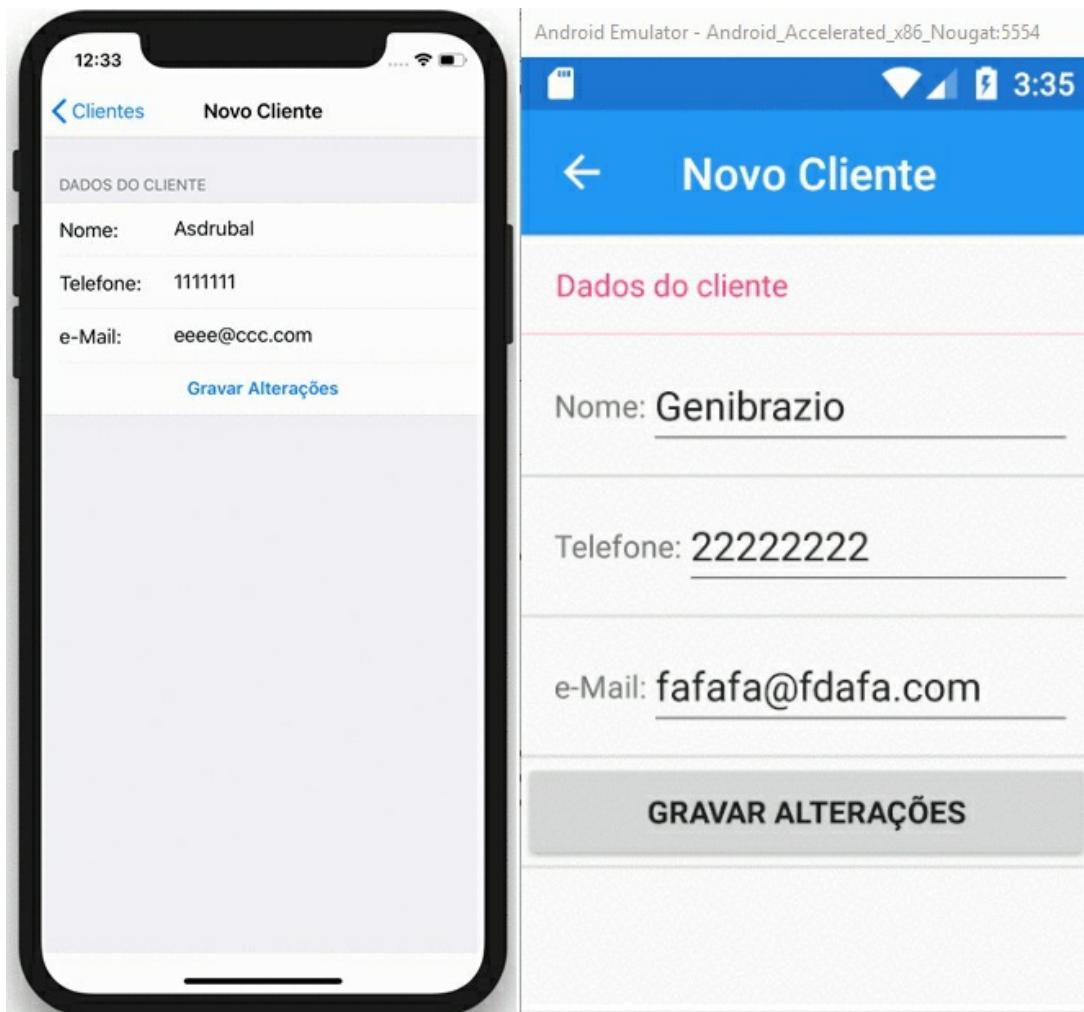


Figura 5.8: Visão de inserção e alteração de um cliente

Embora eu tenha dito que a visão que criamos é a mesma para inserção e alteração, nós não temos ainda a funcionalidade de alteração de um cliente implementada. Se você verificar nosso código da classe `ListagemViewModel`, temos lá um objeto chamado `clienteSelecionado` e, se você pressionar um cliente na listagem, ele fica marcado. O que precisamos fazer para que, ao se pressionar um cliente, a aplicação entenda que queremos alterar os dados ou apenas visualizá-los de uma maneira mais completa? Precisamos mapear uma propriedade para o objeto comentado e, em sua atribuição, realizar o `push` para a visão, passando para ela o cliente selecionado. Nós já temos em nosso XAML o mapeamento da propriedade `SelectedItem`, falta-nos apenas o código na classe `ListagemViewModel`, que é o que apresento na sequência. Veja que enviamos o cliente que foi selecionado para a mensagem, que já temos implementada.

```

public Cliente ClienteSelecionado
{

```

```

get { return clienteSelecionado; }

set
{
    if ( value != null )
    {

        clienteSelecionado = value ;
        MessagingCenter.Send<Cliente>(clienteSelecionado, "Mostrar" );
    }
}
}

```

Vamos abrir um espaço para discutir um problema na atualização do `Listview`. Selecione um cliente da listagem, altere seu telefone, grave e retorne para ela. A listagem não aparece com o número atualizado, correto? Isso é facilmente explicado pela Orientação a Objetos. Quando alteramos um objeto, em nosso caso, um cliente, as alterações repercutem em todo objeto que o utiliza, em nosso caso, a `observableCollection Clientes`. Quando chamamos nosso método de extensão, os itens recuperados da base estão na mesma ordem de classificação que a coleção ligada ao `Listview`, logo, este item não será atualizado na listagem. Poderíamos pensar, também orientado a objetos, que pelo fato de o objeto alterado pertencer a um `ObservableCollection`, o `Listview` seria automaticamente atualizado, mas infelizmente isso não ocorre. Precisamos de um *workaround* para contornar esta situação e alterar este item na coleção que popula o `Listview`. O primeiro passo para isso é, na classe `ClienteIDProperty`, adicionarmos a seguinte propriedade.

```

[NotMapped]

public bool NotificarListView { get ; set ; }

```

Veja que a propriedade `NotificarListView` é precedida do atributo `[NotMapped]`, que indicará ao EF Core que esta propriedade não deve ter um campo criado na tabela e será usada apenas em tempo de execução para resolver nosso problema de atualização do `Listview`. Quando você digitar o atributo, ocorrerá um erro de compilação e o Visual Studio vai sugerir que você adicione a referência para o *assembly* que contém este atributo, basta você confirmar isso. Caso ocorra de não aparecer esta opção, será preciso adicionarmos o Nuget `Microsoft.EntityFrameworkCore` ao projeto.

Dando sequência, precisamos alterar nosso `CRUDViewModel` para atribuir `true` a esta propriedade, quando a gravação ocorrer em um cliente que já tenha sido incluído, ou seja, quando alterarmos um item. Sendo assim, no método `AtualizarPropriedadesParaVisao()`, insira o código a seguir, após o fechamento da chave do `if()`.

```

else
{
    this.Cliente.NotificarListView = true ;
}

```

Por fim, precisamos corrigir nosso método de extensão `SincronizarColecoes()` para verificar se há alteração no item que se está sincronizando. Veja o código a seguir que deve estar na sequência do `if()` do primeiro `for()`. Neste código, recuperamos dinamicamente a nossa propriedade `flag` e, caso ela exista no modelo e tenha seu valor lógico verdadeiro, atualizamos o item ligado à coleção do `ListView`.

```

else
{
    var notificarListView = destino[i].GetType().GetProperty( "NotificarListView" ).GetValue(destino[i], null );

    if (notificarListView != null && ( bool ) notificarListView

```

```

    destino[i] = origem[i];
}

```

Muito bem, vamos agora testar nossa aplicação, selecione um cliente, altere seus dados, grave a modificação e retorne para a listagem. Tudo funcionando certinho, não é mesmo?

## 5.11 Remoção de cliente já cadastrado

Nossa visão de listagem já traz um item de menu para a remoção de um cliente. No iOS, basta arrastar para a esquerda um item da lista e, no Android, pressione-o por alguns instantes. Você viu que o menu aparece, mas nos falta a funcionalidade. De acordo com o que estamos seguindo, vamos implementar esta funcionalidade em nossa classe `DALBase`, no projeto `SQLiteEF`. Veja o código na sequência.

```

public virtual async Task DeleteAsync (T item)

{
    using ( var context = DatabaseContext.GetContext(dbPath))
    {
        context.Remove(item);
        await context.SaveChangesAsync();
    }
}

```

Precisamos consumir este serviço em nossa `ViewModel`, pois já temos inclusive a ligação implementada em nossa visão, faltando só esta implementação na classe `ListagemViewModel`. Veja que na remoção não precisaremos repopular a propriedade ligada ao `ListView`, podemos remover o objeto diretamente dela.

```

public async Task EliminarClienteAsync (Cliente cliente)

{
    await clientesDAL.DeleteAsync(cliente);
    clientes.Remove(cliente);
}

```

Mas como este método será consumido? Dê uma verificada em nosso `EliminarCommand`, dentro do `RegistrarCommands()`. Veja que estamos invocando uma mensagem, cujo corpo já temos declarado na `ListagemView`, falta-nos apenas o comportamento. Vamos fazê-lo! Veja a listagem que segue. O trecho inicial define o método que será disparado quando a mensagem ocorrer e, em seguida, apresenta-se a instrução de registro da mensagem. O que acha de testar sua aplicação? Após a listagem, você pode verificar a figura representando esta funcionalidade.

```

private async Task RemoverClienteAsync (Cliente cliente)

{
    if ( await DisplayAlert ( "Confirmação" ,
        $ "Confirma remoção de {cliente.Nome.ToUpper()}?" , "Yes" , "No" ) )
    {
        await this .viewModel.EliminarClienteAsync(cliente);
        await DisplayAlert ( "Informação" , "Cliente removido com sucesso" , "Ok" );
    }
}

```

```

}

// OnAppearing

MessagingCenter.Subscribe<Cliente>( this , "Confirmação" , async (cliente) => await RemoverClienteAsync (cliente) );

```

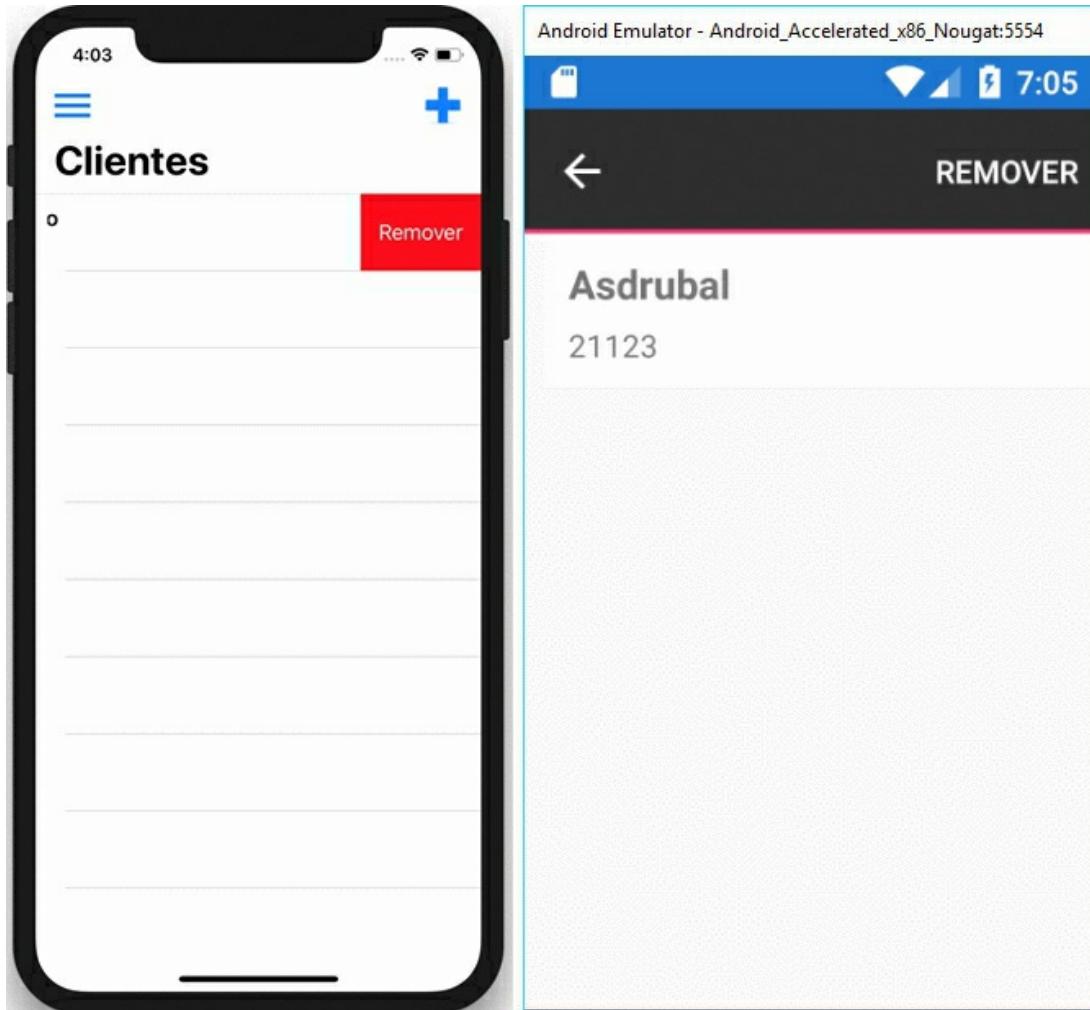


Figura 5.9: Visão remoção de um Cliente

## 5.12 Interação e manutenção dos dados de Serviço

Se você verificar a nossa visão que representa o menu de opções de nossa aplicação, lá já consta a de serviços . A funcionalidade que precisamos implementar para esta opção é a mesma que fizemos para clientes. Note que os dados se referem aos tipos de serviços que a oficina oferecerá e não ao registro de entrada de veículos na oficina. Como o trabalho a ser realizado é semelhante ao que já fizemos, deixarei a seu cargo a implementação, mas deixo o conteúdo disponível em [https://github.com/evertonfoz/xamarin\\_mvvm\\_efcore/tree/master/Capitulo05/](https://github.com/evertonfoz/xamarin_mvvm_efcore/tree/master/Capitulo05/) .

## 5.13 Atualização da base de dados EF Core com Migrations

Quando executamos nossa aplicação com a implementação dos serviços referentes a Clientes, a base de dados foi criada e a tabela para os dados de Clientes também. Agora, temos uma situação de adição de uma nova tabela (Serviço) à base de dados. Não basta inserir a coleção para serviços na classe de contexto para que a tabela seja criada.

Para atualizar a base de dados com o EF Core, as opções que temos são: remover a base de dados, com o `this.Database.EnsureDeleted();` e, em seguida, recriá-la, com o `this.Database.EnsureCreated();`, no construtor da classe `DbContext`; ou então, mudar o nome do arquivo que representa a base de dados na classe `DatabasePath` no projeto específico da plataforma. Em ambos os casos, perdemos os dados que temos nas tabelas já existentes em nossa aplicação e isso não é produtivo, mesmo que sejam dados de testes.

Para estes problemas, a Microsoft tem uma solução chamada `Migrations`, que podemos fazer funcionar em nossa aplicação Xamarin, mas precisamos aplicar um *workaround*, pois o EF Core `Migrations` não funciona, por definição, em um projeto .NET Standard, tampouco nas plataformas nos projetos específicos de plataforma. Quem sabe no futuro isso possa ocorrer. Existem algumas propostas e artigos na internet que propõem alternativas para este problema, mas eu prefiro ir para uma solução mais prática.

Vamos criar em nossa solução um projeto específico para o uso do `Migrations`. Desta maneira, clique com o botão direito do mouse no nome da solução e, em seguida, em `Adicionar -> Novo projeto`. Dentro da categoria `.NET Core`, escolha `Aplicativo de Console (.NET Core)`. Eu nomeei meu projeto de `Capitulo05Migrations`. Adicione ao projeto, após sua criação, a referência ao projeto `OficinaModels`.

Precisamos instalar alguns componentes em nosso novo projeto. Podemos fazer isso utilizando o gerenciamento de Pacotes Nuget, como vimos em situações anteriores, ou então, clicando com o botão direito do mouse sobre o nome do projeto e em `Editar Capítulo05Migrations.csproj`. No código que se exibe, você verá um elemento `<ItemGroup>`, que você deverá inserir logo após o `<PropertyGroup>` que aparece no arquivo. Veja o código apresentado na sequência. As versões apresentadas seguem o já comentado quando introduzimos o EF Core.

```
<ItemGroup>

<PackageReference Include = "Microsoft.EntityFrameworkCore" Version = "2.0.3" />

<PackageReference Include = "Microsoft.EntityFrameworkCore.Sqlite" Version = "2.0.3" />

<PackageReference Include = "Microsoft.Data.Sqlite.Core" Version = "2.0.1" />

<PackageReference Include = "Microsoft.EntityFrameworkCore.Design" Version = "2.0.3" />

<PackageReference Include = "Microsoft.EntityFrameworkCore.Sqlite.Core" Version = "2.0.3" />

<PackageReference Include = "Microsoft.EntityFrameworkCore.Tools" Version = "2.0.3" />

<DotNetCliToolReference Include = "Microsoft.EntityFrameworkCore.Tools.DotNet" Version = "2.0.3" />

</ItemGroup>
```

Vamos entender como o `Migrations` trabalha. Assim que concluirmos nossas adaptações e executarmos alguns instruções para habilitarmos o `Migrations` em nosso projeto, mudaremos para `Migrate()`, no `DbContext`, o método que carregará a base de dados. A cada migração que faremos, em decorrência de alterações em nosso modelo de negócio, o `Migrations` criará arquivos que refletem, em código C# para o EF Core, como estas alterações

devem ser aplicadas na base de dados.

Vamos para a implantação do Migrations. Crie no novo projeto uma pasta chamada `DataAccess` e copie, do projeto `SQLiteEF`, a classe `DbContext` para essa nova pasta. No projeto `Migrations`, na classe `DbContext`, você pode retirar os construtores e definições de objetos privados. Precisaremos apenas das declarações de conjunto e do método `OnConfiguring()`. Para auxiliar, veja o código interno da classe na sequência.

```
public DbSet<Cliente> Clientes { get; set; }

public DbSet<Servico> Servicos { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlite($"Filename=OficinaMigrations.db");
}
```

Vamos agora partir para a habilitação do Migrations. Para isso, clique com o botão direito do mouse sobre o nome do projeto de console e então escolha a opção `Abrir pasta no Gerenciador de Arquivos`. Na caixa de texto com o endereço físico do projeto, antes do endereço, digite `cmd` e pressione `ENTER` para que a janela de console seja aberta na localização do projeto.

Na janela de console, digite `dotnet restore` para garantir que todas as dependências sejam carregadas. Em seguida, precisamos habilitar o Migrations no projeto. Para isso, digite agora `dotnet ef migrations add MapeamentoInicial`, onde `MapeamentoInicial` será o nome da primeira configuração para uso do Migrations. Como em nosso `DbContext` temos um `DbSet` para `Clientes` e outro para `Servicos`, serão criadas instruções para a criação das tabelas, relativas a estas coleções, na base de dados.

Volte ao Visual Studio e verifique que uma nova pasta foi criada em seu projeto de console, chamada `Migrations`. Precisamos copiar toda a pasta para nosso projeto `SQLiteEF`. Após a cópia, no `SQLiteEF` precisamos acertar o namespace dos arquivos criados (3) para que façam parte da estrutura lógica de nosso projeto. Pode ser que em seu Visual Studio um dos arquivos (Designer) esteja colapsado no arquivo da migração. Neste caso, expanda-o para vê-lo. Em meu caso, o namespace para os três arquivos ficou sendo `casaDoCodigo.DataAccess.Migrations`.

Partiremos para uma última alteração em nossa classe `DbContext`, agora no projeto `SQLiteEF`. No construtor da classe precisamos executar um método que faça uso do Migrations. Desta maneira, deixe seu construtor tal qual é apresentado no código a seguir. Na primeira execução insira o método `this.Database.EnsureDeleted()`, que está comentado no código. Logo após a aplicação executar, sua base de dados estará atualizada com o Migrations também atualizado. Infelizmente, isso levou à perda de dados que existiam na tabela, mas essa situação pode ser evitada, bastando, no arquivo criado pelo Migrations que tem o sufixo `MapeamentoInicial`, comentar a chamada aos métodos `migrationBuilder.CreateTable()` que se refiram às tabelas já criadas. Fica a dica registrada, caso surjam problemas no futuro. Após a execução, volte ao código e retire este comentário. Vamos testar a aplicação?

```
private DbContext()
{
    //this.Database.EnsureDeleted();

    this.Database.Migrate();
}
```

É importante eu ressaltar que o Migrations não é a solução perfeita para todas as atualizações que teremos de realizar em uma base de dados. Ele possui algumas limitações, principalmente em relação ao SQLite e elas podem ser verificadas em <https://docs.microsoft.com/pt-br/ef/core/providers/sqlite/limitations/>. O que fazer quando nos

deparamos com uma destas limitações? A resposta é termos alguma rotina para armazenar os dados existentes na aplicação, remover a base antiga, criar a nova e recuperar os dados. Mas isso quando se trata de uma aplicação final. Durante o desenvolvimento, normalmente a solução está em eliminar a base, criar de novo e popular as tabelas com dados de testes. Em algumas situações, durante a escrita do livro, ocorreu a necessidade de remover todo o conteúdo da pasta `Migrations` e recriá-la com os passos apresentados anteriormente.

Nas figuras que se seguem estão as imagens que representam as visões em execução nos emuladores do iOS e Android.

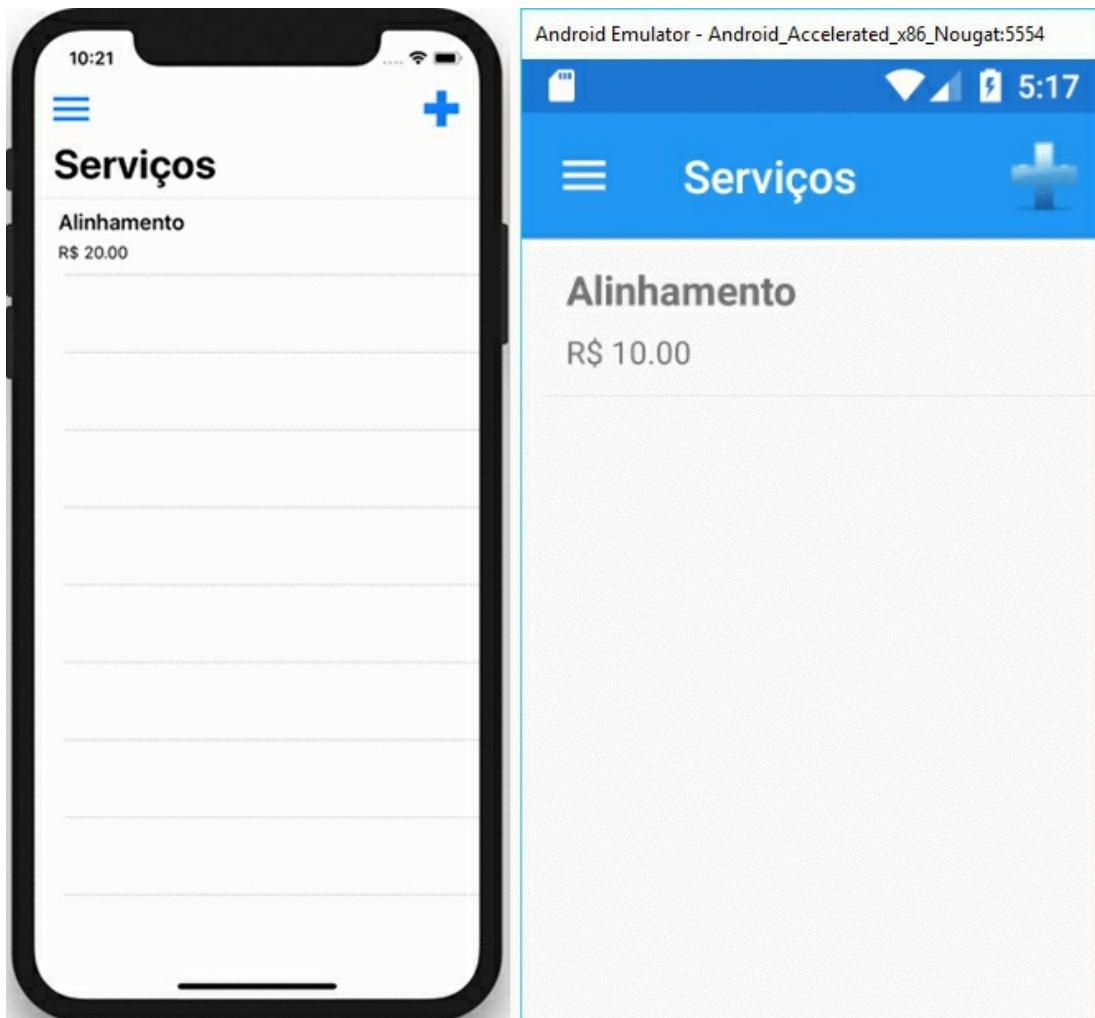


Figura 5.10: Listagem de serviços e visão para inserção

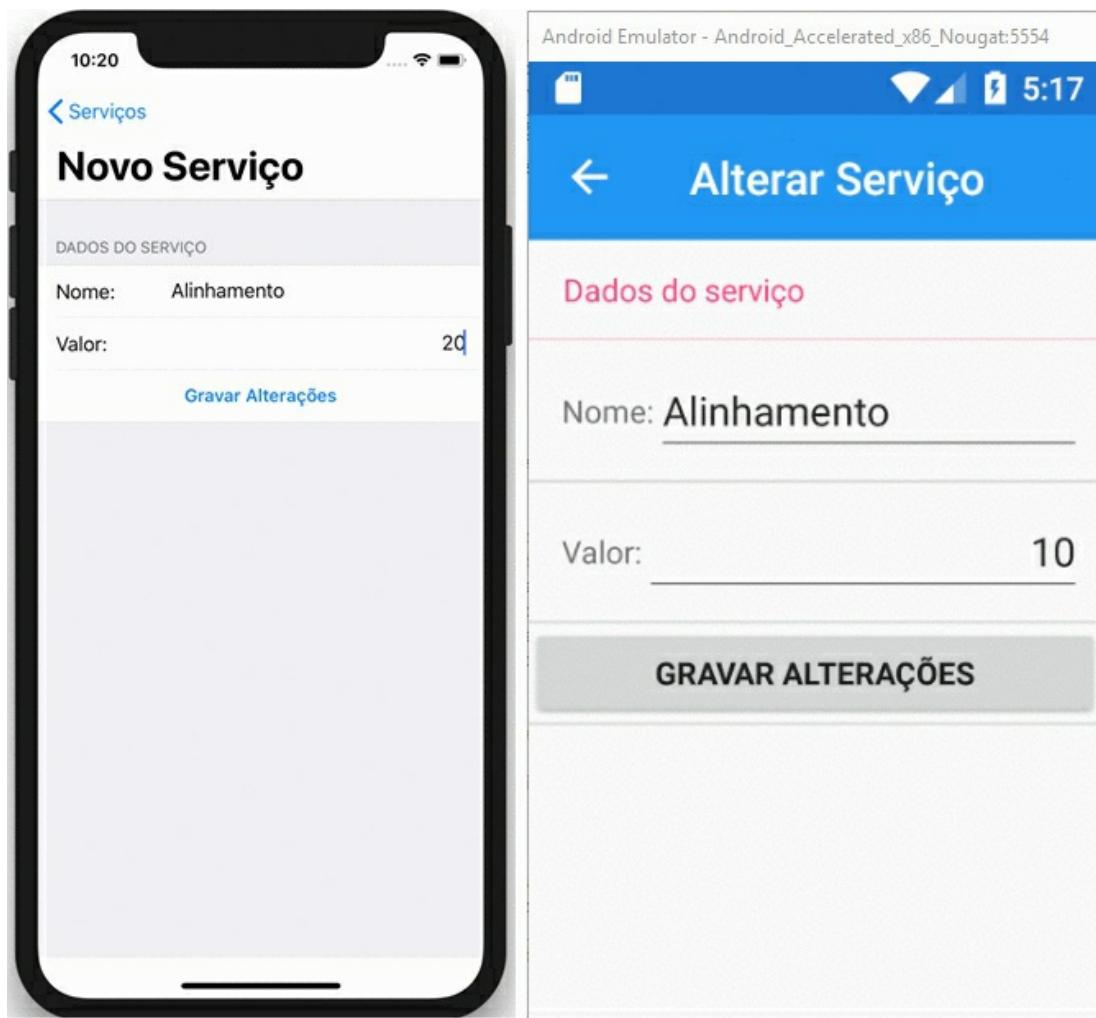


Figura 5.11: Visão de alteração de serviços e remoção

## 5.14 Manipulação da base de dados do SQLite

Muitas vezes se faz necessário acessar a base de dados que nosso dispositivo manipula, neste caso, nossos emuladores. Existem diversas ferramentas para isso, e aqui vamos utilizar o `db Browser for SQLite`, que você pode obter em <http://sqlitebrowser.org/> e instalar facilmente.

Para acessar a base de dados criada para os emuladores, eu recomendo que você obtenha o caminho por meio de depuração, inserindo um *break-point* na classe que criamos como `Dependency Service` em cada projeto e verificando o valor do caminho. Veja isso na figura a seguir.

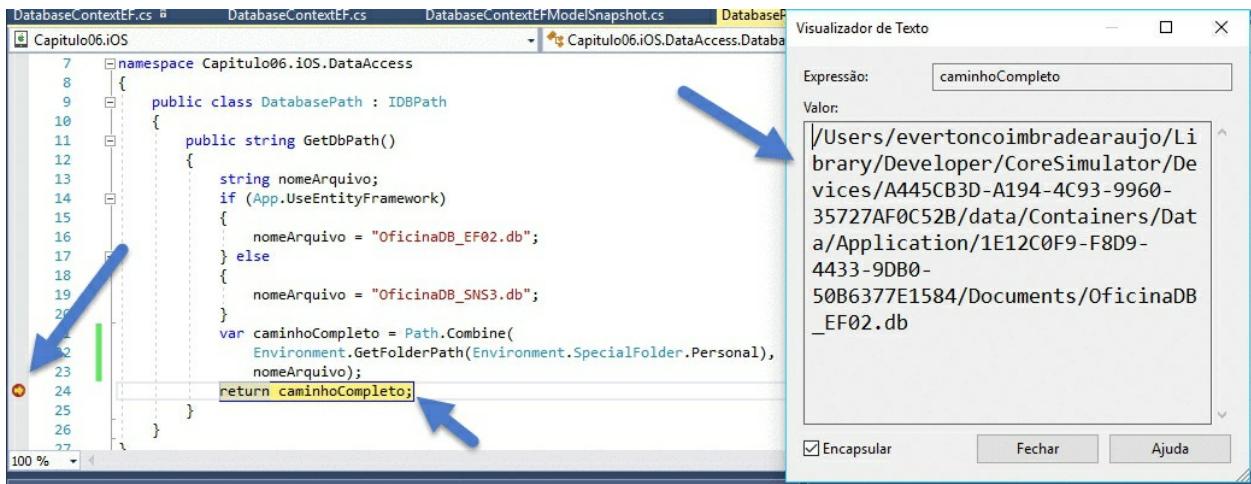


Figura 5.12: Break-point para obter o valor da variável com o caminho da base de dados

Com este caminho em mãos, precisamos agora abrir o arquivo no DB Browser for SQLite . Clique em Abrir banco de dados , no topo da janela ou clique no menu Arquivo e selecione esta opção. A figura a seguir apresenta o aplicativo com a base de dados aberta. Procure navegar pelas janelas e investigar seu uso. É uma ferramenta simples e útil.

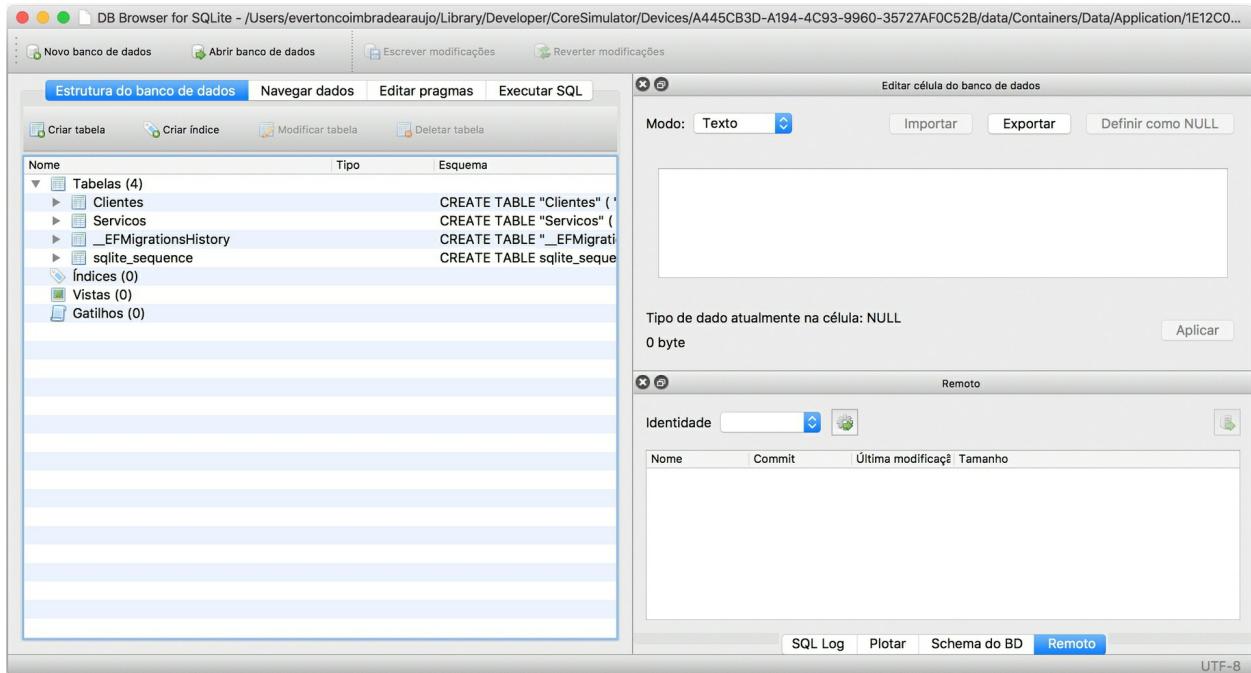


Figura 5.13: DB Browser for SQLite com a base de dados aberta

O acesso ao caminho do emulador do iOS foi simples, como dito anteriormente. Para acessar o arquivo do emulador Android foi algo mais trabalhoso. Algumas documentações trazem que basta acessar o `Android Device Monitor DDMS`, acessível pelo último ícone do grupo de ícones relativos ao Android na barra de tarefas do Visual Studio Windows, ou, no Mac, em `Ferramentas->Device Monitor` . Com a ferramenta aberta, é preciso identificar o arquivo na guia `File Explorer` e então clicar em um botão que está no topo da janela, à direita, chamado `Pull a file from the device` . Minha base de dados estava no caminho `/data/user/0/com.casadocodigo.Capítulo05/files/Oficina.db` .

Mas o que seria das soluções se não existissem problemas, não é mesmo? Ocorre que os dados estão dentro de uma

pasta mais externa ao caminho exibido, chamada `data` . Desta maneira, no `File Explorer` , precisamos acessar `data/data/user/0/com.casadocodigo.Capitulo05/files/` , clicar no arquivo `oficina.db` e depois no botão para obter o arquivo. Em alguns casos, clicando na primeira pasta `data` , os subníveis não são exibidos, por segurança no acesso. O primeiro passo, antes de acessarmos o DDMS para termos condições de obter o arquivo, é acessar o caminho via console onde estão instaladas as ferramentas da plataforma Android. Em minha máquina Windows, este caminho é `C:\Program Files (x86)\Android\android-sdk\platform-tools` .

Estando na pasta, execute a instrução `adb shell` . No terminal que se abre, digite `su` para alterar o usuário. Acesse o caminho do arquivo com `cd data/data/com.casadocodigo.Capitulo05/files` e, em seguida, execute `chmod 777 /data` para mudar os atributos de acesso ao diretório. Se você não conseguir acessar o caminho anterior, talvez precise alterar as permissões para todas as pastas com a instrução `chmod 777 /data /data/data/*` . Tente obter o arquivo agora. Se ainda não conseguir, pois pode ser que apareça um erro durante a transferência, relacionado a permissões, novamente no console, digite `adb root` , estando na pasta `platform-tools` . Agradeço muito a Jerry Zhao, que trouxe essa pérola em seu link <https://www.dev2qa.com/android-device-monitor-cannot-open-data-folder-resolve-method/> .

## 5.15 Conclusão

Chegamos ao final deste capítulo sobre persistência em uma base de dados. Foi um capítulo extenso, muito código, explicações e técnicas. Foi possível trabalhar com o Entity Framework Core, que é o sistema de mapeamento objeto-relacional da Microsoft. Fizemos bastante uso de Orientação a Objetos, buscando coesão e acoplamento corretos.

Armazenamos, recuperamos, manipulamos e removemos dados de tabelas relacionais. Também foi possível executar nossa aplicação em dispositivos físicos, o que permitirá a você uma visualização mais real dos layouts desenhados para suas visões.

No próximo capítulo, trabalharemos mais um pouco o acesso a dados, trazendo o tema de associações. Veremos também alguns controles novos para interação com o usuário. Vamos tomar uma água agora.

## C APÍTULO 6

# Associações, Pesquisa, DatePicker, TimePicker e ActionSheet

No capítulo anterior, implementamos dois CRUDs acessando uma base de dados SQLite. O capítulo foi extenso e produtivo: preparamos a base de dados para a implementação de entrada do veículo na oficina, que é o objetivo principal deste capítulo que estamos começando. Trarei aqui a associação entre classes mapeadas para a base de dados, pois nossa ordem de serviço (`Atendimento`) terá um cliente associado a ela.

Como novos recursos, apresentarei o uso dos controles para data (`DatePicker`), hora (`TimePicker`) e uma relação de opções que podem ser disponibilizadas ao usuário conhecida como `Action Sheet`. Veremos também como habilitar ou desabilitar, exibir ou ocultar controles de entrada de dados, com base em propriedades ligadas. O capítulo se concluirá com a apresentação de `converters`, que possibilita a customização da renderização de controles, por meio de conversões nos valores em objetos ligados a propriedades.

Reforço que toda implementação realizada neste capítulo tem como base o capítulo anterior e servirá como base para o próximo capítulo, desta maneira, siga as implementações diretamente na solução do capítulo anterior.

## 6.1 Classe de modelo e de acesso a dados

Para o registro da entrada de um veículo na oficina, faremos uso de uma classe específica para isso, que será nosso modelo de negócio, chamada `Atendimento`, mas, antes dela, precisamos implementar nossa classe relacionada ao ID. Desta maneira, nos projetos específicos para isso, crie a classe apresentada na sequência. Observe que, além da propriedade identificadora, estamos implementando identidades relacionadas à associação que estabeleceremos, pois todo atendimento será registrado para um cliente.

```
namespace CasaDoCodigo.Models
{
    public class AtendimentoIDProperty
    {
        public long? AtendimentoID { get; set; }
        public long? ClienteID { get; set; }

        [NotMapped]
        public virtual bool EstaFinalizado => false;

        [NotMapped]
        public bool NotificarListView { get; set; }
    }
}
```

Na sequência, em nosso projeto `oficinaModels`, precisamos implementar nossa classe `Atendimento` que estenderá a classe específica para os identificadores. Veja-a no código a seguir. Adicione-a a uma pasta nova, chamada `Atendimentos`, que você deverá criar. Observe que temos uma propriedade para um objeto `cliente`. O EF populará este objeto sempre que a estratégia de carga informar essa necessidade.

Atente para a propriedade `DataHoraEntrega` que está definida como anulável e que o construtor define o valor das propriedades `DataHoraChegada` e `DataHoraPrometida` com a data e hora obtida no momento da instanciação da classe. A `DataHoraEngrega` é inicializada como `null` e será o flag que indicará a finalização do processo de atendimento. Para facilitar, ao final do código implementaremos um método de predicado para indicar quando o atendimento está finalizado.

```
namespace CasaDoCodigo.Models
```

```

{
    public class Atendimento : AtendimentoIDProperty
    {
        public string Veiculo { get; set; }
        public DateTime DataHoraChegada { get; set; }
        public DateTime DataHoraPrometida { get; set; }
        public DateTime? DataHoraEntrega { get; set; }

        public Cliente Cliente { get; set; }

        public Atendimento()
        {
            this.DataHoraChegada = DateTime.Now;
            this.DataHoraPrometida = DateTime.Now;
            this.DataHoraEntrega = null;
        }

        public override bool EstaFinalizado => DataHoraEntrega != null;

        public override bool Equals(object obj)
        {
            return AtendimentoID.Equals((obj as Atendimento).AtendimentoID);
        }

        public override int GetHashCode()
        {
            var hashCode = -1711974840;
            hashCode = hashCode * -1521134297 + EqualityComparer<string>.Default.GetHashCode(AtendimentoID.ToString());
            return hashCode;
        }
    }
}

```

Nossa próxima implementação para registrar os atendimentos deverá ser realizada em nossa classe de contexto com a base de dados. Desta maneira, no construtor da classe `DatabaseContext` insira a propriedade `public DbSet<Atendimento> Atendimentos { get; set; }.`

Na sequência, precisamos implementar a classe de acesso a dados para atendimentos, entretanto, antes disso, precisamos discutir algumas particularidades para este novo conjunto de dados. Vamos começar pela recuperação com o método `GetAllAsync()`. Se verificarmos a implementação deste método na classe `DALBase`, a classificação está sendo feita pelo método `orderBy()`, que a realiza por ordem crescente (ascendente), o que foi perfeito para nossos dados de clientes e serviços, pois usamos a propriedade `Nome`. Agora, para nossos dados de atendimento, gostaríamos que a classificação fosse pela data e hora de chegada do veículo na oficina, mas em ordem decrescente (descendente), ou seja, começando pelo atendimento mais recente. Isso nos leva a uma nova verificação em nosso método `GetAllAsync()` da classe `DALBase`. E, se vamos fazer uma verificação, o método precisa receber um valor. Poderíamos pensar em mandar uma nova string para o método, mas strings podem ter caracteres digitados de maneira errada. Vamos então trabalhar com enumeradores. Para isso, no projeto `Interfaces`, na interface `IDAL`, dentro da pasta `DataAccess`, antes da definição da interface, insira a declaração do enumerador mostrado na sequência.

```
public enum OrderByType { NaoClassificado = 0, Ascendente, Descendente };
```

Agora, sim, precisamos alterar nosso método `GetAllAsync()` e começaremos pela interface, na mesma classe que

inserimos o código anterior. Adapte a assinatura do método para ser igual ao apresentado na sequência. Veja que o padrão é a ordem crescente.

```
Task<List<T>> GetAllAsync( string campoClassificacao = null , OrderByType orderByType = OrderByType.NaoClassificado);
```

Precisamos alterar a implementação deste método em nossa classe `DALBase`, para que, caso seja necessário, o resultado possa ser classificado em qualquer uma das duas maneiras possíveis. Veja na sequência o novo trecho de código para a classificação. Lembre-se de alterar também o método para a nova assinatura, igual à declaração anterior.

```
if (orderByType == OrderByType.Descendente)
    query = query.OrderByDescending(sortExpression);
else
    query = query.OrderBy(sortExpression);
```

Ainda em relação à expressão de ordenação, como para atendimentos ela será realizada por uma propriedade `DateTime`, precisamos alterar a maneira como esta expressão é criada, pois, como está, ocorrerá uma exceção `Expression of type "System.DateTime" cannot be used for return type "System.Object"`. Precisamos realizar uma conversão para a propriedade. Veja o código na sequência, que você deverá substituir também. Esta alteração manterá o funcionamento para as classes já implementadas.

```
var autoBoxing = Expression.Convert(Expression.Property(parameter, campoClassificacao), typeof ( object ));
var sortExpression = Expression.Lambda<Func<T, object >>(autoBoxing, parameter);
```

Temos uma situação especial também para a classe de atendimentos, que é a associação com clientes. O EF Core tem sua carga de objetos conhecida como `Fetch`, que define como os objetos associados são recuperados. No caso de termos um objeto armazenado para atendimento, que esteja associado a um cliente, se utilizarmos nosso método `GetAllAsync()` como está até o momento, apenas a propriedade `ClienteID` terá o valor recuperado com o atendimento. Seria necessária uma nova consulta para recuperar o cliente. Este `Fetch` é conhecido como `Lazy` (tardio) e é melhor para as performances. Entretanto, na visão de listagem que implementaremos, queremos exibir o nome do cliente para cada atendimento.

Precisamos de um `join` entre as duas tabelas da base de dados. A realização desse procedimento no EF Core é chamada de `Eager Fetch` (carga imediata/forçada) e podemos realizar isso adicionando um método, chamado `Include()`, às instruções de recuperação em nosso método `GetAllAsync()`. Mas esta situação não ocorre para toda vez que o método for invocado, pois ela é necessária apenas quando houver associações. Nós vamos implementar esta funcionalidade nas classes especializadas do DAL que necessitem dela, como `NOSSO AtendimentoDAL`. Mas antes, vamos separar a preparação dos dados que está no `GetAllAsync()` para outro método, protegido, que poderá ser invocado nas subclasses. Veja este método na sequência, seguido pela nova implementação para o `GetAllAsync()`.

```
protected IQueryable<T> PrepareDataToGetAll (DbContext context, string campoClassificacao, OrderByType orderByType)
{
    var query = context.Set<T>().AsNoTracking();

    if (! string .IsNullOrEmpty(campoClassificacao))
    {
        var parameter = Expression.Parameter( typeof ( T));
        var autoBoxing = Expression.Convert(Expression.Property(parameter, campoClassificacao), typeof ( object ));
        var sortExpression = Expression.Lambda<Func<T, object >>(autoBoxing, parameter);
    }
}
```

```

        if (orderByType == OrderByType.Descendente)
            query = query.OrderByDescending(sortExpression);

        else
            query = query.OrderBy(sortExpression);
    }

    return query;
}

public async virtual Task<List<T>> GetAllAsync( string campoClassificacao = null , OrderByType orderByType =
OrderByType.NaoClassificado)
{
    using ( var context = DatabaseContext.GetContext( this .dbPath))
    {

        var query = PrepareDataToGetAll(context, campoClassificacao, orderByType);

        return await query.ToListAsync();
    }
}

```

Muito bem, vamos agora criar nossa classe `AtendimentoDAL`, dentro da pasta `DAL`, do projeto `SQLiteEF`. Nela, implemente o código apresentado na sequência. Observe que definimos a propriedade para classificação, o seu tipo e o `Include` que deverá ser utilizado. Precisaremos adaptar as classes `ClienteDAL` e `ServicoDAL` para a atribuição de `orderByType`, tal qual estamos fazendo no código que se segue.

```

public async override Task<List<Atendimento>> GetAllAsync( string campoClassificacao = null , OrderByType orderByType =
OrderByType.NaoClassificado)
{

    campoClassificacao = string .IsNullOrEmpty(campoClassificacao) ? nameof(Atendimento.DataHoraChegada) :
campoClassificacao;

    orderByType = orderByType == OrderByType.NaoClassificado ? OrderByType.Descendente : orderByType;

    using ( var context = DatabaseContext.GetContext(dbPath))
    {

        var query = PrepareDataToGetAll(context, campoClassificacao, orderByType);
        query = query.Include(a => a.Cliente);

        return await query.ToListAsync();
    }
}

```

Precisamos agora declarar a propriedade `Atendimentos` no `DatabaseContext` no projeto `SQLiteEF` e executar o procedimento já conhecido para o Migrations, que realizamos no final do capítulo anterior.

## 6.2 A listagem de atendimentos

Acabamos de implementar as instruções necessárias para que, na próxima execução da aplicação, a nova tabela seja criada em nossa base de dados. Ou seja, não haverá dados para serem exibidos. Entretanto, estamos utilizando o

padrão de apresentar a listagem de itens armazenados e, a partir dela, chegar ao processo de inserção e alteração. Desta maneira, na pasta `views`, do projeto Xamarin Forms, crie uma nova pasta, chamada `Atendimentos` e, dentro dela uma Content Page XAML chamada `ListagemView.xaml`. A implementação inicial para esta visão, já com o controle `ListView`, é apresentada na sequência.

Observe, no `ListView`, a definição da propriedade `ItemsSource`, que define a fonte de dados para popular o controle. Verifique a definição do `ItemTemplate` e `DataTemplate` e, dentro deles um `StackLayout` com dois controles `Label`, que estão ligados (`Binding`) com propriedades de cada objeto recuperado pelo `ItemsSource`. Uma diferença que temos, em relação ao que já víhamos trabalhando, é no `Binding` de `Cliente.Nome`, que é a propriedade de uma propriedade. Por isso implementamos os `Includes` na seção anterior.

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class = "Capitulo05.Views.Atendimentos.ListagemView"
    xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    ios:Page.UseSafeArea = "true"
    Title = "Atendimentos" >

    <ContentPage.Content>

        <StackLayout Padding = "10, 0, 0, 0" VerticalOptions = "FillAndExpand" >

            <ListView x:Name = "listView" HasUnevenRows = "True" ItemsSource = "{Binding Atendimentos}" >
                <ListView.ItemTemplate>
                    <DataTemplate>
                        <ViewCell>
                            <StackLayout Padding = "10" >
                                <Label Text = "{Binding Veiculo}" FontSize = "18" FontAttributes = "Bold" />
                                <Label Text = "{Binding Cliente.Nome}" FontSize = "14" />
                            </StackLayout>
                        </ViewCell>
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

```

        </DataTemplate>

    </ListView.ItemTemplate>

</ListView>

</StackLayout>

</ContentPage.Content>

</ContentPage>

```

Resgatando o conceito do MVVM, precisamos criar uma classe que será a ViewModel para esta visão. Pois é ela que fornecerá os dados para a propriedade `ItemSource` do `ListView`. Para isso, na pasta `ViewModels` crie uma nova pasta, chamada `Atendimentos` e nela adicione uma classe chamada `ListagemViewModel` com o código apresentado na sequência. Veja a criação do método `AtualizarAtendimentosAsync()`, com a mesma finalidade dos métodos de atualização que criamos no capítulo anterior.

```

namespace Capitulo05.ViewModels.Atendimentos
{
    public class ListagemViewModel
    {
        private IDAL<Atendimento> atendimentoDAL;
        public ObservableCollection<Atendimento> Atendimentos { get; set; }

        public ListagemViewModel()
        {
            atendimentoDAL = new AtendimentoDAL(DependencyService.Get<IDBPath>().GetDbPath());
            Atendimentos = new ObservableCollection<Atendimento>();
        }

        public async Task AtualizarAtendimentosAsync()
        {
            var atendimentos = await atendimentoDAL.GetAllAsync();
            Atendimentos.SincronizarColecoes(atendimentos);
        }
    }
}

```

Agora precisamos informar na classe de code-behind da visão `ListagemView` que nossa `ListagemViewModel` será a fonte de ligação de dados. Para isso, implemente o seguinte código.

```

namespace Capitulo05.Views.Atendimentos
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class ListagemView : ContentPage
    {
        private ListagemViewModel viewModel;

        public ListagemView ()
        {
            InitializeComponent ();
        }
    }
}

```

```

        viewModel = new ListagemViewModel();
        BindingContext = viewModel;
    }
}
}
}

```

Precisamos implementar a invocação de nosso método de atualização de atendimentos, que está na ViewModel e faremos isso no `OnAppearing()` da visão. Veja o código a seguir e verifique que é semelhante ao que já utilizamos no capítulo anterior.

```

protected override void OnAppearing ()
{
    base.OnAppearing();

    Device.BeginInvokeOnMainThread( async () =>
    {
        await viewModel.AtualizarAtendimentosAsync();
    });

    if (listView.SelectedItem != null )
        listView.SelectedItem = null ;
}

```

Para finalizarmos e podermos testar nossa aplicação para ver se a visão de listagem é renderizada sem erros, precisamos dar ao usuário essa opção na visão que apresenta o menu da aplicação a ele. Lembra qual é? Isso mesmo, a `MasterPageView`, que está na pasta `Views`. Precisamos inserir uma nova opção no construtor, que é a `new Models.MenuItem { Id = 2, Title = "Atendimentos", TargetType = typeof(Atendimentos.ListagemView), IconSource="tab_atendimento.png"}`. Lembre-se de se certificar que a imagem que está utilizando está nos projetos específicos das plataformas. Execute agora sua aplicação. As figuras a seguir apresentam a visão do menu atualizado e a de listagem de atendimentos, ainda vazia.

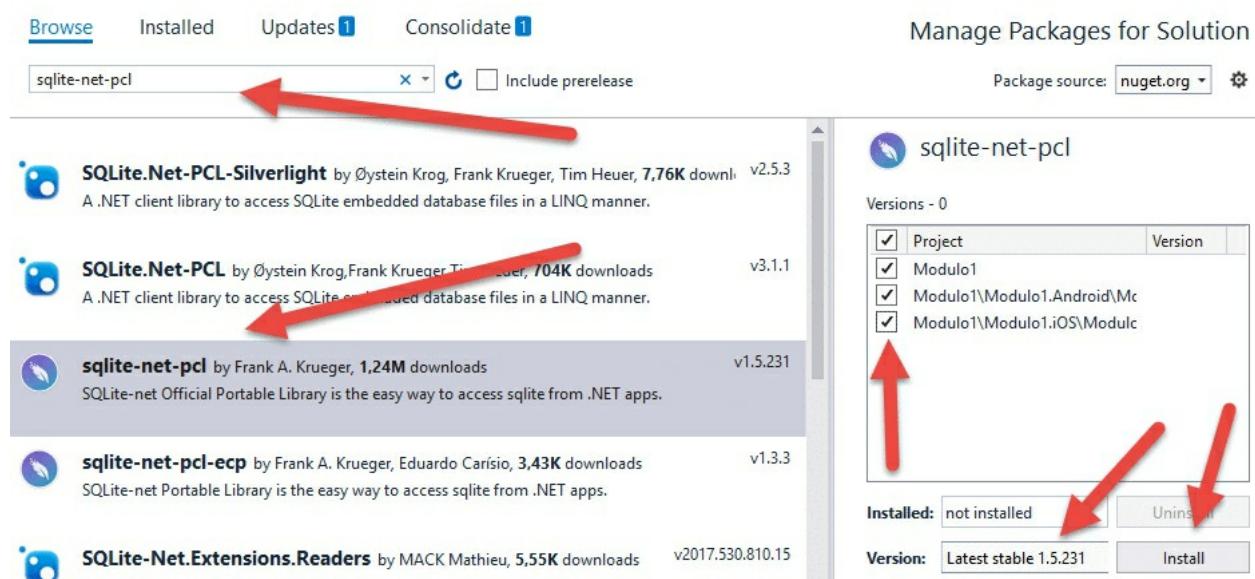


Figura 6.1: Visão principal com menu de opções

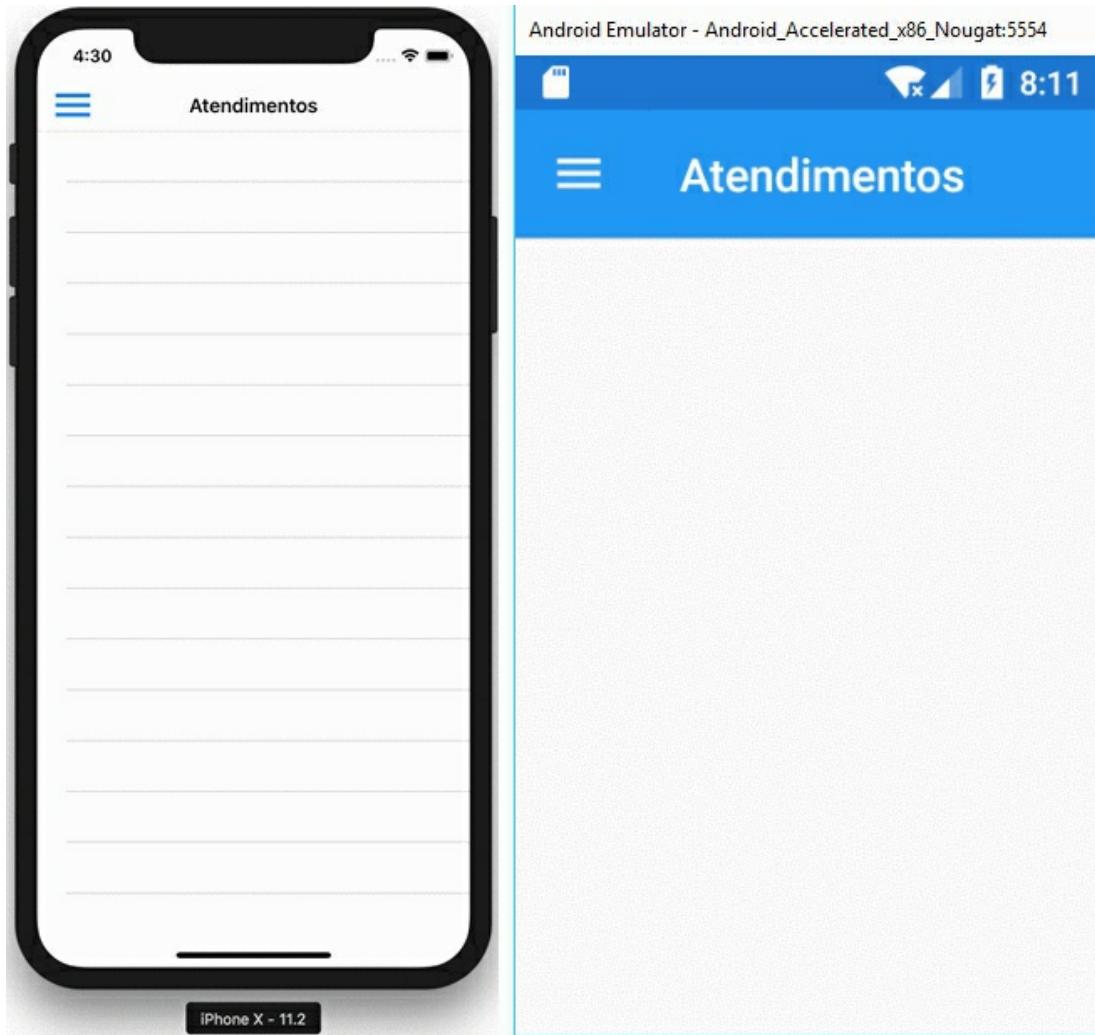


Figura 6.2: Visão de listagem de atendimentos

### 6.3 Pesquisa por clientes para atendimento

Muito bem, precisamos agora alterar a visão de listagem para que ofereça ao usuário a possibilidade de inserir novos atendimentos. Faremos isso exibindo uma barra de tarefas na visão. Abra sua visão `ListagemView` e, antes do `<ContentPage.Content>`, insira as instruções apresentadas a seguir. Observe o uso de uma imagem, que você deverá ter em seu projeto, e a definição do `Command NovoCommand` para que possamos implementar o código necessário para a exibição da visão de inserção de um novo atendimento.

```
<ContentPage.ToolbarItems>
```

```
    <ToolbarItem Icon = "plus.png" Command = "{Binding NovoCommand}" />
</ContentPage.ToolbarItems>
```

Agora, antes de apresentar a implementação para `NovoCommand`, precisamos implementar a visão que o usuário verá quando quiser registrar um novo atendimento. Desta maneira, na pasta `Views/Atendimentos` adicione um novo content Page chamado `CRUDView.xaml`. A princípio não codificaremos nada nesta visão. Desta maneira, deixe-a como apresentado na sequência.

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class = "Capítulo05.Views.Atendimentos.CRUDView"
    xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    ios:Page.UseSafeArea = "true" >

    <ContentPage.Content>
    </ContentPage.Content>
</ContentPage>
```

No code-behind de nossa visão, precisamos implementar um construtor parametrizado, que receberá a instância do objeto que será manipulado por ela e seu título. Veja isso no código a seguir. Note que mantemos o construtor padrão também.

```
public CRUDView ()
{
    InitializeComponent();
}

public CRUDView (Atendimento atendimento, string title) : this ()
{
    this.Title = title;
}
```

Com a visão para inserção criada, podemos agora implementar o comportamento para o Command `NovoCommand` visto anteriormente. Para isso, na classe `ListagemViewModel` de atendimentos, implemente o código a seguir. Note que a criação do `Command` está dentro de um método privado. Sendo assim, no construtor da classe você deve invocar o método `RegistrarCommands()`. Implemente isso em seu construtor logo abaixo da instanciação de `Atendimentos`. Você precisará também adicionar o `using` para `System.Windows.Input`.

```
public ICommand NovoCommand { get ; set ; }

private void RegistrarCommands ()
```

```
NovoCommand = new Command(() =>
{
    MessagingCenter.Send<Atendimento>( new Atendimento(), "Mostrar" );
});
```

Veja no código a seguir as implementações para manipulação da mensagem enviada na listagem anterior.

```
// OnAppearing()

MessagingCenter.Subscribe<Atendimento>( this , "Mostrar" , async (atendimento) => { await Navigation.PushAsync( new
CRUDView(atendimento, "Novo Atendimento" )); });

// OnDisappearing()

MessagingCenter.Unsubscribe<Atendimento>( this , "Mostrar" );
```

Vamos agora executar nossa aplicação. Nas figuras a seguir, estão a nova visão para a listagem de atendimentos e a janela de inserção de um atendimento, ainda sem nenhum controle.

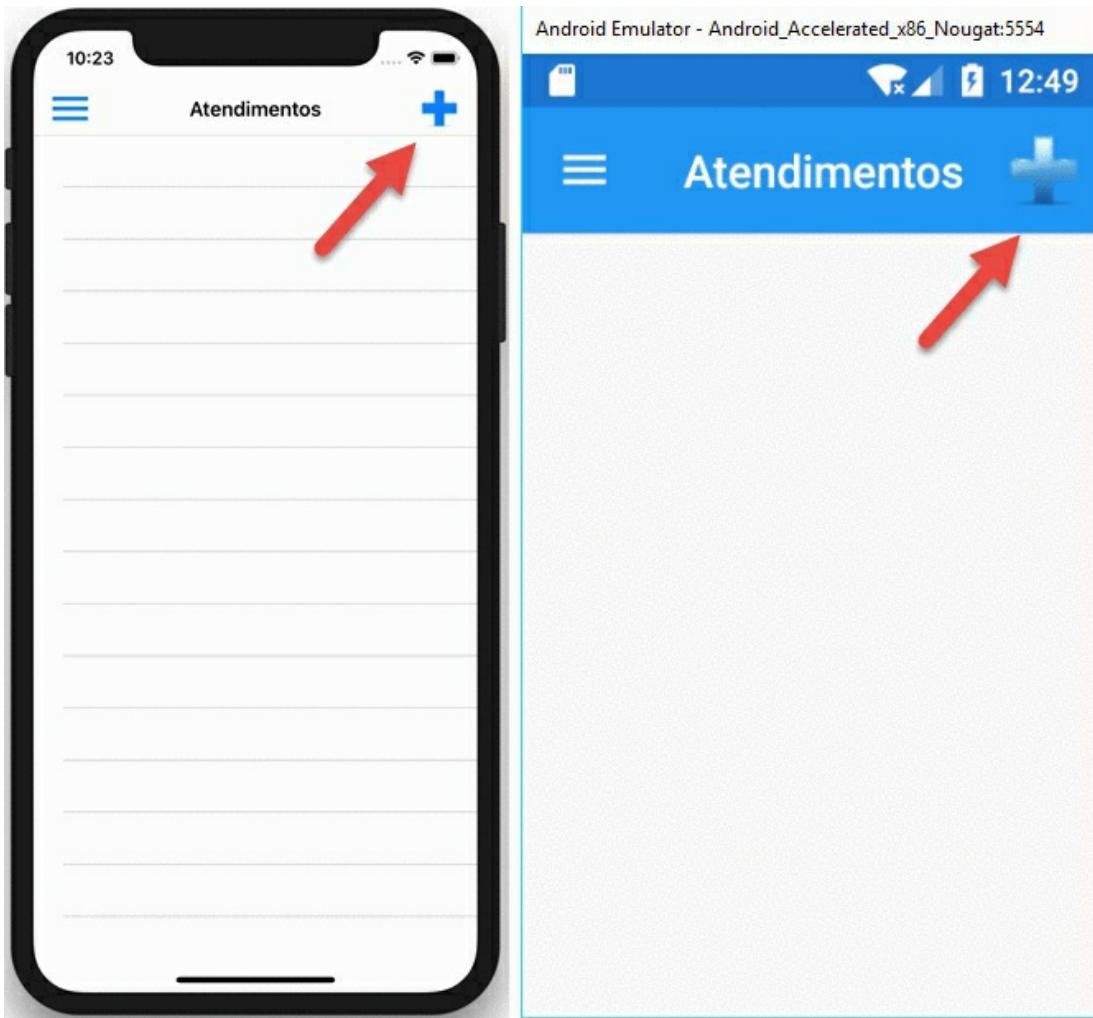


Figura 6.3: Exibição da barra de tarefas com opção de inserir novo atendimento

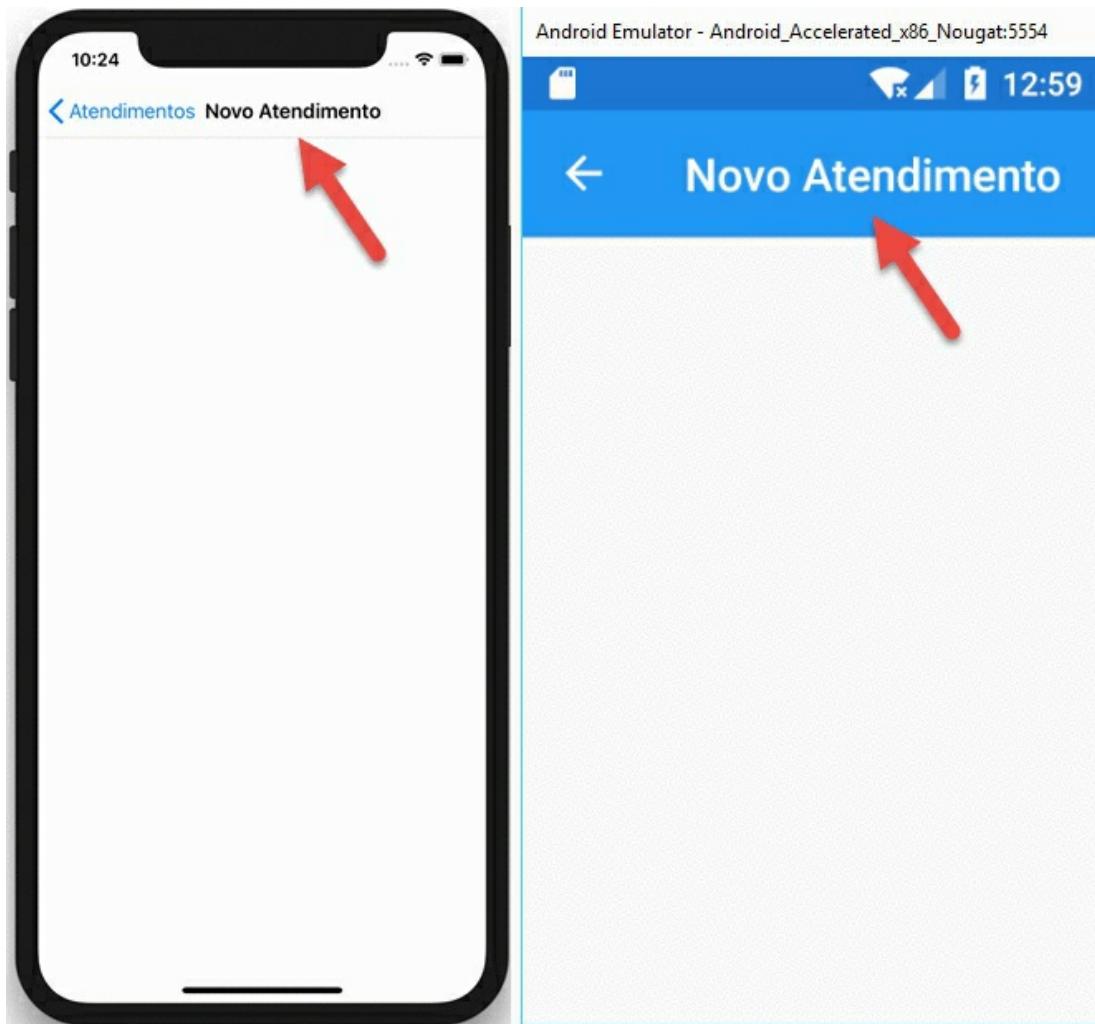


Figura 6.4: Visão para inserir novo atendimento

Precisamos agora criar a classe que representará a ViewModel para a visão anterior. Desta maneira, na pasta `ViewModels/Atendimentos`, crie uma classe chamada `CRUDViewModel` e implemente nela, a princípio, o seguinte código. Note e relembrre a extensão para `BaseViewModel`.

```
namespace Capitulo05.ViewModels.Atendimentos
{
    public class CRUDViewModel : BaseViewModel
    {
        private IDAL<Atendimento> atendimentoDAL;
        private Atendimento Atendimento { get; set; }

        public CRUDViewModel(Atendimento atendimento)
        {
            atendimentoDAL = new AtendimentoDAL(DependencyService.Get<IDBPath>().GetDbPath());
            this.Atendimento = atendimento;
        }
    }
}
```

```
 }
}
```

Com a classe ViewModel implementada, precisamos retornar ao code-behind da visão `CRUDView` e registrar esta classe como a de ligação para a visão. Logo, antes do construtor, declare um campo que representará a ViewModel, tal qual o código a seguir. Depois, no construtor que recebe dois argumentos, insira as declarações também apresentadas na sequência.

```
// Antes do construtor

private CRUDViewModel crudViewModel;

// No construtor parametrizado

this .crudViewModel = new CRUDViewModel(atendimento);

this .BindingContext = this .crudViewModel;
```

Pronto! Com isso já podemos começar a implementação da nossa camada de visão para a inserção de um novo atendimento. O primeiro conjunto de controles que implementaremos refere-se ao Cliente que está sendo atendido e a codificação para ele está no código a seguir, que deve ser inserido logo abaixo do `<contentPage.Content>`, no `CRUDView.xaml`. Veja que está sendo configurado um `<Grid>` com 3 colunas e 2 linhas. Na primeira linha, é inserido o texto `Cliente`, ocupando duas colunas, a partir da primeira. Na segunda linha, também ocupando duas linhas, está um `<Label>` ligado a uma propriedade chamada `clienteNome`, que logo apresento. Por fim, na terceira coluna, ocupando duas linhas, está uma imagem que você deverá ter em seus projetos de plataformas e que terá um `command` chamado `PesquisarCommand`, ligado ao reconhecimento de gestos que podem ocorrer nesta imagem, como o pressionar com o dedo. Já apresento este método também, mas verifique que não estamos obrigados a usar apenas botões para capturar gestos do usuário. Veremos gestos de maneira mais específica no capítulo 9. Por enquanto, dê uma lida no código.

```
<StackLayout>

<Grid HorizontalOptions = "Fill" Margin = "5, 5, 0, 0" Padding = "10, 10, 0, 0" >

    <Grid.ColumnDefinitions>

        <ColumnDefinition Width = "Auto" />

        <ColumnDefinition Width = "*" />

        <ColumnDefinition Width = "Auto" />

    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>

        <RowDefinition Height = "Auto" />

        <RowDefinition Height = "Auto" />

    </Grid.RowDefinitions>
```

```

<Label Text = "Cliente:" Grid.Column = "0" Grid.ColumnSpan = "2" Grid.Row = "0" FontAttributes = "Bold" />

<Label Text = "{Binding ClienteNome}" Grid.Column = "0" Grid.ColumnSpan = "2" Grid.Row = "1" TextColor = "Blue" />

<Image Source = "consultar.png" HeightRequest = "38" WidthRequest = "38" Grid.Column = "2" Grid.Row = "0" Grid.RowSpan = "2" >

<Image.GestureRecognizers>

<TapGestureRecognizer Command = "{Binding PesquisarCommand}" />

</Image.GestureRecognizers>

</Image>

</Grid>

</StackLayout>

```

Na classe `CRUDViewModel`, de `Atendimentos`, vamos implementar a propriedade `clienteNome`, que utilizamos no código anterior. Veja que utilizamos um operador ternário para o retorno de valor da propriedade, que é apenas de leitura.

```

public string ClienteNome
{
    get { return this .Atendimento.Cliente == null ? "Localize o cliente" : this .Atendimento.Cliente.Nome; }
}

```

Execute sua aplicação e acesse a visão de inserção de um novo atendimento. Você verá uma tela semelhante à apresentada na figura a seguir.

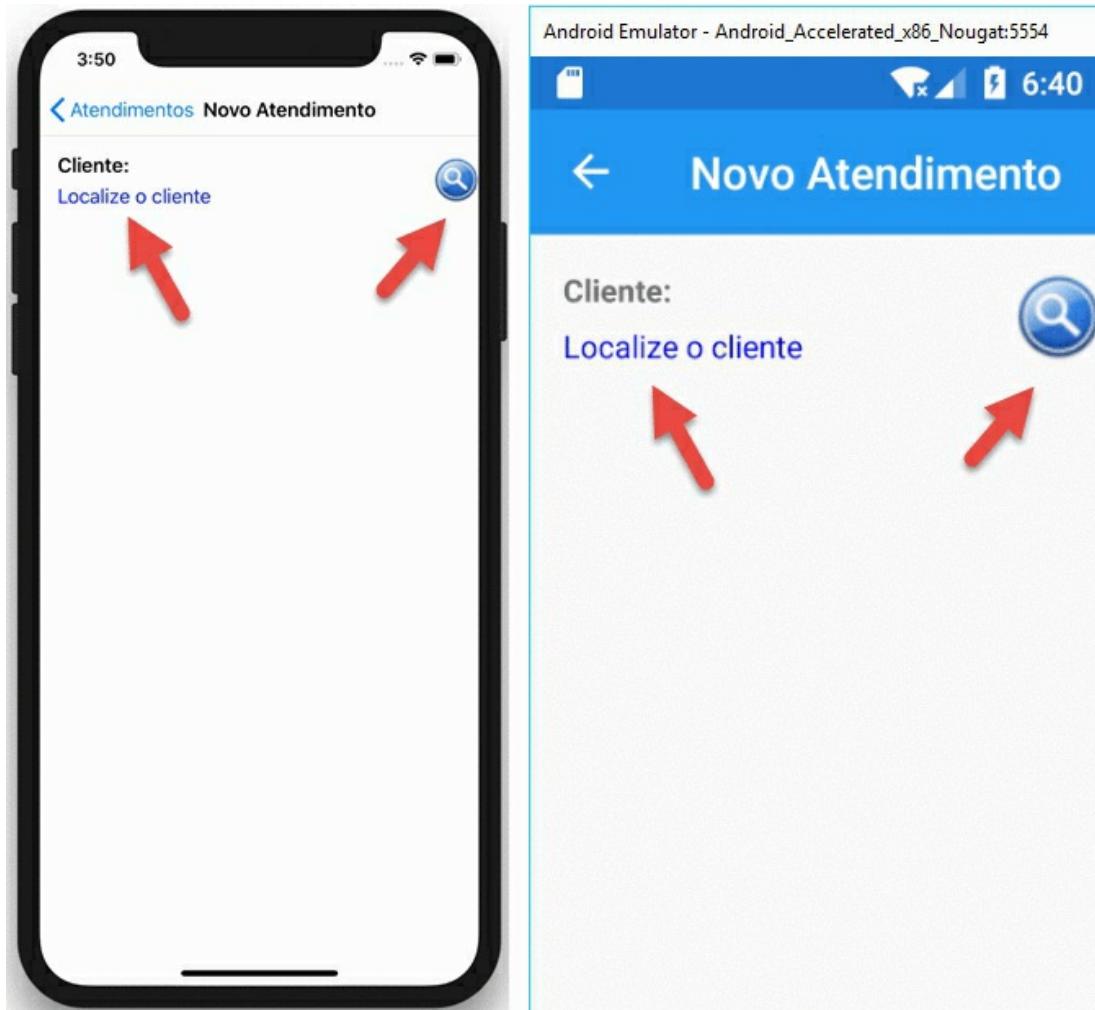


Figura 6.5: Controles para seleção de cliente

No capítulo anterior implementamos todo o CRUD para clientes. Desta maneira, já temos dados persistidos e que podem ser utilizados para registrar o atendimento. A metodologia que adotaremos para o usuário informar o cliente é a de realizar uma pesquisa na base de dados e então selecionar um cliente já cadastrado. Para isso, faremos uso de um novo controle, o `<SearchBar>`, que se refere a uma caixa de texto utilizada como padrão para realização de pesquisas.

Para o uso do `<SearchBar>` criaremos uma nova visão na pasta `Views/Cientes`, chamada `PesquisarView` e que, inicialmente, terá o código apresentado na sequência. Veja, na declaração do controle, as ligações (bindings) para `SearchCommand` e `SearchCommandParameter`, que definem os comportamentos para a pesquisa quando o botão de pesquisar ou a tecla `ENTER` for pressionada. Veja também o parâmetro que será enviado para o comando `PesquisarCommand`. Já vimos este tipo de ligação entre controles antes. Um detalhe especial precisa ser dado à propriedade `HeightRequest` do `<SearchBar>`. Sem essa definição explícita, o controle não é renderizado em um dispositivo físico Android. Talvez, quando você estiver lendo este livro, este bug já tenha sido corrigido. Se o valor de `25` for pouco, para testar em seu dispositivo, fique à vontade para aumentá-lo.

Na declaração do `ListView` temos os bindings para a fonte de dados e para o comportamento de quando um item da listagem for selecionado. Logo apresentarei todos estes códigos. Observe, no `<Label>` para o a propriedade `clienteID`, que ela não será visível ao usuário.

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

```

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class = "Capitulo05.Views.Clientes.PesquisarView"
    xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    ios:Page.UseSafeArea = "true"
    Title = "Pesquisa por cliente" >

    <ContentPage.Content>

        <StackLayout Orientation = "Vertical" HorizontalOptions = "FillAndExpand" Padding = "5,20,5,0" >

            <SearchBar x:Name = "searchBar" HeightRequest = "25" Placeholder = "Digite o nome do cliente..." TextColor = "Black" SearchCommand = "{Binding PesquisarCommand}" SearchCommandParameter = "{Binding Source={x:Reference searchBar}, Path=Text}" />

            <ListView x:Name = "lvClientes" HasUnevenRows = "True" ItemsSource = "{Binding ClientesEncontrados}" SelectedItem = "{Binding ClienteLocalizadoSelecionado}" Margin = "10, 5, 0, 0" >

                <ListView.ItemTemplate>

                    <DataTemplate>

                        <ViewCell>

                            <StackLayout Orientation = "Vertical" >

                                <Label Text = "{Binding ClienteID}" TextColor = "Blue" IsVisible = "False" />

                                <Label Text = "{Binding Nome}" TextColor = "Blue" FontSize = "Large" />

                            </StackLayout>

                        </ViewCell>

                    </DataTemplate>

                </ListView.ItemTemplate>

            </ListView>
        </ContentPage.Content>
    </ContentPage>

```

```

</StackLayout>

</ContentPage.Content>

</ContentPage>

```

Para que possamos realizar a pesquisa na base de dados com base em um valor que será digitado pelo usuário, precisamos pensar em um método em nosso DAL que possa realizar a pesquisa. Veja o método da sequência, que deve ser implementado em nossa classe `DALBase`. Estamos fazendo uso de reflexão no código apresentado para criar expressões dinâmicas com o uso do LINQ. O objetivo do método é montar uma expressão lambda, com base no tipo de dados da classe instanciada, com a propriedade e valores que serão recebidos pelo método. O assunto não é difícil, mas demandaria explicações que estão fora do escopo do livro. Eu quis trazer esta implementação para aguçar sua curiosidade e para que você possa ver as possibilidades oferecidas pelo C# e o LINQ. Já usamos um pouco deste tipo de implementação no `GetAllAsync()`.

```

public async Task<IEnumerable<T>> GetStartsWithByFieldAsync( string field, string value )
{
    using ( var context = DatabaseContext.GetContext(dbPath) )

    {
        ParameterExpression parameterExpression = Expression.Parameter( typeof(T), "t" );
        MemberExpression memberExpression = Expression.Property(parameterExpression, field);

        ConstantExpression constantExpression = Expression.Constant( value , typeof(string) );

        MethodInfo methodInfo = typeof(string).GetMethod( "StartsWith" , new Type[] { typeof(string) } );
        Expression call = Expression.Call(memberExpression, methodInfo, constantExpression);

        Expression<Func<T, bool >> lambda = Expression.Lambda<Func<T, bool >>(call, parameterExpression);

        return await context.Set<T>().Where(lambda).ToArrayAsync();
    }
}

```

O método anterior receberá dois argumentos, sendo o primeiro responsável pela propriedade da classe que será utilizada na pesquisa à base de dados, e o segundo contendo o valor que será pesquisado. O método está limitado, neste caso, a valores literais (`string`). Como a consulta a dados utiliza, no LINQ, expressões lambdas como padrão e estamos implementando um método genérico, precisamos fazer uso de um recurso da linguagem chamado `Reflexão` (`Reflection`).

Reflexão refere-se ao recurso de podermos obter informações de uma classe, de um tipo de dado ou método (inclusive executá-lo), a partir de dados fornecidos, como, em nosso caso, o nome da propriedade. Desta maneira, a variável `parameterExpression` terá o valor `t`, sabendo que se refere ao tipo de dado `T`, que virá definido na instanciação do DAL; a `memberExpression` conterá o valor `t.<nomeDaPropriedade>`; a `constantExpression`, o valor recebido pela variável `value`, entre aspas; `methodInfo` receberá a forma como o método `startswith` é invocado; `call` comporá o método a ser invocado e, como exemplo, digitando a letra "A", o valor desta variável será `t.Nome.StartsWith("A")`; e o retorno define a expressão que deverá ser executada na pesquisa, que também será, em nosso exemplo, `t.Nome.StartsWith("A")`.

Já temos o serviço para recuperação de dados implementado e, como já de costume, seguindo o padrão MVVM, criaremos agora a classe que representará o `ViewModel`. Desta maneira, na pasta `ViewModels/Clientes` crie uma classe chamada `PesquisarViewModel` e nela implemente o código apresentado na sequência. Optei aqui por apresentar todo o código para a classe, pois são poucas funcionalidades. Dê uma lida nele e verifique que: 1) definimos um DAL para obter os clientes que serão pesquisados, uma propriedade que armazenará estes clientes e, se você voltar na definição da `ListView` na visão, verá que é esta propriedade que está ligada à propriedade `ItemsSource`. Concluímos a

primeira etapa declarando o `Command PesquisarCommand`, que está ligado com o `<SearchBar>`; 2) a segunda parte do código define o construtor para a classe, onde o DAL e a propriedade `ClientesEncontrados` são instanciados e o método `RegistrarCommands()` é invocado. Observe, no `Command`, a chamada ao método `GetStartsWithByFieldAsync()`, que acabamos de implementar; 3) na sequência, o método `RegistrarCommands()` é codificado, pois o invocamos no construtor; e 4) a última parte implementada na classe refere-se à propriedade `ClienteLocalizadoSelecionado`, que está ligada ao `Listview` pela propriedade `SelectedItem`. Nesta propriedade, é enviada uma mensagem para O `MessagingCenter` quando o usuário selecionar um cliente da listagem retornada.

```
namespace Capitulo05.ViewModels.Clientes
{
    public class PesquisarViewModel
    {
        private IDAL<Cliente> clienteDAL;
        public ObservableCollection<Cliente> ClientesEncontrados { get; set; }
        public ICommand PesquisarCommand { get; set; }

        public PesquisarViewModel()
        {
            clienteDAL = new ClienteDAL(DependencyService.Get<IDBPath>().GetDbPath());
            ClientesEncontrados = new ObservableCollection<Cliente>();
            RegistrarCommands();
        }

        private void RegistrarCommands()
        {
            PesquisarCommand = new Command<string>(async (cliente) =>
            {
                ClientesEncontrados.Clear();
                var clientesEncontrados = await clienteDAL.GetStartsWithByFieldAsync(nameof(Cliente.Nome), cliente);
                foreach (var c in clientesEncontrados)
                {
                    ClientesEncontrados.Add(c);
                }
            });
        }

        private Cliente clienteLocalizadoSelecionado;
        public Cliente ClienteLocalizadoSelecionado
        {
            get { return clienteLocalizadoSelecionado; }
            set
            {
                if (value != null)
                {
                    clienteLocalizadoSelecionado = value;
                    MessagingCenter.Send<Cliente>(clienteLocalizadoSelecionado, "ClienteSelecionado");
                }
            }
        }
    }
}
```

```
}
```

Vamos nos preparar para testar a execução de nossa aplicação, ver se a visão de pesquisa é exibida de maneira correta e se a pesquisa realmente ocorre. Desta maneira, na classe `CRUDViewModel` de atendimentos, precisamos implementar o `Command PesquisarCommand` que será acionado quando o usuário pressionar no botão específico para isso e que já temos implementado. Veja o código na sequência. Observe a declaração do método `RegistrarCommands()`, que você deverá invocar no construtor da classe.

```
public ICommand PesquisarCommand { get; set; }

private void RegistrarCommands()
{
    PesquisarCommand = new Command(() =>
    {
        MessagingCenter.Send<Atendimento>(Atendimento, "MostrarPesquisarCliente");
    });
}
```

Precisamos implementar a sobrescrita dos métodos para registro e cancelamento de registros de mensagem na classe `CRUDView` para atendimentos. Veja este código:

```
/// OnAppearing()

MessagingCenter.Subscribe<Atendimento>( this, "MostrarPesquisarCliente", async (atendimento) => { await
Navigation.PushAsync( new PesquisarView()); });

// OnDisappearing()

MessagingCenter.Unsubscribe<Atendimento>( this, "MostrarPesquisarCliente" );
```

Para que possamos enfim testar nossa aplicação, no code-behind da visão `PesquisarView`, precisamos definir a `PesquisarViewModel` como o objeto utilizado para a ligação de nossos controles visuais. Veja o código a seguir. Antes do construtor definimos a `ViewModel`, e no construtor a atribuímos para o `BindingContext`.

```
private PesquisarViewModel viewModel;

public PesquisarView()
{
    InitializeComponent();
    viewModel = new PesquisarViewModel();
    BindingContext = viewModel;
}
```

Agora sim. Teste sua aplicação, acesse a opção de atendimentos, na `Toolbar` pressione o botão de adicionar atendimentos, e na visão que aparece pressione o botão de pesquisar cliente. Você deverá ter uma visão semelhante à apresentada na figura a seguir. Digite alguma letra e pressione `ENTER` para a pesquisa ocorrer.

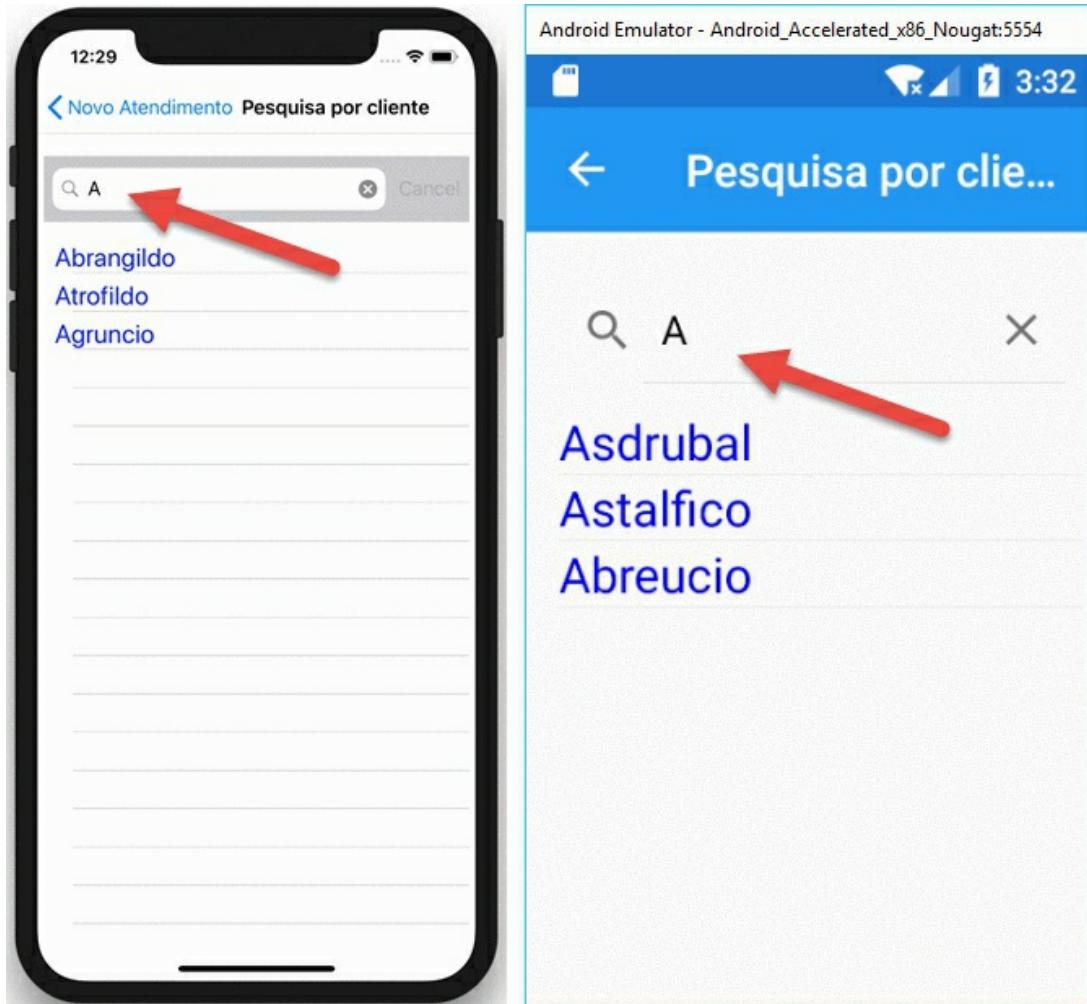


Figura 6.6: Pesquisa por clientes já gravados na base de dados

Complementando nossa implementação, precisamos codificar no code-behind da classe `PesquisarView` o registro e cancelamento para a mensagem `clienteSelecionado`. Entretanto, precisamos pensar que nossa classe de pesquisa precisará retornar o cliente que foi selecionado para a visão de CRUD do atendimento.

Sabemos que a mensagem não poderá retornar um valor quando ela for executada, pois estará dentro de um método sobrescrito, o `OnAppearing()`. Também sabemos que o binding de nossa visão CRUD está ligado à classe `CRUDViewModel` e ela tem um objeto privado para o atendimento atual. Podemos criar uma propriedade que receberá o cliente selecionado e o atribuirá ao atendimento. Veja este método na sequência. Observe que notificamos a visão para que atualize a propriedade `clienteNome`, pois um cliente foi selecionado.

```
public Cliente Cliente
{
    get { return this.Atendimento.Cliente; }

    set
    {
        this.Atendimento.Cliente = value;
        OnPropertyChanged(nameof(ClienteNome));
    }
}
```

```
    }  
}
```

Precisamos agora pensar em como vamos atualizar a propriedade `cliente`. Esta atualização precisará ser realizada na classe `PesquisarView`. Poderíamos pensar em injetar no construtor da visão a ViewModel do CRUD de atendimentos e então atualizar a propriedade `cliente`. Isso resolveria, mas nos traria outro problema. E se quiséssemos utilizar a pesquisa de clientes por outra ViewModel? Teríamos que injetá-la também? Vamos pensar que a visão `PesquisarView` terá a responsabilidade de fornecer o cliente selecionado na pesquisa. Desta maneira, a visão CRUD do atendimento buscaria esta propriedade na pesquisa sempre que estivesse ativa. Vamos criar uma propriedade estática em `PesquisarView`. Veja o código na sequência.

```
public static Cliente ClienteSelecionado { get ; set ; }
```

Precisamos, com isso, a cada instanciação da classe `PesquisarView`, atribuir nulo a essa propriedade, para que ela não fique com valores de pesquisas anteriores. Assim, no construtor dela, implemente, ao final, a instrução `ClienteSelecionado = null;`.

Vamos agora implementar nosso registro e cancelamento para a mensagem de seleção de um cliente, também na classe `PesquisarView`. Veja o código na sequência. Embora a *arrow function* possua mais de uma instrução, preferi deixar direto no registro da mensagem, mas você poderia criar um método para ela, o que acha?

```
protected override void OnAppearing ()  
{  
    base.OnAppearing();  
  
    MessagingCenter.Subscribe<Cliente>( this , "ClienteSelecionado" ,  
        (cliente) =>  
    {  
        ClienteSelecionado = cliente;  
        Navigation.PopAsync();  
    });  
}  
  
protected override void OnDisappearing ()  
{  
    base.OnDisappearing();  
    MessagingCenter.Unsubscribe<Cliente>( this , "ClienteSelecionado" );  
}
```

Agora, precisamos buscar por este valor e o faremos em nosso método sobrescrito `OnAppearing()`, na classe `CRUDView` de atendimentos, pois ele é invocado quando a janela de pesquisa se fecha e volta a exibir o CRUD. Veja o código na sequência, que verifica se existe valor atribuído. Implemente-o logo após a chamada ao método `base`.

```
if (PesquisarView.ClienteSelecionado != null )  
    crudViewModel.Cliente = PesquisarView.ClienteSelecionado;
```

Precisamos agora, quando a visão de CRUD perder o foco, atribuir `null` a `clienteSelecionado`, para que o valor atribuído a ele em uma operação anterior não seja exibido. Veja o código na sequência. Implemente-o no `OnDisappearing()` do `CRUDview`.

```
PesquisarView.ClienteSelecionado = null ;
```

Muito bem, agora vamos executar nossa aplicação e verificar se a pesquisa por clientes já está funcionando. Realize os passos que fizemos há pouco para visualizar a pesquisa, mas agora selecione o cliente retornado pelos critérios que você informar. Veja se ocorre o retorno para a visão CRUD de atendimentos e se o nome do cliente selecionado é exibido.

## 6.4 Veículo, datas e horas de entrada, previsão e entrega

Com nosso cliente selecionado, precisamos registrar seu veículo, a data e hora de entrada na oficina e previsão para entrega. A informação do veículo é simples e pode ser realizada com controles que já foram vistos no capítulo anterior. Logo após o fechamento do `<Grid>`, em `CRUDView.xaml` de `Atendimentos`, insira o código apresentado na sequência. Veja que criamos um `<TableView>` com um `<EntryCell>` ligado a uma propriedade chamada `Veiculo`, que está implementada após o XAML da sequência e que você deverá inserir na classe `CRUDViewModel` de `Atendimentos`.

```
<TableView Intent = "Form" >

<TableRoot>

<TableSection Title = "Dados do Atendimento" >

<EntryCell Label = "Veículo:" Text = "{Binding Veiculo}" ></EntryCell>

</TableSection>

</TableRoot>

</TableView>

public string Veiculo
{
    get { return this .Atendimento.Veiculo; }

    set
    {
        this .Atendimento.Veiculo = value ;
        OnPropertyChanged();
    }
}
```

Vamos agora partir para a implementação dos controles relacionados às datas. Para isso, utilizaremos dois novos controles, o `<DatePicker>` e o `<TimePicker>`, para data e hora respectivamente. A renderização da aplicação será com os recursos oferecidos por cada plataforma. Vamos ao XAML. Como estamos utilizando `TableView`, cada dado deve ser informado em uma única `<viewCell>`. Na listagem a seguir, para a data de chegada, você verá dois controles inseridos, graças ao uso de `<Grid>`. Veja, no `<DatePicker>`, que a data a ser exibida/armazenada está ligada a uma propriedade chamada `DataChegada`, que tem sua implementação após o XAML da sequência. Lembre-se de que ela deve ser implementada no `CRUDViewModel` de `Atendimentos`. É possível também notar a formatação que deve ser utilizada para exibir a data de chegada. Este código deve ir logo após o `<EntryCell>` de veículo. A implementação do `set` de `DataChegada` acusará um erro para a propriedade `Horachegada`, que já vamos implementar.

```
<viewCell>
```

```

<Grid HorizontalOptions = "Fill" VerticalOptions = "FillAndExpand" Margin = "5, 3, 0, 0" Padding = "10, 0, 0, 0" >

<Grid.ColumnDefinitions>

<ColumnDefinition Width = "Auto" />

<ColumnDefinition Width = "*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height = "Auto" />

</Grid.RowDefinitions>

<Label Text = "Chegada:" VerticalOptions = "CenterAndExpand" Grid.Column = "0" Grid.Row = "0" />

<DatePicker VerticalOptions = "FillAndExpand" HorizontalOptions = "StartAndExpand" Date = "{Binding DataChegada}" Grid.Column = "1" Grid.Row = "0" >

<DatePicker.Format> dd/MM/yyyy </DatePicker.Format>

</DatePicker>

</Grid>

</ViewCell>

public DateTime DataChegada
{
    get { return this .Atendimento.DataHoraChegada; }

    set
    {
        this .Atendimento.DataHoraChegada = new DateTime( value .Year, value .Month, value .Day, HoraChegada.Hours, HoraChegada.Minutes, 0 );
        OnPropertyChanged();
    }
}

```

Agora vamos para a implementação do controle referente à informação da hora de chegada. Vamos ao XAML. Implemente o código seguinte logo após o anterior, referente à data de chegada. Observe a formatação e o binding nas respectivas propriedades do XAML. No código da propriedade, que está também em seguida, é possível verificar que o tipo de dado para a hora é o `TimeSpan` e não o `DateTime`, como visto para a data. Veja como a atribuição da hora é realizada no `set()` da propriedade. Novamente, a propriedade deverá ser implementada no `CRUDViewModel` e resolverá o problema anterior, referente à propriedade `HoraChegada`.

```

<ViewCell>

<Grid HorizontalOptions = "Fill" VerticalOptions = "FillAndExpand" Margin = "5, 3, 0, 0" Padding = "10, 0, 0, 0" >

<Grid.ColumnDefinitions>

<ColumnDefinition Width = "120" />

<ColumnDefinition Width = "*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height = "Auto" />

</Grid.RowDefinitions>

<TimePicker VerticalOptions = "FillAndExpand" HorizontalOptions = "StartAndExpand" Time = "{Binding HoraChegada}" Grid.Column = "1" Grid.Row = "0" >

<TimePicker.Format> HH:mm </TimePicker.Format>

</TimePicker>

</Grid>

</ViewCell>

public TimeSpan HoraChegada
{
    get { return new TimeSpan( this .Atendimento.DataHoraChegada.Hour, this .Atendimento.DataHoraChegada.Minute, 0); }

    set
    {
        this .Atendimento.DataHoraChegada = new DateTime(DataChegada.Year, DataChegada.Month, DataChegada.Day, value .Hours
value .Minutes, 0);
        OnPropertyChanged();
    }
}

```

As figuras a seguir apresentam os controles que são renderizados pelas plataformas para a seleção de datas e horas. Você pode pensar em colocar as horas, na visão, na mesma linha das datas. O que acha?

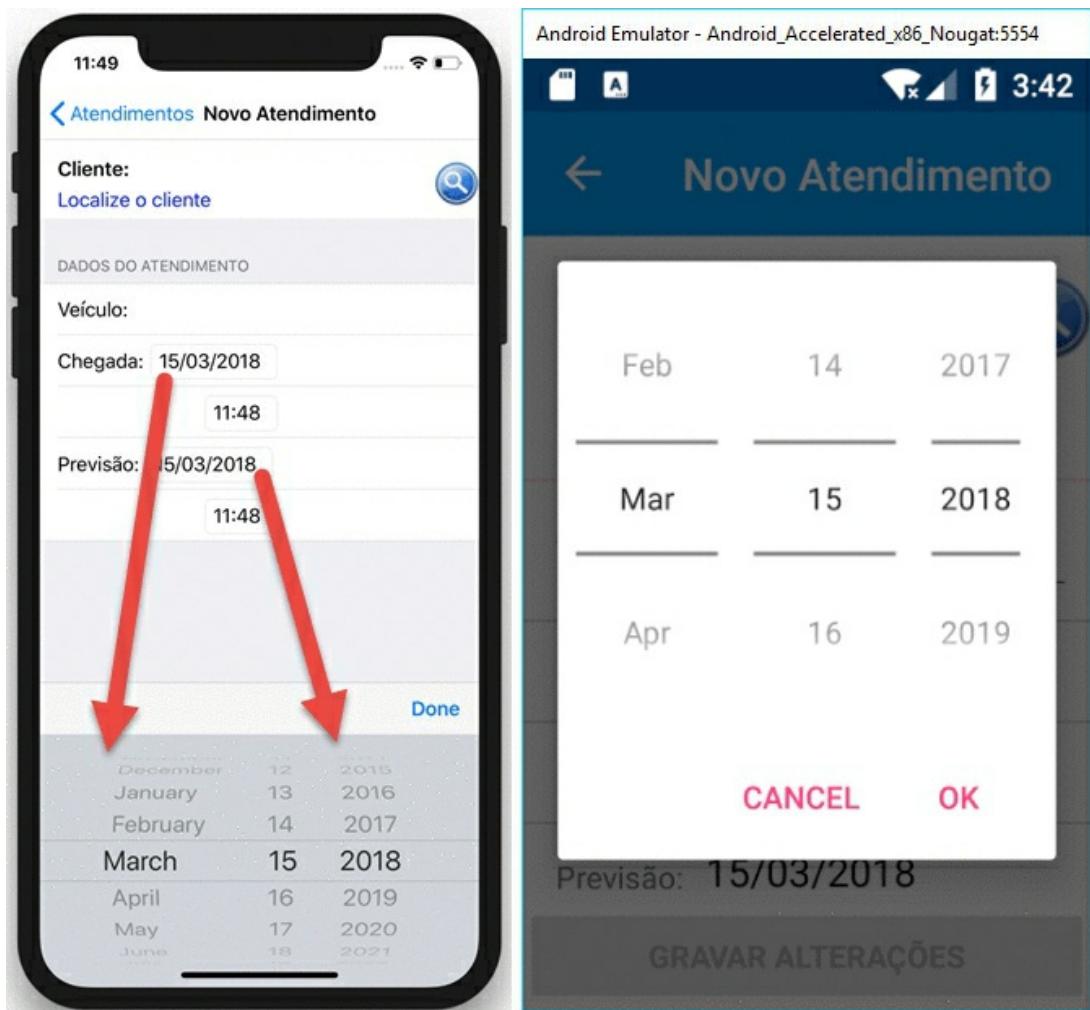


Figura 6.7: Informando datas pelo DatePicker

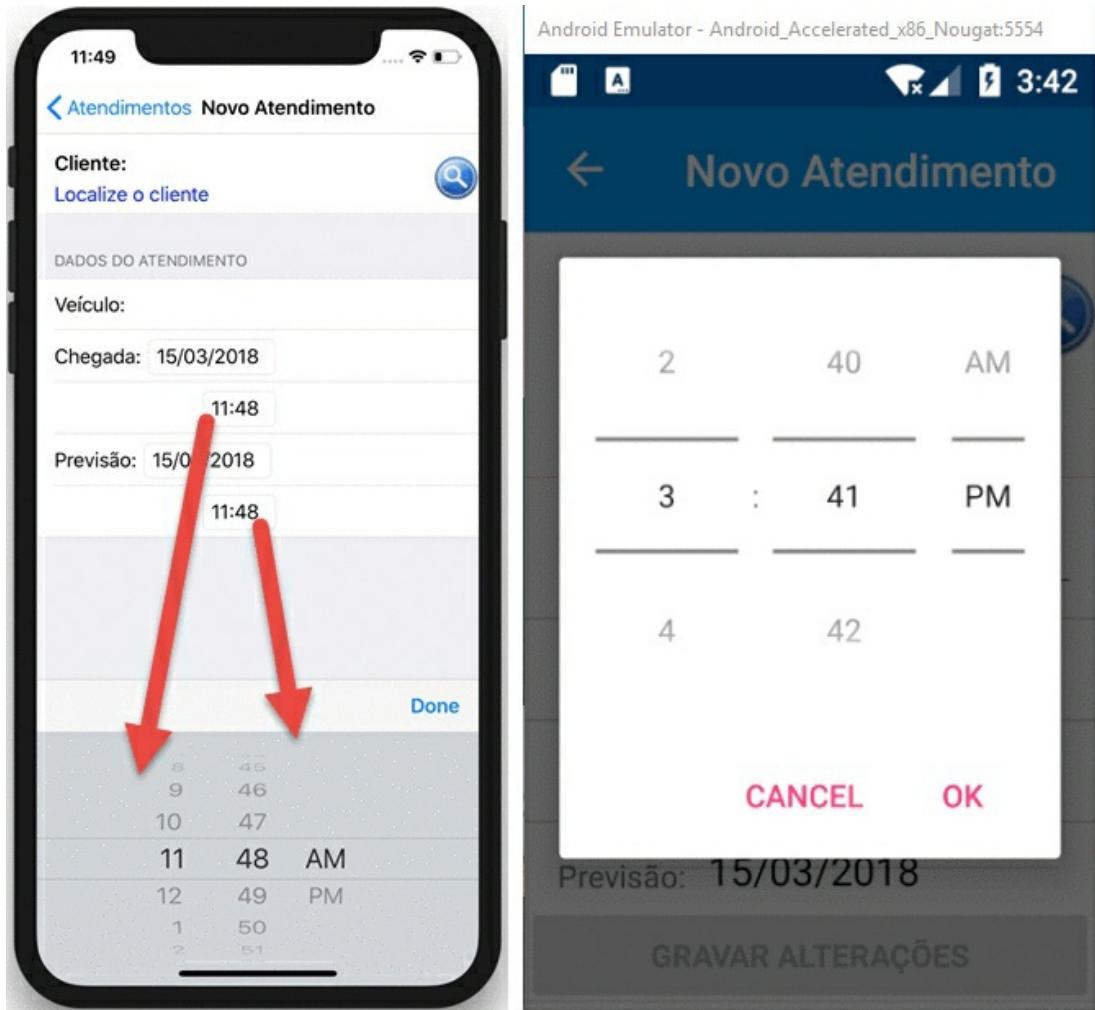


Figura 6.8: Informando horas pelo TimePicker

Agora precisamos implementar controles semelhantes para a data e hora previstas para a conclusão dos serviços. O XAML a seguir apresenta os códigos para os controles de data e hora e, na sequência, as propriedades ligadas a estes controles. Veja que são semelhantes aos vistos anteriormente. Implemente-os na sequência de seu código.

```
<ViewCell>

<Grid HorizontalOptions = "Fill" Margin = "5, 3, 0, 0" Padding = "10, 0, 0, 0" >

    <Grid.ColumnDefinitions>

        <ColumnDefinition Width = "Auto" />

        <ColumnDefinition Width = "*" />

    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
```

```

<RowDefinition Height = "Auto" />

</Grid.RowDefinitions>

<Label Text = "Previsão:" VerticalOptions = "CenterAndExpand" Grid.Column = "0" Grid.Row = "0" />

<DatePicker VerticalOptions = "FillAndExpand" HorizontalOptions = "StartAndExpand" Date = "{Binding DataPrometida}"
Grid.Column = "1" Grid.Row = "0" >

<DatePicker.Format> dd/MM/yyyy </DatePicker.Format>

</DatePicker>

</Grid>

</ViewCell>

<ViewCell>

<Grid HorizontalOptions = "Fill" Margin = "5, 3, 0, 0" Padding = "10, 0, 0, 0" >

<Grid.ColumnDefinitions>

<ColumnDefinition Width = "120" />

<ColumnDefinition Width = "*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height = "Auto" />

</Grid.RowDefinitions>

<TimePicker VerticalOptions = "FillAndExpand" HorizontalOptions = "StartAndExpand" Time = "{Binding HoraPrometida}"
Grid.Column = "1" Grid.Row = "0" >

<TimePicker.Format> HH:mm </TimePicker.Format>

</TimePicker>

</Grid>

</ViewCell>

```

```

public DateTime DataPrometida
{
    get { return this .Atendimento.DataHoraPrometida; }

    set
    {
        this .Atendimento.DataHoraPrometida = new DateTime( value .Year, value .Month, value .Day, HoraPrometida.Hours,
        HoraPrometida.Minutes, 0);
        OnPropertyChanged();
    }
}

public TimeSpan HoraPrometida
{
    get { return new TimeSpan ( this .Atendimento.DataHoraPrometida.Hour, this .Atendimento.DataHoraPrometida.Minute, 0); }

    set
    {
        this .Atendimento.DataHoraPrometida = new DateTime(DataPrometida.Year, DataPrometida.Month, DataPrometida.Day, vali
        .Hours, value .Minutes, 0);
        OnPropertyChanged();
    }
}

```

## 6.5 Botão e Command para realizar a gravação do atendimento

No capítulo anterior, tanto para Clientes como para Serviços, tínhamos um botão que era exibido no final da tela, para que o usuário pudesse clicar nele e realizar a inserção (ou atualização) dos dados informados. Este botão, no entanto, só ficava habilitado quando os valores da visão fossem válidos, e aqui vamos seguir a mesma ideia. Na classe `CRUDViewModel` de `Atendimentos`, declare o `Command` tal qual é exibido no código a seguir.

```
public ICommand GravarCommand { get ; set ; }
```

Com o `Command` declarado, precisamos implementá-lo no método `RegistrarCommands()`, que já temos em nossa classe. Faça isso com o código apresentado na sequência, antes do fechamento do método. Observe as regras para que o botão possa estar habilitado.

```

GravarCommand = new Command( async () =>
{
    if (Atendimento.AtendimentoID != null )

        Atendimento.NotificarListView = true ;

    await atendimentoDAL.UpdateAsync(Atendimento, Atendimento.AtendimentoID);

    MessagingCenter.Send< string >( "Atualização realizada com sucesso." , "InformacaoCRUD" );

}, () =>
{

    return (( this .Atendimento.Cliente != null ) && ! string .IsNullOrEmpty( this .Atendimento.Cliente.Nome) && ! string

```

```
.IsNullOrEmpty( this .Atendimento.Veiculo) && ( this .Atendimento.DataHoraPrometida > this .Atendimento.DataHoraChegada));
});
```

Veja a seguir o código que precisamos implementar no `OnAppearing()` e, após ele, no `OnDisappearing()` do code-behind de `CRUDView`.

```
MessagingCenter.Subscribe<string>( this , "InformacaoCRUD" , async (msg) => { await DisplayAlert ( "Informação" , msg , "ok" );
});
```

```
MessagingCenter.Unsubscribe<string>( this , "InformacaoCRUD" );
```

Para que o usuário possa indicar que deseja gravar o atendimento, novo ou não, precisamos inserir um XAML do botão em nossa `CRUDView` e ele está na sequência. Insira-o após o último `<ViewCell>` existente. Precisaremos agora invocar o método que valida a habilitação ou não do botão. Você deverá implementar o código que está após o XAML ao final do método `set()` de todas as propriedades que são ligadas com a visão, para que, a cada atualização nas propriedades ligadas, o status do botão possa ser verificado e atualizado.

```
<ViewCell>
```

```
<Button Text = "Gravar Alterações" FontAttributes = "Bold" VerticalOptions = "End" Command = "{Binding GravarCommand}" />
</ViewCell>

((Command)GravarCommand).ChangeCanExecute();
```

Vamos conversar um pouco sobre a atualização ou inserção de um objeto pelo EF Core, que possua uma associação que já esteja persistida na base de dados. Precisamos saber e entender que o EF Core trabalha com contextos gerenciáveis de objetos mapeados de e para uma base de dados, para que ele possa rastrear alterações e efetivá-las na chamada ao método `SaveChangesAsync()`. Quando recuperamos um cliente pela pesquisa, fizemos uso de um contexto criado especificamente para esta operação e atribuímos o objeto à propriedade `Cliente` do atendimento que será inserido ou atualizado. Quando formos gravar o atendimento, o EF Core identificará que temos um objeto associado a ele, mas que não faz parte do mesmo contexto, embora seja um objeto já persistido. Então, ele assumirá que este objeto (cliente) não existe, e tentará inseri-lo junto com o atendimento.

Ocorre que, ao tentar esta inserção, uma *constraint* do banco é disparada, a que especifica que a chave primária deve ser única (*UNIQUE constraint*), pois o objeto associado (Cliente) já possui um valor para sua propriedade identificadora. Mas tudo bem, há como corrigir isso. Todo objeto, quando se usa o EF Core, possui um estado e nós precisamos apenas informar que o estado do objeto associado não sofreu alteração no contexto atual. Precisaremos realizar algumas mudanças em nossa `IDAL` e `DALBase`, pois precisaremos poder informar os objetos associados ao que será persistido. Veja, a seguir, a nova assinatura para o método `UpdateAsync()`, na `IDAL`.

```
Task<T> UpdateAsync (T item, long ? itemID, params object [] associatedObjects);
```

Observe que definimos um novo parâmetro, novamente com o `params`, que, opcionalmente, receberá um ou mais objetos que estejam associados ao item que será persistido. Vamos agora à nossa classe `DALBase`. Altere também a assinatura do método nela. Depois, logo no início do bloco `using()`, insira o código a seguir. Veja como a informação do estado para os objetos associados ocorre. Se você tiver interesse, quando tiver disponibilidade, seria legal dar uma lida em <https://www.learnentityframeworkcore.com/dbcontext/change-tracker/> para maiores detalhes sobre os estados de objetos para o EF Core.

```
foreach ( var associated in associatedObjects)
{
    context.Entry(associated).State = EntityState.Unchanged;
}
```

Precisamos alterar também a invocação ao método `UpdateAsync()`, que ocorre no `GravarCommand`, dentro de `RegistrarCommands()` do `CRUDViewModel` de `Atendimentos`. Veja a alteração na instrução a seguir.

```
Atendimento = await atendimentoDAL.UpdateAsync(Atendimento, Atendimento.AtendimentoID, Atendimento.Cliente);
```

Execute sua aplicação e insira um novo atendimento. Após a inserção, note que os controles não são reinicializados. Tratei desta maneira aqui pelo fato de que nos próximos capítulos trabalharemos a inserção dos serviços e fotos, o que poderá ser feito depois da inserção do atendimento, ou, quando um atendimento estiver sendo alterado, que é o que veremos mais à frente. Retorne para a visão de listagem de atendimentos e você verá lá seu atendimento inserido, tal qual pode ser visto na figura a seguir.

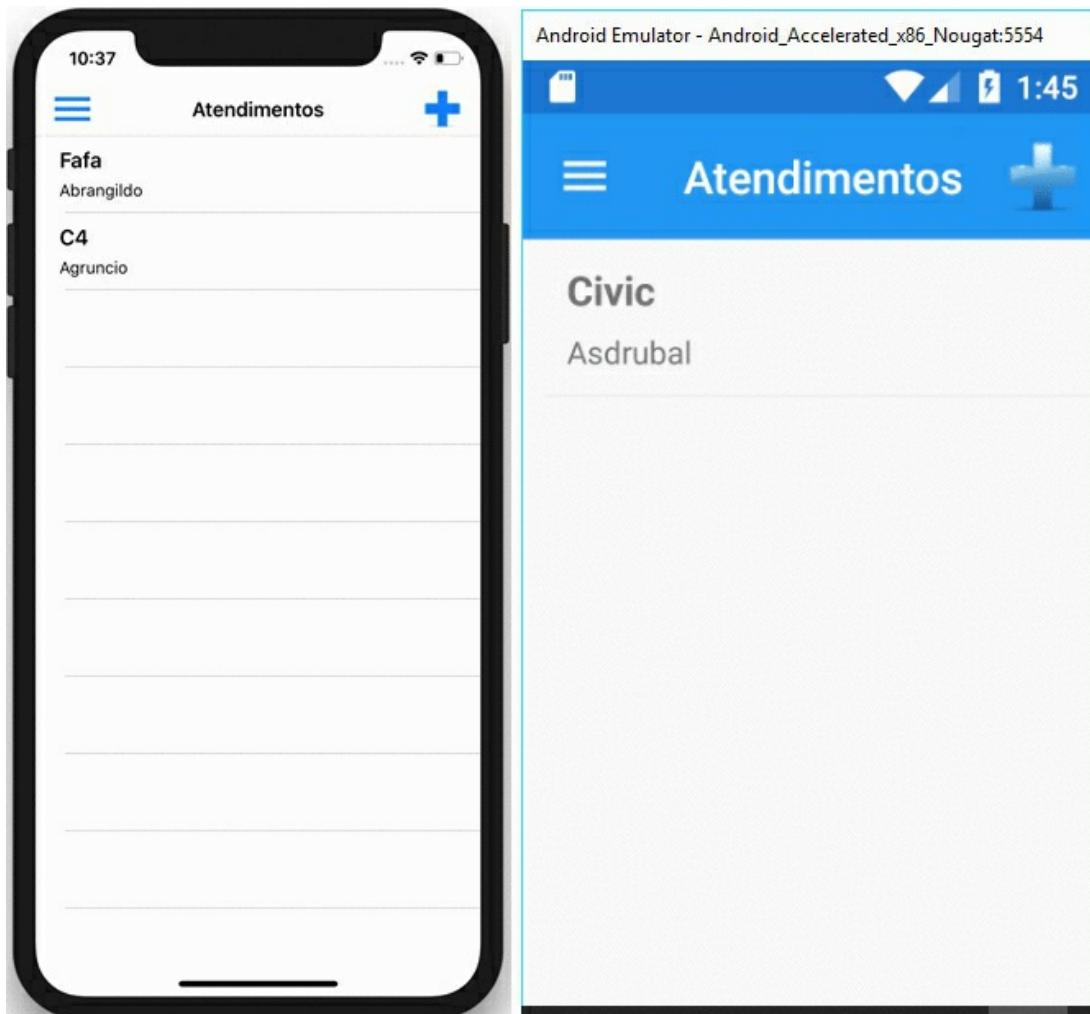


Figura 6.9: Listagem com atendimento registrado

## 6.6 Operações com atendimentos registrados

No capítulo anterior, oferecemos ao usuário a possibilidade de remover um cliente ou serviço por meio dos `Context Actions`, que exibiam as opções disponíveis para o item selecionado. Você certamente se lembra de que, no iOS, para exibir as `Context Actions` é preciso arrastar o item do `Listview` para a esquerda, e, no Android, é preciso pressionar por alguns instantes o item desejado.

Existe outra opção para exibição destas operações e são conhecidas por `Action Sheets`. Este recurso exibe as opções em forma de um menu vertical na parte inferior da visão, o que dá um melhor visual para a aplicação. Nós implementaremos esta funcionalidade por meio do MVVM e `Messaging Center`, quando o usuário selecionar um atendimento. É bem simples e o resultado é legal!

Começaremos a implementação definindo a propriedade `SelectedItem` no `ListView` de atendimentos e isso pode ser feito acrescentando o código `SelectedItem="{Binding AtendimentoSelecionado}"` na tag do `<ListView>` de `ListagemView` de atendimentos. Precisamos, na sequência, implementar a propriedade `AtendimentoSelecionado` em nossa classe `ListagemViewModel`, tal qual o código a seguir.

```
private Atendimento atendimentoSelecionado;

public Atendimento AtendimentoSelecionado
{
    get { return atendimentoSelecionado; }

    set
    {
        if ( value != null )
        {

            atendimentoSelecionado = value ;
            MessagingCenter.Send<Atendimento>(atendimentoSelecionado, "MostrarOpcoes" );
        }
    }
}
```

Agora, no code-behind da visão `ListagemView` precisamos registrar e cancelar a assinatura na mensagem utilizada no código anterior. Você pode conferir a listagem a seguir. Veja, no método registrado no `Subscribe()`, duas chamadas ao método `DisplayActionSheet()`, cada chamada com uma assinatura diferente. O primeiro parâmetro se refere ao título para as opções que serão exibidas, e o segundo refere-se à opção que fechará as opções, que será a última a ser exibida no menu. O terceiro parâmetro refere-se à primeira opção, que aparecerá em destaque e, os demais parâmetros referem-se às opções que deverão complementar o menu a ser oferecido para o usuário. Este método retorna uma `string` que conterá o texto da opção escolhida pelo usuário. A determinação de quais opções serão disponibilizadas ao usuário está diretamente relacionada ao estado do atendimento, se está finalizado, ou não. Veja a condição avaliada no `if`. Observe que a última instrução a ser executada refere-se a um método que ainda não temos, mas que já veremos. Ainda, este método só será executado caso o usuário pressione alguma opção. Caso ele pressione uma área da visão, fora da `Action Sheet`, a variável `result` conterá o valor `null`.

```
private async Task ExibirOpcoesAsync (Atendimento atendimento)
{
    viewModel.AtendimentoSelecionado = null ;

    string result;

    if (atendimento.EstaFinalizado)

        result = await DisplayActionSheet ( "Opções para o Atendimento " + atendimento.AtendimentoID, "Cancelar" ,
        "Consultar" );

    else

        result = await DisplayActionSheet ( "Opções para o Atendimento " + atendimento.AtendimentoID, "Cancelar" , "Alterar" ,
        "Registrar Entrega" , "Remover OS" );
}
```

```

    if (result != null)
        ProcessarOpcãoRespondidaAsync(atendimento, result);
}

// OnAppearing()

MessagingCenter.Subscribe<Atendimento>( this , "MostrarOpções" , async (atendimento) => { await ExibirOpçõesAsync
(atendimento); });

// OnDisappearing()

MessagingCenter.Unsubscribe<Atendimento>( this , "MostrarOpções" );

```

Como temos várias opções que serão oferecidas ao usuário pelo Action Sheet, precisamos tratar a opção do usuário e, buscando coesão, delegamos esta responsabilidade para o método `ProcessarOpcãoRespondidaAsync()`, que recebe, em sua assinatura, o atendimento que foi selecionado e a opção escolhida pelo usuário. Na sequência, você pode ver a implementação necessária para este método, que deve ser implementado na `ListagemView`. No código a seguir, temos a invocação de alguns métodos que ainda não implementamos, mas que veremos adiante. Com isso, é importante que você se atente à funcionalidade, que será comentada, e depois partiremos para a implementação destes métodos.

Vamos lá. A primeira condição avaliada no método `ProcessarOpcãoRespondidaAsync()` refere-se à consulta ou alteração do atendimento selecionado. Veja a definição do título para a visão e a chamada ao construtor dela. Bem semelhante ao que fizemos para a inserção de um novo atendimento. Para o registro de entrega do veículo, que é a segunda condição avaliada, invocamos o método `RegistrarEntregaAsync()`. Caso nenhuma das duas condições seja verdadeira, então, a terceira e última verifica a intenção de remoção do atendimento, e, caso positivo, invoca o método `EliminarAtendimentoAsync()`, depois de o usuário confirmar esta intenção.

```

private async void ProcessarOpcãoRespondidaAsync (Atendimento atendimento, string result)

{
    if (result.Equals( "Consultar" ) || result.Equals( "Alterar" ))
    {
        var title = result + " Atendimento " + atendimento.AtendimentoID;

        await Navigation.PushAsync( new CRUDView(atendimento, title));
    }

    else if (result.Equals( "Registrar Entrega" ) )
    {
        await viewModel.RegistrarEntregaAsync(atendimento);

        await DisplayAlert ( "Informação" , "Entrega registrada com sucesso." , "Ok" );

        listView.SelectedItem = null ;
    }

    else if (result.Equals( "Remover OS" ) )
    {
        if ( await DisplayAlert ( "Confirmação" ,
            $"Confirma remoção da OS {atendimento.AtendimentoID}?" , "Yes" , "No" ) )
    }
}

```

```

        await viewModel.EliminarAtendimentoAsync(atendimento);

        await DisplayAlert ( "Informação" , "Atendimento removido com sucesso" , "Ok" );
    }
}
}

```

Muito bem, para finalizarmos as implementações necessárias para esta etapa precisamos implementar os métodos `RegistrarEntregaAsync()` e `EliminarAtendimentoAsync()` na classe `ListagemViewModel`, e, na sequência, apresento-os. Observe a maneira como atualizamos nossa coleção `Atendimentos`, que está ligada à `ListView`. O procedimento é semelhante ao que realizamos nos métodos que atualizam as coleções quando as visões são exibidas.

```

public async Task RegistrarEntregaAsync (Atendimento atendimento)

{
    atendimento.DataHoraEntrega = DateTime.Now;

    var indiceAtendimento = Atendimentos.IndexOf( await atendimentoDAL.UpdateAsync(atendimento,
atendimento.AtendimentoID));

    Atendimentos.RemoveAt(indiceAtendimento);
    Atendimentos.Insert(indiceAtendimento, atendimento);
}

public async Task EliminarAtendimentoAsync (Atendimento atendimento)

{
    await atendimentoDAL.DeleteAsync(atendimento);
    Atendimentos.Remove(atendimento);
}

```

Agora, enfim, podemos executar nossa aplicação e testá-la. Sugiro que registre mais alguns atendimentos para testar as opções que serão disponibilizadas e que estão apresentadas na figura a seguir. Veja como o Action Sheet é apresentado em cada plataforma. A princípio, altere um atendimento, depois remova um e, para finalizar, registre a entrega de um. Em seguida, selecione um atendimento já entregue e veja que as opções apresentadas ao usuário são diferentes das apresentadas para um atendimento não entregue.

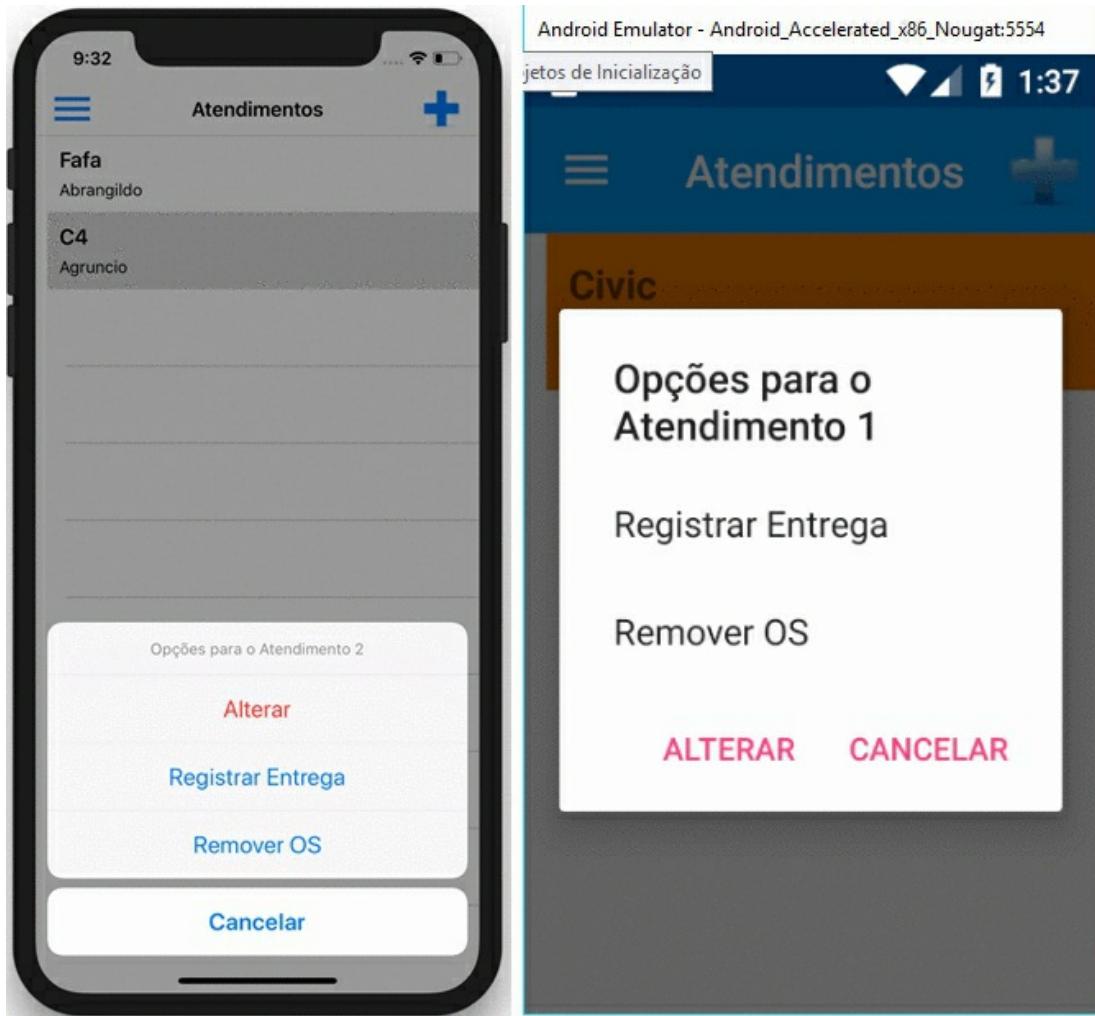


Figura 6.10: Opções de trabalho com atendimento selecionado

Seria interessante, em nosso `Listview`, que os atendimentos já finalizados, que foram entregues, pudessem aparecer em destaque, para que o usuário possa verificar essa situação. Logo implementaremos essa situação.

## 6.7 Implementações para a consulta e alteração de um atendimento

Com o que temos implementado já podemos testar o processo de alteração de um atendimento, lembrando, que de acordo com a nossa lógica adotada, um atendimento só poderá ser alterado enquanto o veículo não tiver sido entregue. Mas e a consulta? Vimos que a metodologia adotada para exibir a visão é a mesma, mudamos apenas o título, mas o que diferencia a consulta de uma inserção ou alteração? A resposta é que os dados não podem ser alterados.

Com esta introdução realizada, vamos à prática. Para a consulta, a imagem para localizar um cliente e o botão de gravar as alterações não podem ser exibidos e os controles que solicitam o nome do veículo, datas e horários não podem estar habilitados. Os controles visuais possuem duas propriedades, que manipularemos para este objetivo. Para a imagem e o botão (`CRUDview`), insira em suas tags a propriedade `isVisible="{Binding HabilitaAlteracao}"`, para ligar a visibilidade destes controles a uma propriedade em nossa classe `CRUDviewModel`. Veja a implementação.

```

public bool HabilitaAlteracao
{
    get { return !Atendimento.EstaFinalizado; }
}

```

Para os controles de entrada de dados, usaremos outra propriedade, a `IsEnabled` (está habilitado) e ela deve ser configurada nas tags dos controles de entrada de dados como `IsEnabled="{Binding HabilitaAlteracao}"`. Agora, tente consultar o atendimento cuja entrega você registrou e veja o resultado visual de não visibilidade e não habilitação para os controles que trabalhamos nessa seção.

Para finalizarmos o trabalho, precisamos exibir a data de entrega do veículo e, para realizarmos isso, vamos implementar o código XAML apresentado na sequência, abaixo da `ViewCell` da hora de previsão. Observe que o `Label` tem sua propriedade `Text` ligada a uma propriedade da ViewModel chamada `EntregaVeiculo`, que está implementada após o XAML.

```

<ViewCell>

    <StackLayout Orientation = "Horizontal" HorizontalOptions = "Fill" Margin = "5, 3, 0, 0" Padding = "10, 5, 0, 0" >

        <Label Text = "{Binding EntregaVeiculo}" VerticalOptions = "CenterAndExpand" HorizontalOptions = "CenterAndExpand" />

    </StackLayout>

</ViewCell>

public string EntregaVeiculo
{
    get {
        return (HabilitaAlteracao)
            ? "" : "Entregue em " + ((DateTime) this .Atendimento.DataHoraEntrega).ToString( "dd/MM/yyyy HH:mm:ss" );
    }
}

```

Execute novamente sua aplicação e consulte o atendimento entregue. A data de entrega deverá estar exibida, tal qual mostra a figura a seguir.

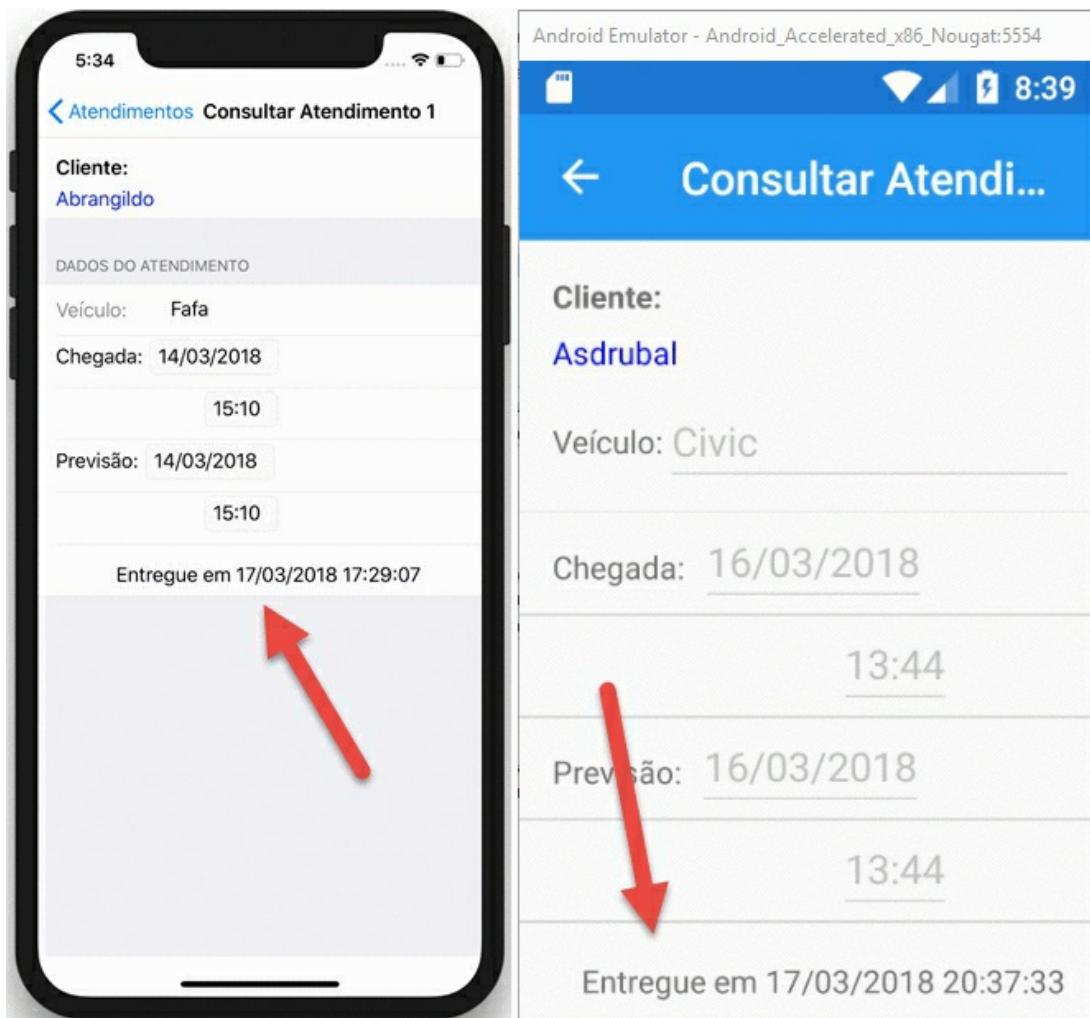


Figura 6.11: Visão de consulta com controles ocultos, desabilitados e dados de entrega

## 6.8 Destaque de um atendimento finalizado pelo uso de Converters

Como dito anteriormente, em nosso `Listview`, seria interessante que os atendimentos já finalizados, que foram entregues, pudessem aparecer em destaque, para que o usuário verificar essa situação. Para resolvemos esta questão, faremos uso de Converters.

No projeto Xamarin Forms, crie uma pasta chamada `Converters` e, dentro dela, uma classe chamada `AtendimentoFinalizadoColor`, tal qual o código a seguir. Note que ela implementa a interface `IValueConverter`. Veja que, no método `Convert()`, verificamos o valor recebido, que será proveniente da propriedade `HabilitaAlteracao` e, de acordo com o valor recebido, uma cor é retornada.

```
namespace Capitulo05.Converters
{
    public class AtendimentoFinalizadoColor : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
        {
```

```

        if ((bool)value)
            return Color.Yellow;
        return Color.White;
    }

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

```

Vamos agora implementar o uso deste conversor em nosso XAML. Na tag `<ContentPage>`, antes do `Title`, insira o código a seguir. Por meio dele definimos um objeto chamado `conv` que referencia nosso Converter.

```
xmlns:conv="clr-namespace:Capitulo05.Converters;assembly=Capitulo05"
```

Depois, antes da tag `<ContentPage.Content>`, precisamos disponibilizar o objeto para uso em nossos controles visuais, no XAML. Faça isso implementando o código.

```

<ContentPage.Resources>

<ResourceDictionary>

<conv:AtendimentoFinalizadoColor x:Key = "colorConvert" />

</ResourceDictionary>

</ContentPage.Resources>

```

Enfim, podemos agora utilizar nosso Converter e o faremos no `StackLayout` dos `Labels` do `ItemTemplate`, tal como é apresentado no código a seguir. Pode testar novamente sua aplicação e verificar o resultado na exibição do atendimento entregue.

```
<StackLayout Padding = "10" BackgroundColor = "[Binding EstaFinalizado, Converter={StaticResource colorConvert}]" >
```

Esta foi uma maneira simples de utilizarmos `Converters` e você pode utilizar desta técnica para alterar comportamentos padrões de renderização de propriedades e controladores. Uma dica que você pode pensar para sua aplicação também seria um novo botão na barra de tarefas, que possibilitasse visualizar apenas atendimentos em aberto. O que acha de fazer isso?

## 6.9 O método para recuperar um único objeto

Nós realizamos várias implementações em nosso `DALBase`, mas ficou faltando um método simples, que nós não implementamos por não ter sido necessário até este momento. Entretanto, para finalizarmos o CRUD, é interessante que o implementemos. Com o que vimos para o atendimento, sabemos que temos a situação de recuperação de um objeto já com suas associações, por meio do `Include()`. Faremos também uma recuperação síncrona, o que nos leva a inserir em nossa interface o método a seguir.

```
T GetById ( long ? id, params string [] includeProperties);
```

Agora sim, podemos implementar nosso método que recuperará um objeto em específico da base de dados. Veja o código a seguir, que deve ser inserido na `DALBase`. Nossa opção para a sincronicidade se deve à simplicidade da implementação.

```
public virtual T GetById ( long ? id, params string [] includeProperties )  
{  
    using ( var context = DatabaseContext.GetContext(dbPath) ) {  
        var result = context.Set<T>().Find(id);  
        for ( int i = 0; i < includeProperties.Count(); i++ )  
        {  
            context.Entry(result).Reference(includeProperties[i]).Load();  
        }  
        return result;  
    }  
}
```

Uma rápida implementação para uma recuperação assíncrona pode ser vista no código a seguir, que deve estar na classe `AtendimentoDAL`. Caso você queira implementá-la de maneira genérica, na `DALBase`, poderá fazer isso com os recursos de reflexão que utilizamos em métodos anteriores, mas vou deixar isso para você, ok?

```
public override async Task<Atendimento> GetByIdAsync ( long ? id )  
{  
    return await context.Atendimentos.Include(c => c.Cliente).SingleOrDefaultAsync(a => a.AtendimentoID == id);  
}
```

## 6.10 Conclusão

Chegamos ao final deste capítulo. Trabalhamos associação entre classes e mapeamos essa associação para a base de dados. Vimos como recuperar e atualizar dados associados pelo EF Core. Utilizamos novos controles para entrada de data e hora e implementamos `Action Sheets`. Fizemos também o uso de `Converters` para diferenciar um atendimento já finalizado. Foi possível aprendermos boas técnicas e conhecer algumas características do EF Core.

No próximo capítulo trabalharemos o registro de serviços que serão realizados no veículo. Teremos como base o projeto que implementamos nestes dois últimos capítulos.

## C APÍTULO 7

# Associações com coleções

No capítulo anterior, vimos o mapeamento de classes associadas por meio de uma propriedade para uma base de dados, onde registramos um atendimento a um veículo e a este atendimento tínhamos associado um cliente. Fizemos uso de controles específicos de plataforma para a informação de data e hora. Este projeto será utilizado como base para este novo capítulo.

O objetivo deste sétimo capítulo está em implementarmos uma associação baseada em coleções e a mapearmos para a base de dados. Teremos duas associações com coleções: na primeira delas, foco deste capítulo, implementaremos os serviços que serão ou foram realizados no veículo do atendimento e, na segunda, registraremos fotos, que o usuário poderá obter utilizando a câmera do dispositivo ou selecionar diretamente do álbum. Esta segunda associação será concluída no próximo capítulo. Para facilitar a implementação, montaremos o básico para o trabalho com fotos, pois é uma tarefa comum à de serviços. Isso minimizará nosso esforço no próximo capítulo.

Este capítulo possui bastante código e algumas técnicas interessantes, o que nos leva a uma atenção especial na sequência das implementações, para que tudo possa ocorrer conforme o que leremos, mas o esforço será recompensado.

## 7.1 Classe de modelo e de acesso a dados

No capítulo anterior, criamos a classe `Atendimento`, que tinha uma associação com `Cliente`. Agora, teremos duas classes que estarão associadas a `Atendimento`, são elas: `AtendimentoItem` e `AtendimentoFoto`. A primeira ainda estará associada à classe `Servico`. Com a implementação destas classes, propiciaremos ao usuário registrar vários serviços a serem realizados no veículo e também inserir diversas fotos do automóvel no momento da entrada na oficina. Entretanto, sabemos que, antes de definirmos as classes, precisamos implementar os IDs para elas. Desta maneira, no projeto `PropertiesEF`, implemente na pasta `Models` a classe `AtendimentoItemIDProperty` e `AtendimentoFotoIDProperty`, tal qual o código a seguir.

```
namespace CasaDoCodigo.Models
{
    public class AtendimentoItemIDProperty
    {
        public long? AtendimentoItemID { get; set; }
        public long? AtendimentoID { get; set; }
        public long? ServicoID { get; set; }

        [NotMapped]
        public bool NotificarListView { get; set; }
    }
}
```

```
namespace CasaDoCodigo.Models
{
    public class AtendimentoFotoIDProperty
    {
        public long? AtendimentoFotoID { get; set; }
        public long? AtendimentoID { get; set; }

        [NotMapped]
    }
}
```

```

        public bool NotificarListView { get; set; }

    }
}

```

Agora podemos implementar nossas classes de negócio para os itens de serviço e para as fotos. Os códigos para elas estão apresentados na sequência. Crie-as na pasta `Atendimentos`, do projeto `oficinaModels`. No código da classe `AtendimentoItem`, observe a sobreescrita para os métodos `Equals()` e `GetHashCode()`, tal qual fizemos nos capítulos anteriores, mas agora levamos a associação em consideração para a identidade dos objetos da classe. Essa implementação é necessária para que, no registro dos serviços para o atendimento, o usuário não insira o mesmo serviço mais de uma vez. Não julguei necessário implementar esta solução na classe `AtendimentoFoto`, pois não há uma identificação única para ela em nosso modelo.

```

namespace CasaDoCodigo.Models
{
    public class AtendimentoItem : AtendimentoItemIDProperty
    {
        public int Quantidade { get; set; }
        public double Valor { get; set; }
        public double SubTotal { get { return Quantidade * Valor; } }

        public Atendimento Atendimento { get; set; }

        public long? ServicoID { get; set; }
        public Servico Servico { get; set; }

        public override bool Equals(object obj)
        {
            var item = (obj as AtendimentoItem);
            return (item.AtendimentoID == this.AtendimentoID && item.ServicoID == this.ServicoID);
        }

        public override int GetHashCode()
        {
            var hashCode = -1711974841;
            hashCode = hashCode * -1521134298 + EqualityComparer<string>.Default.GetHashCode((AtendimentoID +
ServicoID).ToString());
            return hashCode;
        }
    }
}

namespace CasaDoCodigo.Models
{
    public class AtendimentoFoto : AtendimentoFotoIDProperty
    {
        public string CaminhoFoto { get; set; }
        public string Observacoes { get; set; }

        public Atendimento Atendimento { get; set; }
    }
}

```

Para concluirmos a associação e permitirmos que ela tenha navegabilidade bidirecional, vamos implementar as coleções para elas na classe `Atendimento`, tal qual o código a seguir apresenta.

```
public virtual List<AtendimentoItem> Servicos { get ; set ; }

public virtual List<AtendimentoFoto> Fotos { get ; set ; }
```

Precisamos também instanciar esta coleção sempre que um atendimento for inicializado. Desta maneira, no construtor da classe, implemente o código a seguir.

```
this .Servicos = new List<AtendimentoItem>();

this .Fotos = new List<AtendimentoFoto>();
```

Vamos agora modificar nossa classe de contexto. Na classe `DatabaseContext`, para o mapeamento no contexto do EF Core, precisamos implementar o código apresentado na sequência.

```
public DbSet<AtendimentoItem> AtendimentoItens { get ; set ; }

public DbSet<AtendimentoFoto> AtendimentoFotos { get ; set ; }
```

Neste momento, precisaremos rever o comportamento de nossos métodos DAL, pois temos uma condição preestabelecida para recuperar todos os itens de atendimento para nossa listagem, que é a de recuperar apenas os itens do atendimento que se está visualizando, ou seja, os serviços associados ao atendimento. Temos ainda uma nova situação para a classificação dos dados na listagem, que deverá ser feita pelo nome do serviço, isto é, uma propriedade de uma propriedade que é uma associação dos objetos da classe de pesquisa. O EF Core faz isso sem maiores problemas, mas a maneira como estamos fazendo, por meio de `string`, não funcionará, pois a `string` se refere a uma propriedade da classe cujos dados estão sendo recuperados. Precisamos mudar o tipo de dados do parâmetro para classificação.

Buscando trabalhar sempre em nossa classe `DALBase`, para que ela atenda de maneira genérica às requisições que possam ser comuns, precisamos alterar a assinatura de nosso método `GetAllAsync()` em nossa `IDAL`, para a que está na sequência, conforme os requisitos do parágrafo anterior. Veja a mudança para o primeiro argumento, que agora recebe uma `Expression` para uma *arrow function*, que poderá ser utilizada no `OrderBy()` e `OrderByDescending()`, sem precisarmos de qualquer conversão. Veja a mudança do nome do argumento de `campoClassificacao` para `expression`.

```
Task<List<T>> GetAllAsync(Expression<Func<T, object >> expression = null , OrderByType orderByType =
OrderByType.Ascendente);
```

Você pode retirar, ou comentar, as linhas que declaram as variáveis `parameter`, `autoBoxing` e `sortExpression` do método `PrepareDataToGetAll()`. Para a atribuição da expressão de classificação, vamos adaptar o código para que fique semelhante ao apresentado a seguir, que deve estar abaixo da definição de `query`.

```
if (expression != null )
{
    if (orderByType == OrderByType.Descendente)
        query = query.OrderByDescending(expression);

    else
        query = query.OrderBy(expression);
}
```

Além da assinatura no `GetAllAsync()` e da alteração no `PrepareDataToGetAll()`, é necessário ajustar o nome deste argumento no corpo dos métodos da classe `DALBase`. Lembre-se de fazer isso, ok?

Precisamos adaptar nossas classes especializadas de DAL para enviar corretamente as expressões para o primeiro

argumento. Estes códigos estão na sequência.

```
// ServicoDAL

public async override Task<List<Servico>> GetAllAsync(Expression<Func<Servico, object>> expression = null, OrderByType
orderByType = OrderByType.NaoClassificado)
{
    expression = (expression == null) ? (s => s.Nome) : expression;
    orderByType = orderByType == OrderByType.NaoClassificado ? OrderByType.Ascendente : orderByType;
    return await base.GetAllAsync(expression, orderByType);
}

// ClienteDAL

public async override Task<List<Cliente>> GetAllAsync(Expression<Func<Cliente, object>> expression = null, OrderByType
orderByType = OrderByType.NaoClassificado)
{
    expression = (expression == null) ? (c => c.Nome) : expression;
    orderByType = orderByType == OrderByType.NaoClassificado ? OrderByType.Ascendente : orderByType;
    return await base.GetAllAsync(expression, orderByType);
}

// AtendimentoDAL

public async override Task<List<Atendimento>> GetAllAsync(Expression<Func<Atendimento, object>> expression = null,
OrderByType orderByType = OrderByType.NaoClassificado)
{
    expression = (expression == null) ? (a => a.DataHoraChegada) : expression;
    orderByType = orderByType == OrderByType.NaoClassificado ? OrderByType.Descendente : orderByType;

    using (var context = DatabaseContext.GetContext(dbPath))
    {
        var query = PrepareDataToGetAll(context, expression, orderByType);
        query = query.Include(nameof(Atendimento.Cliente));
        return await query.ToListAsync();
    }
}
```

Você viu que utilizamos um pouco o operador ternário ?: para validação de valores nulos? O C# tem uma alternativa interessante, que é o `null-coalescing operator`, representado por ?? . Fica a dica para você ver em <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/null-coalescing-operator/> quando puder.

Nosso próximo passo será implementarmos `AtendimentoItemDAL` . Lembre-se de que esta classe DAL é criada na pasta `DataAccess` do `SQLiteEF` . No construtor, observe uma implementação diferente: ele recebe um argumento a mais agora, que é o atendimento ao qual se referem os itens que serão manipulados nessa classe. Antes do construtor, note que temos uma propriedade privada, que receberá o atendimento registrado no construtor. Veja o método `GetAllAsync()` e observe as atribuições dos argumentos, caso sejam recebidos na classe especializada.

```
namespace CasaDoCodigo.DAL
```

```

{
public class AtendimentoItemDAL : DALBase<AtendimentoItem>
{
    private Atendimento Atendimento { get; set; }

    public AtendimentoItemDAL(Atendimento atendimento, string dbPath) : base(dbPath)
    {
        this.Atendimento = atendimento;
    }

    public async override Task<List<AtendimentoItem>> GetAllAsync(Expression<Func<AtendimentoItem, object>> expression = null, OrderByType orderByType = OrderByType.NaoClassificado)
    {
        expression = (expression == null) ? (i => i.Servico.Nome) : expression;
        orderByType = orderByType == OrderByType.NaoClassificado ? OrderByType.Descendente : orderByType;

        using (var context = DatabaseContext.GetContext(dbPath))
        {
            var query = PrepareDataToGet1All(context, expression, orderByType);
            query = query.Include(i => i.Servico);
            query = query.Where(i => i.AtendimentoID == Atendimento.AtendimentoID);
            return await query.ToListAsync();
        }
    }
}
}

```

Com a recuperação dos objetos concluída, temos agora a tarefa de implementar a atualização para `AtendimentoItem`. Temos duas associações em cada objeto, com `Atendimento` e com `Servico`. Lembra o que comentamos no capítulo anterior quando precisamos persistir objetos que possuíam associação? Precisamos ajustar o estado deles, para que o EF Core não tente inseri-los na base, por não estarem no contexto atual. Nós estávamos enviando as propriedades associativas diretamente na ViewModel, mas agora vamos trazer esta responsabilidade para nossa especialização do DAL, o que melhorará os métodos envolvidos. Desta maneira, veja a sobrescrita para o método `UpdateAsync()`, em `AtendimentoItemDAL`.

```

public override Task<AtendimentoItem> UpdateAsync (AtendimentoItem item, long ? itemID, params object [] associatedObjects
{
    return base .UpdateAsync(item, itemID, item.Atendimento, item.Servico);
}

```

Embora tenhamos implementado apenas o DAL para itens de serviço, já podemos utilizar o Migrations para que as novas tabelas sejam criadas. Para isso, lembre-se de que precisamos mudar nossa classe `DatabaseContext` do projeto `Capitulo05Migrations`, para que contenha a declaração dos conjuntos para as novas classes de negócio. Você já tem este código em seu contexto, basta copiá-lo para o projeto do Migrations. Agora precisamos executar o Migrations para ele criar o código necessário para que a nova tabela seja criada na base de dados. São os mesmos passos que já executamos nos capítulos anteriores.

## 7.2 Acesso às visões para itens de serviço e fotos do veículo

Vamos começar a implementar a operacionalidade para que o usuário possa inserir serviços ao atendimento. Começaremos com a criação da visão que listará os serviços já registrados no atendimento. Na pasta Views/Atendimentos , crie uma ContentPage chamada ServicosListagemView e, inicialmente, implemente o seguinte código para ela em seu XAML. Estamos definindo na visão apenas o StackLayout que conterá nossos controles visuais.

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class = "Capitulo05.Views.Atendimentos.ServicosView"
    Title = "Serviços Atendimento"
    xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    ios:Page.UseSafeArea = "true" >

    <ContentPage.Content>
        <StackLayout VerticalOptions = "FillAndExpand" HorizontalOptions = "FillAndExpand" Orientation = "Vertical" >
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Também na pasta Views/Atendimentos , criaremos outra ContentPage , agora chamada FotosListagemView , cujo código inicial é apresentado na sequência. A princípio, teremos apenas o ScrollView , que trabalhamos nos capítulos 2 e 3 e que permitirá a rolagem da tela, tanto horizontal quanto verticalmente (Orientation=Both ).

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class = "Capitulo05.Views.Atendimentos.FotosListagemView"
    xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    ios:Page.UseSafeArea = "true"
    Title = "Fotos registradas" >
```

```

<ContentPage.Content>

    <ScrollView Orientation = "Both" >

    </ScrollView>

</ContentPage.Content>

</ContentPage>

```

Precisamos agora fornecer ao usuário acesso a estas duas visões. Este acesso será permitido por meio da visão de atendimento, mas só estará disponível quando o usuário estiver visualizando um atendimento já registrado. Se estiver inserindo um atendimento, ele precisará gravá-lo para que as opções possam ser disponibilizadas. Na visão CRUView de atendimentos, temos a última ViewCell para o botão de gravar. Vamos alterá-la tal qual o código que segue, para que exiba mais dois botões. Veja o uso do Grid para organizá-los. Observe que os novos também estão com a propriedade IsEnabled ligada à propriedade HabilitarBotoes . É possível verificar que também temos a definição de Commands para cada botão.

```

<ViewCell>

    <Grid HorizontalOptions = "FillAndExpand" Margin = "5, 3, 0, 0" Padding = "10, 0, 0, 0" >

        <Grid.ColumnDefinitions>

            <ColumnDefinition Width = "1*" />

            <ColumnDefinition Width = "1*" />

            <ColumnDefinition Width = "1*" />

        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>

            <RowDefinition Height = "Auto" />

        </Grid.RowDefinitions>

        <Button Text = "Gravar" FontAttributes = "Bold" Grid.Column = "0" Grid.Row = "0" Command = "{Binding GravarCommand}" IsVisible = "{Binding HabilitaAlteracao}" />

        <Button Text = "Serviços" FontAttributes = "Bold" Grid.Column = "1" Grid.Row = "0" HorizontalOptions = "Center" Command = "{Binding ServicosCommand}" IsEnabled = "{Binding HabilitarBotoes}" />

        <Button Text = "Fotos" FontAttributes = "Bold" Grid.Column = "2" Grid.Row = "0" HorizontalOptions = "Center" Command = "{Binding FotosCommand}" IsEnabled = "{Binding HabilitarBotoes}" />
    
```

```
</Grid>  
</ViewCell>
```

Vamos implementar os `Commands` e propriedades que são utilizadas na listagem anterior. Começaremos pela propriedade `HabilitarBotoes`. Lembre-se de que ela deve ser implementada na classe `CRUDViewModel`, que está na pasta `ViewModels/Atendimento`. Depois, no `Command GravarCommand`, após o `Send()`, insira a notificação de alteração de propriedade `onPropertyChanged("HabilitarBotoes");`, para o estado dos novos botões possa ser alterado.

```
public bool HabilitarBotoes  
{  
    get { return this .Atendimento.AtendimentoID != null ; }  
}
```

Agora, vamos partir para a implementação dos `Commands`, que têm suas definições na sequência.

```
public ICommand ServicosCommand { get ; set ; }  
public ICommand FotosCommand { get ; set ; }
```

Precisamos implementar o comportamento para eles, o que é feito no método `RegistrarCommands()`. Vamos inserir o código seguinte ao final do método.

```
ServicosCommand = new Command(() =>  
{  
    MessagingCenter.Send<Atendimento>(Atendimento, "MostrarServicos" );  
});  
  
FotosCommand = new Command(() =>  
{  
    MessagingCenter.Send<Atendimento>(Atendimento, "MostrarFotos" );  
});
```

Observe que fazemos uso do `MessagingCenter`, o que nos remete a implementar o registro para a mensagem e o cancelamento de registro. Sendo assim, no code-behind da visão `CRUDView`, de atendimentos, implemente o código a seguir no método `OnAppearing()`. Verifique que instanciamos as visões e as inserimos na pilha de navegação, tornando-as visíveis.

```
MessagingCenter.Subscribe<Atendimento>( this , "MostrarServicos" , async (atendimento) => { await Navigation.PushAsync( new ServicosListagemView(atendimento)); };  
  
MessagingCenter.Subscribe<Atendimento>( this , "MostrarFotos" , async (atendimento) => { await Navigation.PushAsync( new FotosListagemView(atendimento)); }; );
```

Ao implementar o código anterior, você se deparará com um erro de inexistência de um construtor que receba um argumento do tipo `Atendimento`. Precisamos corrigir isso. Lembre-se de que os serviços e fotos deverão saber a que atendimento eles se referem. A adaptação, no momento, é simples. Veja na sequência os construtores das classes de visão.

```
public ServicosListagemView (Atendimento atendimento)  
{
```

```

    InitializeComponent ();
}

public FotosListagemView (Atendimento atendimento)
{
    InitializeComponent ();
}

```

Finalizando esta etapa, para cancelar o registro das mensagens, no método `onDisappearing()`, implementaremos o código apresentado na sequência.

```

MessagingCenter.Unsubscribe<Atendimento>( this , "MostrarServicos" );
MessagingCenter.Unsubscribe<Atendimento>( this , "MostrarFotos" );

```

Vamos testar nossa aplicação. Selecione um atendimento existente na listagem de atendimentos, ou insira um, informe os dados e o grave. Verifique que os botões para acessar as novas visões estão disponíveis, como mostra a figura a seguir.

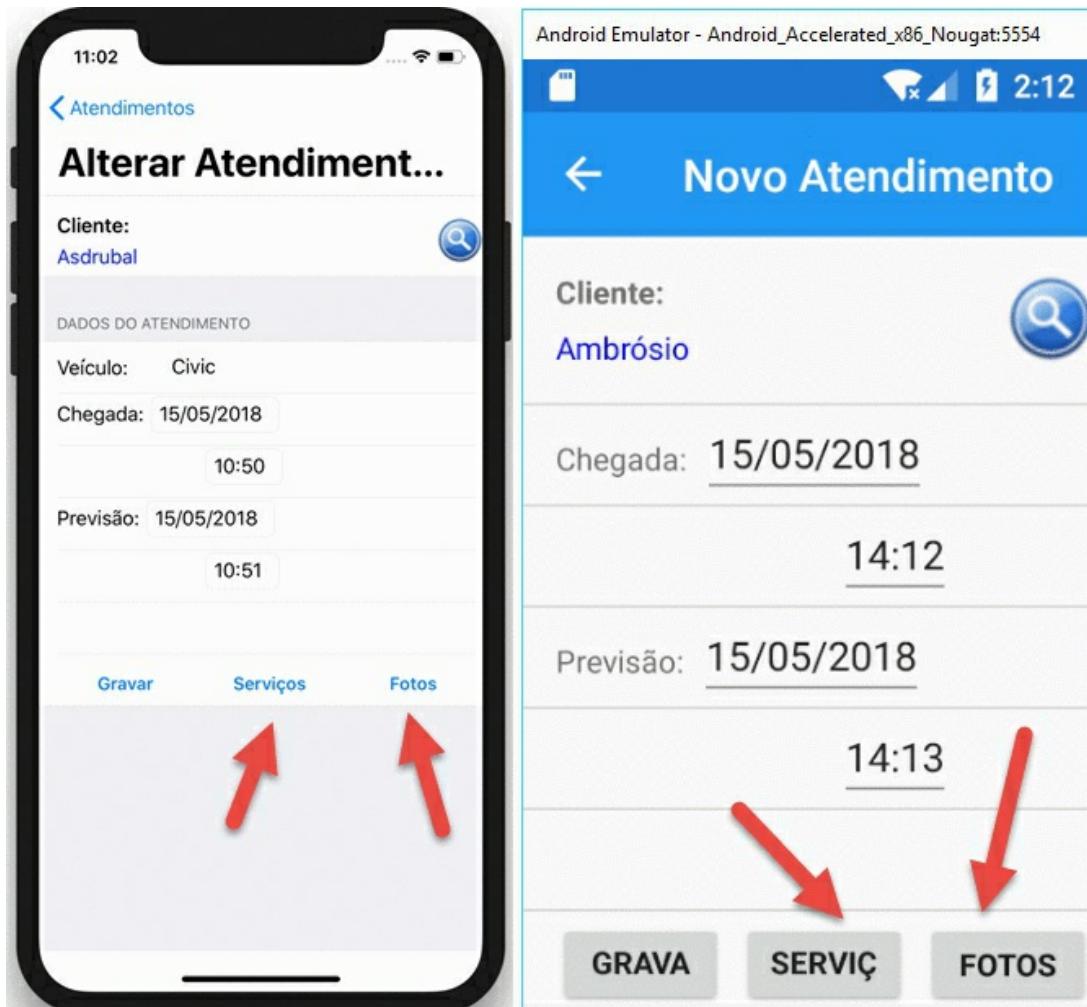


Figura 7.1: Visualização dos botões para serviços e fotos

### 7.3 A listagem e remoção de serviços registrados para o atendimento

Quando o usuário pressionar o botão relacionado à funcionalidade de serviços para o atendimento em foco, inicialmente será exibida uma visão com a listagem de serviços já registrados para o atendimento. Nós já temos a visão criada, falta inserir nela os controles visuais e os comportamentos esperados para eles. Desta maneira, vamos começar com o ListView para o XAML ServicosListagemView, que deverá estar inserido no <ContentPage.Content>, como apresentado na sequência. Como já trabalhamos praticamente todo o conteúdo que o código apresenta, já o veremos de uma vez e, após ele, conforme formos implementando as funcionalidades, vamos comentando e aprendendo.

```
<ListView x:Name = "listView" HasUnevenRows = "True" ItemsSource = "{Binding ItensAtendimento}" SelectedItem = "{Binding ServicoSelecionado}" >

<ListView.Footer>

<StackLayout Orientation = "Vertical" HorizontalOptions = "FillAndExpand" >

<BoxView HeightRequest = "10" HorizontalOptions = "FillAndExpand" BackgroundColor = "Black" />

<Label Text = "{Binding SubTotalItens, StringFormat='SubTotal: {0:C2}'}" FontSize = "14" FontFamily = "Courier"
HorizontalTextAlignment = "End" />

</StackLayout>

</ListView.Footer>

<ListView.ItemTemplate>

<DataTemplate>

<ViewCell BindingContextChanged = "OnBindingContextChanged" >

<ViewCell.ContextActions>

<MenuItem Command = "{Binding Path=BindingContext.EliminarItemCommand, Source={x:Reference listView}}" CommandParameter =
{Binding .}" Text = "Remover" IsDestructive = "True" />

</ViewCell.ContextActions>

<StackLayout Padding = "10" Orientation = "Vertical" >

<Label Text = "{Binding Servico.Nome}" FontSize = "18" FontAttributes = "Bold" />

<Grid>
```

```

<Grid.ColumnDefinitions>

    <ColumnDefinition Width = "20*" />

    <ColumnDefinition Width = "10*" />

    <ColumnDefinition Width = "30*" />

    <ColumnDefinition Width = "10*" />

    <ColumnDefinition Width = "30*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

    <RowDefinition Height = "Auto" />

</Grid.RowDefinitions>

<Label Text = "{Binding Quantidade}" Grid.Column = "0" FontSize = "14" FontFamily = "Courier" HorizontalTextAlignment = "End" />

<Label Text = "*" FontSize = "14" Grid.Column = "1" FontFamily = "Courier" />

<Label Text = "{Binding Valor, StringFormat='{0:C2}'}" Grid.Column = "2" FontSize = "14" FontFamily = "Courier" HorizontalTextAlignment = "End" />

<Label Text = "=" FontSize = "14" Grid.Column = "3" FontFamily = "Courier" />

<Label Text = "{Binding SubTotal, StringFormat='{0:C2}'}" Grid.Column = "4" FontSize = "14" FontFamily = "Courier" HorizontalTextAlignment = "End" />

</Grid>

</StackLayout>

</ViewCell>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

```

Precisamos implementar a classe `servicosListagemViewModel` dentro de `viewModels/Atendimento` e o código inicial para ela

está apresentado a seguir. Observe que definimos o construtor parametrizado, que recebe o atendimento no qual serão manipulados seus serviços associados.

Este objeto é armazenado em uma propriedade, definida antes do construtor. Na tag `<ListView>`, a propriedade `ItemsSource` está ligada a uma propriedade chamada `Servicos`, que também temos implementada na listagem que se segue. Para terminar a declaração do `ListView`, precisamos implementar a propriedade `servicoSelecionado`, que está ligada à propriedade `selectedItem` do `Listview` e é este método que termina o código a seguir. Ele invoca uma mensagem que já registraremos. Como os itens recuperados não possuem um objeto associado a `Atendimento`, atribuímos o atendimento da visão que possui este objeto atualizado ao item que será visualizado. Observe a extensão para a classe `BaseViewModel`. Também declaramos nossa DAL.

```
namespace Capitulo05.ViewModels.Atendimentos
{
    public class ServicosListagemViewModel : BaseViewModel
    {
        private Atendimento Atendimento { get; set; }
        private AtendimentoItem AtendimentoItem { get; set; }
        public ObservableCollection<AtendimentoItem> ItensAtendimento { get; set; }
        private IDAL<AtendimentoItem> atendimentoItemDAL;

        public ServicosListagemViewModel(Atendimento atendimento)
        {
            this.Atendimento = atendimento;
            this.ItensAtendimento = new ObservableCollection<AtendimentoItem>();
            atendimentoItemDAL = new AtendimentoItemDAL(atendimento, DependencyService.Get<IDBPath>().GetDbPath());
        }

        public AtendimentoItem ServicoSelecionado
        {
            get { return this.AtendimentoItem; }
            set
            {
                if (value != null)
                {
                    this.AtendimentoItem = value;
                    this.AtendimentoItem.Atendimento = this.Atendimento;
                    MessagingCenter.Send<AtendimentoItem>(ServicoSelecionado, "Mostrar");
                }
            }
        }
    }
}
```

Como estamos atribuindo o `Atendimento` à propriedade `Atendimento` do objeto `AtendimentoItem`, que foi recuperado da base de dados, teremos um problema se, na visão de CRUD, o usuário retornar para a listagem e tentar remover este mesmo serviço, pois, o EF Core identificará uma mudança no objeto, em relação à sua última recuperação e tentará trabalhar sua inserção. No capítulo anterior, já vimos como evitar isso pela implementação que fizemos do `UpdateAsync()`. Vamos ajustar isso também no `DeleteAsync()`, inserindo a instrução a seguir, antes da invocação ao `Remove()`.

```
context.Entry(item).State = EntityState.Unchanged;
```

Em nossa visão de listagem de serviços para o atendimento, agora precisamos instanciar nosso ViewModel e defini-lo como contexto de ligação para a visão no construtor. Fazemos isso com o código apresentado a seguir

```
private ServicosListagemViewModel viewModel;  
  
private Atendimento Atendimento;  
  
  
public ServicosListagemView (Atendimento atendimento)  
{  
    InitializeComponent ();  
    this.Atendimento = atendimento;  
  
    BindingContext = viewModel = new ServicosListagemViewModel(atendimento);  
}
```

Dando sequência às ligações existentes no `ListView` exibido anteriormente, precisamos agora implementar a propriedade `SubTotalItens`, que apresentará a soma dos serviços registrados no atendimento atual. Podemos ver essa propriedade no próximo código. Veja o uso do método `Sum()`, que requer que adicionemos o `using System.Linq;` no início da classe.

```
public double SubTotalItens  
{  
    get  
    {  
        return this .ItensAtendimento.Sum(st => st.SubTotal);  
    }  
}
```

Precisamos agora realizar uma implementação importante. Nas visões de listagens anteriores, estamos trabalhando a remoção de um item da listagem por meio de `Context Actions`, lembra-se delas? Mas temos um problema, pois esta opção só poderá ser exibida para os itens enquanto o atendimento estiver aberto, ou seja, enquanto o carro não for entregue. Na tag `ViewCell`, dentro de `ListView.ItemTemplate`, observe a propriedade `BindingContextChanged`, que aponta para um método chamado `OnBindingContextChanged`. É nele que definiremos a lógica para a exibição ou não do `Context Actions` de remoção de itens. Veja o seu código adiante, que deve ser implementado na classe `ServicosListagemView`.

```
protected void OnBindingContextChanged ( object sender, EventArgs e)  
{  
    base.OnBindingContextChanged();  
  
    if ( this .Atendimento.EstaFinalizado)  
    {  
        ViewCell theViewCell = ((ViewCell)sender);  
        theViewCell.ContextActions.Clear();  
    }  
}
```

Ainda em relação às `Context Actions`, no menu que será exibido, temos ligações com a `ListView`, com um `Command` chamado `EliminarItemCommand` e passamos o objeto ao qual pertence a `Context Action` como parâmetro para o `Command` que será executado. Veja o código para ele na sequência. Lembre-se de que ele deve ser implementado na classe `ServicosListagemViewModel` e não se esqueça de, no construtor, invocar o método `RegistrarCommands()`.

```

public ICommand EliminarItemCommand { get ; set ; }

private void RegistrarCommands ()
{
    EliminarItemCommand = new Command<AtendimentoItem>((atendimentoItem) =>
    {
        MessagingCenter.Send<AtendimentoItem>(atendimentoItem, "Confirmação");
    });
}

```

Para que os itens apareçam na listagem, falta-nos o método responsável pela recuperação dos serviços registrados para o atendimento. Sendo assim, em nossa ViewModel de listagem, implemente o método a seguir. Observe que os objetos recuperados são atribuídos para a propriedade `servicos` de `Atendimento`. Quando recuperamos os atendimentos para a listagem, optamos por não realizar o `Include()` para `Servicos`, pois, no momento da listagem, estes dados associados não eram necessários. Agora, quando se acessa a listagem de serviços do atendimento, como recuperamos estes dados, já os atribuímos para o atendimento em foco. Veja, ao final do método, a atualização invocada para a propriedade `SubTotalItens`.

```

public async Task AtualizarItensAtendimentoAsync ()
{
    Atendimento.Servicos = await atendimentoItemDAL.GetAllAsync();
    ItensAtendimento.SincronizarColecoes(Atendimento.Servicos);
    OnPropertyChanged(nameof(SubTotalItens));
}

```

Como implementamos o `Command` para a remoção de um item de atendimento, vamos já implementar todo o processo para esta funcionalidade. Veja as implementações da sequência, que começa com o método que invocará a remoção, quando assim for confirmado pelo usuário. Depois, na sequência, estão as sobrescritas para os métodos `OnAppearing()` e `OnDisappearing()` da classe `ServicosListagemView`.

```

private async Task RemoverItemAsync (AtendimentoItem item)
{
    if ( await DisplayAlert ( "Confirmação" ,
        $"Confirma remoção de {item.Servico.Nome.ToUpper()}?" , "Yes" , "No" ) )
    {
        await this.viewModel.EliminarItemAtendimentoAsync(item);
        await DisplayAlert ( "Informação" , "Serviço removido com sucesso" , "Ok" );
    }
}

protected override void OnAppearing ()
{
    base.OnAppearing();
}

```

```

Device.BeginInvokeOnMainThread( async () =>
{
    await viewModel.AtualizarItensAtendimentoAsync();
});

if (listView.SelectedItem != null )
    listView.SelectedItem = null ;

MessagingCenter.Subscribe<AtendimentoItem>( this , "Confirmação" , async (item) => await RemoverItemRemoverItem (item) )

MessagingCenter.Subscribe<AtendimentoItem>( this , "Mostrar" , async (item) => { await Navigation.PushAsync( new
ServicosCRUDView(item, "Serviço Atendimento" )); });

}

protected override void OnDisappearing ()
{
    base.OnDisappearing();
    MessagingCenter.Unsubscribe<AtendimentoItem>( this , "Confirmação" );
    MessagingCenter.Unsubscribe<AtendimentoItem>( this , "Mostrar" );
}

```

Veja que, caso o usuário confirme o desejo de remover o item, ocorre uma chamada ao método `EliminarItemAtendimentoAsync()`, que já vamos implementar. Ele termina com o aviso à visão de que a propriedade `SubTotalItens` sofreu alteração, o que permitirá sua atualização após a remoção do item de serviço. Aproveitamos também para atualizar a propriedade `Servicos`, pois desta visão o usuário pode ir para a de CRUD e necessitar, nela, da propriedade atualizada, sem ter novamente que a popular da base. Lembre-se de que este código deve ser implementado na classe `ServicosListagemViewModel`.

```

public async Task EliminarItemAtendimentoAsync (AtendimentoItem atendimentoItem)
{
    await atendimentoItemDAL.DeleteAsync(atendimentoItem);

    this.ItensAtendimento.Remove(atendimentoItem);
    this.Atendimento.Servicos.Remove(atendimentoItem);
    OnPropertyChanged(nameof(SubTotalItens));
}

```

Se você estiver curioso, comente o `MessagingCenter` para o `ServicosCRUDView` no `OnAppearing()` e pode executar sua aplicação e acessar a visão de serviços de um atendimento que já esteja gravado. A visão funcionará, mas ainda não temos nenhum item registrado nela. Faremos isso a partir da próxima seção.

## 7.4 A pesquisa de serviços para o atendimento

Nossa visão de serviço precisará ter controles que possibilitem ao usuário selecionar um serviço, o qual será localizado tal qual fizemos para a informação do cliente para o atendimento. Com o serviço selecionado, o usuário

precisará informar a quantidade e valor cobrado para ele. Precisaremos garantir que um serviço não seja inserido em duplicidade e que, uma vez inserido, ele possa ser removido ou alterado. Vamos lá?

Nosso primeiro passo será desenhar a parte que conterá o serviço a ser registrado no atendimento. Para isso, insira uma Content Page chamada ServicosCRUDView na pasta Views/Atendimentos , com o código apresentado na sequência. Veja que temos um StackLayout com um Grid e, neste grid, inserimos os controles que serão referentes ao nome do serviço e acesso à visão de pesquisa. No segundo Label , observe que há o binding com uma propriedade chamada ServicoNome e no Image temos um Command ligado chamado PesquisarCommand .

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

    x:Class = "Capitulo05.Views.Atendimentos.ServicosCRUDView" >

    <ContentPage.Content>
        <StackLayout>

            <Grid HorizontalOptions = "Fill" VerticalOptions = "Start" Margin = "5, 5, 0, 0" Padding = "10, 10, 0, 0" >
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width = "Auto" />
                    <ColumnDefinition Width = "*" />
                    <ColumnDefinition Width = "Auto" />
                </Grid.ColumnDefinitions>
                <Grid.RowDefinitions>
                    <RowDefinition Height = "Auto" />
                    <RowDefinition Height = "Auto" />
                </Grid.RowDefinitions>
            </Grid>
        </ContentPage.Content>
    </ContentPage>
<Label Text = "Serviço:" Grid.Column = "0" Grid.ColumnSpan = "2" Grid.Row = "0" FontAttributes = "Bold" />
<Label Text = "{Binding ServicoNome}" Grid.Column = "0" Grid.ColumnSpan = "2" Grid.Row = "1" TextColor = "Blue" />
```

```

<Image Source = "consultar.png" HeightRequest = "38" WidthRequest = "38" Grid.Column = "2" Grid.Row = "0" Grid.RowSpan = "2" >

<Image.GestureRecognizers>

<TapGestureRecognizer Command = "{Binding PesquisarCommand}" />

</Image.GestureRecognizers>

</Image>

</Grid>

</StackLayout>

</ContentPage.Content>

</ContentPage>

```

Como comentado anteriormente, temos propriedades e métodos ligados e, como estamos fazendo uso do MVVM, precisamos de uma classe que representará a camada ViewModel, onde teremos estes elementos de ligação. Desta maneira, na pasta `ViewModels/Atendimentos`, crie uma classe chamada `ServicosCRUDViewModel` e a implemente, inicialmente, com o código apresentado na sequência. Note a definição dos objetos privados para o atendimento a que se referirá a visão atual e o item de atendimento que está sendo trabalhado na visão. Veja o construtor que inicializa o DAL, que está declarado também na classe.

```

namespace Capitulo05.ViewModels.Atendimentos
{
    public class ServicosCRUDViewModel : BaseViewModel
    {
        private IDAL<AtendimentoItem> itemDAL;
        private Atendimento Atendimento { get; set; }
        private AtendimentoItem AtendimentoItem { get; set; }
        public ICommand PesquisarCommand { get; set; }

        public ServicosCRUDViewModel(AtendimentoItem atendimentoItem)
        {
            itemDAL = new AtendimentoItemDAL(atendimentoItem.Atendimento, DependencyService.Get<IDBPath>()
                .GetDbPath());
            this.Atendimento = atendimentoItem.Atendimento;
            this.AtendimentoItem = atendimentoItem;
        }

        public string ServicoNome
        {
            get { return this.AtendimentoItem.Servico == null ? "Localize o serviço" : this.AtendimentoItem.Servico.Nome; }
            set
            {
                OnPropertyChanged();
            }
        }
    }
}

```

```
 }
}
```

Antes de implementarmos o código para o `Command PesquisarCommand`, vamos criar e implementar a visão que será responsável pela pesquisa de serviços. Seguindo o mesmo que fizemos para o cliente do atendimento, na pasta `Views/Servicos`, crie uma `ContentPage` chamada `PesquisarView` e a implemente como o código a seguir. Veja que é simples e semelhante ao que implementamos para a pesquisa de cliente. Temos um `StackLayout`, um `SearchBar` e um `ListView`. Observe que nos dois últimos temos bindings que deverão ser implementados na `ViewModel` para esta visão. Lembre-se de ajustar `HeightRequest` do `searchBar` de acordo com a necessidade de seu dispositivo.

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class = "Capitulo05.Views.Servicos.PesquisarView"
    xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOS;assembly=Xamarin.Forms.Core"
    ios:Page.UseSafeArea = "true"
    Title = "Pesquisa por serviço" >

    <ContentPage.Content>
        <StackLayout Orientation = "Vertical" HorizontalOptions = "FillAndExpand" Padding = "5,20,5,0" >
            <SearchBar x:Name = "searchBar" HeightRequest = "25" Placeholder = "Digite o nome do serviço..." TextColor = "Black" SearchCommand = "{Binding PesquisarCommand}" SearchCommandParameter = "{Binding Source={x:Reference searchBar}, Path=Text}" />
            <ListView x:Name = "lvServicos" HasUnevenRows = "True" ItemsSource = "{Binding ServicosEncontrados}" SelectedItem = "{Binding ServicoLocalizadoSelecionado}" Margin = "10, 5, 0, 0" >
                <ListView.ItemTemplate>
                    <DataTemplate>
                        <ViewCell>
                            <StackLayout Orientation = "Vertical" >
                                <Label Text = "{Binding ServicoID}" TextColor = "Blue" IsVisible = "False" />
                                <Label Text = "{Binding Nome}" TextColor = "Blue" FontSize = "Large" />
                            </StackLayout>
                        </ViewCell>
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
        </StackLayout>
    </ContentPage.Content>

```

```

    </StackLayout>

    </ViewCell>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

</StackLayout>

</ContentPage.Content>

</ContentPage>

```

Como já temos a visão, precisamos do código necessário para a ViewModel. Sendo assim, na pasta `ViewModels/Servicos`, crie uma nova classe, chamada `PesquisarViewModel`, e nela implemente o código apresentado na sequência. Para poder explicar com calma, a implementação inicial é apenas a estrutura básica da classe. Fique atento aos `using`s que precisam ser inseridos conforme o código vai evoluindo.

```

namespace Capitulo05.ViewModels.Servicos
{
    public class PesquisarViewModel
    {
        private IDAL<Servico> servicoDAL;
        public ObservableCollection<Servico> ServicosEncontrados { get; set; }
        public ICommand PesquisarCommand { get; set; }
    }
}

```

Com a estrutura básica e os objetos que utilizaremos na classe declarados, vamos implementar nosso construtor no código a seguir. Veja que instanciamos o DAL, inicializamos a coleção de serviços e implementamos o comportamento para o `command` que realizará a pesquisa com base no valor digitado pelo usuário na visão. Veja, também no código, o método `RegistrarCommands()`.

```

public PesquisarViewModel ()
{
    servicoDAL = new ServicoDAL(DependencyService.Get<IDBPath>().GetDbPath());

    ServicosEncontrados = new ObservableCollection<Servico>();
    RegistrarCommands();
}

private void RegistrarCommands ()
{
    PesquisarCommand = new Command< string >((servico) =>
    {
        ServicosEncontrados.Clear();
    });
}

```

```

        var serviciosEncontrados = servicioDAL.GetStartsWithByFieldAsync( "Nome" , servicio).Result;
        foreach ( var c in serviciosEncontrados)
        {
            ServicosEncontrados.Add(c);
        }
    });
}

```

Como você deve ter notado que invocamos o método `GetStartsWithByFieldAsync()` em nosso objeto DAL. Já temos este método implementado na classe `DALBase`. Vamos voltar para nossa ViewModel. Precisamos concluir-la, implementando a propriedade `ServicoLocalizadoSelecionado`, como apresentada na sequência. Note que nessa propriedade precisamos fazer uso do `MessagingCenter`.

```

private Servico servicioLocalizadoSelecionado;
public Servico ServicoLocalizadoSelecionado
{
    get { return servicioLocalizadoSelecionado; }

    set
    {
        if ( value != null )
        {
            servicioLocalizadoSelecionado = value ;
            MessagingCenter.Send<Servico>(servicioLocalizadoSelecionado, "ServicoSelecionado" );
        }
    }
}

```

No código anterior, a mensagem enviada para o `MessagingCenter` envia o serviço que for selecionado pelo usuário no resultado da pesquisa. Desta maneira, na ViewModel de registro de serviços para o atendimento, precisamos de uma propriedade que terá esta informação, pois é dela que sairá o nome do serviço e seu valor inicial a ser cobrado do cliente. Na classe `ServicosCRUDViewModel`, em `viewModels/Atendimentos`, implemente a propriedade a seguir. Observe que invocamos a mudança de valor na propriedade `ServicoNome`, que já temos implementada.

```

public Servico Servico
{
    get { return this .AtendimentoItem.Servico; }

    set
    {
        this .AtendimentoItem.Servico = value ;
        this .AtendimentoItem.Quantidade = ( value != null ) ? 1 : 0;
        this .AtendimentoItem.Valor = ( value != null ) ? value .Valor : 0;
        OnPropertyChanged(nameof(ServicoNome));
    }
}

```

Precisamos começar agora a implementação do code-behind para a visão `PesquisarView`, que terá, também, o registro e cancelamento de mensagem no `MessagingCenter` que foi utilizado anteriormente. Veja o pequeno código completo da estrutura básica da classe `PesquisarView` de serviços na sequência. Veja a declaração dos objetos que serão utilizados e a inicialização deles no construtor. No registro da mensagem, note o desempilhamento da visão atual na navegação e atribuição do serviço selecionado na ViewModel chamadora. Da mesma maneira que fizemos para a pesquisa de clientes, no capítulo anterior, precisamos agora pensar em como vamos atualizar a propriedade `Servico` na visão de registro de serviços para o atendimento. Esta atualização precisará ser realizada na classe `PesquisarView` de Serviços e, como explicado anteriormente, vamos criar uma propriedade estática em `PesquisarView`. Veja que no construtor inicializamos o objeto com `null` e atribuímos um valor a ele quando um serviço for selecionado.

```
namespace Capitulo05.Views.Servicos
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class PesquisarView : ContentPage
    {
        private PesquisarViewModel viewModel;
        public static Servico ServicoSelecionado { get; set; }

        public PesquisarView()
        {
            InitializeComponent();
            viewModel = new PesquisarViewModel();
            BindingContext = viewModel;
            ServicoSelecionado = null;
        }

        protected override void OnAppearing()
        {
            base.OnAppearing();
            MessagingCenter.Subscribe<Servico>(this, "ServicoSelecionado",
                (servico) =>
            {
                ServicoSelecionado = servico;
                Navigation.PopAsync();
            });
        }

        protected override void OnDisappearing()
        {
            base.OnDisappearing();
            MessagingCenter.Unsubscribe<Servico>(this, "ServicoSelecionado");
        }
    }
}
```

Agora, vamos retomar a visão que registrará os itens de serviço para o atendimento. Vamos para o code-behind da visão de itens de serviços para o atendimento, à classe `servicosCRUDView`, dentro de `Views/Atendimentos`. A princípio, implementaremos o básico para ela, tal qual o código adiante.

```
namespace Capitulo05.Views.Atendimentos
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
```

```

public partial class ServicosCRUDView : ContentPage
{
    private ServicosCRUDViewModel crudViewModel;

    public ServicosCRUDView ()
    {
        InitializeComponent ();
    }

    public ServicosCRUDView(AtendimentoItem item, string title) : this()
    {
        this.Title = title;
        this.crudViewModel = new ServicosCRUDViewModel(item);
        this.BindingContext = this.crudViewModel;
    }
}
}

```

Vamos resgatar a implementação dessa visão, pois só temos seu XAML implementado. Lembra que nela temos uma imagem que, ao ser clicada, abrirá a visão de pesquisa de serviços que acabamos de implementar? Pois bem, vamos a ela agora. Logo poderemos testar nossa implementação. Ao final da classe `ServicosCRUDViewModel` de `ViewModels/Atendimentos`, crie o método `RegistrarCommands()`, tal qual o apresento na sequência. Lembre-se de invocá-lo em seu construtor.

```

private void RegistrarCommands ()
{
    PesquisarCommand = new Command(() =>
    {
        MessagingCenter.Send<AtendimentoItem>(AtendimentoItem, "MostrarPesquisarServico");
    });
}

```

Veja que invocamos a mensagem `MostrarPesquisarServico` ao `MessagingCenter`. Já sabe o que fazer? Isso mesmo. Precisamos registrar esta mensagem e cancelar este registro no code-behind da visão `ServicosCRUDView`, de `Views/Atendimentos`. A seguir, veja que já buscamos o valor para o serviço que o usuário selecionou na pesquisa, caso exista. Já quando a tela for fechada, atribuiremos `null` a ele.

```

protected override void OnAppearing ()
{
    base.OnAppearing();

    if (PesquisarView.ServicoSelecionado != null)
        crudViewModel.Servico = PesquisarView.ServicoSelecionado;

    MessagingCenter.Subscribe<AtendimentoItem>( this , "MostrarPesquisarServico" , async (item) => { await
Navigation.PushAsync( new PesquisarView()); });
}

protected override void OnDisappearing ()

```

```

{
    base.OnDisappearing();

    PesquisarView.ServicoSelecionado = null;

    MessagingCenter.Unsubscribe<AtendimentoItem>( this , "MostrarPesquisarServico" );
    MessagingCenter.Unsubscribe<AtendimentoItem>( this , "Confirmação" );
}

```

Muito bem, para podermos testar nossa aplicação, precisamos oferecer ao usuário a possibilidade para realizar essa operação. No XAML de `ListagemServicosView`, antes da tag `<ContentPage.Content>`, insira o código a seguir, nosso já conhecido `Toolbar`. Observe o uso de uma imagem, você precisará dela em seus projetos específicos de plataforma. Também precisaremos implementar o `NovoCommand` na classe `ListagemServicosViewModel`, que tem seu código após o XAML, sendo o primeiro trecho de código uma declaração na classe, e o segundo devendo ser implementado no método `RegistrarCommands()`, já existente. Atente ao `Command` para a condição de exibição para o botão da barra de tarefas.

```

<!-- XAML -->
<ContentPage.ToolbarItems>

<ToolbarItem Icon = "plus.png" Command = "{Binding NovoCommand}" />

</ContentPage.ToolbarItems>

// Antes do construtor

public ICommand NovoCommand { get ; set ; }

// Dentro do RegistrarCommands()

NovoCommand = new Command(() =>
{
    MessagingCenter.Send<AtendimentoItem>( new AtendimentoItem(), "Mostrar" );
}, () =>
{
    return ! this .Atendimento.EstaFinalizado;
});

```

Como já sabemos, precisamos sobrescrever os métodos `OnAppearing()` e `OnDisappearing()` na classe `ServicosCRUDView`, tal qual o código a seguir, de maneira respectiva.

```

MessagingCenter.Subscribe<AtendimentoItem>( this , "Mostrar" , async (item) => { await Navigation.PushAsync( new
ServicosCRUDView(item, "Novo Atendimento" )); });

```

```

MessagingCenter.Unsubscribe<AtendimentoItem>( this , "Mostrar" );

```

Você pode testar sua aplicação. Acesse atendimentos, selecione um já inserido, pressione o botão `Serviços` e na listagem pressione o botão de adição no topo da visão. Quando a visão aparecer, pressione a opção de pesquisar por serviços, tal qual tínhamos para pesquisas de clientes. Veja, nas figuras a seguir, a visão com acesso à pesquisa de

serviços e, em seguida, a visão que efetiva a pesquisa de serviços.

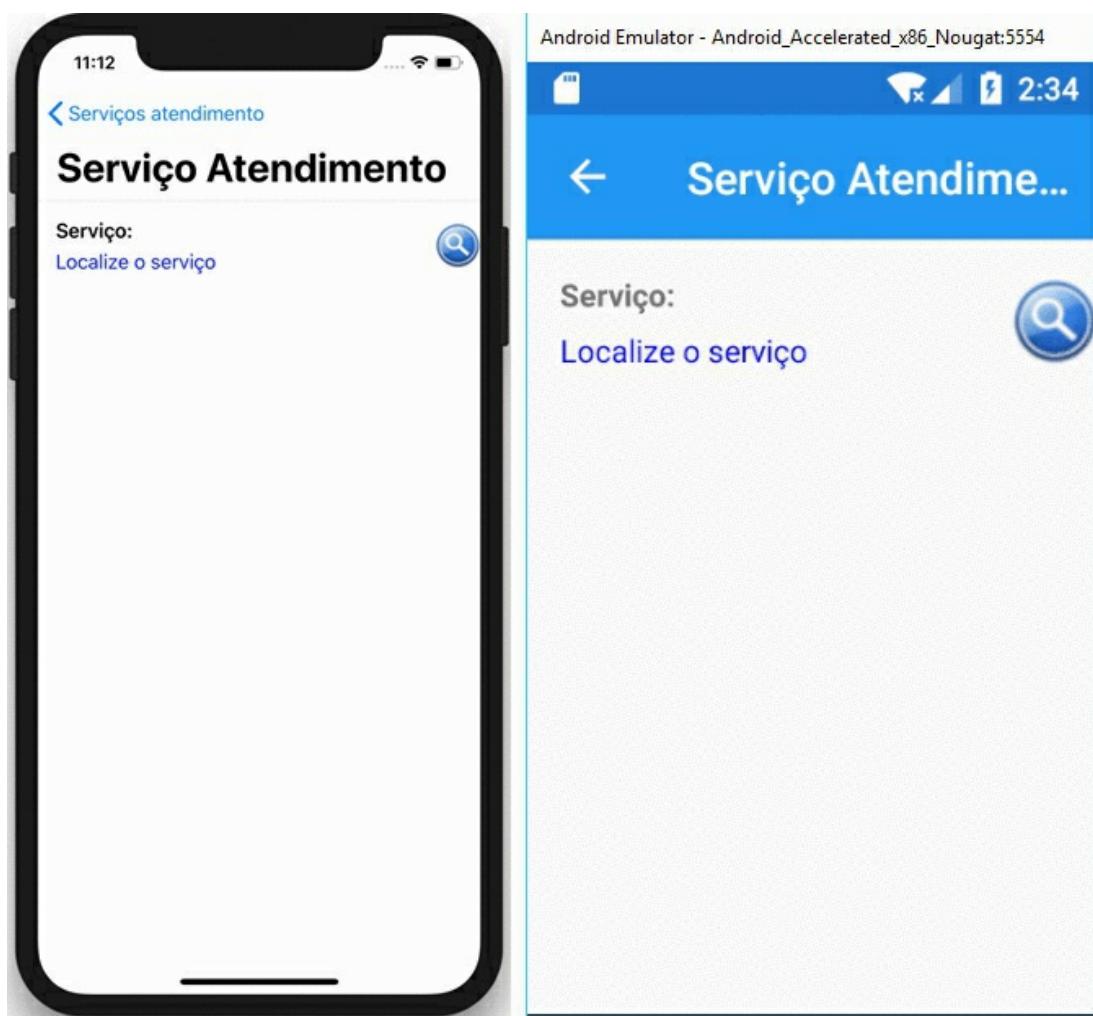


Figura 7.2: Visão de serviços com dado do serviço selecionado

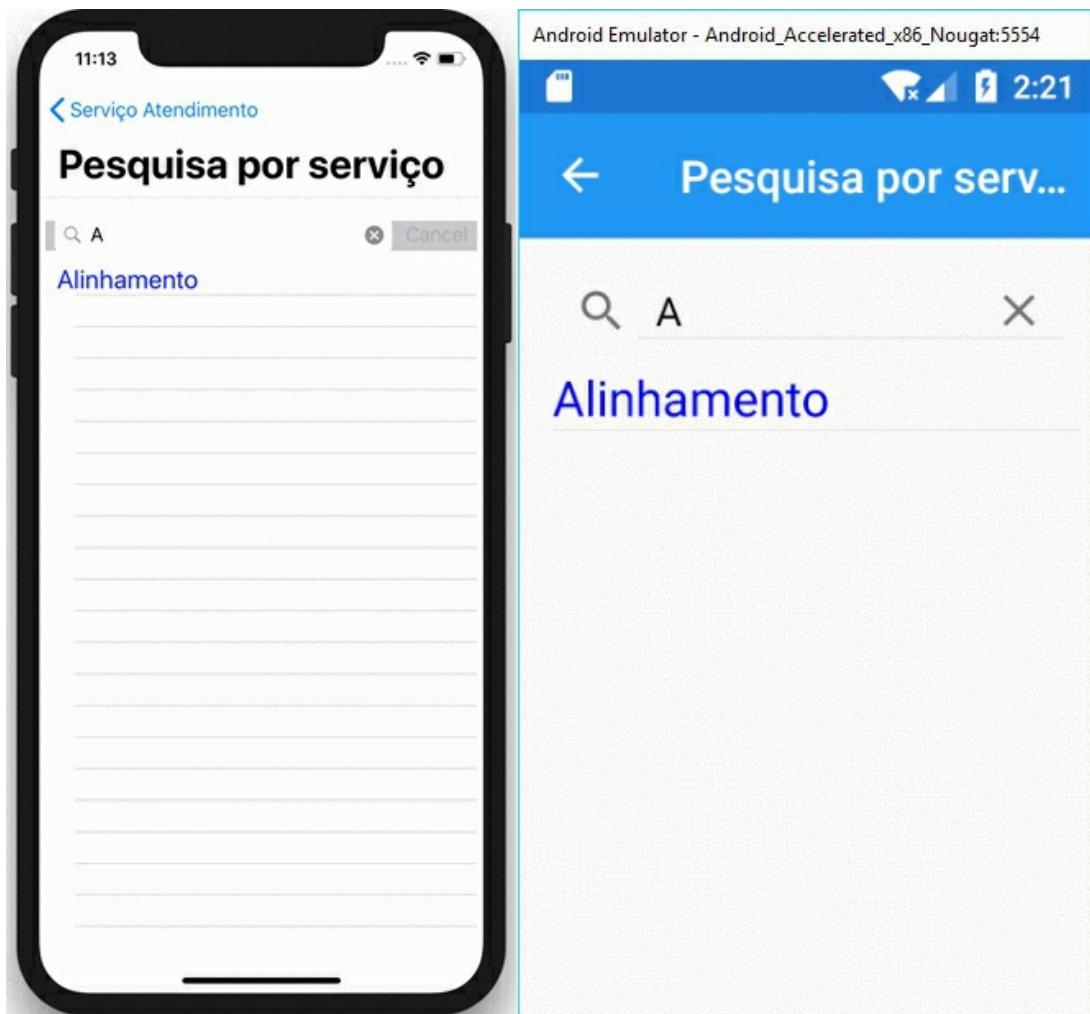


Figura 7.3: Pesquisa de serviços para o atendimento

## 7.5 Registro dos serviços a serem realizados

Dando continuidade ao processo de informar quais serviços serão registrados no atendimento, precisamos agora alterar nossa visão para que permita o registro e exibição de dados do serviço, como quantidade e valor. Para isso, na visão `ServicosCRUDView` de `Views/Atendimentos`, após o `Grid` do nome do serviço, implemente o seguinte código XAML. Veja que temos três propriedades ligadas, a quantidade, valor e subtotal. Precisaremos criá-las na `ViewModel`. Da mesma maneira, podemos verificar a existência da ligação do botão que realizará a gravação do serviço da visão (pelo `Binding`) com o `Command GravarItemCommand`. Outro detalhe importante é o estado de habilitado ou não para o `TableView`, que está ligando a propriedade `IsEnabled` com `HabilitaAlteracao`. Observe o uso de `Keyboard` nos `EntryCell`.

```
<TableView Intent = "Form" IsEnabled = "{Binding HabilitaAlteracao}" >

<TableRoot>

<TableSection Title = "Dados do Serviço" >
```

```

<EntryCell Label = "Quantidade:" Text = "{Binding Quantidade}" Keyboard = "Numeric" ></EntryCell>

<EntryCell Label = "Valor:" Text = "{Binding Valor}" Keyboard = "Numeric" ></EntryCell>

<EntryCell Label = "Subtotal:" Text = "{Binding SubTotal}" IsEnabled = "False" ></EntryCell>

<ViewCell>

<StackLayout>

<Button Text = "Gravar Item" FontAttributes = "Bold" HorizontalOptions = "Center" Command = "{Binding GravarItemCommand}" />

</StackLayout>

</ViewCell>

</TableSection>

</TableRoot>

</TableView>

```

Vamos agora criar as propriedades que receberão e informarão dados de cada item registrado na visão. Na classe `ServicosCRUDViewModel` de `ViewModels/Atendimentos`, insira o código a seguir. Observe a lógica para informação dos valores na visão (método `get`) e as instruções executadas no recebimento do valor, que, além de atribuí-lo à propriedade do item que está sendo inserido, invoca os métodos `OnPropertyChanged()` para informar a visão em cujo modelo houve alteração de que ela precisa ser atualizada. Observe que as propriedades `Quantidade` e `Valor` estão definidas como `string`. É o mesmo procedimento que utilizamos para o registro de `Serviços`, no capítulo 5, para registrar alterações quando o usuário apagar todo o conteúdo do controle na visão.

```

private string quantidade;

public string Quantidade
{
    get { return quantidade; }

    set
    {
        this .quantidade = value ;

        this .AtendimentoItem.Quantidade = string .IsNullOrEmpty( value ) ? 0 : Convert.ToInt16( value ); ;
        OnPropertyChanged();
        OnPropertyChanged(nameof(SubTotal));
    }
}

private string valor;

public string Valor

```

```

{
    get { return valor; }

    set
    {
        this.valor = value;

        this.AtendimentoItem.Valor = string.IsNullOrEmpty( value ) ? 0 : Convert.ToDouble( value );
        OnPropertyChanged();
        OnPropertyChanged(nameof(SubTotal));
    }
}

public double SubTotal
{
    get { return AtendimentoItem.Valor * AtendimentoItem.Quantidade; }

    set
    {
        OnPropertyChanged();
    }
}

```

Com a criação das propriedades anteriores, podemos alterar a propriedade `servico` para fazer uso das novas propriedades, pois ela estava atribuindo valores ao objeto do item recuperado. Veja o novo código na sequência. Desta maneira, mantemos a lógica de atribuição de valores nas novas propriedades.

```

public Servico Servico
{
    get { return this.AtendimentoItem.Servico; }

    set
    {
        this.AtendimentoItem.Servico = value;

        Quantidade = ( value != null ) ? "1" : "0";

        Valor = ( value != null ) ? Convert.ToString( value.Valor ) : "0";
        OnPropertyChanged(nameof(ServicoNome));
    }
}

```

É importante lembrarmos que, quando formos alterar um item, ele já possuirá dados para `valor` e `quantidade`, mas estes valores são numéricos, e precisamos transformá-los em `string`. Desta maneira, insira as instruções a seguir no construtor, antes da invocação do `RegistrarCommands()`.

```

this.valor = string.Format( "{0:N}" , atendimentoItem.Valor);

this.quantidade = string.Format( "{0}" , atendimentoItem.Quantidade);

```

Agora, precisamos implementar o `Command` que será responsável pela gravação do serviço. Na sequência, apresento sua declaração, seguido do código que deve ser inserido no método `RegistrarCommands()`. No `Command`, veja que

atribuímos ao item atual da visão os dados referentes ao atendimento, para garantir a associação. Em seguida, fazemos uma verificação que garante que o serviço que se deseja gravar, se for novo, não está já registrado no atendimento. A variável `countServicosItem` auxilia quando se altera o valor de uma ou mais propriedades para um serviço que já existe na relação.

Na sequência, o serviço é inserido ou atualizado por meio da invocação ao método `UpdateAsync()`. Veja que, após a mensagem de sucesso, o item que estará relacionado com a visão é reinstantiado. Terminando o `command`, temos a lógica que tornará o botão habilitado para que possa ser clicado. Veja que a definição do `Command` tem muitas instruções. O que acha de implementar um método que se responsabilize por isso e apenas o referenciar na declaração? Já fizemos isso antes, para `Creditos` e `Servicos`.

```
//Antes do construtor

public ICommand GravarItemCommand { get ; set ; }

// Dentro do RegistrarCommands()

GravarItemCommand = new Command( async () =>
{
    AtendimentoItem.Atendimento = Atendimento;
    AtendimentoItem.AtendimentoID = Atendimento.AtendimentoID;

    if (AtendimentoItem.AtendimentoItemID != null )
        AtendimentoItem.NotificarListView = true ;

    var countServicosItem = this .Atendimento.Servicos.Where(s => s.ServicoID == AtendimentoItem.ServicoID).Count();

    if (countServicosItem > 1)
    {
        MessagingCenter.Send< string >( "Serviço já existe no atendimento." , "InformacaoCRUD" );
        return ;
    }

    await itemDAL.UpdateAsync(AtendimentoItem, AtendimentoItem.AtendimentoItemID);
    MessagingCenter.Send< string >( "Atualização realizada com sucesso." , "InformacaoCRUD" );
    this .AtendimentoItem = new AtendimentoItem();
    this .Serviço = null ;
}, () =>
{
    return ( this .AtendimentoItem.Servico != null && this .AtendimentoItem.Quantidade > 0 && this .AtendimentoItem.Valor > 0);
});
```

Com a lógica de habilitação para o botão de gravar o serviço implementada, precisamos invocar o método que a implementa, sempre que houver mudança em uma das propriedades relacionadas a ela. Mas no modelo temos uma propriedade que é invocada sempre que as demais se alteram, o que nos permite focar este comportamento nela.

Estou falando da propriedade `subtotal`. Implemente o código a seguir no final do método `set()` dela.

```
((Command)GravarItemCommand).ChangeCanExecute();
```

Observe que, no `GravarItemCommand`, temos o envio de uma mensagem para o Messaging Center. Precisamos registrar esta mensagem na visão. Veja o primeiro techo de código para o `onAppearing()` e o segundo para o `onDisappearing()`, ambos na classe `ServicosCRUDView`.

```
MessagingCenter.Subscribe< string >( this , "InformacaoCRUD" , async (msg) => { await DisplayAlert ( "Informação" , msg , "ok" )};
```

```
MessagingCenter.Unsubscribe< string >( this , "InformacaoCRUD" );
```

Com estas implementações realizadas, podemos testar novamente nossa aplicação. Execute-a, pesquise um serviço e veja as propriedades sendo atualizadas, tal qual pode ser comprovado pela figura a seguir.

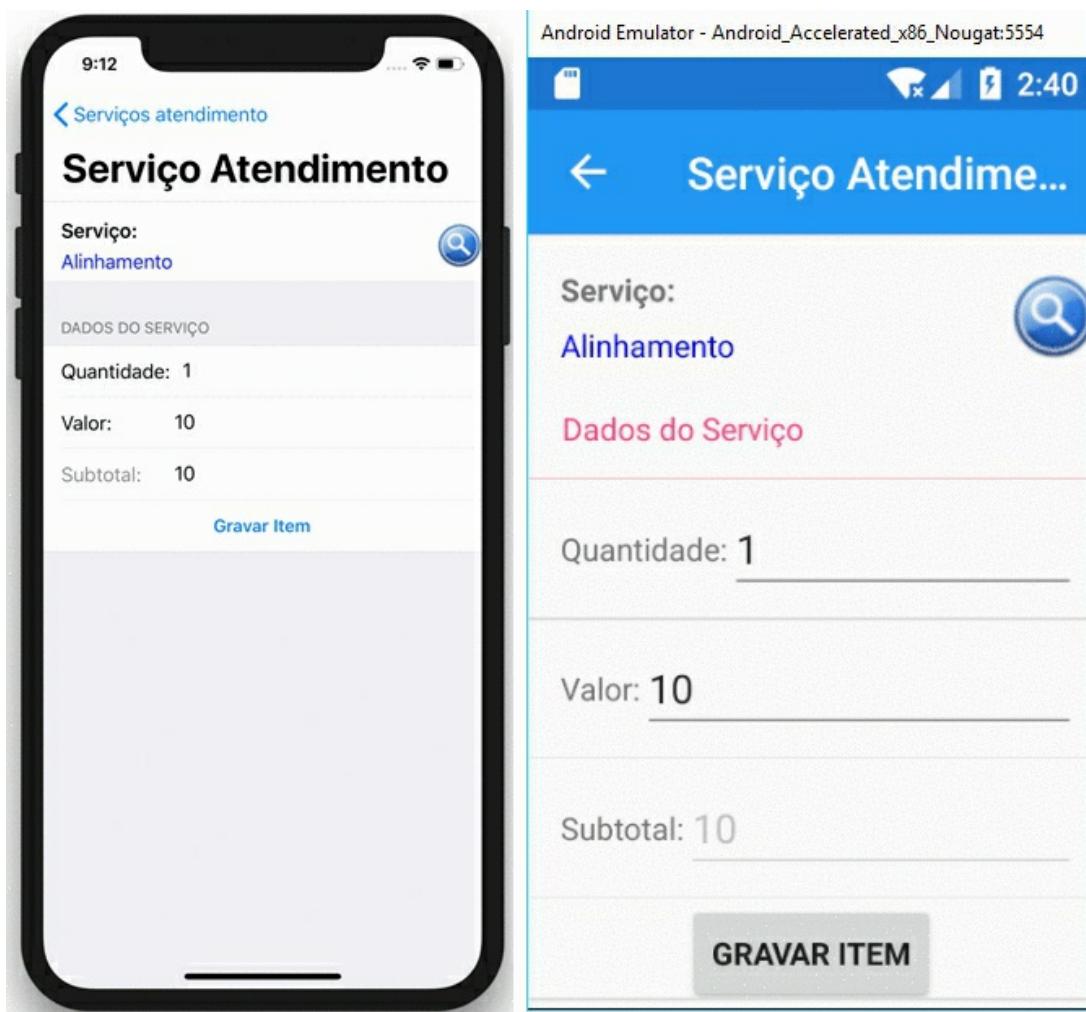


Figura 7.4: Visão com dados relacionados ao serviço a ser registrado

Após inserir seus serviços ao atendimento, volte para a listagem. Ela deverá estar semelhante ao apresentado na figura a seguir.

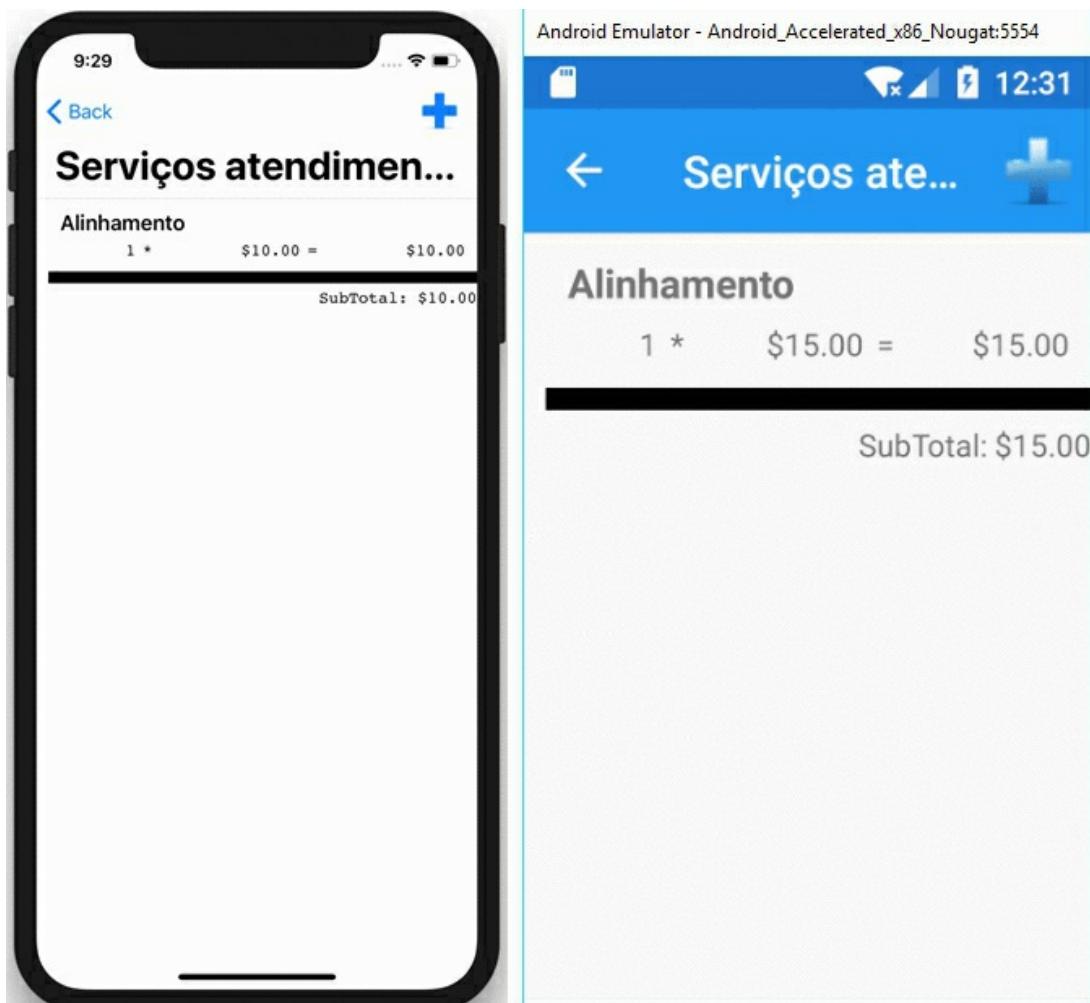


Figura 7.5: Visão com os serviços registrados para o atendimento

## 7.6 Conclusão

Chegamos ao final deste capítulo. Aprendemos associação entre classes e mapeamos essa associação para a base de dados. Vimos como recuperar dados associados pelo EF Core. Trabalhamos a habilitação de determinados controles de acordo a um estado retornado por propriedades, removemos `Action Contexts` por meio de código C#, de maneira dinâmica e configuramos o rodapé de um `ListView`.

Nos dois próximos capítulos concluirímos a etapa de associação de um atendimento com fotos que poderão ser tiradas de um carro, ou obtidas diretamente do álbum do dispositivo. Teremos como base o projeto que concluímos neste capítulo.

## C APÍTULO 8

# Uso de câmera e álbum

No capítulo anterior, vimos o mapeamento de classes associadas por meio de uma propriedade para uma base de dados que representa uma coleção, onde registramos um atendimento a um veículo e a este atendimento tínhamos associados serviços a serem realizados.

O objetivo deste novo capítulo está em implementarmos outra associação baseada em coleções, cujo básico já criamos no capítulo anterior. Mas para esta coleção faremos uso de fotos, que o usuário poderá obter utilizando a câmera do dispositivo ou selecionar diretamente do álbum. Para esta interação faremos uso de um componente chamando Media Plugin. Não armazenaremos a imagem na base, mas sim seu endereço no dispositivo.

Veremos também um recurso novo, vindos do Xamarin Forms 3, que é o uso de CSS e um novo layout, chamado **FlexLayout**.

Reforço que toda implementação realizada aqui tem como base o capítulo anterior e servirá como base para o próximo. Muita coisa boa, nova e útil.

## 8.1 Registro das fotos do veículo

Teremos agora um trabalho semelhante ao que fizemos no capítulo anterior, pois permitiremos a adição de várias fotos ao atendimento em foco. No capítulo anterior, nós criamos a visão `FotosListagemView`, que exibirá as fotos do veículo e, seguindo nosso padrão, o acesso ao registro de fotos se dará por meio desta visão, fazendo uso da nossa já conhecida `Toolbar`. Vamos então inserir as tags seguintes, que devem estar antes da tag `<ContentPage.Content>`. Atente para a imagem e para o `Command` que precisaremos implementar.

```
<ContentPage.ToolbarItems>

<ToolbarItem Icon = "plus.png" Command = "{Binding NovoCommand}" />

</ContentPage.ToolbarItems>
```

Vamos para a parte burocrática de nossa implementação. Sabemos que, se temos um `Command`, estamos usando MVVM e, se estamos usando MVVM, precisamos de uma classe que representa a camada ViewModel. Sendo assim, na pasta `ViewModels/Atendimento`, crie a classe `FotosListagemViewModel` e, inicialmente, implemente o seguinte código para ela. Neste código estamos prevendo o recebimento, pelo construtor, do atendimento selecionado para visualização das fotos. Em demais métodos da classe precisaremos acessar a foto, representada pelo objeto de `AtendimentoFoto`, que estamos declarando no início da classe. Instanciamos também no construtor o DAL para `AtendimentoFoto`. Temos também a declaração para o `Command` que implementamos a invocação na listagem anterior. O `Command` é registrado pelo método `RegistrarCommands()`, que está apresentado ao final da listagem e que é invocado ao final do construtor.

```
namespace Capitulo05.ViewModels.Atendimentos

{
    public class FotosListagemViewModel
    {
        private Atendimento Atendimento { get; set; }
        private AtendimentoFoto AtendimentoFoto { get; set; }
        public ICommand NovoCommand { get; set; }
        private IDAL<AtendimentoFoto> atendimentoFotoDAL;

        public FotosListagemViewModel(Atendimento atendimento)
```

```

    {
        this.Atendimento = atendimento;
        atendimentoFotoDAL = new AtendimentoFotoDAL(atendimento, DependencyService.Get<IDBPath>().GetDbPath());
        RegistrarCommands();
    }
}

private void RegistrarCommands()
{
    NovoCommand = new Command(() =>
    {
        var atendimentoFoto = new AtendimentoFoto() { Atendimento = this.Atendimento, AtendimentoID =
this.Atendimento.AtendimentoID };
        MessagingCenter.Send<AtendimentoFoto>(atendimentoFoto, "Mostrar");
    }, () =>
    {
        return !this.Atendimento.EstaFinalizado;
    });
}
}

```

Observe que na execução do `Command` está sendo invocada uma mensagem; já até sabíamos que isso ocorreria, não é? Então, já podemos imaginar que precisaremos implementar o registro e cancelamento dela na visão `FotosListagemView`, nos métodos já conhecidos `OnAppearing()` e `OnDisappearing()`, tal qual segue na listagem.

```

// OnAppearing()

MessagingCenter.Subscribe<AtendimentoFoto>(this , "Mostrar" , async (foto) => { await Navigation.PushAsync( new
FotosCRUDView(foto, "Foto Atendimento" )); };);

// OnDisappearing()

MessagingCenter.Unsubscribe<AtendimentoFoto>(this , "Mostrar" );

```

Ao implementarmos o código anterior, notaremos um erro na instanciação de `FotosCRUDView`, o que é até esperado, pois não temos nossa visão ainda implementada. Vamos fazer isso agora. Na pasta `Views/Atendimentos`, crie um Content Page chamado `FotosCRUDView`, inicialmente com o código apresentado na sequência para o XAML.

```

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"

xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

x:Class = "Capitulo05.Views.Atendimentos.FotosCRUDView" >

<ContentPage.Content>

<ScrollView>

<StackLayout>

```

```

</StackLayout>

</ScrollView>

</ContentPage.Content>

</ContentPage>

```

Só isso não resolverá ainda nosso erro, pois na instanciação chamamos um construtor que recebe dois argumentos. Precisamos implementá-lo em nossa classe de code-behind para a visão. Veja o código inicial dessa classe na listagem a seguir. Com isso, teremos resolvido o erro de sintaxe comentado anteriormente.

```

namespace Capitulo05.Views.Atendimentos
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class FotosCRUDView : ContentPage
    {
        public FotosCRUDView ()
        {
            InitializeComponent ();
        }

        public FotosCRUDView(AtendimentoFoto foto, string title) : this()
        {
            this.Title = title;
        }
    }
}

```

Antes de podermos testar o que fizemos até aqui, precisamos atribuir em nossa `FotosListagemView` a ligação de contexto com a `FotosListagemViewModel`. Desta maneira, insira o primeiro código da listagem a seguir antes do construtor, e o segundo dentro do construtor. Como implementamos no capítulo anterior o acesso à visão de listagem das fotos, após as seguintes implementações já podemos executar nossa aplicação e ver o resultado. Acesse um atendimento, pressione o botão de fotos. Você receberá uma visão vazia, o que é normal, pois não temos nada. Pressione o botão de adição de foto e você terá uma nova visão, também vazia, pois ainda não implementamos os controles para o registro de fotos. Faremos isso na sequência.

```

private FotosListagemViewModel viewModel;

BindingContext = viewModel = new FotosListagemViewModel(atendimento);

```

## 8.2 Visão para registro das fotos

Vamos agora partir para a implementação de nossa visão de registro de fotos, a `FotosCRUDView`, criada no início deste capítulo. Nela faremos uso de CSS, recurso trazido pela versão 3 do Xamarin Forms. Começaremos nossa implementação com a criação do arquivo de folha de estilo, que consumiremos nessa visão. Para isso, no projeto Xamarin Forms, crie uma nova pasta, chamada `styles` e dentro dela crie um arquivo de folha de estilo chamado `fotoscrud.css`. Insira nele o código apresentado na sequência. Não sou especialista em CSS, mas defini alguns elementos básicos para nosso uso, o que permitirá uma introdução a este novo recurso do Xamarin Forms.

```

^ ContentPage {

```

```
background-color : gray ;  
}
```

```
.header {  
  
    font-size : large ;  
  
    text-align : center ;  
  
    margin-bottom : 5;  
}
```

```
editor {  
  
    height : 90;  
  
    background-color : white ;  
}
```

```
#botaoGravar {  
  
    background-color : darkblue ;  
  
    color : white ;  
  
    font-size : 16;  
  
    height : 30;  
  
    width : 90;  
}
```

Após a criação do arquivo, clique sobre o nome dele com o botão direito do mouse e então em propriedades. A propriedade `Ação Requerida` deve estar com o valor `Recurso Inserido (Embedded Resource)`. Na versão do Xamarin Forms que estamos utilizando isso já vem configurado automaticamente, mas versões preliminares não traziam esta configuração, então achei importante apontá-la aqui.

Vamos a uma explicação do código anterior. O primeiro elemento, precedido pelo circunflexo, não é nativo do CSS, mas sua funcionalidade é selecionar todos os elementos filhos do tipo especificado (classe base), em nosso caso o `ContentPage`. Com a seleção, podemos definir propriedades para os elementos. Em nossa visão, estamos atribuindo uma cor para o `background` deles. O segundo elemento já é conhecido, pois é uma definição de classe de estilo. O terceiro, `editor`, define características para controles do tipo `<Editor>`, e o último define um `id` para um estilo.

Vamos então para a nossa visão, cujos códigos trarei por partes, para que explicar de maneira gradativa. As primeiras instruções referem-se ao uso de nosso arquivo CSS na visão. Desta maneira, antes da tag `<ContentPage.Content>`, insira o código a seguir. Observe o valor da propriedade `Source`, veja que existe um recuo de níveis de pastas, pois nossa visão está em um terceiro nível e o estilo está na raiz do projeto. Se sua visão estivesse também na raiz, apenas `Styles/fotoscrud.css` seria suficiente.

```
<ContentPage.Resources>
```

```
<StyleSheet Source = "../..../Styles/fotoscrud.css" />  
</ContentPage.Resources>
```

Agora, dentro do `<ScrollView>` que já temos na página, vamos inserir uma imagem e abaixo dela outro `StackLayout`. Este `StackLayout` conterá os controles relacionados a observações em relação à foto que será registrada e os botões de funcionalidade, que logo veremos. Observe no `Label` o uso da propriedade `StyleClass`, que registra o valor `header`, que temos definido em nosso arquivo CSS. Um estilo também foi definido para a tag `Editor` e será aplicado naturalmente. O `ScrollView` possibilitará a visualização dos controles, que no emulador do Android ficam ocultos quando o rotacionamos. Veja que, para a imagem, demos nome ao controle, pois trabalharemos suas dimensões via código.

```
<Image x:Name = "fotoCarro" Source = "consultar.png" Margin = "10" />

<ScrollView>

<StackLayout HorizontalOptions = "FillAndExpand" VerticalOptions = "Start" >

<Label Text = "Observações" StyleClass = "header" />

<StackLayout BackgroundColor = "Black" Padding = "1" >

<Editor Text = "{Binding Observacoes}" />

</StackLayout>

</StackLayout>

</ScrollView>
```

Para finalizar nossa interface com o usuário, precisamos desenhar na visão os botões que fornecerão funcionalidade a ela. Sendo assim, após o `StackLayout` do `Editor` insira o código da sequência. No último `Button`, veja a definição do estilo pelo seu `ID`.

```
<Grid HorizontalOptions = "Fill" Margin = "5, 5, 0, 0" Padding = "10, 10, 0, 0" >

<Grid.ColumnDefinitions>

<ColumnDefinition Width = "33*" />

<ColumnDefinition Width = "34*" />

<ColumnDefinition Width = "33*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height = "Auto" />

</Grid.RowDefinitions>
```

```

<Button Text = "Câmera" HorizontalOptions = "Center" Grid.Column = "0" Grid.Row = "0" Command = "{Binding CameraCommand}" />

<Button Text = "Álbum" HorizontalOptions = "Center" Grid.Column = "1" Grid.Row = "0" Command = "{Binding AlbumCommand}" />

<Button Text = "Gravar" HorizontalOptions = "Center" Grid.Column = "2" Grid.Row = "0" Command = "{Binding GravarFotoCommand}" 
StyleId = "botaoGravar" />

</Grid>

```

Com essa implementação, podemos executar nossa aplicação e ver como está ficando nossa visão de registro de imagens. A imagem seguir traz as visões nos emuladores. Vamos observar alguns pontos: 1) a figura padrão está pequena, precisamos dela maior, pois ela depois representará uma foto tirada ou selecionada; 2) nossa barra de títulos não está na mesma cor do conteúdo; 3) a cor do texto dos botões, no iOS, está difícil de se visualizar.

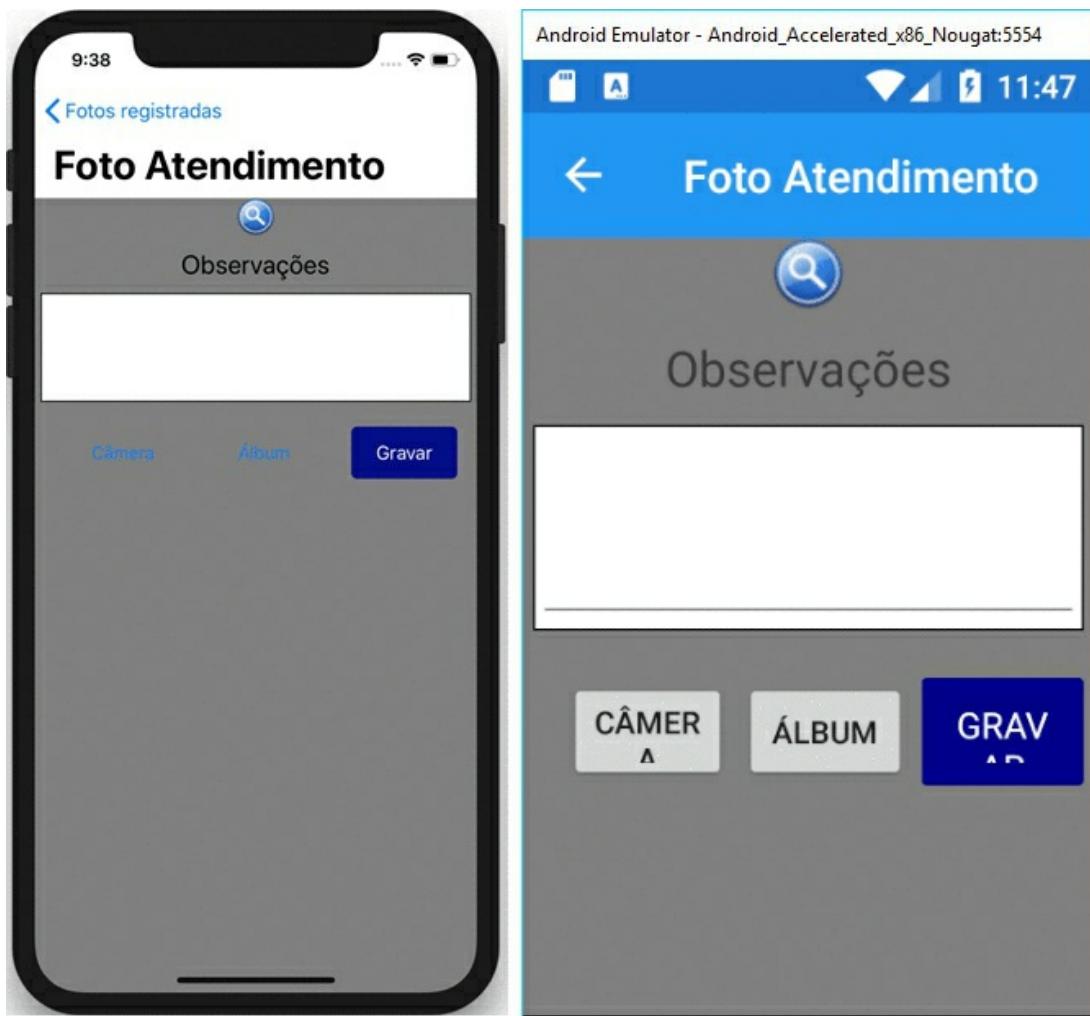


Figura 8.1: Visualização do registro de fotos

Vamos resolver estes problemas. O primeiro é relacionado ao tamanho da imagem que é exibida. Para resolvê-lo, faremos uso de uma técnica/recurso que ainda não vimos. Na visão, temos um método que é sempre invocado quando há a necessidade de alocação de tamanho para a visão em questão: é o `OnSizeAllocated()`, que podemos

sobrescrever da maneira que se encontra na sequência.

```
protected override void OnSizeAllocated ( double width, double height)  
{  
    base.OnSizeAllocated(width, height);  
}
```

Podemos verificar que este método recebe os valores de largura e altura e os submete ao método da classe base. Não remova esta invocação, pois isso impedirá a execução da aplicação. Com o conhecimento deste método e dos valores que ele recebe, podemos trabalhar com eles para definir as dimensões de nossa imagem. Entretanto, vamos fazer isso apenas quando o tamanho da visão mudar. E isso ocorrerá quando ela for exibida, e depois quando o dispositivo for rotacionado. Como podemos identificar isso? Vamos criar duas variáveis, fora do método (na classe) que manterá os valores, em nível de objeto. A aplicação só realizará o redimensionamento quando estes valores mudarem. Veja o novo código na sequência.

```
private double width;  
  
private double height;  
  
protected override void OnSizeAllocated ( double width, double height)  
{  
    base.OnSizeAllocated(width, height);  
  
    if (width != this.width || height != this.height)  
    {  
        this.width = width;  
  
        this.height = height;  
        fotoCarro.WidthRequest = width;  
        fotoCarro.HeightRequest = height / 2;  
    }  
}
```

Poderíamos trabalhar também para que, quando o dispositivo estiver rotacionado, e sua largura é maior que a altura, a altura da imagem seja menor. Veja a nova atribuição para a altura na sequência. Depois, teste sua aplicação e rotacione o emulador ou dispositivo.

```
fotoCarro.HeightRequest = (width < height) ? height / 2 : height / 4;
```

Um detalhe importante! Para que você possa rotacionar seu dispositivo com sua aplicação ativa, é preciso que esta autorização seja registrada no arquivo `info.plist` do projeto iOS. Na listagem a seguir, as quatro possíveis rotações estão permitidas. No Android não há restrições.

```
<key> UISupportedInterfaceOrientations </key>  
  
<array>  
  
<string> UIInterfaceOrientationPortrait </string>  
  
<string> UIInterfaceOrientationPortraitUpsideDown </string>
```

```

<string> UIInterfaceOrientationLandscapeLeft </string>

<string> UIInterfaceOrientationLandscapeRight </string>

</array>

```

Vamos para o segundo problema, que é a barra de título em cor diferente da visão. Precisaremos adotar um *workaround* aqui, pois a barra de títulos existe apenas para páginas derivadas de `NavigationPage` e nossas visões são `ContentPage`. O que ocorre é que, quando chamamos `Navigation.PushAsync()` enviando a `ContentPage`, o sistema de navegação do Xamarin Forms transforma nossa `ContentPage` em uma `NavigationPage`, mas não temos acesso, durante a codificação, à propriedade que permite essa configuração. Desta maneira, vamos criar um objeto estático em nossa classe `App`, na raiz do projeto Xamarin Forms. Veja o código a ser inserido antes do construtor.

```
public static Xamarin.Forms.NavigationPage navigationPage { get ; set ; }
```

Agora, precisamos atribuir a este objeto nossa primeira página de navegação da pilha e isso deve ser feito em nossa classe `MainPageView`, no método `ListView_ItemSelected()`, antes da atribuição da página ao `Detail`. Veja o código na sequência.

```
App.navigationPage = navigationPage;
```

Por fim, quando instanciarmos nossa visão de registro de fotos, precisamos acessar nosso objeto estático e definir nele a cor para a barra de títulos. Veja o código a seguir, que deve ser implementado, inicialmente, no construtor padrão de `FotosCRUDView`. Veja que alteramos também a cor do texto da barra de navegação.

```
App.navigationPage.BarBackgroundColor = Color.Gray;
App.navigationPage.BarTextColor = Color.Black;
```

Execute sua aplicação, acesse a visão de registro. Veja que a barra de títulos agora está com a mesma cor que a visão. Volte para a visão anterior. Não retornou à cor anterior, certo? Pois é, precisamos garantir isso no método `OnDisappearing()` da visão, mas, antes, precisamos guardar a cor que tínhamos anteriormente, pois as plataformas possuem temas diferentes. Para isso, antes de seu construtor, implemente os dois primeiros códigos da sequência, depois, antes da atribuição da nova cor, no construtor, implemente as três últimas instruções. Veja que retiramos os títulos grandes no iOS, para termos mais espaço para a foto a ser tirada.

```

private Color previousBarBackgroundColor;
private Color previousBarTextColor;

this.previousBarBackgroundColor = App.navigationPage.BarBackgroundColor;
this.previousBarTextColor = App.navigationPage.BarTextColor;

App.navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetPrefersLargeTitles( false );

```

Agora sim, vamos restaurar as cores anteriores da barra de títulos e a característica do título em tamanho maior.

```

protected override void OnDisappearing ()
{
    base.OnDisappearing();

    App.navigationPage.BarBackgroundColor = this.previousBarBackgroundColor;
    App.navigationPage.BarTextColor = this.previousBarTextColor;
}

```

```

        App.navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetPrefersLargeTitles( true );
    }

```

Vamos ao terceiro problema, que é a cor do texto dos botões. Este problema poderia ser resolvido facilmente com a criação de uma classe de estilo que definisse, para as duas plataformas, as características visuais para os botões, como fizemos para o botão gravar . Entretanto, vamos relembrar os estilos oferecidos pelo Xamarin Forms, que vimos no capítulo 3. Em nossa visão XAML, dentro de Resources , abaixp da definição do arquivo CSS, insira a instrução a seguir.

```
<OnPlatform x:TypeArguments = "Color" Android = "Black" iOS = "White" x:Key = "buttonTextColor" />
```

Precisamos ligar nossos botões a este estilo. Para isso, ao final das tags <button> dos dois primeiros botões, insira a declaração `TextColor="{StaticResource buttonTextColor}"` . Pronto. Teste sua aplicação novamente. Ficamos com um problema ainda, que é a barra de status do dispositivo mostrada pelo Android. Também já vimos esta característica resolvida no capítulo 3, pela invocação ao método `Window.SetStatusBarColor()` , diretamente na classe `MainActivity` do projeto `Android` .

Terminando esta seção deixo três links interessantes para você, que recomendo ler. Os dois primeiros referem-se ao uso de CSS, que é um recurso trazido com a versão 3 do Xamarin Forms, e o terceiro, algumas técnicas para o trabalho com rotação do dispositivo.

- <https://www.telerik.com/blogs/xamarin-forms-styling-with-css/>
- <https://docs.microsoft.com/pt-br/xamarin/xamarin-forms/user-interface/styles/css/>
- <https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/layouts/device-orientation?tabs=vswin/>

## 8.3 O acesso à câmera

Para que o usuário possa visualizar a câmera, ao clicar no botão `câmera` , apresentado anteriormente, precisamos implementar `ICommand CameraCommand` em nossa classe `FotosCRUDViewModel` , que ainda não criamos. Vamos fazer isso agora. Na pasta `ViewModels/Atendimento` , crie esta classe. No código a seguir já a apresento com a definição do `Command` e do método `RegistrarCommands()` , que é chamado no construtor. Já sabemos que precisaremos do objeto referente à foto que será manipulada pela visão, sendo assim, já implementamos o construtor recebendo este argumento e o atribuindo para um objeto também declarado.

```

namespace Capitulo05.ViewModels.Atendimentos
{
    public class FotosCRUDViewModel
    {
        public ICommand CameraCommand { get; set; }
        public AtendimentoFoto AtendimentoFoto { get; set; }

        public FotosCRUDViewModel(AtendimentoFoto atendimentoFoto)
        {
            this.AtendimentoFoto = atendimentoFoto;
            RegistrarCommands();
        }

        private void RegistrarCommands()
        {
            CameraCommand = new Command(() =>
            {

```

```

        MessagingCenter.Send<AtendimentoFoto>(this.AtendimentoFoto, "Camera");
    });
}
}
}

```

Para que nossa visão possa realizar a ligação de seus controles por meio de nossa classe ViewModel, da listagem anterior, precisamos declarar nela um objeto para isso e então inicializá-lo em nosso construtor. Veja o primeiro trecho referente à declaração do objeto, que deve ser implementado antes do construtor, e o segundo trecho, que deve estar dentro do construtor parametrizado, que recebe o objeto foto .

```

private FotosCRUDViewModel viewModel;

BindingContext = viewModel = new FotosCRUDViewModel(foto);

```

Para que possamos fazer uso do acesso à câmera e ao álbum de fotos, utilizaremos um componente que minimiza nosso trabalho e opera da mesma maneira tanto para iOS como para o Android, o `MediaPlugin`, de James Montemagno. Temos também a situação de que, para testarmos essas funcionalidades, no caso do iOS, precisamos de um dispositivo, pois não é possível o acesso a estes recursos no emulador.

Vamos à instalação do plugin. Clique com o botão direito do mouse no nome da solução e escolha a opção `Gerenciar Pacotes do Nuget para a Solução`. Pesquise por "MediaPlugin" e selecione o `Xam.Plugin.Media`, que no momento da escrita deste livro está na versão 4.0.1.1. Marque os projetos em que o plugin deverá ser instalado e confirme o processo. Os projetos são o do Xamarin Forms, do Android e do iOS. Após a instalação, um arquivo de observações necessárias para a configuração nos projetos específicos para cada plataforma é exibido pelo Visual Studio.

Como optei por realizar os testes primeiramente em um dispositivo iOS, começaremos pelas alterações neste projeto. Clique com o botão direito do mouse sobre o nome do arquivo `Info.plist` e escolha a opção `Abrir com...`. Nas opções oferecidas, escolha `Editor (Texto) de XML`. No arquivo que se abre, insira as seguintes configurações, antes do fechamento da tag `</dict>`. Observe que se trata de um conjunto de pares chave/valor, específicos para o acesso à câmera e ao álbum. No momento do primeiro acesso da aplicação aos recursos, uma janela será exibida com a mensagem específica, solicitando autorização.

```

<key> NSCameraUsageDescription </key>

<string> Esta aplicação precisa acessar a câmera para tirar fotos. </string>

<key> NSPhotoLibraryUsageDescription </key>

<string> Esta aplicação precisa acessar suas fotos. </string>

<key> NSMicrophoneUsageDescription </key>

<string> Esta aplicação precisa acessar seu microfone. </string>

<key> NSPhotoLibraryAddUsageDescription </key>

<string> Esta aplicação precisa acessar sua galeria de fotos. </string>

```

Nossa aplicativo não armazenará a imagem em nossa base de dados, mas sim seu caminho físico no dispositivo. Desta maneira, precisaremos de uma propriedade em nossa ViewModel para manipular a atualização e obtenção deste caminho. Em uma aplicação, o recomendado é termos nossas imagens persistidas na internet, o que minimiza

o armazenamento físico no dispositivo. Então a obtenção das URLs destes arquivos poderia ser feita por um serviço Web, como um REST, ou ainda com seu caminho direto. Nos capítulos 10 e 11, teremos a criação e consumo de serviços REST. Para o momento, na classe `FotosCRUDViewModel`, crie a propriedade a seguir. Lembre-se de estender a classe `BaseViewModel` no início, se não a invocação ao método `OnPropertyChanged()` não será reconhecida pelo compilador.

```
public string CaminhoFoto
{
    get { return this .AtendimentoFoto.CaminhoFoto; }

    set
    {
        this .AtendimentoFoto.CaminhoFoto = value ;
        OnPropertyChanged();
    }
}
```

Agora, vamos ao trabalho referente ao `MessagingCenter`, que deve ser implementado no code-behind da visão `FotosCRUDView`. Lembra? Precisamos registrar no `OnAppearing()` e cancelar a assinatura em `OnDisappearing()`.

Veja o código da mensagem `Camera` na listagem a seguir. Alguns usings serão necessários. Fique atento. Observe a chamada ao método `CrossMedia.Current.Initialize()` que inicializa o plugin instalado anteriormente. Depois, na sequência, é verificado se a câmera está disponível e se é possível tirar uma foto. Se sim, a foto é tirada e o arquivo que aponta para ela, gravado no dispositivo, é armazenado na variável `file`. Observe o nome que será dado ao arquivo, composto por data e hora.

Com o sucesso da operação, atribuímos o arquivo armazenado ao controle `Image`, que definimos na visão com o nome `fotoCarro`. Como adotamos a estratégia de armazenar o nome e caminho do arquivo na base de dados, gravamos essa informação na propriedade `CaminhoFoto` do objeto ligado à nossa ViewModel. Veja que na invocação do método `TakePhotoAsync()` algumas propriedades são configuradas. Definimos uma pasta que será utilizada para armazenar a foto. O tamanho para a foto está atribuído como `Full`, mas existem outras opções. Terminamos atribuindo o nome que será dado ao arquivo para a foto retirada. Recomendo que, caso tenha interesse, você acesse a página do plugin e verifique as possibilidades de configurações. O link é <https://github.com/jamesmontemagno/MediaPlugin/>.

```
private async Task<bool> TirarFotoAsync (AtendimentoFoto foto)
{
    await CrossMedia.Current.Initialize();

    if (!CrossMedia.Current.IsCameraAvailable || !CrossMedia.Current.IsTakePhotoSupported)
    {
        await DisplayAlert ( "Sem Câmera" , "A câmera não está disponível." , "OK" );
        await Task.FromResult( false );
    }

    string fileName = String.Format( "{0:ddMMyy_HHm}" , DateTime.Now ) + ".jpg" ;

    var file = await CrossMedia.Current.TakePhotoAsync( new Plugin.Media.Abstractions.StoreCameraMediaOptions
    {
```

```

        Directory = "Fotos" ,
        PhotoSize = Plugin.Media.Abstractions.PhotoSize.Full,
        Name = fileName
    });

    if (file == null )
        return await Task.FromResult( false );

fotoCarro.Source = ImageSource.FromStream(() =>
{
    var stream = file.GetStream();

    return stream;
});

viewModel.CaminhoFoto = file.Path;

return await Task.FromResult( true );
}

// OnAppearing

MessagingCenter.Subscribe<AtendimentoFoto>( this , "Camera" , async (foto) => { await TirarFotoAsync (foto); });

// OnDisappearing

MessagingCenter.Unsubscribe<AtendimentoFoto>( this , "Camera" );

```

Com essa implementação, você já pode testar sua aplicação em seu dispositivo iOS e tirar uma foto. A figura a seguir apresenta a tela capturada diretamente do celular durante esse processo. Pode ocorrer que a aplicação solicite diversas confirmações suas para que possa ter acesso ao recurso. Responda positivamente a elas.

Um detalhe precisa ser dito em relação ao comportamento do plugin, pois existe um bug nele. Se depois que abrir a câmera, você rotacionar o dispositivo, pode acontecer de a imagem não respeitar a orientação paisagem (*landscape*) e os controles dela desaparecerem. Se você voltar à posição de retrato (*portrait*) os controles reaparecem. Entretanto, se você rotacionar o dispositivo para paisagem, antes de clicar no botão da câmera, o funcionamento é perfeito. Vi discussões sobre isso na internet com o próprio autor, que está trabalhando para resolver este problema. Talvez, quando você for usar, já tenha sido inclusive resolvido. Também, quando você tira a foto em paisagem, na hora de exibir no plugin, ele a mostra em retrato, mas a gravação ocorre em paisagem. Este problema não ocorreu no Android. Entretanto, no emulador do Android, as fotos tiradas na orientação de retrato são sempre exibidas nos `Images` como paisagem. Em testes em um dispositivo Motorola C Plus este problema não ocorreu.

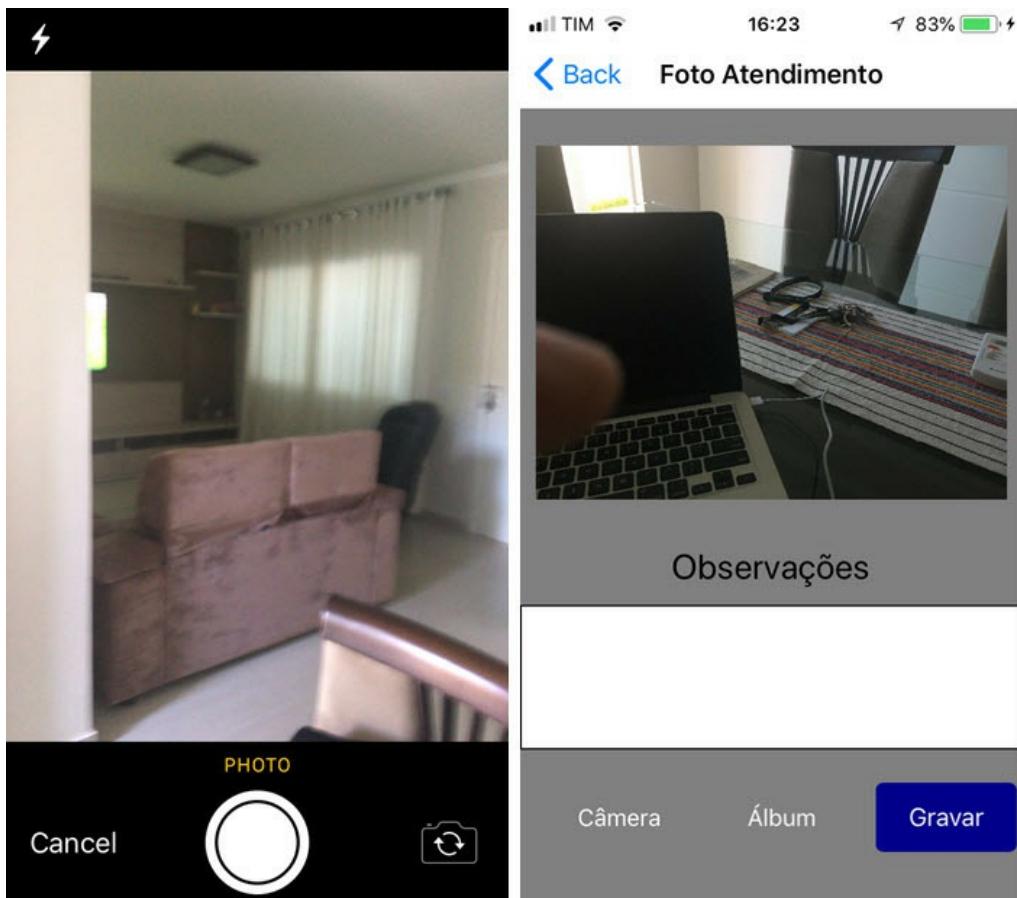


Figura 8.2: Acesso à câmera e visualização da foto tirada

Na imagem da direita, veja que nossa barra de título voltou à cor original dela, mas por quê? Quando a câmera é ativada, nossa visão perde o foco, o que dispara o método `OnDisappearing()`, que retorna os valores iniciais para a barra de título. Mas é fácil corrigir isso. Traga as instruções a seguir, que estão no construtor, para o método `OnAppearing()` da classe `FotosCRUDView`.

```
App.navigationPage.BarBackgroundColor = Color.Gray;
App.navigationPage.BarTextColor = Color.Black;

App.navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetPrefersLargeTitles( false );
```

## 8.4 Tirando foto com o Android

Muito bem, tudo funcionando no iOS. Vamos agora testar o uso da câmera no Android. A boa notícia é que é possível realizar o teste no próprio emulador, mas o plugin exige algumas configurações que precisaremos fazer. A primeira delas é configurarmos o projeto Android para que tenha (ou você verifique) o nome do pacote para a aplicação. Para isso, clique com o botão direito do mouse sobre o nome do projeto Android e, então, em Propriedades. Na janela que se abre, ao lado esquerdo, escolha a opção Manifesto do Android e localize o campo Nome do pacote. Em meu exemplo, deixei `com.casadocodigo.Capitulo05`. Guarde este nome, pois o usaremos.

O próximo passo é implementarmos a sobrescrita de um método na classe `MainActivity`, de nosso projeto Android. Antes do final do delimitador da classe, insira o método apresentado na sequência. Este método está relacionado a permissões que são solicitadas para uso de recursos do dispositivo, o que internamente será feito pelo `Media Plugin`.

```
public override void OnRequestPermissionsResult ( int requestCode, string [] permissions, Android.Content.PM.Permission[] grantResults)
{
    Plugin.Permissions.PermissionsImplementation.Current.OnRequestPermissionsResult(requestCode, permissions,
    grantResults);
}
```

Ainda na classe `MainActivity`, agora no método `onCreate()`, precisamos inserir logo em seu início a instrução a seguir.

```
Plugin.CurrentActivity.CrossCurrentActivity.Current.Activity = this ;
```

Precisamos inserir, no arquivo `AndroidManifest.xml`, que está dentro de `Propriedades` no projeto Android, a configuração apresentada na sequência. Esta configuração deverá estar entre as tags `<Application>` e `</Application>`, e é necessária para o acesso e registro de arquivos no Android.

```
<provider android:name = "android.support.v4.content.FileProvider" android:authorities =
"com.casadocodigo.Capitulo05.fileprovider" android:exported = "false" android:grantUriPermissions = "true" >

<meta-data android:name = "android.support.FILE_PROVIDER_PATHS" android:resource = "@xml/file_paths" ></meta-data>

</provider>
```

Agora, terminando a parte burocrática exigida pelo plugin, na pasta `Resources`, crie outra, chamada `xml`. Dentro dela, crie um arquivo XML chamado `file_paths.xml`, que, se você notou, já estamos utilizando na listagem anterior. Este arquivo precisará do código apresentado na sequência. Em nosso caso, como trabalharemos apenas com fotos, elas serão gravadas na pasta `Pictures` do dispositivo, que é o nome dado na configuração a seguir.

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

<paths xmlns:android = "http://schemas.android.com/apk/res/android" >

<external-files-path name = "my_images" path = "Pictures" />

<external-files-path name = "my_movies" path = "Movies" />

</paths>
```

Maiores detalhes e informações sobre a necessidade imposta pelo Android para o acesso à câmera pelo plugin, podem ser obtidas pelo link <https://github.com/jamesmontemagno/MediaPlugin/>. Agora você já pode testar sua aplicação no seu dispositivo Android ou em seu emulador. A figura a seguir apresenta a execução e a fotografia capturada no emulador. Algumas mensagens solicitando permissões podem aparecer para você.

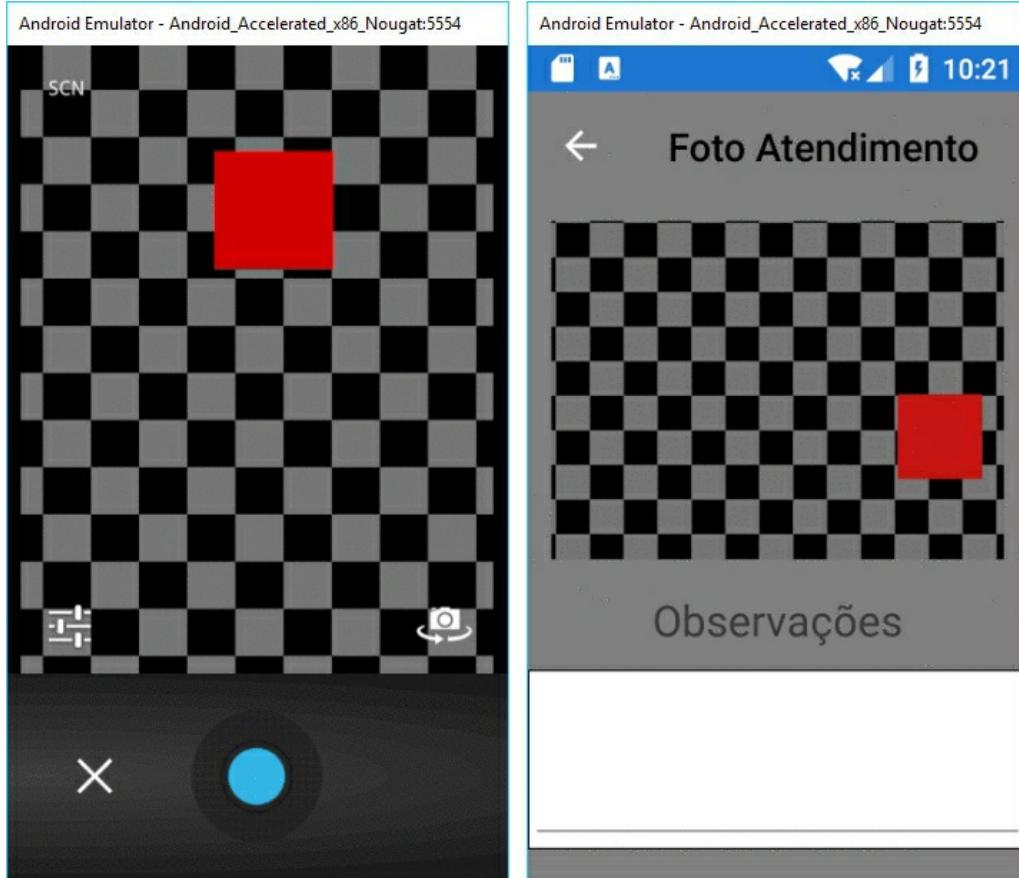


Figura 8.3: Acesso à câmera e visualização da foto tirada no emulador Android

## 8.5 Problemas no caminho para a imagem gravada

Quando formos recuperar uma foto armazenada e atribuí-la para um `Image`, precisaremos utilizar o caminho armazenado na captura da imagem pelo plugin. Ocorre que cada plataforma possui suas particularidades em relação a esta situação. No iOS, basta o nome do arquivo combinado com o caminho para a pasta de documentos, mas para o Android é preciso o caminho completo de onde a imagem foi gravada e isso, no iOS não funciona. Na realidade, eu penso que isso seja um bug do plugin em relação ao Android. Com isso, precisamos atribuir à propriedade `CaminhoFoto` o caminho correto para as duas plataformas. Programação específica para plataformas e consumo de maneira transparente do serviço nos leva ao `Dependency Service`, lembra? Muito bem! Então, em nosso projeto `Interfaces`, vamos criar uma pasta chamada `Fotos` e, dentro dela, a interface chamada `IFotoLoadMediaPlugin` com o código semelhante ao apresentado na sequência.

```
namespace Interfaces.Fotos
{
    public interface IFotoLoadMediaPlugin
    {
        string SetPathToPhoto(string caminhoCompleto);
    }
}
```

Precisamos agora implementar a interface anterior em classes em nossos projetos específicos de plataformas. Para

isso, nos projetos iOS e Android, crie uma pasta chamada `Fotos` e, dentro dela, uma classe chamada `FotoLoadMediaPlugin` tal qual os códigos apresentados na sequência. O primeiro é para o iOS e o segundo para o Android. Veja que no código do iOS aplicamos um filtro para recuperar apenas o nome do arquivo da string que será recebida. Logo veremos onde utilizaremos o método. Já para o Android, o caminho completo é retornado. No caso, o mesmo valor recebido pelo método.

```
[assembly: Xamarin.Forms.Dependency(typeof(FotoLoadMediaPlugin))]  
namespace Capitulo05.iOS.Fotos  
{  
    public class FotoLoadMediaPlugin : IFotoLoadMediaPlugin  
    {  
        public string SetPathToPhoto(string caminhoCompleto)  
        {  
            return caminhoCompleto.Substring(caminhoCompleto.LastIndexOf("/") + 1);  
        }  
    }  
}  
  
[assembly: Xamarin.Forms.Dependency(typeof(FotoLoadMediaPlugin))]  
namespace Capitulo05.Droid.Fotos  
{  
    public class FotoLoadMediaPlugin : IFotoLoadMediaPlugin  
    {  
        public string SetPathToPhoto(string caminhoCompleto)  
        {  
            return caminhoCompleto;  
        }  
    }  
}
```

E agora? Onde consumiremos nosso método criado anteriormente? Na mensagem que realiza a captura da fotografia. Como última instrução dela, temos o código `viewModel.CaminhoFoto = fileName;`. Precisamos mudar, retirando o `fileName` e inserindo `DependencyService.Get<IFotoLoadMediaPlugin>().SetPathToPhoto(file.Path);`. Veja que enviamos para o método o caminho completo de onde o plugin armazenou a foto tirada, mas atribuiremos o valor requerido por cada plataforma. Uma vez mais o `Dependency Service` nos auxiliando.

Caso você queira verificar no console do Visual Studio os valores das variáveis e objetos, durante a execução de sua aplicação, você pode fazer uso da instrução `Debug.WriteLine()`, enviando como argumento a variável/objeto que deseja verificar. Mas você também tem sempre os pontos de interrupção do depurador do Visual Studio, que auxiliam bastante.

## 8.6 O acesso ao álbum de fotos

Já temos o necessário para usar a câmera do dispositivo para tirar uma foto. Agora precisamos oferecer ao usuário a possibilidade de obter uma foto já armazenada no álbum de fotografias. Para que nossa aplicação acesse o álbum de fotos, a lógica do processo é a mesma do que utilizamos para o acesso à câmera. Entretanto, teremos que trabalhar a escrita do arquivo da foto selecionada no dispositivo, pois, quando o plugin a recupera, grava-a em uma pasta temporária e o caminho do arquivo não estará disponível em execuções futuras. Como já comentado, existem diferenças em relação às plataformas e os locais de gravação e recuperação das fotos. No Android, pela configuração que fizemos em XML, o caminho final para elas é `O Pictures` e, no iOS, é a pasta `Documents`. Vamos recorrer ao nosso

`Dependency Service` para resolver o problema. Insira o seguinte método na interface que estamos trabalhando para fotos. Logo após o método da interface, seguem códigos que devem ser implementados no projeto iOS e Android. Veja o uso de `MyDocuments` para iOS e `MyPictures` para o Android.

```
string GetDevicePathToPhoto ();  
  
public string GetDevicePathToPhoto ()  
{  
    return Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments), "Fotos");  
}  
  
public string GetDevicePathToPhoto ()  
{  
    return Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.MyPictures), "Fotos");  
}
```

Precisamos recuperar as imagens armazenadas no dispositivo e já temos o caminho dela na base de dados. Entretanto, não podemos simplesmente usá-lo, pois, na recuperação, o Android precisa do caminho absoluto, que é o que foi gravado, e o iOS precisa de uma composição do nome do arquivo com o caminho da aplicação, já que a cada deploy um novo caminho físico diferente é criado para ela. Devido a isso, não podemos gravar o caminho físico também para o iOS. Vamos uma vez mais fazer uso do `Dependency Service`. Em nossa interface `IFotoLoadMediaPlugin`, declare mais um método, tal qual apresento na sequência.

```
string GetPathToPhoto ( string caminhoArmazenado);
```

Agora, precisamos implementar as especificidades de cada plataforma. Veja o código a seguir, que traz primeiro a implementação para o iOS e depois para o Android. Veja que no iOS realizamos a composição do caminho da pasta desejada da aplicação, com o nome do arquivo que foi gravado. Já no caso do Android, o caminho é o utilizado para a gravação, completo. Estes códigos são nas classes `FotoLoadMediaPlugin`, na pasta `Foto` dos projetos de plataforma.

```
public string GetPathToPhoto ( string caminhoArmazenado)  
{  
    return Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments), "Fotos", caminhoArmazenado);  
}  
  
public string GetPathToPhoto ( string caminhoArmazenado)  
{  
    return caminhoArmazenado;  
}
```

Com este problema contornado, em nossa `FotosCRUDView`, crie o método a seguir. Observe que criamos uma variável com o caminho desejado para que a foto seja gravada. Verificamos a existência deste caminho e, caso não exista, o criamos. Após termos o caminho válido, nós o armazenamos na variável `caminhoCompleto`. Realizamos a escrita do arquivo e requisitamos nosso método `SetPathPhoto()` para obtermos, de acordo com a plataforma, o caminho que deverá ser armazenado na base de dados. Alguns usings serão requisitados, fique atento. O  `FileMode.Create`

sobrescreverá o arquivo caso ele já exista.

Você poderia utilizar a mesma lógica que fizemos para verificação de existência do arquivo e tratar a situação como for necessário. Para isso, utilize os métodos da classe `File`. Reforço que, no Android, quando tiramos uma foto pelo plugin, ele utiliza um caminho absoluto em relação ao dispositivo e precisaremos deste caminho para obter a foto depois. Já na captura da foto, estamos gravando-a em um caminho relativo à aplicação, que é o que seria mais correto. Note que, logo no início do método, adotamos a mesma lógica para atribuição do nome do arquivo para a foto que adotamos quando utilizamos a câmera. A remoção do arquivo está ocorrendo apenas para o caso de o usuário ter tirado uma foto pela câmera e depois selecionado uma do álbum.

```
public string SaveFotoFromAlbum ( string caminhoFoto, Plugin.Media.Abstractions.MediaFile file)

{

    string nomeArquivo;

    if (caminhoFoto == null || caminhoFoto.StartsWith( "http" ))

    {

        nomeArquivo = String.Format( "{0:ddMMyy_HHm}" , DateTime.Now) + ".jpg" ;

    }

    else

    {

        if (File.Exists(DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(caminhoFoto)))

            File.Delete(DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(caminhoFoto));



        nomeArquivo = (caminhoFoto.LastIndexOf( "/" ) > 0) ? caminhoFoto.Substring(caminhoFoto.LastIndexOf( "/" ) + 1) : caminhoFoto;

    }

    var caminhoFotos = DependencyService.Get<IFotoLoadMediaPlugin>().GetDevicePathToPhoto();

    if (!Directory.Exists(caminhoFotos))

        Directory.CreateDirectory(caminhoFotos);

    string caminhoCompleto = Path.Combine(caminhoFotos, nomeArquivo);

    using (FileStream fileStream = new FileStream(caminhoCompleto, FileMode.Create))

    {

        file.GetStream().CopyTo(fileStream);

    }

    return DependencyService.Get<IFotoLoadMediaPlugin>().SetPathToPhoto(caminhoCompleto);

}
```

Agora, vamos declarar e implementar o `Command` que será executado quando o usuário optar por selecionar uma foto do álbum. Veja o código a seguir. A primeira instrução implementa a declaração e a segunda deve ser inserida no método `RegistrarCommands()`. Isso tudo na classe `FotosCRUDViewModel`.

```
public ICommand AlbumCommand { get ; set ; }
```

```
AlbumCommand = new Command(() =>
{
```

```

        MessagingCenter.Send<AtendimentoFoto>( this .AtendimentoFoto, "Album" );
    });

```

Para que a mensagem seja executada, precisamos registrá-la no `MessagingCenter`. Para isso, implemente o código a seguir no método `OnAppearing()` do code-behind de `FotosCRUDView`. Observe a similaridade com o código para acessar a câmera. Entretanto, agora é verificado se existe a permissão para selecionar uma foto e o método invocado refere-se a essa seleção. Veja que a variável `imagePath` recebe o retorno do método que criamos anteriormente, o `SaveStreamToFile()`, que retornará o caminho de acordo a cada plataforma.

```

private async Task SelecionarFotoDoAlbumAsync (AtendimentoFoto foto)

{
    await CrossMedia.Current.Initialize();

    if (!CrossMedia.Current.IsPickPhotoSupported)
    {
        await DisplayAlert ( "Álbum não suportado" , "Não existe permissão para acessar o álbum de fotos" , "OK" );
        return ;
    }

    var file = await CrossMedia.Current.PickPhotoAsync();

    if (file == null )
        return ;

    var imagePath = SaveFotoFromAlbum(foto.CaminhoFoto, file);

    fotoCarro.Source = ImageSource.FromStream(() =>
    {
        var stream = file.GetStream();
        return stream;
    });

    viewModel.CaminhoFoto = imagePath;
    return ;
}

```

Como já sabemos, precisamos agora registrar e cancelar a assinatura da mensagem `Album` no `MessagingCenter` e isso está no código a seguir, que deve ser inserido nos métodos `OnAppearing()` e `OnDisappearing()`.

```

// OnAppearing()

MessagingCenter.Subscribe<AtendimentoFoto>( this , "Album" , async (foto) => { await SelecionarFotoDoAlbumAsync (foto); });

// OnDisappearing()

MessagingCenter.Unsubscribe<AtendimentoFoto>( this , "Album" );

```

Você verificou, na implementação anterior para o método `SaveFotoFromAlbum()`, que invocamos a remoção de arquivo pela invocação ao método `File.Exists()`, para que pudéssemos utilizar o mesmo nome de arquivo para a nova foto? Pois bem, é interessante fazermos isso também na captura de uma fotografia pela câmera. Vamos então alterar um pouco nosso código já implementado para o método `TirarFotoAsync()`. Substitua a atribuição `string fileName = String.Format("{0:ddMMyy_HHmm}", DateTime.Now) + ".jpg";` pelo trecho de código a seguir.

```
string fileName;
if (foto.CaminhoFoto == null)
{
    fileName = String.Format("{0:ddMMyy_HHmm}", DateTime.Now) + ".jpg";
}
else
{
    File.Delete(DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(foto.CaminhoFoto));
    fileName = (foto.CaminhoFoto.LastIndexOf("/") > 0) ? foto.CaminhoFoto.Substring(foto.CaminhoFoto.LastIndexOf("/") + 1) : String.Format("{0:ddMMyy_HHmm}", DateTime.Now) + ".jpg";
}
```

Na atribuição para nome do arquivo, no código anterior, veja que, caso o objeto já possua um valor, ele será atribuído e o arquivo já existente é removido para que outro possa ser criado. Se não fizermos isso, o plugin cria um arquivo com o nome adicionado de (1). Isso evitará que arquivos que não são mais utilizados continuem ocupando espaço, ou seja, para cada foto, teremos sempre um arquivo, mesmo que sejam tiradas várias fotos na mesma operação.

Muito bem, com estas implementações que fizemos já é possível testarmos nossa aplicação. Acesse a visão de registro de fotos e clique no botão Álbum. O teste pode ser feito tanto no iOS como no Android.

Nós ainda temos um problema em relação aos arquivos de fotos tiradas ou selecionadas do álbum. Imagine a situação em que o usuário acessou a tela de registro de foto, tirou uma foto ou selecionou do álbum e não gravou, apenas retornou para a visão de listagem. Isso fará com que a imagem fique gravada no dispositivo e não seja reutilizada, pois o objeto de foto nem chegou a ser gravado. Para resolver este problema, precisaremos utilizar nosso objeto `navigationPage`, que está na classe `App`. Adicionaremos a ele um `EventHandler` para o momento em que ocorrer o `Pop` da visão, ou seja, quando houver a navegação da página de CRUD para a de listagem de fotos. Crie, na classe `FotosCRUDView`, o método a seguir. Veja que temos dois `ifs`, o primeiro para garantir que estaremos trabalhando o `Pop` da visão correta, e o segundo, para garantir que eliminemos um arquivo que respeite as condições que detalhamos neste parágrafo. Ao final, retiramos o método da lista de `handlers` de `Popped`.

```
public void OnPoppedCRUDFoto ( object sender, NavigationEventArgs e )
{
    if (e.Page.GetType() == typeof (FotosCRUDView))
    {
        if (viewModel.AtendimentoFoto.CaminhoFoto != null && viewModel.AtendimentoFoto.AtendimentoFotoID == null )
            File.Delete(DependencyService.Get<IFotoLoadMediaPlugin>()
                .GetPathToPhoto(viewModel.AtendimentoFoto.CaminhoFoto));
    }
    App.navigationPage.Popped -= OnPoppedCRUDFoto;
}
```

No código anterior, removemos o evento de `Popped`, mas e a adição dele? Onde faremos? No construtor. Insira, desta

forma, no construtor da classe `FotosCRUDView`, a instrução a seguir. Você pode obter mais informações sobre registro de eventos no link <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/events/how-to-subscribe-to-and-unsubscribe-from-events/>.

```
App.navigationPage.Popped += OnPoppedCRUDFoto;
```

## 8.7 Gravação do caminho da foto na base de dados

Como manipularemos dados em uma tabela, tal qual fizemos para os serviços, precisamos agora implementar a classe DAL no projeto `SQLiteEF` e ela está na sequência. Não me prenderei a explicações, pois a lógica é semelhante à que implementamos para serviços no capítulo anterior, mas peço que leia com atenção. Fique atento aos namespaces.

```
namespace CasaDoCodigo.DAL
{
    public class AtendimentoFotoDAL : DALBase<AtendimentoFoto>
    {
        private Atendimento Atendimento { get; set; }

        public AtendimentoFotoDAL(Atendimento atendimento, string dbPath) : base(dbPath)
        {
            this.Atendimento = atendimento;
        }

        public async override Task<List<AtendimentoFoto>> GetAllAsync(Expression<Func<AtendimentoFoto, object>> expression = null, OrderByType orderByType = OrderByType.NaoClassificado)
        {
            using (var context = DatabaseContext.GetContext(dbPath))
            {
                var query = PrepareDataToGet1All(context, expression, orderByType);
                query = query.Where(i => i.AtendimentoID == Atendimento.AtendimentoID);
                return await query.ToListAsync();
            }
        }

        public async override Task<AtendimentoFoto> UpdateAsync(AtendimentoFoto foto, long? itemID, params object[] associatedObjects)
        {
            return await base.UpdateAsync(foto, itemID, foto.Atendimento);
        }
    }
}
```

Já temos implementadas as funcionalidades de acesso à câmera e álbum para que o usuário possa escolher ou tirar uma foto do veículo, assim como o campo para que ele possa registrar observações sobre a foto a ser inserida. Entretanto, falta-nos implementar a propriedade `Observacoes`, que podemos verificar no código a seguir, que deve estar na classe `FotosCRUDViewModel`.

```
public string Observacoes
{
    get { return this.AtendimentoFoto.Observacoes; }
```

```

    set {
        this .AtendimentoFoto.Observacoes = value ;
        OnPropertyChanged();
    }
}

```

Agora sim, podemos declarar e implementar a funcionalidade para o `Command GravarFotoCommand`, que está logo a seguir. A primeira instrução declara o `Command` e as demais devem ser inseridas no método `RegistrarCommands()`. Veja que temos a invocação de duas mensagens, `InformacaoCRUD`, que tem uma funcionalidade já conhecida de outras implementações, e uma nova, a `AtualizarFoto`, que será responsável por atribuir uma imagem genérica ao `Image`, após a gravação da foto. Note também que temos uma lógica para habilitação do botão de gravação. Com essa implementação, precisamos inserir a instrução `((Command)GravarFotoCommand).ChangeCanExecute();` no `set()`, nas propriedades `CaminhoFoto` e `Observacoes`. Observe que agora estamos declarando o DAL apenas na gravação, pois não precisamos dele em outra situação. Talvez você queira trocar a cor do botão de gravar, pois quando desabilitado pode não aparecer bem o texto. Mas deixarei isso a seu cargo. Lembre-se apenas que a cor está definida em nosso arquivo de CSS. Também está na sequência os registros e cancelamentos para as mensagens.

```

public ICommand GravarFotoCommand { get ; set ; }

// RegistrarCommands

GravarFotoCommand = new Command( async () =>
{
    AtendimentoFoto.Atendimento = AtendimentoFoto.Atendimento;
    AtendimentoFoto.AtendimentoID = AtendimentoFoto.Atendimento.AtendimentoID;

    var dal = new AtendimentoFotoDAL(AtendimentoFoto.Atendimento, DependencyService.Get<IDBPath>().GetDbPath());
    await dal.UpdateAsync(AtendimentoFoto, AtendimentoFoto.AtendimentoFotoID);

    MessagingCenter.Send< string >( "Atualização realizada com sucesso." , "InformacaoCRUD" );
    MessagingCenter.Send< string >( "consultar.png" , "AtualizarFoto" );

    AtendimentoFoto = new AtendimentoFoto();
    OnPropertyChanged(nameof(Observacoes));
}, () =>
{
    return (! string .IsNullOrEmpty(Observacoes) && ! string .IsNullOrEmpty(CaminhoFoto));
});

// OnAppearing

MessagingCenter.Subscribe< string >( this , "InformacaoCRUD" , async (msg) => { await DisplayAlert ( "Informação" , msg, "ok" );
});

MessagingCenter.Subscribe< string >( this , "AtualizarFoto" , (caminho) => { fotoCarro.Source = caminho; });

// OnDisappearing

MessagingCenter.Unsubscribe< string >( this , "InformacaoCRUD" );

```

```
MessagingCenter.Unsubscribe< string >( this , "AtualizarFoto" );
```

## 8.8 Remoção de objetos de associações

Vamos agora lidar com a remoção de objetos associados. Da maneira como estamos removendo um atendimento, seus serviços e fotos ficam ainda na base de dados e o ideal é que eles sejam removidos em conjunto. O EF Core permite que configuremos a remoção em cascata, sendo que, ao remover um objeto pai, seus objetos associados sejam também removidos. Entretanto, o Migrations tem limitações para adicionar esta *feature* em um banco já existente. É uma das limitações que apresentei no capítulo 5. Desta maneira, trabalharemos a remoção dos serviços e fotos no momento em que formos eliminar um atendimento. Veja no código a seguir a operacionalidade na sobrescrita do método `DeleteAsync`, no `AtendimentoDAL`. Precisamos, entretanto, recuperar os dados de serviços e fotos para o contexto e enviá-los como argumentos para o método `RemoveRange()` de cada conjunto.

```
public async override Task DeleteAsync (Atendimento atendimento)
{
    using ( var context = DatabaseContext.GetContext(dbPath))
    {
        var servicosDAL = new AtendimentoItemDAL(atendimento, dbPath);
        var fotosDAL = new AtendimentoFotoDAL(atendimento, dbPath);

        context.AtendimentoItens.RemoveRange( await servicosDAL.GetAllAsync());
        context.AtendimentoFotos.RemoveRange( await fotosDAL.GetAllAsync());
        await base .DeleteAsync(atendimento);
    }
}
```

Observe que não invocamos o método `SaveChangesAsync()`, pois dependemos ainda da remoção do atendimento, que será realizado na classe `base`. Mas você lembra que nela também temos a captura do contexto? Agora precisamos adaptar o método da classe `base` para que, caso ele receba um contexto, ele seja usado. Vamos adaptar nossa implementação para o `DeleteAsync()` em `DALBase` e, se vamos mudar a assinatura do método, precisamos mudar também a assinatura em nossa interface, tal qual o código a seguir.

```
// Assinatura em IDAL

Task DeleteAsync (T item, object databaseContext = null );

// Assinatura do método em AtendimentoDAL e DALBase

public virtual async Task DeleteAsync (T item, object databaseContext = null )

// using do método

using ( var context = (databaseContext == null ) ? DatabaseContext.GetContext (dbPath) : (DatabaseContext) databaseContext)

// nova invocação na AtendimentoDAL

await base.DeleteAsync (atendimento, context);
```

Mas e nossos arquivos de fotos? Como nós vimos anteriormente, ao remover uma única foto, nós também eliminávamos seu arquivo. Precisaremos fazer isso também quando todo o atendimento for removido. Desta maneira, na classe `ListagemViewModel`, de `Atendimentos`, no projeto Xamarin Forms, adaptaremos nosso método `EliminarAtendimento()` para a implementação a seguir. Observe que, antes de invocarmos o `DeleteAsync()` do DAL de atendimento, guardamos a coleção de fotos em uma variável.

Caso tudo tenha ocorrido bem na remoção do atendimento, uma varredura é realizada na variável `fotos`, para que os arquivos que existiam associados aos objetos possam ser removidos do dispositivo. Se algo der errado durante o processo de remoção do atendimento, possivelmente uma exceção será disparada e a execução, interrompida. Vou deixar para você o tratamento de exceções, ok?

## 8.9 Conclusão

Chegamos ao final deste capítulo. Não trabalhamos ainda todo o processo de CRUD para fotos, pois isso deixaria este nosso capítulo muito longo - realizaremos isso no próximo, que será muito interessante. Mas trabalhamos muita coisa nova e interessante. Demos sequência a associações de objetos, atividade que trabalhamos nos capítulos 6 e 7, interagimos com a câmera e álbum para registro de fotos.

No próximo capítulo, trabalharemos a listagem de fotos registradas para um atendimento, onde utilizaremos a captura e tratamento de gestos do usuário sobre uma imagem. Teremos como base o projeto que concluímos neste capítulo.

## C APÍTULO 9

# Listagem de fotos e manipulação de gestos

Nos capítulos anteriores, vimos o mapeamento de classes associadas por meio de uma propriedade para uma base de dados que representa uma coleção, onde registramos um atendimento a um veículo, ao qual tínhamos associados serviços a serem realizados e fotos retiradas do veículo.

Aqui, veremos um recurso novo, vindo do Xamarin Forms 3, chamado **FlexLayout**, com o qual trabalharemos a troca de orientação do dispositivo e exibição ou não da barra de tarefas. Aproveitaremos o uso de imagens que obtivemos no capítulo anterior para apresentar o uso de captura de gestos (*gesture*) do usuário pelo dispositivo, como dois toques em uma imagem, o arrastar dela, um *drag-and-drop* e operações de zoom. Também veremos mais a técnica de *Dependency Service*.

Ao final, trabalharemos um problema que temos, que não comentei até aqui, mas que tenho certeza de que você já pensou nele: a funcionalidade de remoção de objetos de associações.

Reforço que toda implementação realizada neste capítulo tem como base o capítulo anterior e servirá como base para o próximo. Muita coisa boa, nova e útil.

## 9.1 Listagem das fotos inseridas

Já temos a visão de listagem de fotos registradas para os atendimentos, a visão que as regista no dispositivo e seus caminhos na base de dados, e temos também nosso DAL retornando todas as imagens, tudo implementado nos capítulos anteriores. Precisamos agora concluir nossa visão de listagem e faremos isso utilizando alguns recursos e técnicas que ainda não vimos no livro. Vamos lá. O primeiro passo será complementarmos nosso XAML, tal qual está no código a seguir.

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class = "Capitulo05.Views.Atendimentos.FotosListagemView"
    Title = "Fotos registradas" >

    <ContentPage.ToolbarItems>

        <ToolbarItem Icon = "plus.png" Command = "{Binding NovoCommand}" />

    </ContentPage.ToolbarItems>

    <ContentPage.Content>

        <Grid VerticalOptions = "Center" >

            <Grid.ColumnDefinitions>
```

```

<ColumnDefinition Width = "*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height = "*" />

</Grid.RowDefinitions>

<ScrollView Orientation = "Horizontal" Padding = "10" Grid.Column = "0" Grid.Row = "0" >

<FlexLayout x:Name = "flexLayoutFotos" >

</FlexLayout>

</ScrollView>

</Grid>

</ContentPage.Content>

</ContentPage>

```

Observe que, dentro de `ContentPage.Content`, estamos fazendo uso de um `Grid`, pois queremos centralizar o conteúdo em nossa visão, e o `Grid` é contêiner mais indicado para isso. Mais ao final do código, temos uma novidade, que é o novo tipo de layout, trazido pela versão 3 do Xamarin Forms, o `FlexLayout`.

Em uma primeira vista, pode-se pensar que este layout se assemelha muito ao `StackLayout` (e até alguns posts comentam isso), mas ele tem algumas propriedades exclusivas. Veremos algumas delas em nossa implementação. Por enquanto, apenas definimos um nome para o `FlexLayout`, pois trabalharemos com ele em nosso *code-behind*.

O que esperamos obter com essa visão é algo semelhante ao apresentado pela figura a seguir.

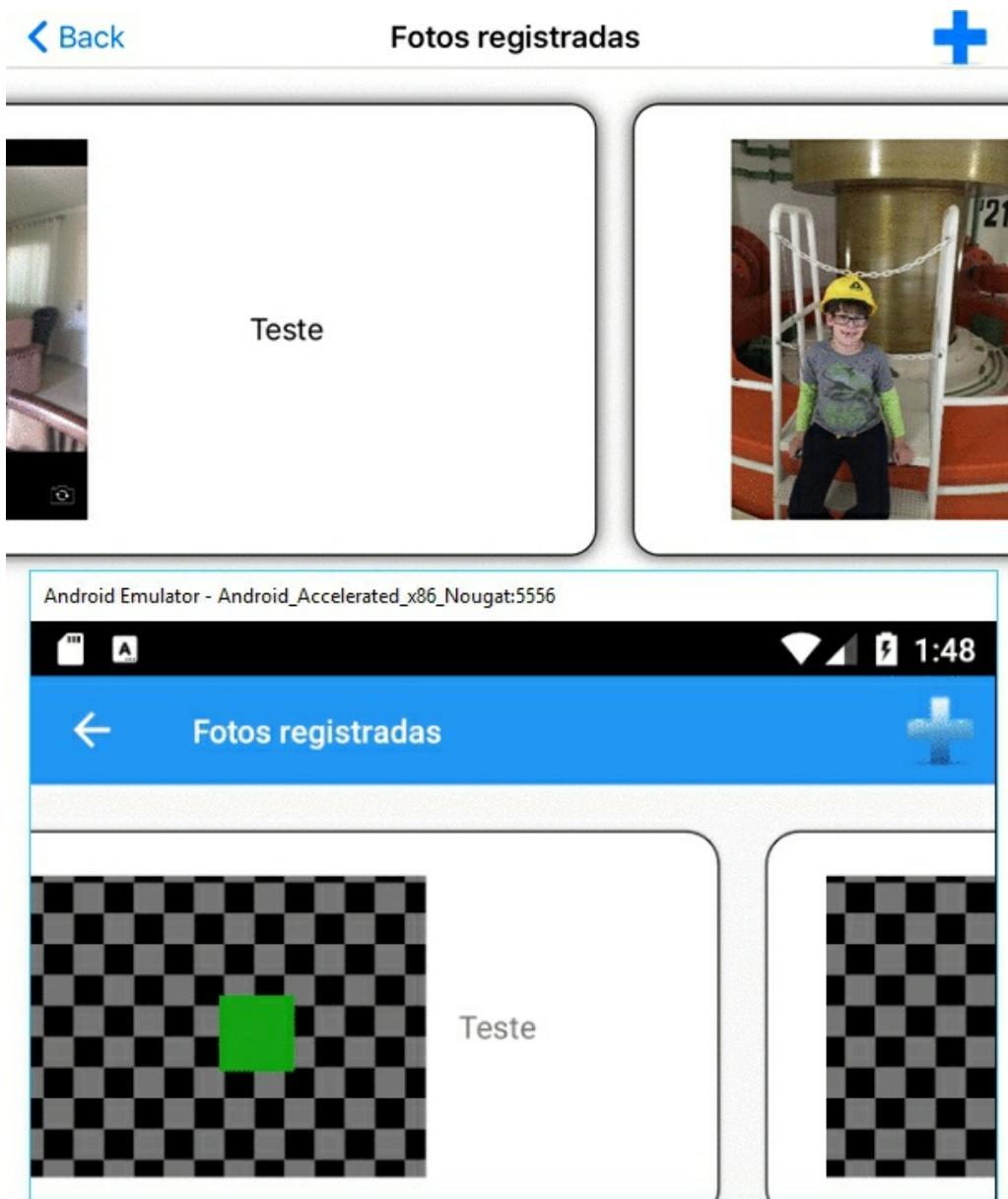


Figura 9.1: Visão com a listagem de fotos

Muito bem, qual é a ideia proposta para esta seção? Vamos criar uma rolagem de todas as fotos registradas para o atendimento. Para isso, trabalharemos a mudança da orientação da visão para paisagem (*Landscape*). Implementaremos essa solução nos projetos específicos de plataforma e o consumiremos por meio de `Dependency Service`.

Uma vez as fotos exibidas, permitiremos ao usuário que, ao pressionar sobre uma delas e arrastá-la para fora da área de exibição, ela seja removida do atendimento. Também, dando dois toques sobre uma foto em específico, ela será exibida para realização de zoom. A observação de cada foto será exibida também na barra de rolagem de imagens. O que acha? Legal, não é? Então vamos ao trabalho.

Em nosso projeto `Interfaces`, crie uma pasta chamada `Devices` e, dentro dela, crie a interface da listagem a seguir. Você se lembrou de que para usarmos `Dependency Service` precisamos de uma interface que deverá ser implementada nos projetos específicos de plataforma?

```

namespace CasaDoCodigo.Devices.Interfaces
{
    public interface IOrientation
    {
        void Landscape();
        void Portrait();
    }
}

```

Agora, nos projetos `ios` e `Android`, crie uma pasta também chamada `Devices`. Veja as classes `Orientation` para os projetos na sequência, começando pelo iOS. Não entrarei em detalhes sobre as implementações, pois são instruções específicas para cada plataforma, que foram mapeadas para C#. Quem tiver familiaridade com o iOS poderá notar similaridade entre os nomes de classes e métodos. Para o código do Android, faremos uso do plugin `Current Activity`, do James Montemagno, mas nós já o temos instalado nos projetos, já que ele é um pré-requisito do `Media Plugin`. É menos custoso fazer o uso do plugin para o Android, pois se usássemos apenas recursos nativos a ele, o processo seria muito mais burocrático e verboso. Observe que antes dos namespaces existe uma anotação para o `Dependency Service`, mas já sabemos disso, certo?

```

[assembly: Xamarin.Forms.Dependency(typeof(Orientation))]
namespace Capitulo05.iOS.Devices
{
    public class Orientation : IOrientation
    {
        public void Landscape()
        {
            UIDevice.CurrentDevice.SetValueForKey(new NSNumber((int)UIInterfaceOrientation.LandscapeLeft), new NSString("orientation"));
        }
    }
}

```

```

public void Portrait()
{
    UIDevice.CurrentDevice.SetValueForKey(new NSNumber((int)UIInterfaceOrientation.Portrait), new NSString("orientation"));
}
}
}

```

```

[assembly: Xamarin.Forms.Dependency(typeof(Capitulo05.Droid.Devices.Orientation))]
namespace Capitulo05.Droid.Devices
{
    public class Orientation : IOrientation
    {
        public void Landscape()
        {
            CrossCurrentActivity.Current.Activity.RequestedOrientation = ScreenOrientation.Landscape;
        }
    }
    public void Portrait()
    {
        CrossCurrentActivity.Current.Activity.RequestedOrientation = ScreenOrientation.Portrait;
    }
}

```

```
 }
}
```

Muito bem, vamos consumir nosso serviço na visão de listagem de fotos. Vamos realizar algumas implementações no momento em que a visão aparece e em que ela perde o foco. Veja estes métodos, já conhecidos nossos, na sequência. Tiramos as configurações específicas do iOS 11 também, para termos um melhor aproveitamento da visão. Observe a invocação dos métodos que acabamos de implementar. Execute sua aplicação e verifique a mudança de orientação da visão quando ela for exibida. Estamos falando da classe `FotosListagemView`.

```
protected override void OnAppearing ()
{
    base.OnAppearing();

    App.navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetPrefersLargeTitles( false );
    App.navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetUseSafeArea( false );

    DependencyService.Get<IOrientation>().Landscape();
}

protected override void OnDisappearing ()
{
    base.OnDisappearing();

    App.navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetPrefersLargeTitles( true );
    App.navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetUseSafeArea( true );

    DependencyService.Get<IOrientation>().Portrait();
}
```

Com a visão de listagem de fotos aberta, rotacione seu emulador ou dispositivo e veja o comportamento. No Android, todas as rotações mantêm a visão em paisagem, mas no iOS há um momento em que a visão retorna para a orientação de retrato. Precisamos corrigir isso quando acontecer. Existe um método para as visões (classe `Page` e suas especializações), chamado `onSizeAllocated()`, que é invocado sempre que há uma mudança no tamanho da visão, em nosso caso, nossa `ContentPage`. Precisamos sobrescrever este método e, quando ocorrer uma alteração em que a altura seja maior que a largura (isto é, está na posição de retrato), precisaremos forçar novamente a paisagem. Para podermos fazer essa verificação, precisamos ter disponível um valor anterior para comparação. Então, vamos definir dois objetos privados em nossa classe, um para largura (`width`) e um para altura (`height`). Veja o código para esta declaração na sequência.

```
private double width;
private double height;
```

Agora vamos implementar a sobrescrita para o método `onSizeAllocated()`, tal como é apresentado na sequência. A primeira instrução, que invoca o próprio método em sua superclasse, não deve ser removida, caso contrário a sua aplicação travará. Veja que nossa primeira instrução é referente à verificação dos valores recebidos pelo método, que é a dimensão solicitada pela a aplicação. Sabemos que, quando definimos tipos primitivos, eles possuem um valor inicial e, em nosso caso, no código anterior, os valores para `width` e `height` serão zeros. Uma vez que a condição tenha sido avaliada como verdadeira, os valores recebidos pelo método são atribuídos aos objetos que declaramos anteriormente.

Com os dados atualizados, verificamos a orientação da visão. Se ela tiver a altura maior que a largura, a chamada ao nosso método `Landscape()` deverá ocorrer. Entretanto, se abrirmos a visão de registro de foto, este método será chamado e a nova visão também estará em paisagem, mesmo tendo sido posicionada em retrato no método `OnDisappearing()`. Poderíamos realizar a implementação a seguir, que verifica, além das dimensões, se a visão atual é do CRUD de Fotos; se for, a paisagem não é forçada.

```
protected override void OnSizeAllocated ( double width, double height )  
{  
    base.OnSizeAllocated ( width, height );  
  
    if ( width != this.Width || height != this.Height )  
    {  
        this.Width = width;  
        this.Height = height;  
  
        if ( this.Height > this.Width && Navigation.NavigationStack [ Navigation.NavigationStack.Count - 1 ].GetType () !=  
            typeof ( FotosCRUDView ) )  
            DependencyService.Get<IOrientation> ().Landscape ();  
    }  
}
```

A técnica anterior funciona bem no iOS, mas no Android, quando se retorna para o CRUD do atendimento, o método é novamente chamado e, como o `if()` é avaliado como verdadeiro, a visão é erroneamente orientada em modo paisagem. A solução para ambas as plataformas foi fazer uso de uma variável de controle, um `flag`. Implemente-a, de acordo ao seguinte código, antes de seu construtor da `FotosListagemView`.

```
private bool forcarLandscape;
```

Para configurar corretamente o momento em que forçar a orientação de paisagem, no `OnAppearing()` atribua `true` à variável, e no `OnDisappearing()` atribua `false`. Para finalizar, no método `OnSizeAllocated()`, altere a verificação para executar o `Landscape()` para a apresentada no código a seguir.

```
if ( this.Height > this.Width && forcarLandscape )
```

Temos ainda um problema a resolver. No iOS, quando nossa visão de listagem estiver ativa, se minimizarmos a aplicação, ao retornar para ela, a visão estará em orientação de retrato. Para resolver isso, precisamos realizar as seguintes codificações na classe `App`, no método `OnResume()`. Observe que começamos obtendo a quantidade de páginas que existem na pilha de navegação do objeto estático `navigationPage`. Como este objeto só deixa de ser nulo quando chegamos à listagem, fazemos uso do operador condicional nulo, `o ? ,` para que, caso o objeto ainda não tenha sido instanciado, a variável `countStackPages` receba `null`. Depois, a condição é por ela mesma compreensível.

```
protected override void OnResume ()  
{  
    int? countStackPages = navigationPage?.Navigation.NavigationStack.Count;  
  
    if ( countStackPages != null && countStackPages > 0 && navigationPage.Navigation.NavigationStack [ ( int ) countStackPages - 1 ].GetType () == typeof ( FotosListagemView ) )  
        DependencyService.Get<IOrientation> ().Landscape ();  
}
```

Voltemos à listagem. A ideia que proponho para exibição das fotos é que cada uma esteja inserida em uma borda e podemos fazer isso utilizando o controle `Frame`. Só que não podemos fazer isso dentro de nosso XAML, pois não

sabemos quantas fotos serão recuperadas. O problema está em gerar uma borda para cada foto. Sendo assim, nosso primeiro problema aqui é a obtenção das fotos do atendimento selecionado na visão anterior. Em nosso ViewModel, já temos a propriedade `Fotos`, precisamos populá-la. Faremos tal qual estamos fazendo no decorrer do livro. Veja o método da sequência, que deve ser implementado na classe `FotosListagemViewModel`. Note que, como não utilizaremos um `ListView`, não precisaremos sincronizar as coleções, apenas atualizamos a propriedade associativa de nosso objeto de atendimento.

```
public async Task AtualizarFotosAsync ()  
{  
    Atendimento.Fotos = await atendimentoFotoDAL.GetAllAsync();  
}
```

Também em nossa ViewModel, precisamos oferecer uma propriedade que possa ser utilizada pela visão. Veja o código dela na sequência. Poderíamos pensar em guardar o atendimento na visão e evitar esta propriedade, mas para seguirmos o MVVM, vamos trabalhar com ela.

```
public List<AtendimentoFoto> FotosAtendimento  
{  
    get { return Atendimento.Fotos; }  
}
```

Agora, no método `OnAppearing()`, no code-behind da visão `FotosListagemView`, insira a instrução `await viewModel.AtualizarFotosAsync();` após a invocação ao método `Landscape()`. Será preciso inserir `async` na assinatura do método.

Precisamos criar nossos `Frames` e faremos isso de maneira programática, e não declarativa, como estamos fazendo por meio do XAML. Como vamos inserir e configurar controles na própria visão, manteremos isso no seu code-behind. Veja o código do método na sequência. Veja que limpamos (`Clear()`) inicialmente a coleção de controles existentes (`children`), em nosso objeto `flexLayoutFotos` que temos definido em nosso XAML. Com essa operação realizada, varremos a coleção de `Fotos` no ViewModel e, para cada foto, criamos um `Frame` e configuramos algumas propriedades para eles. Finalizando, inserimos cada frame como componentes do `flexLayoutFotos`. Execute sua aplicação, tire ou selecione algumas fotos e veja os frames exibidos na visão. Poderíamos pensar em criar um controle específico para nosso `Frame`, mas vou deixar como dica para você fazer, ok?

```
private void CriaFramesFotos ()  
{  
    flexLayoutFotos.Children.Clear();  
  
    for ( int i = 0; i < viewModel.FotosAtendimento.Count; i++ )  
    {  
        Frame frameFoto = new Frame();  
        frameFoto.Margin = 10;  
        frameFoto.BorderColor = Color.Black;  
        frameFoto.CornerRadius = 15;  
        frameFoto.WidthRequest = (width > height ? width * 0.60 : height * 0.60);  
        frameFoto.HeightRequest = (height > width ? height * 0.50 : width * 0.50);  
  
        flexLayoutFotos.Children.Add(frameFoto);  
    }  
}
```

Neste ponto, pelo fato de que vamos trabalhar com fotos e já comentamos que o ideal é que elas venham da Web, precisamos pensar em tráfego. Vamos levantar um problema: no momento em que acessamos a listagem de fotos, todas elas são recuperadas do dispositivo de acordo com o caminho armazenado na base de dados para popular nossa coleção, que será utilizada no código apresentado anteriormente.

Com o que temos implementado, a partir desta visão, o usuário pode acessar a de registro e adicionar novas fotos. A questão é: precisaremos exibir TODAS as fotos novamente? Ou só as novas? A ideia é que sejam só as novas. Vamos tratar isso, mas precisaremos inserir em nossa classe de modelo `AtendimentoFoto` uma propriedade que servirá de sinalizadora para esta situação. Entretanto, não queremos que esta propriedade exista em nossa base de dados, pois ela só terá validade em tempo de execução. Vamos criá-la no projeto `IDPropertiesEF`, conforme a listagem a seguir. Este tipo de propriedade é conhecido como transiente. Já utilizamos desta estratégia em capítulos anteriores.

[`NotMapped`]

```
public bool JaExibidaNaListagem { get ; set ; }
```

Precisamos agora implementar algumas burocracias para que evitemos processamento desnecessário na exibição das fotos que recuperaremos. O primeiro passo será determinar quando as fotos devem ser recuperadas. Temos o método `AtualizarFotosAsync()` na classe `FotosListagemViewModel` sendo chamado a cada vez que a visão for exibida. Com as explicações que já apresentei antes, a princípio, este método deverá ser executado quando a visão for exibida pela primeira vez e quando a visão de registro de fotos for acessada e tiver fotos novas inseridas. Podemos pensar em deixar esta responsabilidade em nosso método, e não em quem o chama. Vamos declarar uma variável que possa ser utilizada como `flag`. No início da classe `FotosListagemViewModel`, declare uma variável privada do tipo `bool`, chamada `atualizarDados`, e atribua `true` a ela, tal qual é mostrado no código a seguir.

```
private bool atualizarDados = true ;
```

Em seguida, no método `AtualizarFotosAsync()`, insira logo no início a verificação apresentada na sequência. No final do método, antes do `return`, insira a instrução que atribui `false` à variável de controle, que está após o primeiro trecho do código apresentado.

```
// Início

if (!atualizarDados)

    return ;

// Final

atualizarDados = false ;
```

Vamos analisar o código anterior. Quando acessarmos a visão de listagem pela primeira vez, a variável `atualizarDados` terá o valor `true`, então precisaremos popular a propriedade `servicos`, caso contrário, o retorno ocorre. Apenas para ficar claro no código, dentro do `foreach`, insira a instrução `f.JaExibidaNaListagem = false;`, para marcarmos que os objetos recuperados ainda não foram exibidos. Isso é desnecessário, pois o valor padrão para `bool` é `false`, mas eu gosto de trabalhar assim.

Já temos os objetos, vamos agora ver como trataremos a criação dos frames, que deverão ser criados apenas para as novas fotos, não recuperadas pelo método que as recupera a primeira vez. Para isso, precisaremos mudar nossa estrutura de repetição, do `for` tradicional para o `foreach`, no método `CriaFrameFotos()`, na classe `FotosListagemView`.

Da maneira como nosso método está, ele está limpando os controles que estão inseridos em nosso `FlexLayout` e depois criando um `Frame` para cada elemento da coleção. Com nossa nova técnica, esta limpeza não é mais necessária, então vamos retirá-la e inserir apenas as novas fotos. Agora, com a propriedade que criamos anteriormente, podemos recuperar sempre as novas fotos apenas, ainda não visualizadas na listagem. Veja na sequência o código para nosso `foreach`, do método `CriaFrameFotos()`. O corpo do bloco se mantém o mesmo. Veja o uso do `LINQ` pelo método `where()`.

.

```

var fotos = viewModel.FotosAtendimento.Where(f => !f.JaExibidaNaListagem).ToList();

foreach ( var f in fotos)

```

Ainda não está tudo pronto para podermos testar, temos mais o que implementar. Não temos nossa foto em nosso frame. A proposta é que a foto seja inserida e, ao lado direito dela, as observações. Ainda, na foto, queremos que ela possa ser removida da coleção quando o usuário pressionar e a arrastar; e dando dois toques sobre ela, que uma nova visão com a imagem ocupando todo seu espaço seja exibida. Estes recursos são chamados de gestos (*gesture*).

Para que o gesto de pressionar e arrastar seja possível nas duas plataformas, é preciso que tenhamos a imagem inserida em um `ContentView` e que tenhamos este novo controle definido em uma classe separada. No iOS, essa regra não é necessária. Vamos criar nosso controle. No projeto Xamarin Forms, crie uma pasta chamada `controls` e dentro dela uma classe tal qual o código apresentado na sequência.

```

namespace Capitulo05.Controls
{
    public class FotoListContainerControl : ContentView
    {
        public AtendimentoFoto Foto { get; set; }

        public FotoListContainerControl(AtendimentoFoto foto)
        {
            this.Foto = foto;
        }
    }
}

```

Vamos à inserção da foto em nosso contêiner, que deve ser implementada no construtor da classe que acabamos de criar para este fim, a `FotoListContainerControl`. Sendo assim, logo no início do construtor, implemente a instrução a seguir. Estamos instanciando a imagem no construtor, pois no Android, se esta imagem for trocada depois da construção do objeto, o processo de arrastar e soltar que pretendemos implementar não funciona. Seria preciso criar um novo controle e atribuir a ele a nova foto.

```
Content = new Image() { Source = DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(foto.CaminhoFoto) };
```

Agora, insira antes da instrução `flexLayoutFotos.Children.Add(frameFoto);` o código a seguir. Observe que instanciamos nosso controle criado anteriormente e encaminhamos nosso objeto `foto` para ele. Na sequência, criamos um `Label`, que exibirá as informações para as fotos. Veja o uso do método `FlexLayout.SetAlignSelf()`, que força o alinhamento do controle para o valor especificado como parâmetro, sobrescrevendo definições em contêineres externos. Em seguida, criamos um novo `FlexLayout`, que será responsável por conter a foto (nossa contêiner) e a observação. Veja a configuração para `Direction`, que fará com que os controles sejam inseridos em forma de linha. Outra opção seria `Column`. Configuramos também a propriedade `JustifyContent`, que inserirá espaços iguais entre os controles, incluindo as margens. Ao final, neste novo `FlexLayout` são inseridas a foto e a observação, e o `FlexLayout` é atribuído ao `content` do `Frame`.

```

var fotoContainer = new FotoListContainerControl(f);

Label observacoes = new Label();
observacoes.Text = f.Observacoes;
observacoes.WidthRequest = frameFoto.WidthRequest * 0.30;
FlexLayout.SetAlignSelf(observacoes, FlexAlignSelf.Center);

FlexLayout flexLayout = new FlexLayout();

```

```

flexLayout.Direction = FlexDirection.Row;
flexLayout.JustifyContent = FlexJustify.SpaceAround;

flexLayout.Children.Add(fotoContainer);
flexLayout.Children.Add(observacoes);

frameFoto.Content = flexLayout;

```

Em nossa aplicação, uma vez uma foto exibida, se formos para a visão de registro de fotos e inserirmos novas fotos, ao retornar, queremos que apenas elas sejam exibidas, minimizando uma carga que existia na execução da consulta feita pelo DAL. Então, o que devemos fazer é que, uma vez a foto exibida, a marquemos como tal. Faremos isso então no método `CriaFramesFotos()` da classe `FotosListagemView`. Antes do final do laço, insira o código a seguir.

```
f.JaExibidaNaListagem = true ;
```

O que achou do que vimos até aqui neste capítulo? Bastante coisa interessante e nova, não é? E ainda veremos mais. Faltam os gestos, que veremos a seguir. Mas, como dito, tivemos muitas coisas novas para a implementação da listagem e uso do plugin de fotos no capítulo anterior. Não tem como eu terminar esta etapa sem deixar alguns links importantes e necessários para sua formação. Você pode escolher o momento mais propício para lê-los.

- <https://docs.microsoft.com/pt-br/xamarin/ios/app-fundamentals/file-system/> fala sobre o trabalho com o sistema de arquivos do iOS.
- <https://developer.android.com/reference/android/support/v4/content/FileProvider/> traz um pouco de teoria para auxiliar você na compreensão da configuração que o plugin de fotos exige para o Android.
- <https://docs.microsoft.com/pt-br/xamarin/xamarin-forms/user-interface/layouts/flex-layout/> apresenta o novo layout do Xamarin, que utilizamos para a listagem de fotos.
- <https://javiersuarezruiz.wordpress.com/2018/03/28/xamarin-forms-primer-vistazo-a-flexlayout/> traz uma visão um pouco mais divertida sobre o FlexLayout também.

## 9.2 O uso de gestos para definir funcionalidades

Comentamos anteriormente que, na listagem de fotos, o usuário poderá pressionar duas vezes a área do frame, nosso contêiner, e então uma nova visão com a foto será exibida. Isso é relativamente simples, mas precisamos saber alguns pontos, que vamos elucidar com o código a seguir. Ele deve ser inserido em nossa classe `FotoListContainerControl`, que criamos na pasta `Controls` do projeto Xamarin Forms. Este método deverá ser invocado no construtor da classe.

```

private void RegistrarTapGestureRecognizer ()
{
    var tapGestureRecognizer = new TapGestureRecognizer();
    tapGestureRecognizer.NumberOfTapsRequired = 2;
    tapGestureRecognizer.Tapped += (s, e) =>
    {
        DependencyService.Get<IOrientation>().Portrait();
        Navigation.PushAsync( new FotoInFocoView(Content as Image));
    };
    GestureRecognizers.Add(tapGestureRecognizer);
}

```

Observe que a primeira instrução instancia a classe `TapGestureRecognizer` para termos um objeto relativo aos gestos de

toques. Configuramos para ser capturado quando forem dados dois toques sobre o contêiner, depois, declaramos o comportamento que deverá ser executado quando esta condição acontecer e, após esta implementação (`Tapped`), o objeto é adicionado à coleção `GestureRecognizers` de nosso container. Bem simples, não é?

As instruções a serem executadas referem-se a tornar a visão em posição de retrato. Precisamos fazer isso antes da instanciação da página, para que uma foto que esteja em posição de retrato seja exibida corretamente. Depois você pode fazer um teste e comentar esta linha para ver este comportamento. Por fim instanciamos uma classe, que ainda não temos, que será responsável por exibir a imagem selecionada. Veja que enviamos para a nova página a imagem que está no contêiner que foi selecionado.

Vamos agora criar nossa visão. Na pasta `Views/Atendimentos`, crie uma nova `Content Page` e a nomeie como `FotoInFocoView`. Não teremos a implementação do MVVM nessa classe, pois ela lidará apenas com a camada de apresentação. No XAML da sequência, verifique a propriedade `BackgroundColor` definida. Esta cor, `Black`, ocupará o espaço nas laterais da foto, quando ela não ocupar todo a tela. Temos também um `StackLayout` que alinhará ao centro, tanto horizontal, como verticalmente, a imagem que nele inserirmos, e damos a ele um nome para que possamos manipulá-lo no code-behind. Para concluir, temos a propriedade `NavigationPage.HasNavigationBar` recebendo `false`, o que causará a ocultação da barra de navegação para a visão. Na estrutura do XAML, temos um `Grid` e, dentro dele, um `ContentView`, que receberá nossa imagem.

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class = "Capitulo05.Views.Atendimentos.FotoInFocoView"

    NavigationPage.HasNavigationBar = "false"

    BackgroundColor = "Black" >

    <ContentPage.Content>
        <Grid>
            <ContentView x:Name = "containerFoto" >
                </ContentView>
            </Grid>
        </ContentPage.Content>
    </ContentPage>
```

Na classe dessa nova visão, precisaremos de acesso em nível de objeto à imagem que está sendo enviada pelo código anteriormente apresentado e à imagem que será criada para ser exibida. Precisamos ter um novo objeto da imagem, pois trabalharemos algumas propriedades dela. Em nosso construtor, registraremos estas imagens em objetos e realizaremos algumas configurações. Veja, na sequência, o código inicial para essa classe. Observe que obtemos a fonte da imagem do objeto que recebemos no construtor. Definimos o `Aspect` para ela dentro do

`StackLayout` e a adicionamos a ele. Acho interessante você testar as outras duas opções para o `Aspect` , que são:  
`AspectFill` e `Fill` .

```
namespace Capitulo05.Views.Atendimentos
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class FotoInFocoView : ContentPage
    {
        private Image imagemSelecionada;
        private Image image;

        public FotoInFocoView (Image imagemSelecionada)
        {
            InitializeComponent();
            Image image = new Image() { Source = imagemSelecionada.Source };
            this.image = image;
            this.imagemSelecionada = imagemSelecionada;
            image.Aspect = Aspect.AspectFit;
            containerFoto.Content = image;
        }
    }
}
```

Vamos testar nossa aplicação. Acesse a listagem de fotos e dê dois toques sobre uma foto que esteja na orientação de retrato e depois na de paisagem. Após a exibição de cada foto, rotacione o dispositivo e veja seu comportamento. Lembre-se de testar com os diferentes `Aspect` .

Observe a existência da barra de tarefas no topo das imagens. Se você navegar na aplicação no iOS e deixar as visões na posição de paisagem, verá que esta barra desaparece, tornando-se ativa na posição de retrato. Este mesmo comportamento não ocorre no Android. Podemos implementar, por meio de `Dependency Service` a ocultação desta barra quando estivermos em nossa visão atual, a que mostra apenas a imagem selecionada, então precisamos de uma interface. Como trabalharemos com o dispositivo e já temos o de orientação, poderíamos pensar em utilizar uma única, mas a refatoração agora exigiria mudança de código, então, por simplicidade, criaremos uma nova interface, na pasta `Devices` do projeto `Interfaces` , tal qual apresenta o código a seguir. Veja o nome `IStatusBar` para a interface.

```
namespace Interfaces.Devices
{
    public interface IStatusBar
    {
        void Exibir();
        void Ocultar();
    }
}
```

Agora, em cada projeto de plataforma, precisamos implementar os recursos que nos possibilitarão ocultar e exibir a barra de tarefa. Nos projetos de plataforma, na pasta `Devices` , crie a classe `StatusBar` , tal qual vemos na sequência, sendo o iOS o primeiro. Veja que para ao Android fazemos mais uma vez uso do plugin que utilizamos para a orientação. O esforço para o iOS é menor. Para o Android, a novidade pode estar no operador `|=` , que adiciona valores em forma de bits na variável. O `|` é conhecido como operador `bitwise or` . O resto do código é de fácil assimilação.

```
[assembly: Xamarin.Forms.Dependency(typeof(StatusBar))]
namespace Capitulo05.iOS.Devices
```

```

{
    public class StatusBar : IStatusBar
    {
        public void Exibir()
        {
            UIApplication.SharedApplication.StatusBarHidden = false;
        }

        public void Ocultar()
        {
            UIApplication.SharedApplication.StatusBarHidden = true;
        }
    }
}

[assembly: Xamarin.Forms.Dependency(typeof(StatusBar))]
namespace Capitulo05.Droid.Devices
{
    public class StatusBar : IStatusBar
    {
        WindowManagerFlags originalFlags;

        public void Exibir()
        {
            var activity = CrossCurrentActivity.Current.Activity;
            var attrs = activity.Window.Attributes;
            attrs.Flags = originalFlags;
            activity.Window.Attributes = attrs;
        }

        public void Ocultar()
        {
            var activity = CrossCurrentActivity.Current.Activity;
            var attrs = activity.Window.Attributes;
            originalFlags = attrs.Flags;
            attrs.Flags |= Android.Views.WindowManagerFlags.Fullscreen;
            activity.Window.Attributes = attrs;
        }
    }
}

```

Para que possamos testar nosso serviço criado anteriormente, no iOS, precisamos configurar a aplicação para que permita a manipulação da visibilidade da *status bar* e isso deve ser feito no arquivo `Info.plist` do projeto iOS, adicionando as duas configurações apresentadas na sequência. Para o Android não é preciso nada semelhante.

```

<key> UIViewControllerBasedStatusBarAppearance </key>
<false/>

```

```
<key> UIStatusBarHidden </key>
```

```
<false/>
```

Vamos ocultar a barra de status agora. Em nosso code-behind da classe `FotoInFocoView`, vamos sobrescrever os métodos `OnAppearing()` e `OnDisappearing()` para invocarmos os métodos de ocultação e exibição, tal como segue.

```
// OnAppearing()  
DependencyService.Get<IStatusBar>().Ocultar();
```

```
//OnDisappearing()  
DependencyService.Get<IStatusBar>().Exibir();
```

Tudo isso seria maravilhoso se as plataformas funcionassem de maneira semelhante. Antes de trabalharmos a *status bar*, o iOS a ocultava de acordo com sua orientação, mas o Android não fazia isso, mantendo-a sempre exibida. Quando ocultamos e depois exibimos a barra no iOS, ele perde esta característica automática. Ou seja, nós precisaremos definir quando a barra deverá ficar exibida ou oculta. Agora, compete a você decidir o que é melhor para sua aplicação, levando em consideração o custo operacional para implementar estas características específicas. Veja que, ao retornarmos da foto para a listagem, a barra é exibida, mesmo estando em `Landscape`, pois nós a habilitamos quando a visão de fotos perder o foco. Você poderia pensar em retirar esta chamada que implementamos na visão da foto e colocá-la na visão da listagem. Fica à sua escolha.

Precisamos agora permitir o fechamento da visão da foto e retornar para a listagem. Trabalharemos também o `Tap Gesture`, com dois toques. Seguiremos o que fizemos anteriormente para chegar à visão. Na classe `FotoInFocoView`, crie o seguinte método e o invoque no construtor. Veja que o comportamento é a função de `Pop` na pilha de navegação.

```
private void RegistrarTapGestureRecognizer ()  
{  
    var tapGestureRecognizer = new TapGestureRecognizer();  
    tapGestureRecognizer.NumberOfTapsRequired = 2;  
    tapGestureRecognizer.Tapped += (s, e) =>  
    {  
        Navigation.PopAsync(); ;  
    };  
    this.image.GestureRecognizers.Add(tapGestureRecognizer);  
}
```

## 9.3 Operação de zoom na imagem selecionada

Até aqui tudo perfeito, mas queremos oferecer a nosso usuário a possibilidade de realizar um zoom na foto que ele selecionou. Isso também é um gesto, conhecido como `pinch`. A ideia é que quando o usuário pressione a imagem com dois dedos e os abra, um zoom seja executado para aumentar a imagem e, quando os dedos se aproximarem, a imagem vá diminuindo de tamanho. É uma operação corriqueira que vemos em aplicativos de fotos, mas existe uma complexidade física a ser resolvida.

Para a implementação proposta, eu tive como base os links:

- <https://docs.microsoft.com/pt-br/xamarin/xamarin-forms/app-fundamentals/gestures/pinch/> , que traz a documentação oficial do Xamarin para implementação do Gesture Pinch ;
- <https://forums.xamarin.com/discussion/83455/how-to-implement-image-zoom-in-and-out/> , onde <https://forums.xamarin.com/profile/CCLIU/> propôs inclusive uma classe que abordasse a solução que estamos propondo; e
- <https://stackoverflow.com/questions/40181090/xamarin-forms-pinch-and-pan-together/> , uma discussão no Stack Overflow , de onde retirei a implementação proposta por um dos colegas que lá discutiam o tema, que foram <https://stackoverflow.com/users/5953643/brandon-minnick/> e <https://stackoverflow.com/users/2076286/g%C3%A1bor/> . Parabéns aos dois e ao Stack Overflow .

Nosso primeiro passo é criar o Gesture Recognizer , tal qual fizemos para os gestos anteriores, referentes a selecionar a foto e retornar para a listagem, mudando apenas o tipo de gesto. Então, na classe FotoInFocoView , vamos implementar o método apresentado na sequência.

```
private void RegistrarPinchGestureRecognizer ()
{
    var pinchGesture = new PinchGestureRecognizer();
    pinchGesture.PinchUpdated += OnPinchUpdated;

    this.image.GestureRecognizers.Add(pinchGesture);
}
```

A operacionalidade toda do zoom estará implementada no método OnPinchUpdated() , que será apresentado a seguir. Neste método, faremos uso de algumas variáveis que precisamos declarar em nível de objeto. Desta maneira, antes do construtor, implemente as declarações apresentadas a seguir. São objetos relativos à escala de zoom da imagem e da posição dela em relação à visão.

```
private double currentScale = 1;
private double startScale = 1;
private double xOffset = 0;
private double yOffset = 0;
```

Agora sim, vamos ao método. Respire um pouco e não se assuste com a complexidade dele, mas como disse, é preciso um pouco de física e matemática para resolver o problema. Alguns comentários estão inseridos no código, que poderão nos ajudar na compreensão. Após a implementação do código, acesse uma foto, pressione com os dois dedos, afaste-os e, com os dedos pressionados, vá arrastando a imagem. Para retornar ao tamanho anterior, aproxime os dedos. Verifique nos ifs os estados que trabalharemos durante o processo.

```
void OnPinchUpdated ( object sender, PinchGestureUpdatedEventArgs e )
{
    if (e.Status == GestureStatus.Started)
    {
        // Armazena o fator de escala atual aplicado à imagem e zera o ponto central da transformação de conversão.
        startScale = Content.Scale;
        Content.AnchorX = 0;
        Content.AnchorY = 0;
    }
    else if (e.Status == GestureStatus.Running)
    {
        // Aqui é onde ocorre a magia. O código calcula a nova escala baseada no movimento dos dedos.
        double currentScale = e.GetCurrentScale();
        double scaleDelta = currentScale - startScale;
        double totalScale = startScale * (1 + scaleDelta);

        Content.Scale = totalScale;
        Content.TranslationX = -xOffset * scaleDelta;
        Content.TranslationY = -yOffset * scaleDelta;
    }
}
```

```

// Calcula o fator de escala a ser aplicado
currentScale += (e.Scale - 1) * startScale;
currentScale = Math.Max(1, currentScale);

// O ScaleOrigin é uma coordenada relativa à imagem para obter o pixel X da coordenada

double renderedX = Content.X + xOffset;
double deltaX = renderedX / Width;
double deltaWidth = Width / (Content.Width * startScale);
double originX = (e.ScaleOrigin.X - deltaX) * deltaWidth;

// O ScaleOrigin é uma coordenada relativa à imagem para obter o pixel Y da coordenada

double renderedY = Content.Y + yOffset;
double deltaY = renderedY / Height;
double deltaHeight = Height / (Content.Height * startScale);
double originY = (e.ScaleOrigin.Y - deltaY) * deltaHeight;

// Calcular as coordenadas de pixel do elemento transformado

double targetX = xOffset - (originX * Content.Width) * (currentScale - startScale);
double targetY = yOffset - (originY * Content.Height) * (currentScale - startScale);

var transX = targetX.Clamp(-Content.Width * (currentScale - 1), 0);
var transY = targetY.Clamp(-Content.Height * (currentScale - 1), 0);

// Aplica a conversão com base na mudança da origem
Content.TranslateTo(transX, transY, 0, Easing.Linear);

// Aplica o fator na imagem
Content.Scale = currentScale;
}

else if (e.Status == GestureStatus.Completed)

{
    xOffset = Content.TranslationX;
    yOffset = Content.TranslationY;

    // Coloca a imagem de novo, dentro dos limites
    Content.TranslateTo(xOffset, yOffset, 500, Easing.BounceOut);
}

```

```
}
```

## 9.4 Remoção de uma foto da listagem

Trabalharemos a remoção de uma foto em um novo processo de captura de gestos, agora o de arrastar, como em uma operação de *drag-and-drop*. O nome deste gesto que manipularemos é `Pan Gesture`. Vamos começar. Nosso primeiro passo será criar um método para registro do `Recognizer` em nosso *container*. Sendo assim, na classe `FotoListContainerControl` implemente o seguinte método. Você precisará invocá-lo no construtor da classe.

```
private void RegistrarPanGestureRecognizer ()  
{  
    var panGesture = new PanGestureRecognizer();  
    panGesture.PanUpdated += OnPanUpdated;  
    GestureRecognizers.Add(panGesture);  
}
```

Precisaremos implementar o método `OnPanUpdated()`, que implementará o comportamento para nosso arrastar de imagem. Veja o código dele na sequência. Precisaremos de duas variáveis em nível de objeto para registrar a movimentação da imagem e aplicá-las como atuais quando o processo se concluir. Verifique no `switch...case` os estados que trabalharemos para o gesto.

```
private double newXPosition, newYPosition;  
  
private void OnPanUpdated ( object sender, PanUpdatedEventArgs e)  
{  
    switch (e.StatusType)  
    {  
        case GestureStatus.Running:  
            newXPosition = Content.TranslationX = e.TotalX;  
            newYPosition = Content.TranslationY = e.TotalY;  
            break ;  
  
        case GestureStatus.Completed:  
            Content.TranslationX = newXPosition;  
            Content.TranslationY = newYPosition;  
            break ;  
    }  
}
```

Com estes códigos implementados, vamos testar nossa aplicação. Na listagem de fotos, clique em cima de uma e a arraste. Ela se movimentou e ficou em sua nova posição? Com esta nova implementação, o arrastar entre imagens já não pode ser feito pressionando a imagem. Para navegar, o usuário precisará pressionar na observação da foto, ou qualquer área que não seja a imagem, pois ela terá agora este comportamento específico de ser arrastada quando pressionada.

Agora é preciso implementar a funcionalidade que deverá ser executada quando o usuário concluir o processo de

arrastar. Para isso faremos uso do `MessagingCenter`. No código anterior, antes do `break` do segundo `case`, insira a seguinte instrução.

```
MessagingCenter.Send<FotoListContainerControl>( this , "Remover" );
```

Vamos agora para nosso code-behind de `FotosListagemView`, no método `OnAppearing()`, para implementarmos a assinatura da mensagem anterior. No código a seguir, veja que, tal qual fizemos em implementações anteriores no processo de remoção, solicitamos uma confirmação do usuário para a remoção do item desejado, em nosso caso, a foto arrastada. Ainda não vamos implementar nada para a situação positiva de desejo de remoção.

```
private async Task RemoverContainerAsync (FotoListContainerControl container)

{
    if ( await DisplayAlert ( "Confirmação" , $ "Confirma remoção da foto?" , "Yes" , "No" ) )
    {
        await viewModel.EliminarFotoAsync(container.Foto);

        (container.Parent.Parent as Frame).IsVisible = false ;

        await DisplayAlert ( "Informação" , "Atendimento removido com sucesso" , "OK" );
    }
    else
    {
        await ReposicionamentoDaImagemNoContainerAsync (container);
    }
    return ;
}

// OnAppearing

MessagingCenter.Subscribe<FotoListContainerControl>( this , "Remover" , async (container) => { await RemoverContainerAsync (container); } );
```

No código anterior, quando o usuário não confirmar o desejo de remover a foto, por talvez tê-la arrastado sem querer, estamos invocando um método ainda não implementado, chamado `ReposicionamentoDaImagemNoContainerAsync()`. Vamos esclarecer. Caso o usuário confirme que não quer remover, queremos que a foto volte para a posição original dela e é neste método que faremos isso. Uma vez mais parabenizo os colegas da discussão <https://stackoverflow.com/questions/40181090/xamarin-forms-pinch-and-pan-together/>, que foi onde me baseei para nossa solução, que está na sequência.

```
private async Task ReposicionamentoDaImagemNoContainerAsync (FotoListContainerControl container)

{
    // Definição de variáveis que auxiliarão no processo de exibição da imagem no local original

    var image = container.Content as Image;
    var currentScale = 1;
    var width = container.Width;
    var height = container.Height;
```

```

    double xOffset = 0, yOffset = 0;

    // Obtenção da posição X para centralizar a imagem

    if (image.Width * currentScale < width && width > height)
        xOffset = (width - image.Width * currentScale) / 2 - container.Content.X;
    else
        xOffset = System.Math.Max(System.Math.Min(0, xOffset), -System.Math.Abs(image.Width * currentScale - width));

    // Obtenção da posição Y para centralizar a imagem

    if (image.Height * currentScale < height && image.Height > width)
        yOffset = (height - image.Height * currentScale) / 2 - container.Content.Y;
    else
        yOffset = System.Math.Max(System.Math.Min((image.Height - (height)) / 2, yOffset), -System.Math.Abs((image.Height * currentScale - height - (image.Height - height) / 2)));

    await container.Content.TranslateTo(xOffset, yOffset, 500, Easing.BounceOut);
}

```

Teste novamente nossa aplicação. Arraste uma foto em nossa listagem, veja a mensagem de confirmação que aparece e pressione a opção de não remover. A foto voltou para sua origem? Que bom.

Agora precisamos efetivamente remover nossa imagem. Para isso, faremos uso de nosso método `DeleteAsync()` de nosso DAL, mas você lembra que para remover uma foto (ou qualquer item) da base de dados, precisamos do ID dele? Foi para isso, também, que colocamos no contêiner uma propriedade chamada `Foto`. Ela tem tudo do que precisamos.

Dando sequência, em nossa classe `FotosListagemViewModel`, precisaremos implementar o método que realizará a remoção da foto de nossa base de dados. Veja-o na sequência. O método é de fácil compreensão, pois já trabalhamos essa lógica em todos os exemplos anteriores. Observe apenas que o arquivo também precisa ser removido do dispositivo.

```

public async Task EliminarFotoAsync (AtendimentoFoto atendimentoFoto)

{
    await atendimentoFotoDAL.DeleteAsync(atendimentoFoto);
    File.Delete(DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(atendimentoFoto.CaminhoFoto));
}

```

Agora, finalmente, vamos retornar para a mensagem de remoção, na classe `FotosListagemView`. Vamos implementar a lógica para o caso de o usuário confirmar o desejo de remover a foto do atendimento. Veja o seguinte código, que deverá estar no bloco do `if` da mensagem de confirmação, lá no `onAppearing()`. Observe que, após a remoção da foto, a tornamos invisível, por meio da recuperação do frame pela hierarquia dos controles. Nosso contêiner tem como pai um `FlexLayout`, que tem como pai o NOSSO `Frame`, que desejamos ocultar.

```

await viewModel.EliminarFotoAsync(container.Foto);

(container.Parent.Parent as Frame).IsVisible = false ;

await DisplayAlert ( "Informação" , "Atendimento removido com sucesso" , "Ok" );

```

Nosso último passo é cancelar a assinatura da mensagem no `Messaging Center`, implementando a seguinte instrução no método `OnDisappearing()` da classe `FotosListagemView`.

```
MessagingCenter.Unsubscribe<FotoListContainerControl>( this, "Remover" );
```

## 9.5 Remoção de objetos de associações

Vamos agora lidar com a remoção de objetos associados. Da maneira como estamos removendo um atendimento, seus serviços e fotos ficam ainda na base de dados, e o ideal é que eles sejam removidos em conjunto. O EF Core permite que configuremos a remoção em cascata, segundo a qual, ao remover um objeto pai, seus objetos associados sejam também removidos. Entretanto, o Migrations tem limitações para adicionar esta *feature* em um banco já existente. É uma das limitações que apresentei no capítulo 5. Desta maneira, trabalharemos a remoção dos serviços e fotos no momento em que formos eliminar um atendimento. Veja no código a seguir a operacionalidade na sobrescrita do método `DeleteAsync`, no `AtendimentoDAL`. Entretanto, temos que recuperar os dados de serviços e fotos para o contexto e então enviá-los como argumentos para o método `RemoveRange()` de cada conjunto.

```
public async override Task DeleteAsync (Atendimento atendimento)
{
    using ( var context = DatabaseContext.GetContext(dbPath))
    {
        var servicosDAL = new AtendimentoItemDAL(atendimento, dbPath);
        var fotosDAL = new AtendimentoFotoDAL(atendimento, dbPath);

        context.AtendimentoItens.RemoveRange( await servicosDAL.GetAllAsync());
        context.AtendimentoFotos.RemoveRange( await fotosDAL.GetAllAsync());
        await base .DeleteAsync(atendimento);
    }
}
```

Observe que não invocamos o método `SaveChangesAsync()`, pois dependemos ainda da remoção do atendimento, que será realizado na classe base. Mas você lembra que nela também temos a captura do contexto? Então, agora, precisamos adaptar o método da classe base para que, caso ele receba um contexto, ele seja usado. Vamos adaptar nossa implementação para o `DeleteAsync()` em `DALBase` e, se vamos mudar a assinatura do método, precisamos mudar também a assinatura em nossa interface, tal qual o código a seguir.

```
// Assinatura em IDAL

Task DeleteAsync (T item, object databaseContext = null );

// Assinatura do método em AtendimentoDAL e DALBase

public virtual async Task DeleteAsync (T item, object databaseContext = null )

// using do método

using ( var context = (databaseContext == null ) ? DatabaseContext.GetContext (dbPath) : (DatabaseContext) databaseContext)
```

```

// nova invocação na AtendimentoDAL

await base.DeleteAsync(atendimento, context);

```

Mas e nossos arquivos de fotos? Como nós vimos anteriormente, ao remover uma única foto, nós também eliminávamos seu arquivo. Precisaremos fazer isso também quando todo o atendimento for removido. Desta maneira, na classe `ListagemViewModel`, de `Atendimentos`, no projeto Xamarin Forms, adaptaremos nosso método `EliminarAtendimentoAsync()` para a implementação a seguir.

```

public async Task EliminarAtendimentoAsync(Atendimento atendimento)

{
    var fotosDAL = new AtendimentoFotoDAL(atendimento, DependencyService.Get<IDBPath>().GetDbPath());

    var fotos = await fotosDAL.GetAllAsync();

    await atendimentoDAL.DeleteAsync(atendimento);

    foreach (var foto in fotos)
    {
        File.Delete(DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(foto.CaminhoFoto));
    }

    Atendimentos.Remove(atendimento);
}

```

Observe que, antes de invocarmos o `DeleteAsync()` do DAL de atendimento, guardamos a coleção de fotos em uma variável. Caso tudo tenha ocorrido bem na remoção do atendimento, uma varredura é realizada na variável `fotos`, para que os arquivos associados aos objetos sejam removidos do dispositivo. Sempre que trabalhamos remoção de objetos e/ou arquivos, seria interessante tratarmos possíveis exceções. Fica a dica para você implementar isto em seus projetos.

Temos ainda duas situações em que ocorre associação e precisamos intervir na remoção. Estou falando de `Cliente`, que está associada a `Atendimento` e `Servico`, que está associada a `AtendimentoItem`. Adotaremos aqui a situação de não permitir a remoção de objetos que possuam vínculos já registrados. Essa é uma medida de segurança comum em aplicações. Se você optar por remover os itens dependentes, você já tem a lógica que aplicamos nos exemplos anteriores. Lembre-se apenas de que, se optar por remover um cliente com cascateamento, todos os atendimentos dele deverão ser removidos e, se optar por remover um serviço com cascateamento, será preciso pensar se remove todos os atendimentos em que ele está inserido, ou apenas os serviços dos atendimentos. É uma decisão importante.

Para validarmos o que comentamos no parágrafo anterior, precisamos criar um método em nosso `ClienteDAL`, tal qual se segue. Observe que a opção foi trazer os registros existentes, pois em algum momento isso pode ser útil para a aplicação. Mas poderíamos pensar em utilizar o método `Any()` do LINQ, que retornaria um valor lógico verdadeiro, caso houvesse algum registro que respeitasse a condição.

```

public async Task<IEnumerable<Atendimento>> GetAtendimentosAsync(Cliente cliente)

{
    return await context.Atendimentos.Where(a => a.ClienteID == cliente.ClienteID).ToListAsync();
}

```

Agora, precisamos utilizar este método, e faremos isso na classe `ListagemViewModel` de `Cientes`, no método `EliminarClienteAsync()`, que precisa ser adaptado tal qual é mostrado na sequência. Precisamos fazer o *cast* para o DAL, pois o novo método não pertence à interface `IDAL`. Veja o uso do `Any()`, como comentado anteriormente. Apenas para reforçar, *cast* é o processo de forçarmos um objeto para um tipo específico. Isso é possível em nosso

exemplo pelo fato de as classes estarem na mesma hierarquia.

```
public async Task EliminarClienteAsync (Cliente cliente)

{
    var atendimentos = await (clientesDAL as ClienteDAL).GetAtendimentosAsync(cliente);

    if (atendimentos.Any())
        throw new Exception ( "O cliente a ser removido possui atendimentos registrados em seu nome" );

    await clientesDAL.DeleteAsync(cliente);
    Clientes.Remove(cliente);
}
```

Nosso novo passo é verificar se a invocação ao método anterior dispara uma exceção. Lembre-se de que estamos invocando este método em nossa mensagem, que está declarada no `OnAppearing()` de `ListagemView`, de `Clientes`. Veja na sequência como deverá ficar nossa invocação agora. Observe que, caso existam registros da associação, uma mensagem é exibida, orientando o usuário e, como visto na listagem anterior, a efetivação da remoção do cliente não ocorre.

```
private async Task RemoverClienteAsync (Cliente cliente)

{
    if ( await DisplayAlert ( "Confirmação" ,
        $"Confirma remoção de {cliente.Nome.ToUpper()}?" , "Yes" , "No" ) )

    {
        try
        {

            await this .viewModel.EliminarClienteEliminado(cliente);

            await DisplayAlert ( "Informação" , "Cliente removido com sucesso" , "Ok" );

        } catch (Exception e)
        {
            await DisplayAlert ( "Erro" , e.Message , "Ok" );
        }
    }
}

// OnAppearing

MessagingCenter.Subscribe<Cliente>( this , "Confirmação" , async (cliente) => { await RemoverClienteAsync (cliente); })
```

Poderíamos ter uma estratégia diferente para esta solução que implementamos. Se tivéssemos em `Cliente` uma propriedade para `Atendimentos`, poderíamos fazer uso dos recursos do EF Core para verificar a quantidade de atendimentos que cada cliente teria. Talvez não devêssemos optar pelo `Eager Fetch`, pois ele pode ser custoso neste caso, mas sim tentar utilizar o método `Load()` para associações. De qualquer maneira, o custo de carga seria o mesmo do que estamos utilizando, pois teríamos a carga dos objetos associados.

## 9.6 Conclusão

Chegamos ao final deste capítulo. Trabalhamos muita coisa nova e interessante. Concluímos associações de objetos, atividade que começamos nos capítulos 6, 7 e 8. Aprendemos a rotacionar a orientação de uma visão, como ocultar a barra de status do dispositivo. Vimos e trabalhamos três maneiras de capturar gestos do usuário em imagens. Fizemos bastante uso de `Dependency Service`. Concluímos o capítulo com remoção de dados que pertenciam a associações e vimos um pouco de refatoração. Conferimos também alguns recursos que chegaram com o Xamarin 3.

No próximo capítulo, trabalharemos o consumo de serviços disponíveis na Web, por meio de interações REST. Não só consumiremos os serviços, como os criaremos e os disponibilizaremos na Web. Além disso, veremos a criação de controles personalizados. Teremos como base o projeto que implementamos neste capítulo.

## C APÍTULO 10

# Custom renderers, login de acesso e consumo de serviços REST

Neste capítulo, trabalharemos o acesso a serviços REST disponibilizados na internet. Vamos criar uma visão para login em nossa aplicação. Ainda, introduziremos o uso de renderizadores personalizados para controles visuais (*custom renderers* ), um recurso extremamente útil para personalização da interface com o usuário, e veremos a criação de propriedades ligadas (*bindable properties* ).

Outro ponto que abordaremos neste capítulo são as APIs disponibilizadas por um projeto da Microsoft chamado `Xamarin Essentials`, que tem como objetivo acessar recursos físicos dos dispositivos onde a aplicação está sendo executada. Também trabalharemos o uso de `Activity Indicators`, que mostrarão ao usuário que alguma atividade está sendo realizada, evitando a sensação de aplicação congelada.

Tal qual fizemos nos últimos capítulos, toda a implementação realizada aqui terá como base o projeto do capítulo anterior. Vamos aos estudos.

## 10.1 A visão de Login

Como primeira atividade neste capítulo, teremos a etapa de realização de login, para garantir que o usuário só acesse as opções da aplicação caso ele esteja autenticado nela. Para isso, seguindo nosso padrão e metodologia adotados em todo o livro, começaremos a criação de nossa visão. Vamos começar na pasta `views`, criando uma nova pasta, chamada `Login` e, dentro dela, uma `Content Page`, chamada `LoginView`. Nela, insira o seguinte código XAML. A implementação do código só traz de novidade a propriedade `IsPassword`, no segundo `Entry`, o que faz com que os caracteres informados tenham sua exibição mascarada.

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

    x:Class = "Capitulo05.Views.Login.LoginView"
    Title = "Acesso à aplicação" >

    <ContentPage.Content>
        <ScrollView>

            <StackLayout HorizontalOptions = "CenterAndExpand" VerticalOptions = "Center" Orientation = "Vertical" >

                <Grid HorizontalOptions = "FillAndExpand" Padding = "20" >
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width = "1*" />
                    </Grid.ColumnDefinitions>
                </Grid>
            </StackLayout>
        </ContentPage.Content>
    </ContentPage>
```

```

<Grid.RowDefinitions>

    <RowDefinition Height = "Auto" />

    <RowDefinition Height = "Auto" />

    <RowDefinition Height = "Auto" />

    <RowDefinition Height = "Auto" />

</Grid.RowDefinitions>

<StackLayout Grid.Column = "0" Grid.Row = "0" HorizontalOptions = "Center" Orientation = "Vertical" >

    <Image Source = "logo.png" HeightRequest = "55" WidthRequest = "55" />

    <Label Text = "Meu Calhambeque" TextColor = "Blue" FontAttributes = "Bold" FontSize = "20" HorizontalTextAlignment = "Center" />

    <Label Text = "Livro Xamarim - CC" TextColor = "Blue" FontSize = "Small" HorizontalTextAlignment = "Center" />

</StackLayout>

<Entry Placeholder = "nome do usuário" Text = "{Binding Nome}" Grid.Column = "0" Grid.Row = "1" />

<Entry Placeholder = "senha de acesso" Text = "{Binding Senha}" IsPassword = "true" Grid.Column = "0" Grid.Row = "2" />

<Button Text = "Acessar" HorizontalOptions = "Center" Command = "{Binding LoginCommand}" Grid.Column = "0" Grid.Row = "3" />

</Grid>

</StackLayout>

</ScrollView>

</ContentPage.Content>

</ContentPage>

```

Na sequência, vamos criar nossa ViewModel. Para isso, na pasta `viewModels` crie outra, chamada `Login` e dentro dela uma nova classe, chamada `LoginViewModel`. Vamos começar nossa classe com o código responsável pela realização da autenticação. Para ilustrar, deixei dados constantes, depois mudaremos isso. Observe que o retorno é um booleano. Veja que a classe estende `BaseViewModel`.

```

namespace Capitulo05.ViewModels
{
    public class LoginViewModel : BaseViewModel
    {
        private bool RealizarLogin(string nome, string senha)
        {
            return (nome.Equals("everton") && senha.Equals("1234"));
        }
    }
}

```

Os valores enviados para o método anterior serão informados na visão e os respectivos controles estão ligados a propriedades que devemos implementar em nossa ViewModel. Inicialmente, vamos criar os campos da classe que armazenarão os valores informados na visão. Veja o código na sequência, que deve estar logo no início do corpo da classe.

```

private string nome;
private string senha;

```

Com as implementações realizadas, precisamos realizar a criação do `command` que será disparado quando o usuário pressionar o botão `Acessar`. A seguir, temos sua definição, a criação do método de seu registro e a invocação deste método no construtor da classe. Veja que no código temos a implementação de verificação de quando o botão deverá estar habilitado para o usuário. Observe também que, no envio da mensagem ao `MessagingCenter`, transformamos o retorno do método para uma `string`. Veremos que isso poderá nos beneficiar nos tratamentos de exceções. Fique atento aos `using`s que serão necessários.

```

public ICommand LoginCommand { get ; set ; }

private void RegistrarCommands ()
{
    LoginCommand = new Command(() => { MessagingCenter.Send< string >(RealizarLogin(nome, senha).ToString(), "Informacao")
    , () => { return ! String.IsNullOrEmpty( this .Nome) && ! string .IsEmpty( this .Senha); }};
}

public LoginViewModel ()
{
    RegistrarCommands();
}

```

Para finalizar a implementação da ViewModel, precisamos apenas implementar as propriedades que estarão ligadas à visão e elas estão na sequência.

```

public string Nome
{
    get { return this .nome; }

    set
{

```

```

        this .nome = value ;
        ((Command)LoginCommand).ChangeCanExecute();
        OnPropertyChanged();
    }
}

public string Senha
{
    get { return this .senha; }

    set
    {
        this .senha = value ;
        ((Command)LoginCommand).ChangeCanExecute();
        OnPropertyChanged();
    }
}

```

No code-behind de nossa visão, precisamos definir nossa ViewModel e configurá-la como `DataBinding Context`. Veja no seguinte código a definição de nossos objetos e a configuração necessária para nosso construtor. Será preciso o `using` para o `LoginViewModel`.

```

private LoginViewModel loginViewModel;

public LoginView ()
{
    InitializeComponent ();
    this.loginViewModel = new LoginViewModel();
    this.BindingContext = this .loginViewModel;
}

```

Precisamos agora implementar o código necessário para o `Messaging Center`. Os métodos devem ser implementados também no code-behind da visão. Caso a mensagem receba um valor verdadeiro, a página principal da aplicação é atualizada. Fique atento aos `usings` que deverão ser inseridos.

```

private async Task ValidarLoginAsync ( bool validacaoLogin)

{
    if (validacaoLogin)
        App.Current.MainPage = new Capitulo05.Views.MainPageView();
    else
        await DisplayAlert ( "Informação" , "Usuário e/ou senha incorretos" , "ok" );
}

protected override void OnAppearing ()

{

```

```

base.OnAppearing();

MessagingCenter.Subscribe< string >( this , "Informacao" , async (resultadoLogin) => { await ValidarLoginAsync
(resultadoLogin.ToLower() .Equals ( "true" ) ) ; });

}

protected override void OnDisappearing ()

{
    base.OnDisappearing();

    MessagingCenter.Unsubscribe< string >( this , "Informacao" );
}

```

Para finalizarmos, precisamos alterar o código de nossa classe `App`, para que initialize a visão `LoginView` como página principal. Fazemos isso com o código apresentado na sequência, que deve ser alterado no construtor da classe. Lembre-se de que esta classe está na raiz de nosso projeto Xamarin Forms.

```
MainPage = new Capitulo05.Views.Login.LoginView();
```

Com isso, já podemos testar nossa aplicação. Execute-a e compare com as imagens da figura a seguir. Você poderia pensar em utilizar o componente `Image Circle`, que já utilizamos em capítulos anteriores, o que acha?

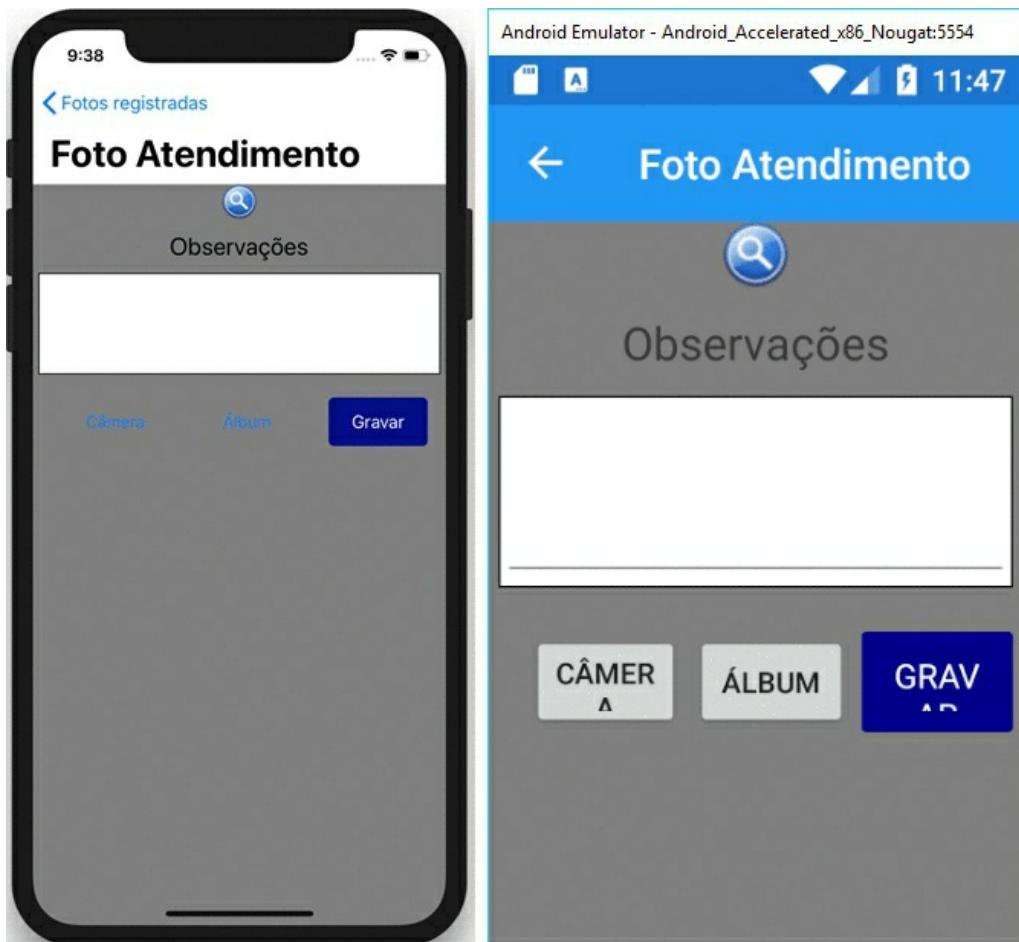


Figura 10.1: Visão de login

## 10.2 Renderizadores personalizados (Custom Renderers)

Já vimos alguns recursos oferecidos pelo Xamarin Forms, como o `Dependency Service`, além de diversos outros com que estamos trabalhando desde o início do livro. Vamos agora a um outro muito importante e interessante, que diz respeito à personalização de controles visuais, de acordo com suas necessidades, ou especificidades da plataforma em que a aplicação é executada.

Nós vamos direto para a prática, mas ao final da seção deixarei alguns links interessantes que poderão agregar valor ao processo de criação e utilização de controles personalizados. O exemplo tem apenas a finalidade didática de demonstrar para você uma enorme flexibilidade do Xamarin para personalização de controles.

Vamos lá. Existem alguns passos que devem ser seguidos sempre que precisamos personalizar um controle (*control custom*). O primeiro é criarmos em nosso projeto Xamarin Forms uma classe que estenda o controle que queremos customizar. Desta maneira, na pasta `Controls` do projeto, crie uma classe chamada `LoginEntry`, com o código apresentado na sequência. Veja que estamos estendendo `Entry`.

```
namespace Capitulo05.Controls
{
    public class LoginEntry : Entry
    {
    }
}
```

O passo seguinte é planejarmos o que o controle terá de maneira adicional, que possa ser configurado pelo usuário (programador), quer seja via código C#, quer seja por XAML. Para o exemplo que trabalharemos, queremos que nosso `Entry` personalizado possua bordas arredondadas e que a borda tenha uma cor quando o foco estiver no controle e outra cor quando o controle perder o foco. Lembre-se de que um `Entry` não possui nativamente propriedades para bordas, quer seja cor, espessura ou se é ou não arredondada. Criaremos duas propriedades. Se você julgar interessante, depois fica a seu cargo implementar as demais. O código a seguir deve estar dentro da classe anteriormente criada. Observe que os valores são recuperados e atribuídos a objetos ainda não criados, mas já os implementaremos.

```
public Color BorderColorOnFocus
{
    get { return (Color)GetValue(BorderColorOnFocusProperty); }
    set { SetValue(BorderColorOnFocusProperty, value); }
}

public Color BorderColorLostFocus
{
    get { return (Color)GetValue(BorderColorLostFocusProperty); }
    set { SetValue(BorderColorLostFocusProperty, value); }
}
```

Com as propriedades criadas, resta-nos configurá-las como `ligáveis` (*bindables*) e realizaremos isso com o código da sequência, que pode ser declarado no início da classe. O código está na sintaxe exigida pelo Xamarin. O método `Create()` recebe o nome da propriedade a que `BindableProperty` se refere, o tipo de dado que será armazenado nela, a

classe onde ela estará disponível e, por fim, o valor padrão para a propriedade, caso ele não seja atribuído.

```
public static readonly BindableProperty BorderColorOnFocusProperty = BindableProperty.Create(nameof(BorderColorOnFocus),  
typeof(Color), typeof(LoginEntry), Color.Red);  
  
public static readonly BindableProperty BorderColorLostFocusProperty =  
BindableProperty.Create(nameof(BorderColorLostFocus), typeof(Color), typeof(LoginEntry), Color.Black);
```

Feito isso, é interessante realizar um build em seu projeto. Você pode fazer isso clicando com o botão direito do mouse sobre o nome do projeto e então em `Compilar`.

Agora vamos utilizar estas propriedades em nosso XAML de login. Na visão `LoginView`, adicione o seguinte código no início do XAML, antes do `Title`. Nós já utilizamos esta declaração de namespaces anteriormente.

```
xmlns:local="clr-namespace:Capitulo05.Controls;assembly=Capitulo05"
```

Agora, vamos substituir o `Entry` que pede o nome do usuário pelo nosso novo controle, com a definição das duas novas propriedades. Veja isso no código a seguir. Você pode substituir os dois `Entries`, se quiser. Mas lembre-se de que nossa implementação tem apenas o fim didático.

```
<local:LoginEntry Placeholder = "nome do usuário" Text = "{Binding Nome}" Grid.Column = "0" Grid.Row = "1" BorderColorOnFocus  
"Red" BorderColorLostFocus = "Black" />
```

Já temos as propriedades implementadas em nosso controle, mas ele ainda não está pronto. Precisamos implementar um renderizador para ele nas duas plataformas. Nos projetos específicos de plataforma, crie uma pasta chamada `CustomRenderers` e dentro dela crie uma classe chamada `LoginEntryCustomRenderer`. Na sequência, veja a implementação para o projeto iOS. Note que, antes da definição do namespace, um atributo registra o novo renderizador para nosso controle `LoginEntry`. A classe estende `EntryRenderer`. A formatação do novo `controller` está no método sobrescrito `OnElementChanged()`. O comportamento do método começa com uma verificação de existência de objeto para `control`, e as quatro primeiras linhas dentro do `if()` referem-se ao `Padding`. Na sequência são registrados comportamentos para os eventos de foco no controle, que é onde as cores das bordas serão lidas e atribuídas. Finalizando o método, temos a definição da borda arredondada e sua largura.

```
[assembly: ExportRenderer(typeof(LoginEntry), typeof(LoginEntryCustomRenderer))]  
namespace Capitulo05.iOS.CustomRenderers  
{  
    public class LoginEntryCustomRenderer : EntryRenderer  
    {  
        protected override void OnElementChanged(ElementChangedEventArgs<Entry> e)  
        {  
            base.OnElementChanged(e);  
            if (Control != null && e.NewElement != null)  
            {  
                Control.LeftView = new UIView(new CGRect(0, 0, 15, 0));  
                Control.LeftViewMode = UITextFieldViewMode.Always;  
                Control.RightView = new UIView(new CGRect(0, 0, 15, 0));  
                Control.RightViewMode = UITextFieldViewMode.Always;  
                Control.Layer.BorderColor = (Element as LoginEntry).BorderColorLostFocus.ToCGColor();  
  
                e.NewElement.Unfocused += (sender, evt) =>  
                {  
                    Control.Layer.BorderColor = (Element as LoginEntry).BorderColorLostFocus.ToCGColor();  
                };  
            }  
        }  
    }  
}
```

```
        };

        e.NewElement.Focused += (sender, evt) =>
    {
    Control.Layer.BorderColor = (Element as LoginEntry).BorderColorOnFocus.ToCGColor();
};

Control.Layer.CornerRadius = 5;
Control.Layer.BorderWidth = 1;
}

}
}
```

Para finalizar esta etapa, vamos implementar a classe no projeto Android. A lógica inicial para o Android é semelhante à que vimos para o iOS. Entretanto, para a borda, precisamos criar um objeto `GradientDrawable` e realizar nele as configurações. Primeiro, é definido o ângulo de arredondamento da borda, depois a largura e cor no momento em que o controle for criado. Na sequência, o `Padding` é definido, de maneira mais simples que no iOS. Após isso, temos o registro dos eventos e, ao final, atribuímos ao controle a nova configuração.

```
[assembly: ExportRenderer(typeof(LoginEntry), typeof(LoginEntryCustomRenderer))]

namespace Capitulo05.Droid.CustomRenderers
{
    public class LoginEntryCustomRenderer: EntryRenderer
    {
        public LoginEntryCustomRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(ElementChangedEventArgs<Entry> e)
        {
            base.OnElementChanged(e);

            if (Control != null && e.NewElement != null)
            {
                GradientDrawable gd = new GradientDrawable();
                gd.SetCornerRadius(10);
                gd.SetStroke(2, (Element as LoginEntry).BorderColorLostFocus.ToAndroid());
                Control.SetPadding(10, 5, 10, 5);

                e.NewElement.Unfocused += (sender, evt) =>
                {
                    gd.SetStroke(2, (Element as LoginEntry).BorderColorLostFocus.ToAndroid());
                };
                e.NewElement.Focused += (sender, evt) =>
                {
                    gd.SetStroke(2, (Element as LoginEntry).BorderColorOnFocus.ToAndroid());
                };
            }

            Control.SetBackground(gd);
        }
    }
}
```

```
 }
}
```

No Android, para quem está acostumado a trabalhar com o Material Design e configurações em arquivos XML, temos uma segunda opção. Na pasta `Resources/drawable`, crie um arquivo XML chamado `loginentrycustomrenderer.xml`. Note que todas as letras estão em minúsculo, pois é um requisito da plataforma. Insira o conteúdo a seguir no arquivo. Faça uma leitura atenciosa e note que definimos no XML os mesmos componentes que temos no código anterior.

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

<selector xmlns:android = "http://schemas.android.com/apk/res/android" >

    <item android:state_focused = "true" >

        <shape android:shape = "rectangle" >

            <gradient
                android:startColor = "@color/entry_background"
                android:endColor = "@color/entry_background"
                android:angle = "270" />

            <stroke
                android:width = "1dp"
                android:color = "#fc0505" />

            <corners
                android:radius = "4dp" />

        </shape>

    </item>

    <item>

        <shape android:shape = "rectangle" >

            <gradient
                android:startColor = "@color/entry_background"
                android:endColor = "@color/entry_background"
                android:angle = "270" />

        </shape>

    </item>

</selector>
```

```

<stroke
    android:width = "1dp"

    android:color = "@color/entry_border" />

<corners
    android:radius = "4dp" />

</shape>

</item>

</selector>

```

Utilizamos algumas cores específicas, que não existem nos recursos disponíveis no ambiente da aplicação. Desta maneira, na pasta `Resources/values`, precisamos criar um arquivo `colors.xml`, com o conteúdo apresentado na sequência.

```

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

<resources>

<color name = "entry_background" > #faf8cd </color>

<color name = "entry_border" > #050505 </color>

</resources>

```

Agora, uma nova versão para o renderizador do Android. No código a seguir, observe que o `drawable` é recuperado do arquivo XML e atribuído ao controle. Este é um recurso interessante para o Android, para quem está acostumado com o Material Designer. Note que não utilizamos no renderizador as propriedades que definimos no controle, mas elas poderiam ser utilizadas para sobrescrever as configurações do XML. Caso `loginentrycustomrenderer` não seja reconhecido, na `Solution Explorer`, clique com o botão direito do mouse sobre o nome do arquivo e em propriedades. Na propriedade `Custom Tool`, informe `MSBuild:UpdateGeneratedFiles`.

```

[assembly: ExportRenderer(typeof(LoginEntry), typeof(LoginEntryCustomRenderer))]
namespace Capitulo05.Droid.CustomRenderers
{
    public class LoginEntryCustomRenderer: EntryRenderer
    {
        public LoginEntryCustomRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(ElementChangedEventArgs<Entry> e)
        {
            base.OnElementChanged(e);
            if (Control != null && e.NewElement != null)
            {
                Control.Background = Context.GetDrawable(Resource.Drawable.loginentrycustomrenderer);
            }
        }
    }
}

```

```
}
```

Muito bem, chegamos ao fim da seção e, como prometido, deixarei alguns links interessantes para você estudar, quando julgar melhor, sobre este interessantíssimo tema.

- <https://docs.microsoft.com/pt-br/xamarin/xamarin-forms/app-fundamentals/custom-renderer/> traz a documentação oficial do Xamarin sobre o assunto, com vários tópicos e exemplos.
  - <https://docs.microsoft.com/pt-br/xamarin/xamarin-forms/xaml/bindable-properties/> , com documentação oficial sobre propriedades ligáveis.
  - <https://docs.microsoft.com/pt-br/xamarin/android/user-interface/android-designer/> , com diversos recursos para quem quer utilizar o Material Design em projetos Android.

Uma última observação antes de fechamos a seção. Você notou que, para criarmos renderizadores personalizados, precisamos conhecer um pouco como cada plataforma trabalha certo? É importante você acessar o link <https://docs.microsoft.com/en-us/xamarin/ios/> que traz bastante informação específica sobre o iOS e o Xamarin e o <https://docs.microsoft.com/en-us/xamarin/android/> sobre o Android. Faça isso quando julgar necessário.

### 10.3 O login com o serviço REST

Precisamos agora adaptar nossa aplicação Xamarin Forms para que valide o nome e senha do usuário em nosso servidor. Nosso primeiro passo será instalar um componente para manipulação de dados em JSON. Para isso, clique com o botão direito no projeto Xamarin Forms, e então em Gerenciar pacotes Nuget . Na janela de pesquisa, digite `Newtonsoft.Json` e o instale.

Para que você possa replicar em sua máquina todo o trabalho que desenvolveremos a partir desta seção, disponibilizarei um documento no apêndice do livro (*13. Apêndice - Criação de serviços REST*), para que você possa, se quiser, criar os serviços também. É importante que você saiba que, para consumirmos um serviço REST, precisamos de especificações técnicas, como o tipo de retorno, nome e tipos de dados que ele recebe, pois o serviço a ser consumido pode ser, e muitas vezes é, uma implementação realizada por terceiros.

Agora, criaremos uma classe que será responsável pela criação do recurso que nos gerará a conexão com nosso serviço REST. No projeto Xamarin Forms, crie uma pasta chamada `services` e, dentro dela, uma classe chamada `ServicesPrepare`, com o código da sequência.

```
namespace Capitulo05.Services
{
    public class ServicesPrepare
    {
        public static HttpClient GetHttpClient()
        {
            var proxy = WebRequest.DefaultWebProxy;
            HttpClientHandler clientHandler = new HttpClientHandler()
            {
                Proxy = proxy,
                Proxy = new WebProxy("http://200.124.29.64:4239"),
                UseProxy = (proxy != null)
            };
            //HttpClient client = new HttpClient(clientHandler);
        }
    }
}
```

```

        client.BaseAddress = new Uri("https://meucalhambeque02.herokuapp.com/");

        return client;
    }
}
}

```

O primeiro ponto é observar que o método `GetHttpClient()` está definido como estático, o que nos permitirá o consumo sem ter que instanciar uma classe. Na primeira instrução da classe, veja que realizamos uma atribuição ao objeto `proxy`, que deverá receber as configurações existentes no dispositivo em relação ao proxy, caso esteja configurado.

Importante: eu tentei utilizar o aplicativo em uma rede com proxy, pelos emuladores, e não foi possível consumir o serviço, entretanto, tudo funcionou quando foi utilizado um dispositivo real. Deixei como comentário a informação de um proxy específico. Note a definição de uma URI base para o objeto. Ao final do método, um `HttpClient` é retornado.

O nome `meucalhambeque02` é um endereço onde a aplicação servidora está funcionando, para que você possa testar. Mas, se você criou seu projeto servidor, com auxílio da documentação auxiliar, disponibilizada no início da seção, utilize o nome de sua aplicação. Deixo claro que você não precisa implementar os serviços do documento complementar para implementar os códigos deste e do próximo capítulo.

Agora mudaremos o método `RealizarLoginAsync()`, que está em nossa `LoginViewModel`, para a implementação apresentada na sequência. Nossa primeira mudança se refere ao tipo de dado retornado pelo método, que deixa de ser um `Boolean` para ser um `String`. Adotaremos esta técnica, pois exceções podem ser disparadas, tanto pelo serviço como pelo nosso método cliente e, a interface que está utilizando o método precisa receber uma mensagem sobre o problema ocorrido. Observe que o cliente para a conexão está sendo obtido dentro de um bloco `using()` e que todo o bloco está inserido em uma cláusula `try..catch`, pois é interessante que você faça tratamentos em caso de problemas com a conexão.

Ainda na invocação ao `PostAsync()`, veja a URI que encaminhamos, ela complementará a que definimos como base na classe `ServicesPrepare`. Em caso de execução bem-sucedida, a resposta é retornada para o método chamador. Caso a conexão ocorra sem exceções, mas retorne um status de falha, o texto do erro é retornado. Da mesma maneira, se ocorrer alguma exceção, a mensagem dela também é retornada. Lembre-se de que nosso serviço em Java foi implementado para retornar um valor lógico, `true` ou `false` e o estamos lendo como `string`, devido à técnica que estamos utilizando.

```

private async Task<string> RealizarLoginAsync ( string nome, string senha)

{
    var json = JsonConvert.SerializeObject( new { nome = nome, senha = senha });

    var content = new StringContent(json, Encoding.UTF8, "application/json" );

    try
    {
        using ( var client = ServicesPrepare.GetHttpClient() )

        HttpResponseMessage response = await client.PostAsync( "autenticacao/login" , content);

        if (response.IsSuccessStatusCode)

            return ( await response.Content.ReadAsStringAsync());

        return response.StatusCode.ToString();
    }
}

```

```

        }
    }

    catch (Exception ex)
    {
        return ex.Message;
    }
}

```

No código anterior, alteramos o tipo de retorno para o método `RealizarLoginAsync()`, o que nos leva a alterar a chamada a este método, em nosso `Command LoginCommand`, tal qual segue na listagem. Observe que apenas tiramos o `ToString()` que estava encadeado ao método e inserimos o `await` em sua invocação. Com estas mudanças, precisamos também alterar a assinatura da mensagem na visão `LoginView` e o método que será disparado quando ela ocorrer. Estas alterações também estão na sequência. Veja que agora podemos exibir ao usuário mensagens que possam ser retornadas por alguma exceção.

```

// Método RegistrarCommands() na classe LoginViewModel

LoginCommand = new Command( async () => { MessagingCenter.Send< string > ( await RealizarLoginAsync ( nome, senha ) ,
"Informacao" ) ; },

() => { return ! string . IsNullOrEmpty( this . Nome ) && ! string . IsNullOrEmpty( this . Senha ); });




// OnAppearing() na classe LoginView

MessagingCenter.Subscribe< string > ( this , "Informacao" , async ( resultadoLogin ) => { await ValidarLoginAsync
( resultadoLogin ); });




// Método também na classe LoginView

private async Task ValidarLoginAsync ( string validacaoLogin )

{
    if ( validacaoLogin . ToLower () . Equals ( "true" ) )

        App . Current . MainPage = new Capitulo05 . Views . MainPageView();

    else if ( validacaoLogin . ToLower () . Equals ( "false" ) )

        await DisplayAlert ( "Informação" , "Usuário e/ou senha incorretos" , "ok" );

    else

        await DisplayAlert ( "Informação" , validacaoLogin , "ok" );
}

```

Como o processo de autenticação depende de uma conexão, pode ocorrer uma certa demora na espera de uma resposta. É interessante exibirmos algo para o usuário, para que ele veja que a aplicação está funcionando. O Xamarin oferece um controle para indicação de atividade e vamos fazer uso dele. Em nossa visão `LoginView`, logo após a tag `</Grid>`, insira a instrução a seguir. Veja que temos a ligação da propriedade `IsRunning` com a propriedade `LoginViewModel`, que precisaremos implementar em nossa classe `ViewModel`, logo após o XAML. Este é um recurso que estamos vendendo pela primeira vez.

```
<ActivityIndicator IsRunning = "{Binding Autenticando}" Color = "Black" />
```

```

private bool autenticando;

public bool Autenticando
{
    get { return this.autenticando; }

    set
    {
        this.autenticando = value;
        OnPropertyChanged();
    }
}

```

Para que possamos usar o controle inserido anteriormente, precisamos alterar novamente nosso `Command` de `LoginCommand`, que está no método `RegistrarCommands()`, tal como é apresentado na sequência. Podemos testar nossa aplicação agora. Tente informar dados errados. Depois, altere a URL do serviço para uma que não exista, em seguida, desative sua rede e teste novamente. Por fim, insira os dados corretos e veja a atualização da página principal da aplicação. Você pode pensar em tratar melhor o erro recebido quando a URL informada estiver errada e quando testarmos sem rede.

```

LoginCommand = new Command( async () => {

    this.Autenticando = true;

    MessagingCenter.Send< string >( await RealizarLoginAsync ( nome, senha ) , "Informacao" );
    this.Autenticando = false;

},
() => { return ! string.IsNullOrEmpty( this.Nome) && ! string.IsNullOrEmpty( this.Senha); });

```

## 10.4 Xamarin Essentials

A Microsoft está desenvolvendo um conjunto de APIs chamada *Xamarin Essentials*, que oferece um conjunto de classes muito importante e útil para manipulação direta de recursos físicos dos dispositivos onde sua aplicação está sendo executada. Nós usaremos a verificação de existência de uma conexão com a internet para que o serviço REST que estamos implementando no material adicional possa ser executado. Um detalhe maior sobre o *Xamarin Essentials* pode ser obtido em <https://docs.microsoft.com/en-us/xamarin/essentials/> e recomendo que você o leia.

Nosso primeiro passo é instalarmos o Nuget em nossos projetos, que são o do *Xamarin Forms* e os específicos de cada plataforma. Lembra como? Botão direito sobre o nome da solução e Gerenciar Pacotes Nuget da Solução. Procure por *Xamarin Essentials*, mas deixe marcada a opção Incluir pré-lançamentos, pois esta API ainda não está no status de release. Clique sobre o pacote da Microsoft e marque, do lado direito, os projetos para instalação e então a realize. Vamos trabalhar com a API `Connectivity`. Esta API nos possibilita trabalhar com estados relacionados a conexões de rede. Nossa intenção será prover, de maneira simples, mecanismos para identificar se existe uma conexão com a internet disponível.

Vamos preparar nossos projetos. Para que possamos usar este recurso em um projeto *Android*, precisamos liberar o acesso para `AccessNetworkState` para a aplicação. Você verá na documentação que existem três maneiras de fazer isso, mas vou apresentar a mais direta, que é realizada no arquivo `AndroidManifest.xml`, dentro da pasta `Properties` do projeto *Android*. Com o arquivo aberto, insira nele a seguinte instrução, dentro da tag `<manifest>`. Para o *iOS*, não há necessidade de nenhuma configuração.

```
<uses-permission android:name = "android.permission.ACCESS_NETWORK_STATE" />
```

Para testarmos nossa aplicação, antes do `return` do método `GetHttpClient()`, da classe `ServicesPrepare`, insira a implementação a seguir. O primeiro teste eu fiz sem a verificação do WiFi em um dispositivo físico com o iOS. Acessei a aplicação e depois desabilitei o acesso a dados e WiFi. Funcionou.

É preciso acessar antes a aplicação, pois sem internet haverá falha, uma vez que é preciso uma certificação na Apple para executar aplicações que não foram baixadas pela Apple Store. O teste no Android também foi tranquilo, sem o problema de certificação que existe da Apple, entretanto, no emulador, foi necessário habilitar o WiFi. A documentação traz algumas observações e limitações desta API quando existe o acesso a uma rede WiFi que esteja sem conexão à internet, retornando a falsa informação de que existe disponibilidade. Mas certamente quando liberarem a versão oficial este problema estará resolvido.

```
if (!Connectivity.Profiles.Contains(ConnectionProfile.WiFi))  
    throw new Exception ( "Sem acesso à WIFI" );  
  
if (Connectivity.NetworkAccess != NetworkAccess.Internet)  
    throw new Exception ( "Sem acesso à internet" );
```

## 10.5 Conclusão

Chegamos ao final deste nosso capítulo. Ele foi rápido, com algumas novidades, recursos e técnicas. Foi possível criarmos serviços REST em Java, distribuí-los na internet (pela documentação complementar) e os consumirmos em nossa aplicação Xamarin. Vimos ainda a criação de renderizadores personalizados para controles Xamarin, com a implementação de propriedades ligáveis e tivemos uma introdução ao *Xamarin Essentials*. Também trabalhamos o uso de `Activity Indicators`, que mostraram ao usuário que alguma atividade estava sendo realizada, para não dar a sensação de aplicação travada. Finalizaremos o livro no próximo capítulo, implementando um CRUD completo com uso de REST.

## C APÍTULO 11

# O CRUD de peças com o consumo de serviços REST

Chegamos ao último capítulo de conteúdo do livro. Concluiremos o acesso a serviços REST disponibilizados na internet, agora com a implementação de um CRUD. Com isso, poderemos trabalhar a sincronização de dados com uma base central. Para que você possa replicar todo o trabalho em sua máquina, lembre-se do documento complementar disponibilizado no apêndice, que será necessário também para este capítulo, para que você possa criar os serviços também.

Veremos também a possibilidade de atualizar um `ListView` com o recurso de puxar para atualizar (*Pull to refresh*). Vamos aos estudos.

## 11.1 A classe Peca, com arquivo de imagem, suas visões e DAL para o REST

Nossa aplicação está persistindo os dados localmente, no dispositivo, o que não está errado. Entretanto, normalmente, as aplicações móveis têm seus dados publicados na internet, e não se preocupam com qual estrutura os mantém. Sabemos que a base de dados principal de uma aplicação pode estar na Web, o que nos leva a rever a situação de nossas chaves primárias, que, em nosso caso, estão sendo autoincrementadas localmente, no dispositivo. O problema disso é que todos os dispositivos que tiverem nossa aplicação instalada inevitavelmente terão, em algum momento, os mesmos valores para a propriedade identificadora (chave primária) dos objetos de classes de nossa aplicação, se for nossa aplicação a geradora de inserções destes dados.

Conhecendo esse problema, vamos implementar um novo modelo de nossa área de negócio, que fará uso de uma chave primária que seja única para todos os dispositivos que utilizem nossa aplicação.

Trabalharemos com `Peças` que podem ser utilizadas em um atendimento. Vamos realizar esta nova implementação por etapas, sendo que a primeira é a definição de nossa chave primária para este novo modelo. No projeto `IDPropertiesEF`, na pasta `Models`, crie uma nova classe chamada `PecaIDProperty` com o código apresentado na sequência, que tem, além do ID, duas propriedades que já conhecemos de capítulos anteriores. Fique atento aos namespaces e observe o tipo de dados, `Guid`, para `PecaID`. Um GUID (*Globally Unique Identifier*) é um valor que o C# garante ser único, sem possibilidade de repetição. Um GUID também pode ser chamado de UUID (*Universally Unique Identifier*). Este valor segue o layout de grupos de 8, 4, 4, 4 e 12 dígitos hexadecimais e minúsculos, separados por hifens. Um exemplo de valor válido para um GUID: 56269d788-517d-46ed-8fed-65a3f77ed1ed.

```
namespace CasaDoCodigo.Models
{
    public class PecaIDProperty
    {
        [Key]
        public Guid PecaID { get; set; }

        [NotMapped]
        public virtual string ValorFormatado { get { return string.Empty; } }

        [NotMapped]
        public bool NotificarListView { get; set; }
    }
}
```

Agora, com o identificador devidamente implementado, vamos definir a classe `Peca`, que deve estar na pasta `Cadastros` do projeto `oficinaModels`, tal qual é apresentado na sequência. Note a existência de uma propriedade que não pertence diretamente ao modelo, a `sincronizado`. Quando trabalhamos com aplicações móveis com base de dados sincronizada, temos o problema de identificarmos quando o dado que inserimos localmente foi efetivamente

sincronizado com a base central. Precisaremos de uma propriedade de controle para isso, que deve resolver esse problema. Note também que teremos uma imagem que, inicialmente, seguirá a mesma lógica que vimos no capítulo anterior.

```
namespace CasaDoCodigo.Models
{
    public class Peca : PecaIDProperty
    {
        public string Nome { get; set; }
        public double Valor { get; set; }
        public string CaminhoImagem { get; set; }
        public bool Sincronizado { get; set; }

        public override string ValorFormatado => string.Format("R$ {0:N2}", this.Valor);
    }
}
```

Vamos abrir um espaço aqui para ponderações, problemas e ideias para dados de aplicações móveis. A solução que estamos implementando para Peças compartilhará com todos os usuários os dados registrados em qualquer dispositivo. Isso pode trazer alguns problemas. Como ficam as atualizações e remoções de dados da base? Você pode alterar ou remover um dado e ter outros usuários o utilizando, em outros dispositivos. Uma solução muito comum é esquecer a persistência local e trabalhar apenas com aplicações online. Esta solução, dependendo do foco de sua aplicação, pode não ser viável, pois ela não poderia ser utilizada se o dispositivo não tiver acesso à internet (ou uma rede local). Neste contexto a sincronização é ótima. Mas voltemos ao problema de alteração e remoção. O que fazer? Uma ideia é tornar estas funcionalidades disponíveis para apenas alguns usuários e enviar notificações para os dispositivos se atualizarem. Mas e se sua aplicação tiver os dados centralizados apenas para permitir que você os acesse de qualquer dispositivo? Isso seria legal também, mas precisaríamos inserir em nosso modelo uma identificação para o usuário e, então, todas as operações de recuperação e atualização de dados levariam em consideração não apenas o atributo identificador, mas também o usuário. Poderíamos mapear isso para uma chave composta, inclusive. Bastante coisa para se pensar, não é?

Com nosso modelo implementado, precisamos agora trabalhar a situação de persistência de seus objetos em uma base de dados. A princípio podemos pensar que está tudo pronto, basta criarmos uma classe DAL para o modelo. Não é bem assim, pois temos nossas classes DAL baseadas em uma interface e, dentro da interface, temos alguns métodos que recebem o ID do objeto que será manipulado e, até então, nosso ID era definido como `long`, o que, para nosso objetivo didático foi a melhor solução. Com a nova definição de tipo para o ID (para a classe `Peca`), uma vez que nossa aplicação fará uso de uma base de dados centralizada, precisamos adaptar nossa interface.

Veja o código que deve ser inserido na `IDAL`, que está na pasta `DataAccess` do projeto `Interfaces`. Você precisará inserir `System` nos usings. Uma discussão interessante e de rápida leitura sobre o tipo de identificador pode ser vista em <https://exceptionnotfound.net/integers-vs-guids-the-great-primary-key-debate/>.

```
Task<T> UpdateAsync (T item, Guid itemID, string propriedadeID, bool sincronizado = false, params object [] associatedObjec
Task<T> GetByIdAsync (Guid id, params string [] includeProperties);
```

Adicionamos dois novos argumentos para o método `UpdateAsync()`. O primeiro receberá o nome da propriedade identificadora da classe, para que, dinamicamente, o valor de um novo GUID possa ser atribuído, antes da inserção. O segundo novo parâmetro receberá, opcionalmente, um valor que indicará se o objeto a ser persistido pelo método deverá ser visto como sincronizado. Logo veremos o uso destes argumentos. Mantenha os métodos anteriores na interface.

Com a inclusão que fizemos na interface, foi gerado um erro na classe `DALBase`, pois os novos métodos não estão implementados nela e precisamos corrigir isso. Para o `GetByIdAsync()`, podemos realizar a nova implementação diretamente no `DALBase`, como segue. Observe o uso de `FindAsync()`. Veja o `Load` para as propriedades de associação.

```

public virtual async Task<T> GetByIdAsync (Guid id, params string [] includeProperties)
{
    using ( var context = DatabaseContext.GetContext(dbPath))
    {
        var result = await context.Set<T>().FindAsync(id);

        for ( int i = 0; i < includeProperties.Count(); i++)
        {
            context.Entry(result).Reference(includeProperties[i]).Load();
        }

        return result;
    }
}

```

Vamos implementar nosso novo método `UpdateAsync()`, na sequência. Compare-o com o que já temos na classe `DALBase`. A lógica é praticamente a mesma. A mudança que temos no código, além do tipo de `itemID`, é a atribuição do valor para a propriedade identificadora. Veja a reflexão para atribuir o valor a ela.

```

public async virtual Task<T> UpdateAsync (T item, Guid itemID, string propriedadeID, bool sincronizado = false , params object[] associatedObjects)
{
    using ( var context = DatabaseContext.GetContext(dbPath))
    {

        foreach ( var associated in associatedObjects)
        {
            context.Entry(associated).State = EntityState.Unchanged;
        }

        if (itemID != Guid.Empty)
        {
            item.GetType().GetProperty( "Sincronizado" ).SetValue(item, sincronizado);
            context.Update(item);
        }
        else
        {
            item.GetType().GetProperty(propriedadeID).SetValue(item, Guid.NewGuid());
            await context.AddAsync(item);
        }

        await context.SaveChangesAsync();
    }

    return item;
}

```

Agora, vamos implementar a especialização de `DALBase` para peças. No projeto `SQLiteEF`, na pasta `DAL`, crie uma classe chamada `PecaDAL`, com o código a seguir.

```

namespace CasaDoCodigo.DAL
{
    public class PecaDAL : DALBase<Peca>
    {
        public PecaDAL(string dbPath) : base(dbPath)
        {
        }
    }
}

```

Trabalharemos agora na criação de nossa interface com o usuário, seguindo os mesmos caminhos que já realizamos para os modelos que temos funcionando na aplicação, e começaremos pela visão que listará todas as peças. Procurarei inserir as implementações específicas ao capítulo por partes, mas aquilo que já é de nosso conhecimento terá seu código diretamente apresentado.

Em nosso projeto Xamarin Forms, na pasta `views`, crie outra pasta, chamada `Pecas` e, dentro dela, uma `Content Page` chamada `ListagemView` com o XAML apresentado na sequência. Note que estamos deixando o layout preparado para receber as imagens.

```

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"
              xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
              x:Class = "Capitulo05.Views.Pecas.ListagemView"

              xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"

              ios:Page.UseSafeArea = "true"

              Title = "Peças" >

    <ContentPage.ToolbarItems>

        <ToolbarItem Icon = "plus.png" Command = "{Binding NovoCommand}" />

    </ContentPage.ToolbarItems>

    <ContentPage.Content>

        <ListView x:Name = "listView" Margin = "5,5,0,0" HasUnevenRows = "True" ItemsSource = "{Binding Pecas}" SelectedItem = "{Bind: PecaSelecionada}" >

            <ListView.ItemTemplate>

                <DataTemplate>

                    <ViewCell>

```

```

<ViewCell.ContextActions>

<MenuItem Command = "{Binding Path=BindingContext.EliminarCommand, Source={x:Reference listView}}" CommandParameter = "{Binding .}" Text = "Remover" IsDestructive = "True" />

</ViewCell.ContextActions>

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width = "60" />

<ColumnDefinition Width = "*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height = "60" />

</Grid.RowDefinitions>

<StackLayout Padding = "5" HeightRequest = "60" VerticalOptions = "Center" HorizontalOptions = "Center" >

<Image Source = "{Binding CaminhoImagen}" Grid.Column = "0" Grid.Row = "0" HeightRequest = "50" />

</StackLayout>

<StackLayout Padding = "10" Grid.Column = "1" Grid.Row = "0" >

<Label Text = "{Binding Nome}" FontSize = "18" FontAttributes = "Bold" />

<Label Text = "{Binding ValorFormatado}" FontSize = "14" />

</StackLayout>

</Grid>

</ViewCell>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

```

```
</ContentPage.Content>
```

```
</ContentPage>
```

Para gerenciar a ligação da visão com nosso modelo e os dados que devem ser manipulados por ela, vamos criar nossa ViewModel. No projeto Xamarin Forms, na pasta `ViewModels`, crie uma nova pasta, chamada `Pecas` e, dentro dela, a classe `ListagemViewModel`, com o código apresentado na sequência. Não se assuste com todo o código, é tudo semelhante ao que já fizemos, apenas focado para Peças.

```
namespace Capitulo05.ViewModels.Pecas
{
    public class ListagemViewModel : BaseViewModel
    {
        private IDAL<Peca> pecasDAL;
        public ObservableCollection<Peca> Pecas { get; set; }

        public ICommand NovoCommand { get; set; }
        public ICommand EliminarCommand { get; set; }

        public ListagemViewModel()
        {
            pecasDAL = new PecaDAL(DependencyService.Get<IDBPath>().GetDbPath());
            Pecas = new ObservableCollection<Peca>();
            RegistrarCommands();
        }

        private Peca pecaSeleccionada;
        public Peca PecaSeleccionada
        {
            get { return pecaSeleccionada; }
            set
            {
                if (value != null)
                {
                    pecaSeleccionada = value;
                    MessagingCenter.Send<Peca>(pecaSeleccionada, "Mostrar");
                }
            }
        }

        public async Task AtualizarPecasAsync()
        {
            var pecas = await pecasDAL.GetAllAsync();
            Pecas.SincronizarColecoes(pecas);
        }

        private void RegistrarCommands()
        {
            NovoCommand = new Command(() =>
            {

```

```

        MessagingCenter.Send<Peca>(new Peca(), "Mostrar");
    });

    EliminarCommand = new Command<Peca>((peca) =>
    {
        MessagingCenter.Send<Peca>(peca, "Confirmação");
    });
}

public async Task EliminarPecaAsync(Peca peca)
{
    await pecasDAL.DeleteAsync(peca);
    Pecas.Remove(peca);
}
}
}
}

```

Vamos agora criar a visão referente ao CRUD para `Pecas`. Na pasta `Views/Pecas`, crie um novo Content Page, chamado `CRUDView`, com o conteúdo apresentado na sequência para seu XAML.

```

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

<ContentPage xmlns = "http://xamarin.com/schemas/2014/forms"

    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"

    xmlns:ios = "clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"

    ios:Page.UseSafeArea = "True"

    x:Class = "Capitulo05.Views.Pecas.CRUDView" >

    <ContentPage.Content>

        <ScrollView>

            <StackLayout>

                <Image Source = "{Binding CaminhoImagen}" Margin = "10" />

                <TableView Intent = "Form" >

                    <TableRoot>

                        <TableSection Title = "Dados da peça" >

                            <EntryCell Label = "Nome:" Text = "{Binding Nome}" />

```

```

<EntryCell Label = "Valor:" Text = "{Binding Valor}" Keyboard = "Numeric" HorizontalTextAlignment = "End" />

<ViewCell>

<Grid HorizontalOptions = "Fill" >

<Grid.ColumnDefinitions>

<ColumnDefinition Width = "50*" />

<ColumnDefinition Width = "50*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height = "Auto" />

</Grid.RowDefinitions>

<Button Text = "Álbum" HorizontalOptions = "Center" Grid.Column = "0" Grid.Row = "0" Command = "{Binding AlbumCommand}" />

<Button Text = "Gravar" HorizontalOptions = "Center" Grid.Column = "1" Grid.Row = "0" Command = "{Binding GravarCommand}" />

</Grid>

</ViewCell>

</TableSection>

</TableRoot>

</TableView>

</StackLayout>

</ScrollView>

</ContentPage.Content>

</ContentPage>

```

Precisamos criar a ViewModel para a visão. Seguindo os exemplos anteriores, na pasta `ViewModels/Pecas`, crie uma classe chamada `CRUDViewModel` e implemente nela o código a seguir. Reforçando, todo o código apresentado já é comum e conhecido de implementações anteriores. Observe atentamente a propriedade `CaminhoImagem`. Quando ela for vazia, uma imagem padrão será exibida. Lembre-se de que você precisa dessa imagem em seus projetos de plataforma específica e, se ela tiver um endereço HTTP, ela será buscada diretamente na internet, sem qualquer esforço adicional.

```

namespace Capitulo05.ViewModels.Pecas
{
    public class CRUDViewModel : BaseViewModel
    {
        private IDAL<Peca> pecasDAL;
        public Peca Peca { get; set; }

        public ICommand GravarCommand { get; set; }
        public ICommand AlbumCommand { get; set; }

        public CRUDViewModel(Peca peca)
        {
            pecasDAL = new PecaDAL(DependencyService.Get<IDBPath>().GetDbPath());
            this.Peca = peca;
            RegistrarCommands();
        }

        public string Nome
        {
            get { return this.Peca.Nome; }
            set
            {
                this.Peca.Nome = value;
                ((Command)GravarCommand).ChangeCanExecute();
                OnPropertyChanged();
            }
        }

        private string valor;
        public string Valor
        {
            get { return valor; }
            set
            {
                this.valor = value;
                this.Peca.Valor = string.IsNullOrEmpty(value) ? 0 : Convert.ToDouble(valor);
                ((Command)GravarCommand).ChangeCanExecute();
                OnPropertyChanged();
            }
        }

        public string CaminhoImagen
        {
            get {
                if (string.IsNullOrEmpty(this.Peca.CaminhoImagen))
                    return "consultar.png";
                else if (this.Peca.CaminhoImagen.StartsWith("http"))
                    return this.Peca.CaminhoImagen;
                else

```

```

        return DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(Peca.CaminhoImagem);
    }
    set
    {
        this.Peca.CaminhoImagem = value;
        OnPropertyChanged();
        ((Command)GravarCommand).ChangeCanExecute();
    }
}

private void RegistrarCommands()
{
    GravarCommand = new Command(async () =>
    {
        await GravarAsync();
        MessagingCenter.Send<string>("Atualização realizada com sucesso.", "InformacaoCRUD");
    }, () =>
    {
        return !string.IsNullOrEmpty(this.Peca.Nome) && this.Peca.Valor > 0;
    });

    AlbumCommand = new Command(() =>
    {
        MessagingCenter.Send<Peca>(this.Peca, "Album");
    });
}

private async Task GravarAsync()
{
    var ehNovaPeca = (this.Peca.PecaID == null ? true : false);
    await pecasDAL.UpdateAsync(Peca, Peca.PecaID, nameof(Peca.PecaID));
    AtualizarPropriedadesParaVisao(ehNovaPeca);
    return;
}

private void AtualizarPropriedadesParaVisao(bool ehNovaPeca)
{
    if (ehNovaPeca)
    {
        this.Peca = new Peca();
        this.Nome = string.Empty;
        this.Valor = string.Empty;
        this.CaminhoImagem = "consultar.png";
    }
    else
    {
        this.Peca.NotificarListView = true;
    }
}

```

}

Partiremos agora para as implementações dos code-behinds de nossas novas visões, começando com o código para a visão `CRUDView`, que está apresentado na sequência. Observe que estamos repetindo a implementação para o método `SaveFotoFromAlbum()` e `SelecionarFotoDoAlbum()`, como fizemos para as fotos do atendimento, no capítulo 8. Aqui poderíamos pensar em ter uma classe auxiliadora, um tipo de `Helper`, que contivesse esta funcionalidade, e apenas invocar este serviço da classe sempre que precisarmos. Note também o tamanho definido para a imagem que está sendo recuperada. Lembre-se de que ela trafegará pela rede.

```
namespace Capitulo05.Views.Pecas
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class CRUDView : ContentPage
    {
        private CRUDViewModel crudViewModel;

        public CRUDView()
        {
            InitializeComponent();
        }

        public CRUDView(Peca peca, string title) : this()
        {
            this.crudViewModel = new CRUDViewModel(peca);
            this.BindingContext = this.crudViewModel;
            this.Title = title;
        }

        private async Task SeleccionarFotoDoAlbumAsync(Peca peca)
        {
            await CrossMedia.Current.Initialize();

            if (!CrossMedia.Current.IsPickPhotoSupported)
            {
                await DisplayAlert("Álbum não suportado", "Não existe permissão para acessar o álbum de fotos", "OK");
                return;
            }

            var file = await CrossMedia.Current.PickPhotoAsync(new PickMediaOptions
            {
                PhotoSize = PhotoSize.Small
            });

            if (file == null)
                return;

            var imagePath = SaveFotoFromAlbum(peca.CaminhoImagem, file);
        }
    }
}
```

```

        crudViewModel.CaminhoImagem = imagePath;
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();
        MessagingCenter.Subscribe<string>(this, "InformacaoCRUD", async (msg) => { await DisplayAlert("Informação", msg, "ok"); });
        MessagingCenter.Subscribe<Peca>(this, "Album", async (peca) => { await SelecionarFotoDoAlbumAsync(peca); });
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();
        MessagingCenter.Unsubscribe<string>(this, "InformacaoCRUD");
        MessagingCenter.Unsubscribe<Peca>(this, "Album");
    }

    public string SaveFotoFromAlbum(string caminhoImagem, Plugin.Media.Abstractions.MediaFile file)
    {
        string nomeArquivo;
        if (string.IsNullOrEmpty(caminhoImagem) || caminhoImagem.StartsWith("http"))
        {
            nomeArquivo = String.Format("{0:ddMMyyyy_HHmm}", DateTime.Now) + ".jpg";
        }
        else
        {
            File.Delete(DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(caminhoImagem));
            nomeArquivo = (caminhoImagem.LastIndexOf("/") > 0) ?
                caminhoImagem.Substring(caminhoImagem.LastIndexOf("/") + 1) : String.Format("{0:ddMMyyyy_HHmm}", DateTime.Now) + ".jpg";
        }

        var caminhoFotos = DependencyService.Get<IFotoLoadMediaPlugin>().GetDevicePathToPhoto();
        if (!Directory.Exists(caminhoFotos))
            Directory.CreateDirectory(caminhoFotos);

        string caminhoCompleto = Path.Combine(caminhoFotos, nomeArquivo);

        using (FileStream fileStream = new FileStream(caminhoCompleto, FileMode.Create))
        {
            file.GetStream().CopyTo(fileStream);
        }
        return DependencyService.Get<IFotoLoadMediaPlugin>().SetPathToPhoto(caminhoCompleto);
    }
}
}

```

Agora vamos implementar o code-behind para a classe `ListagemView`, tal qual é apresentado na sequência. Novamente, não se preocupe, é tudo que já fizemos antes no livro.

```
namespace Capitulo05.Views.Pecas
```

```

{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class ListagemView : ContentPage
    {
        private ListagemViewModel viewModel = new ListagemViewModel();

        public ListagemView ()
        {
            InitializeComponent ();
            BindingContext = viewModel;
        }

        private async Task RemoverPecaAsync(Peca peca)
        {
            if (await DisplayAlert("Confirmação",
                $"Confirma remoção de {peca.Nome.ToUpper()}?", "Yes", "No"))
            {
                await this.viewModel.EliminarPecaAsync(peca);
                await DisplayAlert("Informação", "Peça removida com sucesso", "Ok");
            }
            return;
        }

        protected override void OnAppearing()
        {
            base.OnAppearing();

            Device.BeginInvokeOnMainThread(async () =>
            {
                await viewModel.AtualizarPecasAsync();
            });

            if (listView.SelectedItem != null)
                listView.SelectedItem = null;

            MessagingCenter.Subscribe<Peca>(this, "Mostrar", async (peca) => { await Navigation.PushAsync(new
CRUDView(peca, (peca.PecaID == Guid.Empty) ? "Nova Peça" : "Alterar Peça")); });
            MessagingCenter.Subscribe<Peca>(this, "Confirmação", async (peca) => await RemoverPecaAsync(peca));
        }

        protected override void OnDisappearing()
        {
            base.OnDisappearing();
            MessagingCenter.Unsubscribe<Peca>(this, "Mostrar");
            MessagingCenter.Unsubscribe<Peca>(this, "Confirmação");
        }
    }
}

```

Para finalizarmos o trabalho com as visões e permitirmos acesso a elas, precisamos ter esta nova opção

disponibilizada em nossa visão principal. Vamos apenas inserir a linha a seguir na listagem de itens da classe `MasterPageView`.

```
new Models.MenuItem { Id = 3, Title = "Peças" , TargetType = typeof (Pecas.ListagemView), IconSource= "tab_pecas.png" }
```

Sabemos que, ao executarmos nossa aplicação, precisaremos ter nosso modelo mapeado para uma tabela em nossa base de dados. E precisamos realizar algumas implementações para que isso ocorra. Precisamos adaptar nossa classe `DbContext` no projeto `SQLiteEF` para que o mapeamento seja possível. Sendo assim, insira nessa classe a instrução a seguir.

```
public DbSet<Peca> Pecas { get ; set ; }
```

Mas, como estamos usando o `Migrations`, sabemos que só isso não basta para termos a tabela disponível na base de dados. Temos um procedimento que deve ser executado para isso. Lembra? Estamos utilizando-o desde o capítulo 5, no item *Atualização da base de dados EF Core com Migrations*. Realize-o, então.

Agora, sim, podemos testar nossa aplicação. Acesse o registro de novas peças, informe os dados, selecione uma imagem do álbum e a grave. Retorne para a listagem. Pode ser que a imagem não apareça para você e, se isso acontecer, é pelo problema que comentamos nos capítulos anteriores, sobre como a imagem é armazenada em disco e como ela deve ser recuperada. Mas este problema já está resolvido, precisamos apenas implementá-lo em nossa visão. Devemos usar um recurso que utilizamos no capítulo 5, os *Custom Converters*. Lembra-se dele?

Nossa problema está no fato de que, a cada dado exibido no `ListView`, precisamos obter a imagem a partir de um caminho que está armazenado nos objetos que serão exibidos. No projeto `Xamarin Forms`, na pasta `Converters`, vamos criar uma nova classe, chamada `ImagenPecaConverter`, e vamos implementar nela o código a seguir. O método `Convert()` receberá o valor da propriedade `CaminhoImagem`, pois o configuraremos como propriedade ligada anteriormente. Observe que já estamos preparando o método para ter uma imagem na Web.

```
namespace Capitulo05.Converters
{
    public class ImagemPecaConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
        {
            var caminho = value as string;
            if (string.IsNullOrEmpty(caminho) || caminho.StartsWith("http"))
                return caminho;
            else
                return DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(caminho);
        }

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

Pelo que já utilizamos, sabemos que apenas implementar a classe anterior não é suficiente. Precisamos utilizá-la em nossa visão. O primeiro passo é registrá-la no início de nosso arquivo, antes da propriedade `Title`, tal qual mostra o código a seguir.

```
xmlns:conv="clr-namespace:Capitulo05.Converters;assembly=Capitulo05"
```

Após isso, precisamos registrar nosso `converter` como recurso para a visão e faremos isso inserindo as instruções da sequência, logo após a tag de fechamento da nossa `toolbar`.

```
<ContentPage.Resources>

<ResourceDictionary>

<conv:ImagenPecaConverter x:Key = "imagemConverter" />

</ResourceDictionary>

</ContentPage.Resources>
```

Finalizando, vamos mudar nossa tag `<Image>`, para fazer uso do `converter`. Veja a nova tag no código a seguir. Teste novamente sua aplicação e verifique que a imagem que inseriu para a peça agora aparece.

```
<Image Source= "{Binding CaminhoImagem, Converter={StaticResource imagemConverter}}" Grid.Column= "0" Grid.Row= "0" />
```

Tal qual implementamos a solução para remoção de imagens não utilizadas no capítulo anterior, faremos aqui. Na classe `CRUDView` de Peças, insira o método a seguir. Depois, insira a instrução, que está após o método, no construtor da classe.

```
public void OnPoppedCRUDFoto ( object sender, NavigationEventArgs e )

{
    if (e.Page.GetType() == typeof (CRUDView))
    {
        if (crudViewModel.Peca.CaminhoImagem != null && crudViewModel.Peca.PecaID == Guid.Empty)
            File.Delete(DependencyService.Get<IFotoLoadMediaPlugin>()
                .GetPathToPhoto(crudViewModel.Peca.CaminhoImagem));

    }
    App.navigationPage.Popped -= OnPoppedCRUDFoto;
}

// Construtor
App.navigationPage.Popped += OnPoppedCRUDFoto;
```

## 11.2 Envio de um objeto com uma imagem para o serviço REST

Muito bem, com a aplicação atualizada no Heroku, vamos criar a classe consumidora do serviço. Para isso, no projeto `Xamarin Forms`, na pasta `services`, crie uma classe chamada `PecaService` e implemente o código apresentado na sequência. Parece ser muito código, mas é simples. Como será enviada uma imagem para o REST, a configuração do conteúdo que vai para ele é mais detalhada, pois utilizaremos o `Content-type multipart/form-data`. A classe começa com o método privado `RegistrarContentsParaMultiPartForm()`, que receberá o objeto a ser submetido e o formulário, com esse objeto, que será enviado para o serviço. Verifique a lógica, logo no início do método, para o envio do arquivo.

```
namespace Capitulo05.Services
{
```

```

public class PecaService
{
    private void RegistrarContentsParaMultiPartForm(Peca peca, MultipartFormDataContent form)
    {
        if (!(string.IsNullOrEmpty(peca.CaminhoImagem) || peca.CaminhoImagem.StartsWith("http")))
        {
            peca.CaminhoImagem = DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(peca.CaminhoImagem);
            var fileStream = new FileStream(peca.CaminhoImagem, FileMode.Open);
            var streamContent = new StreamContent(fileStream);
            var imagemContent = new ByteArrayContent(streamContent.ReadAsByteArrayAsync().Result);
            imagemContent.Headers.ContentType = MediaTypeHeaderValue.Parse("multipart/form-data");
            form.Add(imagemContent, "arquivo", Path.GetFileName(peca.CaminhoImagem));

            var caminhoImagemContent = new StringContent(peca.CaminhoImagem);
            caminhoImagemContent.Headers.ContentType = MediaTypeHeaderValue.Parse("multipart/form-data");
            form.Add(caminhoImagemContent, "caminhoImagem");
        }

        var pecaIDContent = new StringContent(peca.PecaID.ToString());
        pecaIDContent.Headers.ContentType = MediaTypeHeaderValue.Parse("multipart/form-data");
        form.Add(pecaIDContent, "pecaID");

        var nomeContent = new StringContent(peca.Nome);
        nomeContent.Headers.ContentType = MediaTypeHeaderValue.Parse("multipart/form-data");
        form.Add(nomeContent, "nome");

        var valorContent = new StringContent(string.Format("{0:N}", peca.Valor).Replace(',', '.'));
        valorContent.Headers.ContentType = MediaTypeHeaderValue.Parse("multipart/form-data");
        form.Add(valorContent, "valor");
    }
}

```

Para terminarmos a classe, vamos à implementação do método que realizará a invocação do serviço REST. Veja este método na sequência, que deve ser implementado logo após o término do método da listagem anterior. Na declaração do formulário, que configuramos no método anteriormente codificado, invocamos este método enviando o objeto recebido e o formulário instanciado. Depois, temos o bloco `using()`, tal qual tínhamos para o `login`. Observe a similaridade do código. Podemos pensar em uma generalização, tal qual temos implementada para nossos DALs. Poderíamos ter uma interface genérica, uma classe base e as especializações que forem necessárias. O que acha? Fica a sugestão para você fazer, ok? O objetivo aqui no livro, neste capítulo, será o de realizar um CRUD com serviços REST.

```

public async Task<string> PostComArquivo (Peca peca)
{
    MultipartFormDataContent form = new MultipartFormDataContent();
    RegistrarContentsParaMultiPartForm(peca, form);

    using ( var client = ServicesPrepare.GetHttpClient())
    {
        try

```

```

    {

        HttpResponseMessage response = await client.PostAsync( "pecas/savecomimagem" , form);

        if (response.IsSuccessStatusCode)

        {

            return await response.Content.ReadAsStringAsync();

        }

        return await Task.FromResult( "false" );

    }

    catch (Exception e)

    {

        return await Task.FromResult(e.Message);

    }

}

}

```

Antes de consumirmos este novo método, vamos recapitular o processo que temos para o login. Ele pode demorar um pouco, certo? Lembre-se de que estamos usando uma ferramenta gratuita. Quando implementamos a autenticação do usuário, definimos que um indicador de atividade seria exibido ao usuário, indicando um processamento, para não dar a impressão de um congelamento (travamento) da aplicação. Vamos então ao nosso XAML para inserir o código apresentado na sequência. É o crudview de peças. Este código deve ser inserido antes da viewCell do Grid de botões. Observe a ligação com uma propriedade chamada Atualizando . Precisamos criá-la em NOSSA CRUDVIEWMODEL , para peças. O código está após o XAML.

```

<ViewCell>

<ActivityIndicator IsRunning = "{Binding Atualizando}" Color = "Black" />

</ViewCell>

private bool atualizando;

public bool Atualizando

{

    get { return this .atualizando; }

    set

    {

        this .atualizando = value ;

        OnPropertyChanged();

    }

}

```

Vamos identificar em que parte do código deveremos invocar nosso método consumidor do serviço REST e podemos assumir que a responsabilidade deve ser do método GravarAsync() de nossa classe CRUDVIEWMODEL , de Peca , pois assim que o usuário gravar uma nova peça, ou atualizar uma já existente, devemos tentar o sincronismo e atualizar a base servidora. Veja o novo código para o método GravarAsync() na sequência. O código cresceu um pouco, mas você verá que é simples.

```
// Antes do construtor
```

```

private PecaService service;

// No construtor

service = new PecaService();

private async Task<string> GravarAsync ()
{
    var ehNovaPeca = ( this .Peca.PecaID == null ? true : false );

    Atualizando = true ;

    await pecasDAL.UpdateAsync(Peca, Peca.PecaID, nameof(Peca.PecaID));

    var result = await service.PostComArquivo( this .Peca);

    Atualizando = false ;

    if (result.ToLower().Equals( "false" ) || !result.ToLower().StartsWith( "http" ))

        return await Task.FromResult(result);

    else

    {
        if (File.Exists(DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(Peca.CaminhoImagem)))
            File.Delete(DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(Peca.CaminhoImagem));

        this .Peca.CaminhoImagem = result;

        await pecasDAL.UpdateAsync(Peca, Peca.PecaID, nameof(Peca.PecaID), true );
    }

    AtualizarPropriedadesParaVisao(ehNovaPeca);

    return await Task.FromResult( "true" );
}

```

Inserimos mudanças de valores para a propriedade `Atualizando`, para que nosso `ActivityIndicator` possa ser exibido e ocultado de acordo com o processo. Invocamos o método `PostComArquivo()`, enviando nossa peça já persistida localmente. Depois, trabalhamos com o resultado retornado pelo método. Caso tenha havido sucesso, removemos o arquivo de imagem do dispositivo, atualizamos a propriedade `CaminhoImagem` para o caminho disponibilizado pelo serviço e invocamos o `UpdateAsync()` novamente, enviando `true` para ser atribuído à propriedade `Sincronizado`. O final do método continua o mesmo que tínhamos implementado antes. O que acha de testar sua aplicação após a implementação? Se você sentir uma demora na exibição da imagem quando retornar à listagem, lembre-se de que agora ela vem da internet, depende da rede e, novamente, estamos utilizando um serviço gratuito.

Vamos alterar nosso `GravarCommand` para as instruções a seguir, pois precisamos trabalhar o retorno do método `Gravar` e informar corretamente o resultado ao usuário.

```

var resultadoGravacao = await GravarAsync ();

if (resultadoGravacao.ToLower().Equals( "true" ))

    MessagingCenter.Send< string >( "Atualização realizada com sucesso." , "InformacaoCRUD" );

else

    MessagingCenter.Send< string >(resultadoGravacao.ToLower().Equals( "false" ) ? "Erro na comunicação" :

```

```
resultadoGravacao, "InformacaoCRUD" );
```

Estamos utilizando o Heroku como ferramenta didática para trabalharmos os exemplos deste capítulo e é preciso informar que ele possui uma limitação em relação à manutenção dos arquivos enviados via upload, em nosso caso, as imagens das peças. Todo arquivo gravado no sistema de arquivos do Heroku é removido depois de um certo tempo, é uma característica dele e pode ser verificada em <https://help.heroku.com/K1PPS2WM/why-are-my-file-uploads-missing-deleted/>. O próprio link traz algumas sugestões de solução para esta situação; as imagens podem ser gravadas na base de dados ou ainda em um servidor de arquivos. Mais à frente, no capítulo, verificaremos como persistir os dados na base de dados. Não adotei a solução de persistir a imagem em um servidor de arquivos pelo fato de que isso também aumentaria a burocracia e complexidade de nosso exemplo deste capítulo. O importante aqui é o aprendizado para enviar um arquivo para um serviço e armazená-lo em disco, para que possamos recuperá-lo de maneira simples, por sua URL.

### 11.3 Recebendo as peças registradas no servidor

Precisamos pensar agora em serviços que complementem nosso CRUD com peças, pois temos apenas o `Update`, que cobre a inserção e alteração de dados. Falta-nos a recuperação de todos os elementos, de uma única peça e a remoção de uma peça. Temos tudo isso implementado em nossos DAL, mas localmente, no dispositivo. Vamos começar essas novas implementações.

Precisamos implementar nosso consumidor do serviço na classe `PecaService`. Este método será simples, curto. Veja o código na listagem a seguir. Note que nossa invocação agora é para o método `GetAsync()`, pois nosso serviço agora é `GET`, não `POST`. O retorno estará em uma coleção de objetos JSON, o que nos leva à necessidade de converter este retorno em uma lista de objetos para a classe `Peca`, e isso é feito pela invocação do método `DeserializeObject()`, que recebe a string retornada pelo serviço. Em caso de exceções, deixaremos para que o consumidor deste método realize o tratamento.

```
public async Task<List<Peca>> GetAllAsync()
{
    using ( var client = ServicesPrepare.GetHttpClient() )
    {
        try
        {
            HttpResponseMessage response = await client.GetAsync( "pecas/findAll" );
            if ( response.IsSuccessStatusCode )
            {
                return JsonConvert.DeserializeObject<List<Peca>>( await response.Content.ReadAsStringAsync() );
            }
            throw new Exception( "Erro ao recuperar peças do servidor " + response );
        }
        catch ( Exception e )
        {
            throw new Exception( e.Message );
        }
    }
}
```

Muito bem, mas onde invocaremos este método? E como atualizaremos nossa base de dados local com o resultado fornecido pelo serviço? Vamos às respostas e às resoluções. Nossa primeiro passo será implementar um método responsável pela atualização da base local, e temos uma classe perfeita para isso, nossa `PecaDAL`. Veja o novo método na sequência. Observe que invocamos o método `RemoveRange()` do conjunto de `Pecas`, enviando a ele o retorno do método `GetAllAsync()`, que receberá um valor `true`. Logo veremos a mudança do método `GetAllAsync()`, que recebe o `bool`. Por enquanto, pode acusar erro de compilação para você. Após a remoção dos objetos, nós gravamos o procedimento realizado, depois, percorrendo a coleção de peças recebidas, adicionamos uma a uma em nosso contexto e, ao final, gravamos toda a operação de inserção. Podemos pensar em inserir este método em nossa interface `IDAL`, se ele for comum a todos os modelos de nossa aplicação, mas deixo isso como uma dica para você.

```
public async Task SincronizaBaseLocal (List<Peca> baseRest)

{
    using ( var context = DatabaseContext.GetContext(dbPath))
    {
        context.Pecas.RemoveRange( await GetAllAsync ( true ) );
        await context.SaveChangesAsync();

        foreach ( var p in baseRest)
        {
            var pl = await GetByIdAsync (p.PecaID);
            if (pl == null )
                await context.AddAsync(p);
        }
        await context.SaveChanges();
    }
}
```

A estratégia que adotei no método anterior foi simplista. Optei por remover os objetos já sincronizados e adicionar os recebidos do servidor. Talvez, em uma aplicação diferente, fosse preciso verificar a questão de associações das classes para adotarmos a estratégia de remoção de objetos, como vimos no capítulo 9. Outra estratégia poderia ser, em vez de remover todos, verificar na base local quais objetos não existem em relação à coleção recebida e inserir apenas os novos. Mas seria necessário também verificar quais existem na local e que não existem na recebida, além, é claro, de conferir se os objetos comuns (mesmo ID) estão diferentes localmente, em relação ao recebido do servidor. O sincronismo pode se tornar algo trabalhoso. Lembre-se das opções que elenquei no início do capítulo anterior em relação às características para uma aplicação com base centralizada.

Voltemos à implementação. Comentamos anteriormente, na explicação do novo método, que o `GetAllAsync()`, recebendo `true`, retornará apenas os objetos locais que já foram sincronizados. Com isso, precisamos alterar nosso método. Poderíamos adaptar nosso método na interface e na `DALBase` para esta situação, mas preferi especializar o método na `PecaDAL`, tal qual pode ser visto na listagem a seguir. A preferência se deu em minimizar, neste momento do livro, a reescrita de métodos e, também, mostrar a criação de um método, de mesmo nome, em uma classe especializada, que não esteja sobrescrito e consuma um método da superclasse.

Veja que o método não é sobreescrito, é um novo método, que, ao final, invoca o método de mesmo nome da superclasse.

```
public async Task<List<Peca>> GetAllAsync( bool apenasSincronizado = false )

{
    using ( var context = DatabaseContext.GetContext(dbPath))
```

```

{
    if (apenasSincronizado)

        return await context.Set<Peca>().Where(p => p.Sincronizado).ToListAsync();

}

return await base.GetAllAsync(p => p.Nome, OrderByType.Ascendente);
}

```

Com a implementação anterior, precisamos forçar nosso `ListagemViewModel` a utilizar este método e não o da classe `DALBase`. Desta maneira, adapte a instrução do método `AtualizarPecasAsync()` para o `cast` da sequência.

```
var pecas = await ((PecaDAL)pecasDAL).GetAllAsync();
```

Enfim, vamos à implementação do consumo destes serviços pela interação do usuário. Adotaremos um recurso novo no livro, conhecido como `Pull to refresh` do `ListView`. Ele possibilita que a aplicação execute um `command` quando o usuário pressionar a listagem e a arrastar para baixo. Precisaremos ajustar nosso código XAML da visão `ListagemView` de peças. Veja o que precisamos adicionar na tag `<ListView>` para esta nova funcionalidade. Observe que temos a propriedade `IsPullToRefreshEnabled` habilitando a funcionalidade desejada, a `RefreshCommand` definindo o `Command` que será executado quando ocorrer o `Pull to refresh` e uma propriedade ligada para `IsRefreshing`, que gerenciará a exibição de um `ActivityIndicator` enquanto o processo ocorre.

```
IsPullToRefreshEnabled="True" RefreshCommand="{Binding RefreshCommand}" IsRefreshing="{Binding Sincronizando}"
```

Vamos começar nossa implementação, para as propriedades adicionadas pelo trecho anterior na classe `ListagemViewModel` de peças. Nosso primeiro código deve ser referente à propriedade ligada `Sincronizando`. Veja-o na sequência.

```

private bool sincronizando;

public bool Sincronizando
{
    get { return this.sincronizando; }

    set
    {
        this.sincronizando = value;
        OnPropertyChanged();
    }
}

```

Na classe `ListagemViewModel` de peças, também utilizaremos a `PecaService`. Desta maneira, realize as seguintes implementações nela.

```

// Antes do construtor

private PecaService service;

// No construtor

service = new PecaService();

```

Precisamos agora implementar o `command RefreshCommand`, que será executado quando o usuário realizar o `Pull to refresh`. Veja o código na sequência, que deve ser implementado no método `RegistrarCommands()`. Todo o processo

está em um `try...catch`, para que possamos retornar ao usuário a exceção disparada, caso ocorra. O método começa com a atribuição de `true` à propriedade `Sincronizando`, e depois o serviço é consumido. Veja o *cast* para o DAL para a execução do método `SincronizaBaseLocal()`. Com a execução deste método, chamamos o já conhecido `AtualizarPecasAsync()`, que atualizará a coleção de objetos que são exibidos no `ListView`.

```
RefreshCommand = new Command( async () =>
{
    try
    {
        Sincronizando = true ;

        var pecasREST = await service.GetAllAsync();

        await (pecasDAL as PecaDAL).SincronizaBaseLocal(pecasREST);

        await AtualizarPecasAsync();

        MessagingCenter.Send< string >( "Sincronização realizada com sucesso." , "Informação" );
    }
    catch (Exception e)
    {
        MessagingCenter.Send< string >(e.Message, "Informação" );
    } finally
    {
        Sincronizando = false ;
    }
});
```

Já podemos testar nossa aplicação no Android, e é para funcionar sem problemas. Entretanto, no iOS, existe um bug quando se utiliza o `Pull to refresh` em conjunto com uma visão que tenha a característica de títulos longos, vinda com o iOS 11. Precisamos desabilitar esta característica para nossa visão de listagem de peças. Faremos isso na classe `MainPageView`, onde não atribuiremos esta característica para a visão que estamos trabalhando. Veja o código na sequência, onde inserimos um `if()` para esta situação, no método `ListView_ItemSelected()`, antes da atribuição `App.navigationPage`. Lembre-se de tirar também a instrução que já existe para o `SetPrefersLargeTitles()`. Depois disso, podemos testar nossa aplicação também para o iOS. Realizei pesquisas sobre este problema, que foi confirmado em <https://github.com/xamarin/Xamarin.Forms/issues/1905/>. O link até comenta que o problema foi corrigido, mas estou testando com o iOS 11.4, com a versão atualizada do Xamarin e o problema persiste.

```
if (page.GetType() != typeof (Views.Pecas.ListagemView))

navigationPage.On<Xamarin.Forms.PlatformConfiguration.iOS>().SetPrefersLargeTitles( true );
```

## 11.4 Remoção de uma peça no servidor

Precisamos codificar o método responsável pelo consumo do serviço para remoção de uma peça, em nossa classe `PecaService`, tal qual é apresentado pelo código a seguir.

```
public async Task Remove (Guid PecaID)

{
    using ( var client = ServicesPrepare.GetHttpClient())
```

```

{
    try
    {
        var content = new StringContent(PecaID.ToString(), Encoding.UTF8, "application/json");

        HttpResponseMessage response = await client.PostAsync("pecas/removebyid", content);

        if (response.IsSuccessStatusCode)
        {
            return;
        }
    }
    catch (Exception e)
    {
        throw new Exception(e.Message);
    }
}
}

```

Agora, vamos para a implementação que será responsável por invocar este método, que deverá ocorrer antes de sua remoção na base local. Na classe `ListagemViewModel`, de peças, temos o método `EliminarPecaAsync()`, que remove a peça da base local. Vamos adaptá-lo para a seguinte implementação.

```

public async Task EliminarPecaAsync (Peca peca)

{
    Sincronizando = true;

    await service.Remove(peca.PecaID);

    await pecasDAL.DeleteAsync(peca);

    Pecas.Remove(peca);

    Sincronizando = false;
}

```

Como estamos trabalhando a possibilidade de exceção no método `Remove()`, precisamos em algum momento envolver o código com este tratamento. Para esta situação, o melhor local é no método que será executado quando for confirmado pelo usuário que ele quer remover uma peça, e temos isso em nossa classe `ListagemView`, no método `RemoverPecaAsync()`. Adapte-o para o código a seguir. O que estamos mudando é apenas o envolvimento da operação pelo bloco `try...catch`.

```

private async Task RemoverPecaAsync (Peca peca)

{
    if ( await DisplayAlert ("Confirmação",
        $"Confirma remoção de {peca.Nome.ToUpper()}?" , "Yes" , "No" ) )

    {
        try
        {

```

```

        await this.viewModel.EliminarPecaAsync(peca);

        await DisplayAlert ("Informação" , "Peça removida com sucesso" , "Ok" );

    }

    catch (Exception e)
    {

        await DisplayAlert ("Erro" , e.Message , "Ok" );
    }
}

return ;
}

```

## 11.5 Recuperação de uma peça específica no servidor

Vamos implementar o método que consumirá o serviço de recuperação de uma peça específica, em nossa classe `PecaService`, mesmo que não o utilizemos em nossa aplicação, pois, como dito, o objetivo é apresentar e disponibilizar a você o serviço para servir como base em um uso futuro. Veja o método na sequência.

```

public async Task<Peca> GetByIdAsync (Guid PecaID)

{
    using ( var client = ServicesPrepare.GetHttpClient() )
    {
        try
        {

            HttpResponseMessage response = await client.GetAsync( "pecas/findbyid?id=" + PecaID );

            var content = new StringContent(PecaID.ToString(), Encoding.UTF8, "application/json" );

            if (response.IsSuccessStatusCode)
            {

                return JsonConvert.DeserializeObject<Peca>( await response.Content.ReadAsStringAsync() );
            }

            throw new Exception ( "Erro ao recuperar peça do servidor " + response );
        }

        catch (Exception e)
        {
            throw new Exception ( e.Message );
        }
    }
}

```

Observe que junto com a URL enviamos uma `QueryString`, na qual `id` receberá o ID da peça. Podemos dizer que uma `QueryString` representa um conjunto de pares/valores anexados à URL. Como visto no código, seu uso é simples, bastando adicionar após a URL o primeiro `conjunto` usando a seguinte sintaxe: `?Chave=Valor`. Quando precisarmos mandar mais de um conjunto, eles devem ser concatenados com o caractere `&`. Esta característica se deve à

forma como o REST foi implementado no servidor. Você verá mais à frente, no método `GetImagemByIdAsync()` uma maneira diferente de enviar o ID.

## 11.6 Envio de uma peça sem imagem para o servidor

Em nossa classe `Pecaservice`, vamos implementar o método que consumirá o serviço que implementamos, pelo documento complementar, para a funcionalidade de enviar uma peça para o servidor, sem imagem. Veja este código na sequência. Observe que convertemos o objeto recebido para um objeto anônimo, que será convertido para JSON, com as propriedades tendo os nomes das propriedades da classe `Peca.java`, que chegará ao nosso serviço. Uma vez criado e convertido o objeto, instanciamos um `StringContent` com a informação que deverá ser enviada para o serviço REST.

```
public async Task<string> PostSemArquivoAsync (Peca peca)

{

    var json = JsonConvert.SerializeObject( new
    {
        pecaID = peca.PecaID,
        nome = peca.Nome,
        valor = peca.Valor
    });

    var content = new StringContent(json, Encoding.UTF8, "application/json");



    using ( var client = ServicesPrepare.GetHttpClient())
    {

        try
        {

            HttpResponseMessage response = await client.PostAsync( "pecas/savesemimagem" , content);

            if (response.IsSuccessStatusCode)
            {

                return await response.Content.ReadAsStringAsync();
            }

            return await Task.FromResult( "false" );
        }

        catch (Exception e)
        {

            return await Task.FromResult(e.Message);
        }
    }
}
```

## 11.7 Um serviço REST que retorna uma imagem

Com o que implementamos até este ponto do capítulo, já teríamos todo o objetivo proposto cumprido, com todas as operações referentes ao CRUD para peças implementadas. Entretanto, quando terminamos o serviço que envia a imagem, comentei sobre a limitação do Heroku na manutenção de arquivos que sejam recebidos por uploads. Vamos criar uma versão para nosso método que recebe a peça com sua imagem, para que ela possa ser persistida na base de dados. Caso você queria ver detalhes da implementação do serviço, ele está implementado no documento complementar. Aqui, a única preocupação é em consumi-lo.

Precisamos consumir estes novos serviços. Para a gravação da imagem em bytes, podemos apenas alterar o endereço do serviço, para o novo que criamos no documento complementar. Veja como ele fica na sequência.

```
 HttpResponseMessage response = await client.PostAsync( "pecas/saveimagemembytes" , form);
```

Para finalizar nossa implementação, consumindo o serviço que recupera a imagem em bytes, para exibi-la em nossa listagem e em nossa visão de CRUD quando estivermos alterando a peça, teremos que implementar algumas mudanças em nossa aplicação Xamarin. A primeira é a adição de uma nova propriedade transiente em nossa classe `PecaIDProperty`, tal qual segue.

```
[NotMapped]  
  
public byte[] ImagemEmBytes { get ; set ; }
```

Precisamos tratar esta nova implementação em nossa propriedade `CaminhoImagem` da classe `CRUDViewModel` de peças. Desta maneira, antes do `if()` que já existe na propriedade, insira o proposto a seguir.

```
if ( this .Peca.ImagemEmBytes != null )  
  
    return string .Empty;
```

Na sequência, a outra mudança será em nossa classe `ImagenPecaConverter`, que define a fonte para a imagem de cada peça a ser exibida em nosso `Listview`. Veja nosso novo método `Convert()` na sequência. Mantivemos a verificação do caminho, para o caso de termos peças não sincronizadas com o servidor.

```
public object Convert ( object value , Type targetType, object parameter, CultureInfo culture)  
  
{  
  
    string caminho = ((Label) parameter).Text;  
  
    byte[] bytes = ( byte [] ) value ;  
  
    if ( ! string .IsNullOrEmpty(caminho))  
  
        return DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(caminho);  
  
    else  
  
        return (bytes == null || bytes.Length == 0) ? "consultar.png" : ImageSource.FromStream ( () => new MemoryStream(bytes  
    )
```

Em nossa classe `PecaService`, precisamos agora implementar o método que consumirá o serviço que implementamos para esta finalidade no documento complementar. Veja o código para ele na sequência. Veja o tipo de dado que o método retorna, observe como compomos a URL no `GetAsync()` e como lemos o retorno, utilizando `ReadAsByteArrayAsync()`.

```
public async Task<Byte[]> GetImagenByIdAsync(Guid PecaID)  
  
{  
  
    using ( var client = ServicesPrepare.GetHttpClient())  
    {
```

```

    try
    {

        HttpResponseMessage response = client.GetAsync( "pecas/getimagem/" + PecaID.ToString()).Result;

        if (response.IsSuccessStatusCode)
        {

            return await response.Content.ReadAsByteArrayAsync();
        }

        return null ;
    }

    catch (Exception e)
    {
        throw new Exception (e.Message);
    }
}
}

```

Uma vez que o processo de recuperação, imagem a imagem, do servidor pode ser algo custoso e sofrer uma certa demora, vamos propor uma implementação que não deixe o usuário com a impressão de aplicação travada. Vamos exibir para ele a listagem das peças, inicialmente sem imagem, e isso é tranquilo, pois as peças estão na base local do dispositivo. Depois, criaremos um processo de atualização dos itens conforme as imagens forem sendo recebidas do servidor. Isso é legal e dá um bom visual para o usuário. Vamos realizar uma alteração no XAML de listagem de peças. Veja o código a seguir. Coloque-o no lugar do <Image>. Veja que temos um *Activity Indicator*, que ficará em execução no lugar da imagem, até que ela seja recuperada. O efeito será bem legal. Note também que temos um <Label> oculto, que nos enviará ao *converter*, como parâmetro, o caminho para a imagem e o *Source* natural para a imagem é a propriedade *ImagenEmBytes*.

```

<StackLayout Padding = "5" HeightRequest = "60" VerticalOptions = "Center" HorizontalOptions = "Center" >

<Label IsVisible = "False" x:Name = "CaminhoImagen" Text = "{Binding CaminhoImagen}" />

<StackLayout Grid.Column = "0" Grid.Row = "0" IsVisible = "{Binding ImagemEmBytes, Converter={StaticResource
imagemNaoCarregadaConverter}, ConverterParameter={x:Reference Name=CaminhoImagen}}" VerticalOptions = "CenterAndExpand"
HorizontalOptions = "CenterAndExpand" >

<ActivityIndicator IsRunning = "True" />

</StackLayout>

<StackLayout Grid.Column = "0" Grid.Row = "0" IsVisible = "{Binding ImagemEmBytes, Converter={StaticResource
imagemCarregadaConverter}, ConverterParameter={x:Reference Name=CaminhoImagen}}" >

<Image Source = "{Binding ImagemEmBytes, Converter={StaticResource imagemConverter}, ConverterParameter={x:Reference
Name=CaminhoImagen}}" Grid.Column = "0" Grid.Row = "0" HeightRequest = "50" />

</StackLayout>

```

```
</StackLayout>
```

Os dois `<StackLayouts>` internos têm a propriedade `IsVisible` ligada a `ImagenEmBytes`, que por sua vez está ligada a conversores, pois apenas um destes layouts poderá estar visível por vez para cada item. Vamos implementar os `converters`. Crie, na pasta `Converters` do Xamarin Project, as duas classes apresentadas na sequência. Veja que a única diferença entre elas é na negação do `return` de `Convert()`.

```
namespace Capitulo05.Converters
{
    public class ImagemCarregadaConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
        {
            return !(value == null) || !string.IsNullOrEmpty(((Label)parameter).Text) ? true : false;
        }

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

namespace Capitulo05.Converters
{
    public class ImagemNaoCarregadaConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
        {
            return (value == null) && string.IsNullOrEmpty(((Label)parameter).Text) ? true : false;
        }

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

Com os `converters` implementados, precisamos registrá-los em nosso XAML, junto com o conversor para a imagem que já temos no código. Veja a implementação a ser acrescentada na sequência, logo no início do arquivo.

```
<conv:ImagenCarregadaConverter x:Key = "imagemCarregadaConverter" />

<conv:ImagenNaoCarregadaConverter x:Key = "imagemNaoCarregadaConverter" />
```

Agora, vamos para a implementação necessária em nosso ViewModel, para que possamos recuperar as imagens para as peças exibidas no `ListView`. Veja os métodos a seguir. Nossa primeira instrução declara uma propriedade chamada `AtualizandoImagens`, que será um flag que nos indicará quando o processo de atualização estiver ocorrendo. Na

sequência, há um método que invoca nosso serviço consumidor do REST para um determinado ID. O método `AtualizarImagem()` é responsável por invocar o método `GetBytesFromImageAsync()` e associar à propriedade `ImagenEmBytes` a imagem recuperada por este método. Por fim, o método `AtualizarImagens()` percorre a coleção de peças e invoca o método `AtualizarImagem()` para recuperar as imagens de cada peça. Em vez do `for()` tradicional, podíamos ter usado o `foreach()` também.

```
public bool AtualizandoImagens = false ;  
  
public async Task<Byte[]> GetBytesFromImageAsync(Guid pecaID)  
{  
    return await service.GetImagenByIdAsync(pecaID);  
}  
  
private void AtualizarImagen (Peca peca)  
{  
    Task.Run( async () => {  
        if (peca.ImagenEmBytes == null )  
        {  
            var i = Pecas.IndexOf(peca);  
  
            peca.ImagenEmBytes = await GetBytesFromImageAsync (peca.PecaID);  
  
            if (peca.ImagenEmBytes == null && string .IsNullOrEmpty(peca.CaminhoImagen))  
                peca.CaminhoImagen = "consultar.png" ;  
  
            this .Pecas.RemoveAt(i);  
            this .Pecas.Insert(i, peca);  
        }  
    });  
}  
  
public void AtualizarImagens ()  
{  
    AtualizandoImagens = true ;  
    for ( int i = 0; i < Pecas.Count; i++)  
    {  
        AtualizarImagen(Pecas[i]);  
    }  
    AtualizandoImagens = false ;  
}
```

Falta invocar este método e o faremos em nosso método `OnAppearing()` da `ListagemView` de peças. Logo após a invocação ao método `AtualizarPecasAsync()`, insira as instruções a seguir.

```
if (!viewModel.AtualizandoImagens)  
    viewModel.AtualizarImagens();
```

## 11.8 Inserção e alteração para a nova estratégia

Precisamos agora adaptar nossa aplicação Xamarin para que, na inserção e alteração, possamos invocar o novo serviço, que armazena a imagem na base de dados Web e, consequentemente, trabalhar a exibição de nossa imagem também com a nova técnica. Nossa primeira alteração será no XAML da visão de CRUD, no qual substituiremos a tag <Image> pelo código a seguir. É bem semelhante ao que fizemos para a listagem.

```
<StackLayout Padding = "5" HeightRequest = "150" VerticalOptions = "Center" HorizontalOptions = "Center" >

<StackLayout Margin = "10" IsVisible = "{Binding CarregandoImagen}" VerticalOptions = "CenterAndExpand" HorizontalOptions = "CenterAndExpand" >

<ActivityIndicator IsRunning = "True" VerticalOptions = "CenterAndExpand" HorizontalOptions = "CenterAndExpand" />

</StackLayout>

<Label IsVisible = "False" x:Name = "CaminhoImagen" Text = "{Binding CaminhoImagen}" />

<StackLayout Margin = "10" HeightRequest = "150" IsVisible = "{Binding CarregandoImagen, Converter={StaticResource booleanNegadoConverter}}" VerticalOptions = "CenterAndExpand" HorizontalOptions = "CenterAndExpand" >

<Image Source = "{Binding ImagemEmBytes, Converter={StaticResource imagemConverter}, ConverterParameter={x:Reference Name=CaminhoImagen}}" HeightRequest = "150" VerticalOptions = "CenterAndExpand" HorizontalOptions = "CenterAndExpand" />

</StackLayout>

</StackLayout>
```

Observe que temos uma propriedade ligada chamada `carregandoImagen`, que precisamos implementar. Temos também o uso de dois *converters*, O `imagemConverter`, que já conhecemos, e O `booleanNegadoConverter`. Vamos ver estes códigos na sequência. Para o *converter* precisaremos criar a classe na pasta `Converters`.

```
// Propriedade em CRUDViewModel
private bool carregandoImagen;
public bool CarregandoImagen
{
    get { return carregandoImagen; }
    set {
        carregandoImagen = value;
        OnPropertyChanged();
    }
}

// Classe para novo conversor
namespace Capitulo05.Converters
{
    public class BooleanNegadoConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
```

```

    {
        var valor = (bool)value;
        return !valor;
    }

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

```

Em nosso XAML do CRUD, precisamos registrar nossos conversores. Veja os códigos na sequência.

```

// Final da tag ContentPage no início do arquivo
xmlns:conv="clr-namespace:Capitulo05.Converters;assembly=Capitulo05"

// Antes de <ContentPage.Content>

<ContentPage.Resources>

<ResourceDictionary>

<ResourceDictionary>

<conv:ImagenPecaConverter x:Key = "imagemConverter" />

<conv:BooleanNegadoConverter x:Key = "booleanNegadoConverter" />

</ResourceDictionary>

</ResourceDictionary>

</ContentPage.Resources>

```

Estamos quase terminando. Na classe `CRUDView`, no construtor, insira a instrução a seguir. Ela garante que o `ActivityIndicator` apareça até que a imagem possa ser recuperada pelo serviço.

```
crudViewModel.CarregandoImagem = true;
```

Agora, no `OnAppearing()` da `CRUDview` de peças, logo após a chamada ao método da classe `base`, insira as instruções a seguir.

```

// OnAppearing
Device.BeginInvokeOnMainThread(() =>
{
    crudViewModel.AtualizarImagenPeca();
});

```

O método que estamos invocando na listagem anterior deve ser implementado na `CRUDviewModel` de peças, e ele está na sequência, juntamente com uma propriedade de leitura. Veja que, no caso de a imagem ser recuperada, precisamos

ajustar o valor de `CaminhoImagem`, pois, se acessarmos um link antes de ele ter a imagem recuperada, ele trará nossa imagem padrão como caminho. Observe que redundamos a implementação de `GetBytesFromImageAsync()`, que fizemos no `ListagemViewModel`. O que acha de pensar em uma reutilização?

```
// Propriedade ligada à imagem em CRUDViewModel

public byte[] ImagemEmBytes { get { return Peca.ImagemEmBytes; } }

// Método em CRUDViewModel

public async Task<Byte[]> GetBytesFromImageAsync(Guid pecaID)
{
    return await service.GetImagenByIdAsync(pecaID);
}

// Método em CRUDViewModel

public void AtualizarImagenPeca ()
{
    Task.Run( async () => {
        if (Peca.ImagemEmBytes == null && Peca.PecaID != Guid.Empty)
        {
            Peca.ImagemEmBytes = await GetBytesFromImageAsync (Peca.PecaID);

            if (Peca.ImagemEmBytes != null )
            {
                Peca.CaminhoImagem = string .Empty;
                OnPropertyChanged(nameof(CaminhoImagem));
            }
            OnPropertyChanged(nameof(ImagenEmBytes));
        }
        CarregandoImagen = false ;
    });
}
```

Assumindo que o usuário também poderá alterar uma peça já registrada, o que inclui sua imagem, precisamos inserir uma instrução ao final do método `SelecionarFotoDoAlbumAsync()`, tal qual segue no código. Vamos testar nossa aplicação com as funcionalidades que implementamos?

```
crudViewModel.Peca.ImagemEmBytes = null ;
```

## 11.9 Peças que não forem sincronizadas automaticamente

Em nosso `ListView`, seria interessante que as peças não sincronizadas apareçam em destaque, para que o usuário verifique essa situação e atualize a base Web. Isso pode acontecer caso ocorra algum problema de comunicação com o servidor no momento de gravação do item. Para resolver esta questão, faremos uso de *converters*.

No projeto Xamarin Forms, em nossa pasta `converters`, vamos criar uma classe chamada `PecaNaoSincronizadaColor` e a

implemente tal qual o código a seguir. Note que ela implementa a interface `IValueConverter`. No método `Convert()`, verificamos o valor recebido, que será proveniente da propriedade `Sincronizado` e, de acordo com ele, uma cor é retornada.

```
namespace Capitulo05.Converters
{
    public class PecaNaoSincronizadaConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
        {
            if (!((bool)value))
                return Color.Red;
            return Color.Black;
        }

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

Vamos agora implementar o uso deste conversor em nosso XAML. Adicione o seguinte conversor à lista dos que já temos registrados em nossa `ListagemView`.

```
<conv:PecaNaoSincronizadaConverter x:Key="colorConvert"/>
```

Enfim, podemos agora utilizar nosso `converter` e o faremos nos `Labels` do `ItemTemplate`. Adapte os dois para ficarem tal qual o código a seguir apresenta. Veja que nosso `binding` está sendo realizado na propriedade `TextColor`.

```
<Label Text = "{Binding Nome}" FontSize = "18" FontAttributes = "Bold" TextColor = "{Binding Sincronizado, Converter=StaticResource colorConvert}" />
```

```
<Label Text = "{Binding ValorFormatado}" FontSize = "14" TextColor = "{Binding Sincronizado, Converter={StaticResource colorConvert}}" />
```

Vamos realizar um teste para ver esta nova mudança em ação. No método `PostComArquivo()` de nossa `PecaService`, altere o endereço do serviço, para que ocorra um erro. Depois, execute sua aplicação e insira uma nova peça e retorne para a listagem. O `converter` funcionou para itens não sincronizados?

Precisamos agora fornecer ao usuário a possibilidade de sincronizar a peça que está apenas na base local. Rapidamente, podemos lembrar de inserir um `Context Action` em nosso `ListView`, tal qual temos para a remoção de uma peça. Vamos lá. Em seu `ListagemView.xaml` adicione o seguinte código, abaixo de seu `Context Action` para remover.

```
<MenuItem Command = "{Binding Path=BindingContext.SincronizarCommand, Source={x:Reference listView}}" CommandParameter = "{Binding .}" Text = "Sincronizar" />
```

Se executarmos nossa aplicação agora, você verá que esta nova opção aparecerá para todas as peças que aparecem na `ListView`, e não apenas para as que não foram sincronizadas. Precisamos corrigir isso. Nossa primeiro passo é alterar a nossa tag `viewCell` para o conteúdo que temos na sequência.

```
<viewCell BindingContextChanged = "OnBindingContextChanged" >
```

Agora, no code-behind de nossa visão, precisamos implementar o método `OnBindingContextChanged()`, tal qual é apresentado no seguinte trecho de código. Observe que o trabalho do método é verificar se a peça que está sendo preparada para ser exibida no ListView está sincronizada e, se estiver, removemos a opção de sincronização. Para recuperar o item a ser removido estamos fazendo uso do LINQ, pelo método `Where()`, que já utilizamos no livro. Precisamos também verificar se a peça obtida não é nula.

```
protected void OnBindingContextChanged ( object sender, EventArgs e)

{
    base.OnBindingContextChanged();
    ViewCell theViewCell = ((ViewCell)sender);

    var peca = (Peca)theViewCell.BindingContext;

    if (peca != null && peca.Sincronizado)
    {
        var itemSincronizar = theViewCell.ContextActions.Where(i => i.Text.Equals( "Sincronizar" )).First();
        theViewCell.ContextActions.Remove(itemSincronizar);
    }
}
```

Teste sua aplicação e veja que as opções de contexto são exibidas de acordo com o estado de cada peça, em relação à sua sincronização com o servidor Web.

Precisamos agora codificar o `Command SincronizarCommand` que apontamos no `Context Action` como implementação para realizar a sincronização da peça da base local com a base Web. Veja na sequência a declaração do `Command` e, logo após, o trecho de código que deve ser inserido no método `RegistrarCommands()`, tudo isso em nossa `ViewModel` de listagem. Observe que o método é semelhante ao que temos no `GravarAsync()` do `CRUDViewModel`. Poderíamos pensar em alguma refatoração, mas deixarei isso com você, ok?

```
public ICommand SincronizarCommand { get ; set ; }

SincronizarCommand = new Command<Peca>( async (peca) =>
{
    Sincronizando = true ;

    peca.CaminhoImagem = peca.CaminhoImagem.Equals( "consultar.png" ) ? null : peca.CaminhoImagem;

    var result = await service.PostComArquivo(peca);

    Sincronizando = false ;

    if (result.ToLower().Equals( "true" ))
    {
        MessagingCenter.Send< string >( "Atualização realizada com sucesso." , "Informação" );

        if ( ! string .IsNullOrEmpty(peca.CaminhoImagem) && File.Exists(DependencyService.Get<IFotoLoadMediaPlugin>()
            .GetPathToPhoto(peca.CaminhoImagem)))
            File.Delete(DependencyService.Get<IFotoLoadMediaPlugin>().GetPathToPhoto(peca.CaminhoImagem));

        peca.CaminhoImagem = string .Empty;

        await pecasDAL.UpdateAsync(peca, peca.PecaID, nameof(Peca.PecaID), true );
        AtualizarImagem(peca);

        this .Pecas[ this .Pecas.IndexOf(peca)] = peca;
    }
}
```

```
    }

    else
    {

        MessagingCenter.Send< string >(result.ToLower().Equals( "false" ) ? "Erro na comunicação" : result, "Informação" );
    }
});
```

Teste agora sua aplicação. Insira uma peça sem imagem, depois retorne para a listagem, altere a peça, insira uma imagem. Tente fazer o mesmo procedimento forçando um erro de comunicação com o servidor, depois tente sincronizar a peça inserida apenas localmente.

Ficará para você a implementação de associação de peças ao atendimento. Já temos tudo preparado para isso, basta você implementar os modelos para a associação, as visões e ViewModels.

## 11.10 Conclusão

Chegamos ao final deste nosso capítulo. Ele trouxe várias novidades, recursos e técnicas. Foi possível criarmos serviços REST em Java, distribuí-los na internet e consumi-los em nossa aplicação Xamarin. Realizamos o processo de sincronismo dos dados de uma base central para nossa base de dados local ao dispositivo. Trabalhamos com o envio e recebimento de imagens para atualização de nossos controles visuais. Vimos também a possibilidade de atualizar um ListView com o recurso puxar para atualizar (*Pull to refresh*). Acredito que tudo que vimos neste capítulo será de muita utilidade em suas aplicações.

## C APÍTULO 12

# Os estudos não param por aqui

Os dispositivos móveis já fazem parte do dia a dia da maioria da população. Você, como programador (ou desenvolvedor), não pode perder essa onda.

São duas as maiores plataformas móveis atualmente disponíveis (iOS e Android) e várias são as ferramentas para desenvolvimento de aplicações para elas, quer seja de forma nativa ou híbrida. Neste livro, você teve acesso à metodologia de implementação de aplicações nativas às plataformas descritas, por meio do Xamarin e Xamarin Forms, utilizando uma única linguagem, o C#.

Fizemos um passeio sobre grande parte dos controles disponibilizados pelo Xamarin Forms. Também testamos as aplicações em emuladores e em dispositivos físicos, o que nos possibilitou uso de recursos como câmera e álbum de fotografia.

Como o foco da aplicação desenvolvida no livro foi comercial, não podíamos deixar de trabalhar com acesso a base de dados, o que fizemos a partir do capítulo cinco, por meio do SQLite. Uma característica comum em aplicações móveis é o sincronismo com aplicações na nuvem, e isso também foi trabalhado.

Adotamos o MVVM como modelo para interação entre nossas camadas de visão e negócio. Para a persistência dos dados na base de dados, fizemos uso do Entity Framework como framework de mapeamento de objetos para e do paradigma relacional.

O livro não esgotou os temas trabalhados. É preciso dedicação para investigação e descoberta de novas tecnologias e recursos. Tenho certeza de que ele foi mais do que um pontapé inicial para o seu desenvolvimento, no que se diz respeito a aplicações móveis.

Espero que, com o conteúdo que trabalhei, tenha sido despertada em você uma grande curiosidade e que o livro tenha desmitificado o desenvolvimento de aplicativos para dispositivos móveis. Agora é bola para a frente na evolução.

Obrigado pela companhia e sucesso.

## C APÍTULO 13

# Apêndice - Criação de serviços REST

Este material é um conteúdo complementar para os capítulos 10 e 11. Por meio deste documento, você poderá criar os serviços REST que são consumidos pela aplicação de exemplo do livro, testá-los localmente em sua máquina e distribuí-los na internet.

Primeiro, teremos que instalar algumas ferramentas para desenvolver a aplicação Web. Faremos uso do Java, Spring e PostgreSQL. Darei orientação para todo este processo. A implantação na Web será feita no Heroku (<https://www.heroku.com/>), pois não necessidade de pagar, tampouco de informar cartão de crédito para o período de avaliação. Se você tiver outra plataforma, fique inteiramente à vontade para usá-la. Para aplicações Web .NET existe o AppHarbor (<https://appharbor.com/>), que também é gratuito.

## 13.1 Preparação do ambiente para o desenvolvimento REST

Como dito, nossos serviços REST serão desenvolvidos com Java, Spring, persistência no PostgreSQL e implantação no Heroku. Desta maneira, precisamos ter em nossa máquina um ambiente que permita este desenvolvimento. A primeira ferramenta, que já deve estar instalada, pois executamos aplicações Android nos projetos do livro, é o Java. Mas, apenas como reforço, vamos precisar do JDK dele e você pode obtê-lo no link <http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html/>, caso não o tenha em sua máquina.

Não me aterei a explicar a plataforma e as ferramentas utilizadas, pois não é o foco do livro e nem deste documento. O objetivo aqui é apenas propiciar que você reproduza todo o ambiente que utilizaremos para que seja possível o consumo dos serviços REST utilizados no livro.

Como a solução proposta faz uso do Spring, que é um framework com diversos projetos que minimizam o esforço no desenvolvimento de aplicações Java, vamos precisar dele disponível em nossa máquina. Buscando facilitar este procedimento, vamos realizar o download do Spring Tool Suite, que é um IDE baseado no Eclipse e que facilita muito o desenvolvimento quando estamos utilizando as APIs fornecidas pelo Spring.

Realize o download do arquivo compactado para a sua plataforma no link <https://spring.io/tools/>. Recomendo que este processo seja feito na raiz de seu disco, pois o arquivo traz vários níveis de pastas e pode ocorrer de aparecer algum problema relacionado a nomes longos para algum arquivo. Em minha máquina, fiquei com a pasta `sts-bundle` na raiz de meu disco e, dentro dela, três pastas, sendo que o aplicativo do ambiente ficou na `sts-3.9.2.RELEASE`. Sua pasta pode variar, devido à versão do ambiente que baixar. O aplicativo `sts` deve ser executado. Você pode criar um atalho para ele. Ao executar a aplicação, será solicitado a você um caminho para a área de trabalho que será aberta pelo ambiente, o `workspace`. Eu também indiquei uma pasta na raiz de meu disco.

A última ferramenta que faz parte de nosso ambiente de desenvolvimento é o PostgreSQL. Precisaremos dele para acessar o banco em nossa máquina, na fase de testes. Você pode realizar o download pelo link <https://www.postgresql.org/download/>. Se você for instalar a ferramenta, fique atento à senha e à porta que serão configuradas. A instalação é simples, praticamente uma sequência de janelas que solicitam confirmações.

## 13.2 Criação do projeto Web no STS

Vamos criar nosso projeto Web, fazendo uso de recursos disponibilizados no STS para minimizar a dificuldade e configurações para a aplicação. Clique no menu `File->New->Spring Starter Project` e, na janela que se abre, informe valores para os campos, tal qual apresenta a figura a seguir. Você pode substituir os valores representados na figura conforme sua necessidade. Atenção: onde você ler `meucalhambeque`, substitua pelo nome que você der à sua aplicação.

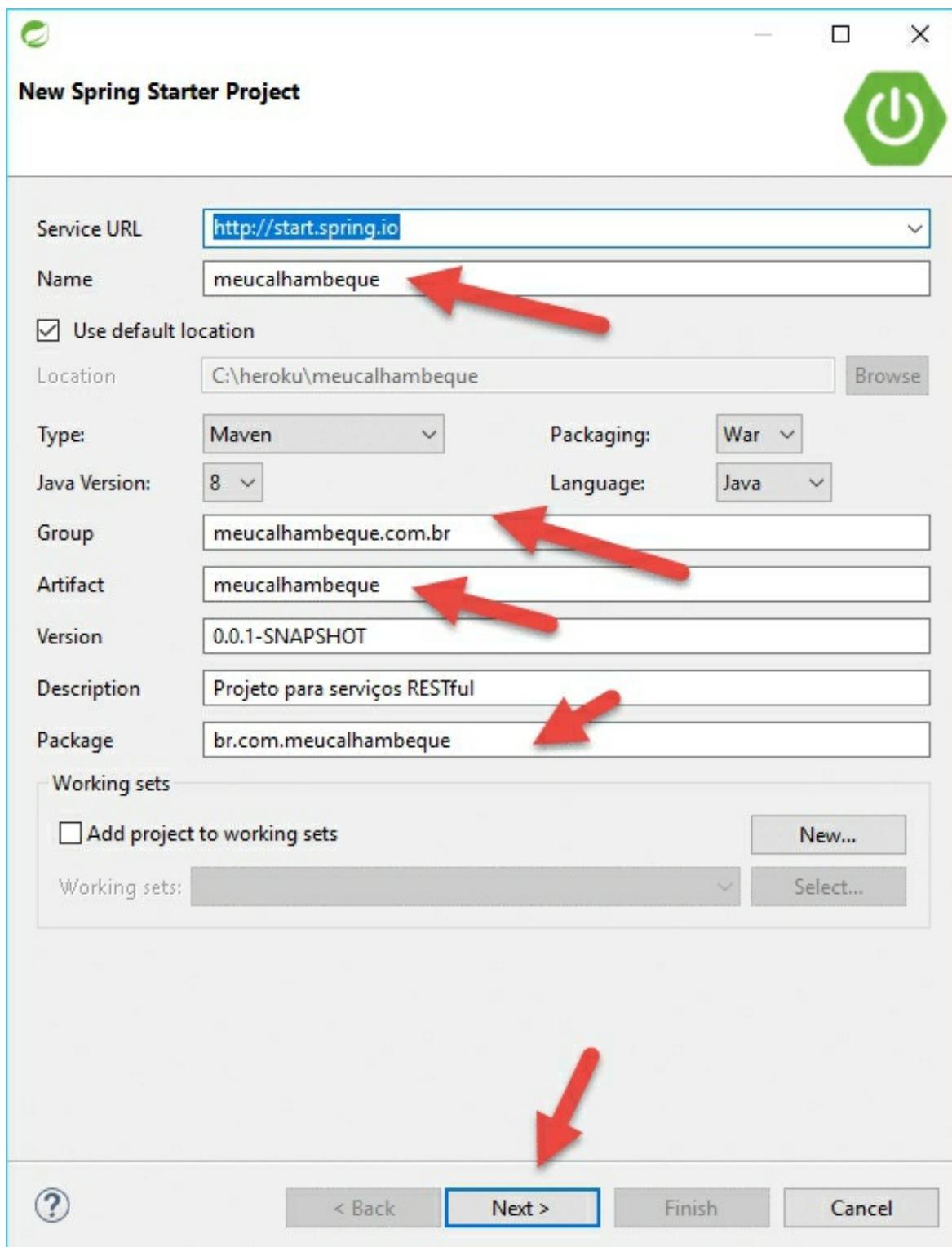


Figura 13.1: Criação do projeto Web - parte 01

Clicando no botão `Next >`, da figura anterior, você será direcionado para a segunda etapa de criação do projeto. Nesta nova etapa, é preciso informar quais os recursos que serão utilizados pela aplicação, para que eles sejam disponibilizados por meio do `Maven`, que é uma ferramenta de automação que nos auxilia na configuração de

dependências de nosso projeto. Você pode obter mais informações sobre o Maven em <https://maven.apache.org/> .

Na figura a seguir, sinalizei a versão do Spring Boot Version que estou utilizando no momento em que criei este projeto. Tentei utilizar uma versão mais atualizada, mas ela apresentava erro e não baixava as APIs necessárias. Talvez estes problemas tenham sido resolvidos quando você estiver lendo o livro.

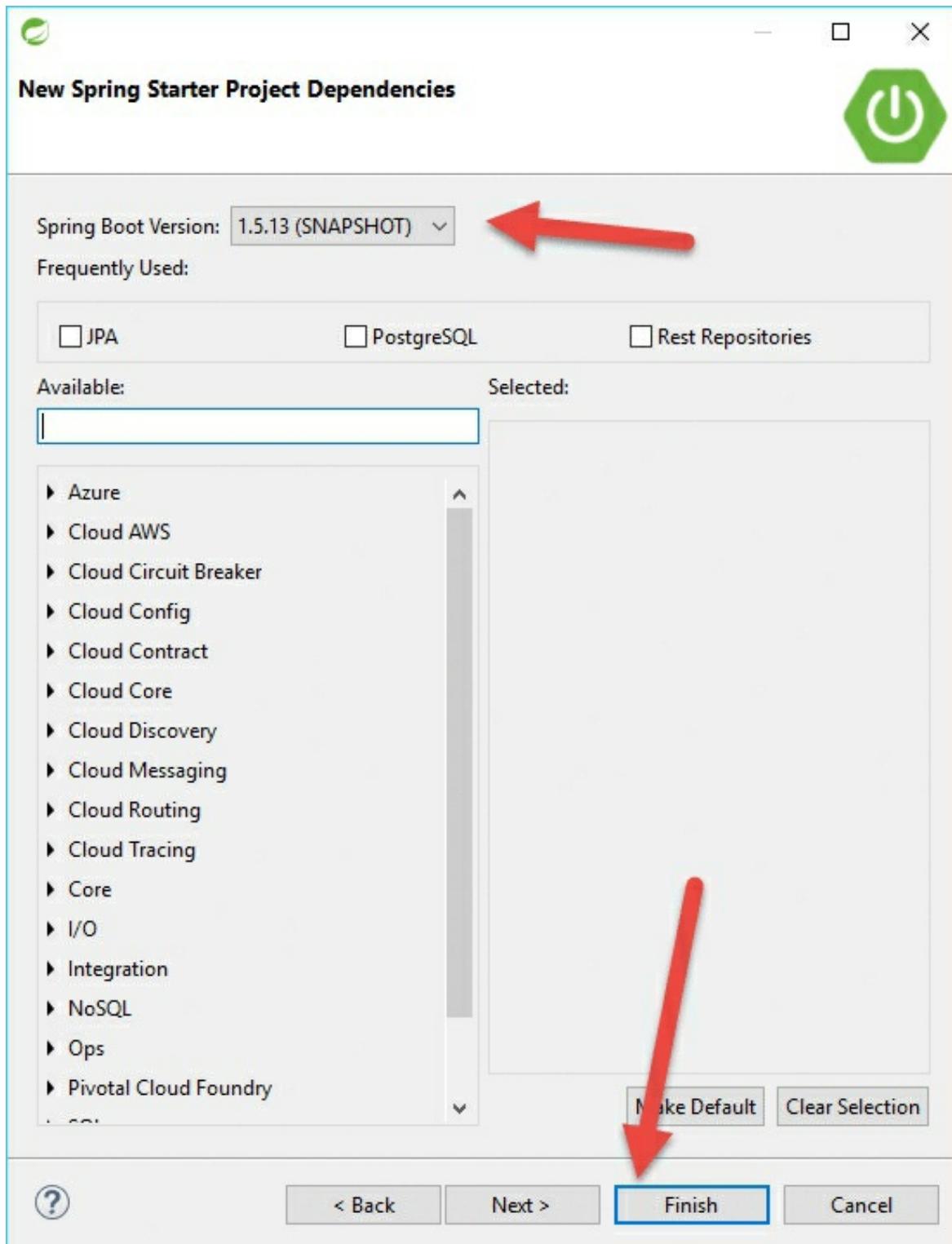


Figura 13.2: Criação do projeto Web - parte 02

Em vez de selecionar os itens e informá-los nessa janela, eu preferi clicar em `Finish` e disponibilizar para você o arquivo de configuração do Maven, o `POM.XML`, que está apresentado na sequência. Este arquivo estará disponível para você na raiz de seu projeto. Atente-se ao `<packaging>` que aponta a aplicação como `JAR`, e não `WAR`, como uma aplicação Java Web tradicional. É um requisito para que a aplicação execute. Isso é uma convenção do Spring e assim será validada a aplicação quando for distribuída no Heroku, onde será reconhecida como aplicação Web.

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

<project xmlns = "http://maven.apache.org/POM/4.0.0" xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd" >

    <modelVersion> 4.0.0 </modelVersion>

    <groupId> meucalhambeque.com.br </groupId>

    <artifactId> meucalhambeque </artifactId>

    <version> 0.0.1-SNAPSHOT </version>

    <packaging> jar </packaging>

    <name> meucalhambeque </name>

    <description> Projeto para serviços RESTful </description>

    <parent>

        <groupId> org.springframework.boot </groupId>

        <artifactId> spring-boot-starter-parent </artifactId>

        <version> 1.5.13.BUILD-SNAPSHOT </version>

        <relativePath/> <!-- lookup parent from repository -->

    </parent>

    <properties>
```

```
<project.build.sourceEncoding> UTF-8 </project.build.sourceEncoding>

<project.reporting.outputEncoding> UTF-8 </project.reporting.outputEncoding>

<java.version> 1.8 </java.version>

</properties>

<dependencies>

<dependency>

<groupId> org.springframework.boot </groupId>

<artifactId> spring-boot-starter-web </artifactId>

</dependency>

<dependency>

<groupId> org.springframework.boot </groupId>

<artifactId> spring-boot-starter-tomcat </artifactId>

<scope> provided </scope>

</dependency>

<dependency>

<groupId> org.springframework.boot </groupId>

<artifactId> spring-boot-starter-test </artifactId>

<scope> test </scope>

</dependency>

</dependencies>

<build>
```

```
<plugins>

<plugin>

<groupId> org.springframework.boot </groupId>

<artifactId> spring-boot-maven-plugin </artifactId>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id> spring-snapshots </id>

<name> Spring Snapshots </name>

<url> https://repo.spring.io/snapshot </url>

<snapshots>

<enabled> true </enabled>

</snapshots>

</repository>

<repository>

<id> spring-milestones </id>

<name> Spring Milestones </name>

<url> https://repo.spring.io/milestone </url>

<snapshots>

<enabled> false </enabled>
```

```
</snapshots>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id> spring-snapshots </id>

<name> Spring Snapshots </name>

<url> https://repo.spring.io/snapshot </url>

<snapshots>

<enabled> true </enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id> spring-milestones </id>

<name> Spring Milestones </name>

<url> https://repo.spring.io/milestone </url>

<snapshots>

<enabled> false </enabled>

</snapshots>

</pluginRepository>

</pluginRepositories>

</project>
```

Agora, na janela `Package Explorer`, que deve estar aberta do lado esquerdo de seu ambiente, expanda o projeto, clicando no `>` que aparece ao lado esquerdo do nome do projeto. Se esta janela não estiver visível para você, clique

no menu `Window->Show View->Package Explorer`. Com a visão do projeto ativa, expanda agora `src/main/java` e você verá um pacote nomeado `br.com.meucalhambeque` e, expandindo-o, uma classe que foi criada pelo IDE para que sua aplicação possa ser executada. Fique à vontade para ver o conteúdo desta classe, é bem simples, mas lembre-se de que por trás dela existe toda a estrutura do Spring.

Para o processo de login, o conteúdo que será enviado de nossa aplicação Xamarin para o serviço REST estará em formato JSON e será atribuído a um objeto. Desta maneira, precisamos criar uma classe que fornecerá este objeto para que, com base nele, possamos realizar o processo de autenticação. Para isso, criaremos um pacote Java que hospedará nossas classes de negócios. Vamos lá. Clique com o botão direito do mouse sobre o nome do pacote `br.com.meucalhambeque` e em `New->Package`. No campo `Name` da janela que se abre, complemente o nome `com.models`. Clique no botão `Finish` para que o pacote seja criado. Você pode obter informações sobre JSON em <https://www.json.org/json-pt.html/>.

Muito bem, vamos criar nossa classe `Login` para o processo de autenticação. No novo pacote criado, clique com o botão direito sobre o nome dele e em `New->Class`, e dê a essa nova classe o nome `Login`. Clique no botão `Finish` para que a classe seja criada e a implemente tal qual o código a seguir. Não entrarei em detalhes sobre a linguagem Java, mas você pode acessar <https://docs.oracle.com/javase/tutorial/index.html/> para obter uma introdução a ela.

```
package br.com.meucalhambeque.models;

public class Login {

    private String nome;
    private String senha;

    public String getNome () {
        return nome;
    }

    public void setNome (String nome) {
        this.nome = nome;
    }

    public String getSenha () {
        return senha;
    }

    public void setSenha (String senha) {
        this.senha = senha;
    }
}
```

Por fim, precisamos criar nossa classe que será responsável por fornecer o serviço de autenticação. Para isso, criaremos novamente um pacote Java que hospedará nossas classes controladoras de REST. O processo é o mesmo que vimos anteriormente. Nomeie o novo pacote como `.controllers`.

Vamos criar nossa classe REST para o processo de autenticação. Siga o procedimento adotado para a classe `Login`, só que agora criando a nova classe no pacote `controllers`. Dê a ela o nome `LoginController`. Veja o código na sequência. A variável `autenticado` foi criada apenas para melhorar a visibilidade do código. Novamente, fiz uso de valores constantes para a validação do usuário, mas você poderá adotar a lógica necessária para sua aplicação. Antes da classe, observe as anotações `@RestController` e `@RequestMapping`, que a definem como um controlador e o mapeamento para ela. Antes do método `login`, temos a definição de mapeamento para como o método será

requisitado, por meio da anotação `@PostMapping`. Por fim, o argumento recebido pelo método é anotado como `@RequestBody`. Observe que o objeto que deve chegar é da classe `Login`. Note também - pois usaremos esse conhecimento na implementação do serviço - que as propriedades (atributos Java) começam com letra minúscula, por convenção. É sempre importante saber detalhes dos nomes e tipos de dados que cada serviço recebe, para poder enviar os dados de maneira correta. Fique atento aos `imports`.

```
package br.com.meucalhambeque.controllers;

import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import br.com.meucalhambeque02.models.Login;

@RestController
@RequestMapping( "autenticacao/" )

public class LoginController {

    @PostMapping( "/login" )

    public ResponseEntity<Boolean> login (@RequestBody Login login) {

        Boolean autenticado = (login.getNome().equals( "everton" ) && login.getSenha().equals( "1234" ));

        return new ResponseEntity<Boolean>(autenticado, new HttpHeaders(), HttpStatus.OK);
    }
}
```

### 13.3 Teste e deploy da aplicação para o Heroku

Quando desenvolvemos serviços, em nosso caso, serviços REST, é interessante que os testemos antes de realizar o deploy para o servidor que o oferecerá. Uma ferramenta comum e eficiente para este serviço é o **Postman**, que pode ser obtido em <https://www.getpostman.com/>. Instale o aplicativo em sua máquina e o execute. Uma janela com atalhos é exibida, você pode fechá-la, mas depois é interessante dar uma investigada nela, caso você queira.

Voltemos ao STS, vamos executar nossa aplicação. Para isso, abra o pacote `br.com.meucalhambeque`, clique com o botão direito do mouse sobre a classe `MeucalhambequeApplication`, escolha a opção `Run as` e então confirme `Spring Boot App`. Com a aplicação sendo executada, vamos voltar ao Postman.

Veja, na figura a seguir, os destaques do que você precisará trabalhar. O primeiro passo é configurar a mensagem como `POST`. Em seguida, informe o endereço completo para o serviço, tendo como base a máquina local, que é onde ele está sendo executado. Em nosso caso, o endereço completo é `http://localhost:8080/autenticacao/login`. Precisamos informar os valores que serão recebidos e armazenados no argumento `login` do método invocado. Para isso, clique em `Body`, depois em `Raw`, selecione `JSON (application/json)` e informe `{"nome":"everton", "senha":"1234"}` no

campo aberto. Observe que estamos enviando um objeto JSON com dois pares chave:valor . Note que as chaves são os nomes dos atributos da classe Login.java . Isso é extremamente importante e necessário para o correto funcionamento do serviço.

Com tudo configurado, clique no botão Send . Quando o método for executado, na caixa Response , você receberá o valor true . Tente mudar os valores enviados para ver o retorno como false . Você pode agora interromper a execução da aplicação no STS clicando no botão de parada (quadrado vermelho).

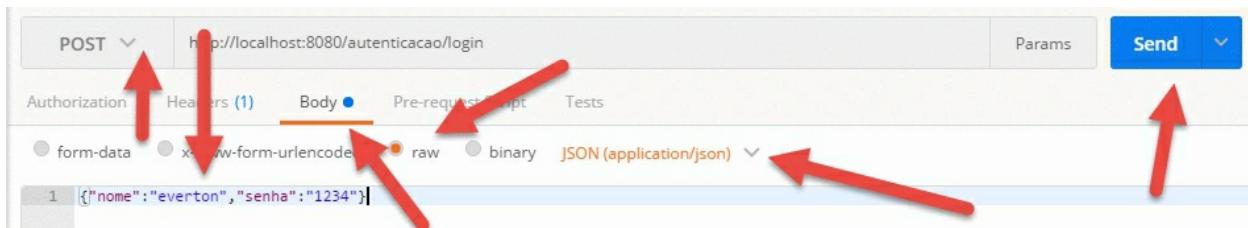


Figura 13.3: Configurando chamada ao serviço REST no Postman

Vamos para o deploy no Heroku? Acesse <https://www.heroku.com/> e, no topo da página, clique em sign up para criar sua conta ou, se já tiver uma, clique em Log in e informe suas credenciais. Eu optei por apresentar a maneira mais simples para criação e deploy de aplicação oferecida pela plataforma, mas, investigando o portal, você poderá obter maiores informações sobre outras maneiras de criação e deploy.

Com sua autenticação realizada, vamos partir para a criação de nossa aplicação. Assim que você se autentica, uma listagem de suas aplicações é exibida. Em nosso caso, se for a primeira vez, nada é mostrado, mas um botão chamado New é exibido. Clique nele e escolha a opção create new app . Na página que se abre, digite o nome para sua aplicação ( meuca1hambeque em meu caso) e mantenha a região dos Estados Unidos. Infelizmente não temos o Brasil nas regiões disponibilizadas e o EUA é o mais próximo.

Após a criação da aplicação, somos redirecionados para a página de Deploy , que pode também ser acessada por uma opção, de mesmo nome, disponibilizada no menu que aparece quando clicamos no nome da aplicação na listagem anteriormente comentada. Para nosso trabalho faremos uso do GitHub. Desta maneira, você precisa ter uma conta criada nele (<https://github.com/> ). O procedimento é simples. Siga as orientações do site. Com a conta criada, crie um repositório; eu chamei de meuca1hambeque . Retorne à área de deploy de sua aplicação no Heroku e marque a opção de deploy pelo GitHub. Siga as orientações de conexão. Quando conectado, o Heroku oferecerá algumas opções de controle de deploy. Não precisa marcar nenhuma.

Precisaremos instalar em nossa máquina o Git e o Heroku CLI. Veja as orientações para este processo nos links: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git/> e <https://devcenter.heroku.com/articles/heroku-cli#download-and-install/> .

Com as ferramentas instaladas, vamos configurar nossa aplicação para ser um repositório local do Git. Se você estiver em uma rede que possua proxy, será preciso configurá-lo no Git. Para isso, na sua pasta c:\users\você , existe um arquivo chamado .gitconfig . Se ele não existir, você precisará criá-lo. Insira nele as configurações a seguir.

```
[http]
proxy = http://server:port
```

Via console, acesse a pasta onde está seu projeto Java. Digite as instruções apresentadas na sequência. Lembre-se de que meuca1hambeque é o nome de nossa aplicação no Heroku. As duas primeiras instruções são relacionadas à criação do repositório local. A terceira registra mudanças realizadas desde o último commit. Seria interessante dar uma estudada sobre os conceitos de commit/pull/push do Git, quando você puder. A quarta instrução registra um repositório remoto, referente à nossa aplicação. A última envia as mudanças realizadas para nosso repositório remoto, o Heroku.

```
git init
```

```
git add .
git commit -am "Commit inicial"
heroku git:remote -a meucalhambeque
git push heroku master
```

Pronto, com isso, nossa primeira versão da aplicação já está distribuída na internet. Agora, voltando para o Postman, vamos testá-la. Na URL a ser invocada, digite <https://meucalhambeque.herokuapp.com/autenticacao/login>. Veja o nome de sua aplicação no endereço. Mantenha os parâmetros anteriores e clique no botão `Send`. Pronto, aplicação distribuída e testada.

### 13.4 O serviço REST que receberá o objeto e imagem para persistência

Com a aplicação Xamarin funcionando, vamos agora para o STS. Precisamos implementar nossos serviços REST, para recuperarmos dados e atualizarmos, pois, relembrando, nossa aplicação terá uma base de dados centralizada, no Heroku. Desta maneira, precisamos preparar nosso ambiente de desenvolvimento para permitir o acesso ao PostgreSQL, que será o banco de dados que utilizaremos para a base de dados centralizada. Insira as duas dependências a seguir em seu `POM.xml`. O conteúdo, a partir desta seção refere-se ao capítulo 11 do livro.

```
<dependency>

<groupId> org.springframework.boot </groupId>

<artifactId> spring-boot-starter-data-jpa </artifactId>

</dependency>

<dependency>

<groupId> org.postgresql </groupId>

<artifactId> postgresql </artifactId>

<scope> runtime </scope>

</dependency>
```

Precisamos implementar, também na nossa aplicação servidora, nossa classe de modelo para as `Peças`. Desta maneira, no pacote `br.com.meucalhambeque.models`, adicione a classe apresentada na listagem que se segue. O Spring utiliza a JPA como mecanismo de mapeamento objeto relacional para as classes, semelhante em funcionalidade com o EF Core. Como dito, não entrarei em detalhes sobre estas ferramentas que utilizaremos para os serviços REST, pois estes assuntos juntos, por si só, resultariam em vários livros, e nosso objetivo aqui é o Xamarin. Atente aos `imports` necessários e perceba que anotamos a classe como `@Entity` e o identificador como `@Id`. Observe que, diferente do C#, o Java utiliza a classe `UUID`, que é um sinônimo para `GUID`. Veja o objeto `arquivo`, que armazenará a imagem que será recebida pelo serviço. Note que ela está anotada como `@Transient`, o que garante que ela não será mapeada para a base de dados.

```
package br.com.meucalhambeque.models;
```

```
import java.util.UUID;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Transient;

import org.springframework.web.multipart.MultipartFile;

@Entity

public class Peca {

    @Id

    private UUID pecaID;

    private String nome;

    private double valor;

    private String caminhoImagen;

    private boolean sincronizado;

    private byte [] arquivoEmBytes;

    public byte [] getArquivoEmBytes() {

        return arquivoEmBytes;

    }

    public void setArquivoEmBytes ( byte [] arquivoEmBytes) {

        this .arquivoEmBytes = arquivoEmBytes;

    }

    @Transient

    private MultipartFile arquivo;

    public MultipartFile getArquivo () {

        return arquivo;

    }

    public void setArquivo (MultipartFile arquivo) {

        this .arquivo = arquivo;

    }

    public UUID getPecaID () {

        return pecaID;
```

```

}

public void setPecaID (UUID pecaID) {
    this .pecaID = pecaID;
}

public String getNome () {
    return nome;
}

public void setNome (String nome) {
    this .nome = nome;
}

public double getValor () {
    return valor;
}

public void setValor ( double valor) {
    this .valor = valor;
}

public String getCaminhoImagen () {
    return caminhoImagen;
}

public void setCaminhoImagen (String caminhoImagen) {
    this .caminhoImagen = caminhoImagen;
}

public boolean isSincronizado () {
    return sincronizado;
}

public void setSincronizado ( boolean sincronizado) {
    this .sincronizado = sincronizado;
}
}

```

Ainda sobre a persistência, o Spring utiliza o conceito de repositórios, em que ele implementa os métodos básicos para acesso aos dados (o CRUD). Ele só precisa que seja criada uma interface , tal qual o código a seguir mostra.

Para implementar esta interface, como você pode confirmar na definição do package , é preciso criar um pacote, da mesma maneira que fizemos anteriormente, com o nome dao .

Veja que a interface é anotada como @Repository e a extensão (extends da classe) especifica para qual classe de negócio a interface será implementada, com o tipo de dado da propriedade que representará a chave primária na tabela.

```
package br.com.meucalhambeque.dao;

import java.util.UUID;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

import br.com.meucalhambeque02.models.Peca;

@Repository

public interface PecaRepository extends CrudRepository<Peca, UUID> { }
```

Vamos enfim passar ao nosso controlador, que implementará nossos serviços REST. No pacote br.com.meucalhambeque.controllers , crie a classe PecasController , com o código apresentado a seguir. A princípio, implementaremos apenas o método que gravará o objeto recebido na base de dados. Como o método receberá dados e um arquivo, a anotação para o argumento é @ModelAttribute . No corpo do método, veja que gravaremos a imagem recebida na pasta imagensPecas , que você precisa criar dentro de src/main/webapp . O tipo de retorno para o método é um String , pois precisamos retornar a exceção, caso ela ocorra. Caso tudo dê certo, o caminho da imagem na Web é retornado para que possa ser atualizado no dispositivo.

```
package br.com.meucalhambeque.controllers;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
import javax.servlet.ServletContext;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
```

```

import org.springframework.transaction.annotation.Transactional;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.ResponseBody;

import org.springframework.web.bind.annotation.RestController;

import br.com.meucalhambeque.dao.PecaRepository;

import br.com.meucalhambeque.models.Cliente;

import br.com.meucalhambeque.models.Peca;

@RestController

@RequestMapping( "pecas/" )

public class PecaController {

@Autowired

PecaRepository repository;

@Autowired

ServletContext context;

@PostMapping( "/savecomimagem" )

@Transactional

public ResponseEntity<String> saveComImagem (@ModelAttribute Peca peca){

    if (peca.getPecaID() == null )

        peca.setPecaID(UUID.randomUUID());

    peca.setSincronizado( true );

    String nomeArquivo = null ;

    if (peca.getArquivo() != null && peca.getCaminhoImagem() != null && !peca.getCaminhoImagem().startsWith( "http" ) ){

        nomeArquivo = (peca.getCaminhoImagem().lastIndexOf( "/" ) > 0) ?

            peca.getCaminhoImagem().substring(peca.getCaminhoImagem().lastIndexOf( "/" ) + 1) : peca.getCaminhoImagem();

        String realPathToUploads = context.getRealPath( "imagensPecas" ) + File.separator + nomeArquivo;

        byte [] bytes;

        try {

            bytes = peca.getArquivo().getBytes();


```

```

        peca.setArquivoEmBytes(bytes);
        Path path = Paths.get(realPathToUploads);
        Files.write(path, bytes);

    } catch (IOException e) {

        return new ResponseEntity<String>(e.getMessage(), new HttpHeaders(), HttpStatus.OK);
    }
}

if (nomeArquivo != null && !peca.getCaminhoImagem().startsWith( "http" ))
    peca.setCaminhoImagem( "https://meucalhambeque02.herokuapp.com/imagensPecas/" + nomeArquivo);
repository.save(peca);

return new ResponseEntity<String>(peca.getCaminhoImagem(), new HttpHeaders(), HttpStatus.OK);
}
}

```

Para finalizar esta etapa, precisamos configurar alguns parâmetros de conexão para a aplicação. Em seu projeto no STS, expanda o pacote `src/java/resources` e dê duplo clique no arquivo `application.properties`. Implemente as seguintes configurações nele. Observe a última propriedade JPA, que fará com que a base de dados seja apagada, caso exista, e criada para a aplicação. Depois de executar os testes, você pode mudar este valor para `update`, para evitar perder os dados já gravados. As duas últimas propriedades configuram o tamanho máximo para o arquivo que será enviado. Você pode rever isso de acordo com sua aplicação.

```

spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=senhainformadanainstalação
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.http.multipart.max-file-size=100MB
spring.http.multipart.max-request-size=100MB

```

Vamos testar os serviços no Postman? Execute sua aplicação no STS. Para isso, abra o pacote `br.com.meucalhambeque`, clique com o botão direito do mouse sobre a classe `MeucalhambequeApplication`, escolha a opção `Run as` e então confirme com `Spring Boot App`. Podemos começar nossos testes gravando uma peça. No Postman, crie uma chamada `POST`, com a URL `http://localhost:8080/pecas/savecomimagem`. Configuraremos os dados a serem enviados de maneira diferente agora, pois enviaremos um arquivo junto.

Em `Body`, escolha a opção `form-data` e registre o conjunto de `chave-valor` na listagem abaixo do campo, tal qual é apresentado pela figura a seguir. As chaves deverão ter o nome das propriedades da classe `Peca.java`. Clique no botão `Send` quando tudo estiver preenchido. É para você receber como retorno o caminho da imagem armazenada no servidor local.

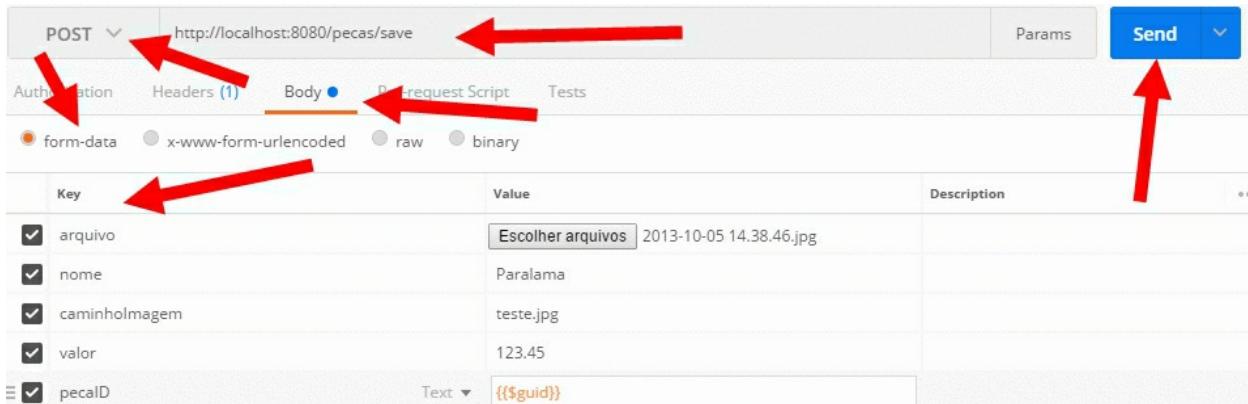


Figura 13.4: Postman com dados para envio de imagem

Você pode executar o **PgAdmin**, ferramenta instalada com o PostgreSQL, para acesso à base de dados. Você precisará adicionar um novo servidor quando a ferramenta estiver aberta, informando os dados de conexão que listamos agora pouco. Expanda a base de dados, depois em *schemas* você verá um nó que representa as tabelas, onde você terá sua tabela `peca`. Clique com o botão direito sobre o nome dela e escolha a opção de visualização dos registros.

### 13.5 Envio de um objeto com uma imagem para o serviço REST

Para que possamos consumir, em nossa aplicação Xamarin, os serviços criados anteriormente, precisamos fazer o deploy deles para o Heroku. Entretanto, como nossa aplicação agora faz uso de uma base de dados PostgreSQL, precisamos configurar isso para ela. Vamos realizar esta configuração.

Acesse o portal do Heroku, autentique-se e accese sua aplicação. No menu de opções, clique em `Resources`. Depois, em `Add-ons`, escolha `Heroku Postgres`. Pode manter o plano `Hobby Dev - Free` e clicar no botão `Provision`. Após este processo, o `Add-on` para o banco estará disponível. Clique sobre o nome dele e uma nova guia do navegador será aberta. Precisamos obter os dados de acesso ao banco no servidor. Na nova guia, clique em `Settings` e depois em `view Credentials`. Vamos agora usar estes valores em nosso arquivo `application.properties`, tal qual está mostrado na sequência. Existe uma observação nesta página informando que estes valores não são permanentes. Certamente em um plano pago isso não ocorre. Lembre-se de que, depois da primeira execução, faremos com testes pelo `Postman`. É interessante você trocar `O create-drop` por `update` e realizar novo commit da aplicação.

```
spring.jpa.database=POSTGRESQL
spring.datasource.platform=postgres
spring.datasource.url=jdbc:postgresql://informe-o-host:5432/informe-o-database?
ssl=true&sslfactory=org.postgresql.ssl.NonValidatingFactory
spring.datasource.username=informe-o-usuário
spring.datasource.password=informe-a-senha
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.http.multipart.max-file-size=100MB
spring.http.multipart.max-request-size=100MB
```

Precisamos agora enviar nossas alterações para o Heroku. Acesse a pasta de sua aplicação e execute as instruções a seguir. Depois, no `Postman`, substitua a URL que estávamos testando localmente para `https://meucalhambeque.herokuapp.com/pecas/savecomimagem`, com os parâmetros igualmente configurados, e envie a

requisição para o servidor. É para tudo dar certo e você receber o caminho em que a imagem está disponibilizada na Web. Se você quiser verificar sua base de dados remota pelo PgAdmin, basta criar um novo servidor e, nos dados, informar os fornecidos pelo Heroku. O inconveniente é que aparecem muitas bases de dados e você deverá localizar a sua para abrir. Mas funciona direitinho.

```
git add .
git commit -am "Pecas save"
git push heroku master
```

## 13.6 Obtenção de dados remotos por meio de um serviço REST

Nosso primeiro serviço a ser implementado nesta seção será referente à obtenção de todas as peças existentes no servidor e, com elas, popularmos nossa base no dispositivo. Para isso, precisamos ir ao STS e, na classe `PecaController.java`, implementar o método que está a seguir. Veja que o método é simples, instanciamos um `ArrayList` para a classe `Peca`, recuperamos todos os objetos por meio do `repository` e pela invocação do método `findAll()`. Com o retorno, invocamos `forEach()` para que todos os elementos recuperados possam ser adicionados à coleção `list`, que é retornada ao chamador. Eu optei por não trabalhar exceções neste método, mas é interessante você pensar nisso e, em caso de ocorrência de alguma, poderia retornar um objeto vazio para o cliente. Fica a sugestão.

```
@RequestMapping( "/findAll" )
public ResponseEntity<List<Peca>> findAll() {
    List<Peca> list = new ArrayList<Peca>();
    repository.findAll().forEach(list::add);
    return new ResponseEntity<List<Peca>>(list, new HttpHeaders(), HttpStatus.OK);
}
```

Execute sua aplicação REST localmente, por meio do STS. Lembra como? Botão direito do mouse sobre o nome do projeto, `Run as->Spring Boot App`. Com a aplicação em execução, vamos ao Postman. Crie agora uma requisição `GET`. Veja no código anterior a anotação `RequestMapping`, que caracteriza este tipo de mensagem. No endereço, informe `http://localhost:8080/pecas/findAll` e clique no botão `Send`. Se tudo estiver OK, na parte de baixo da janela você verá uma coleção de objetos JSON, que foi retornada.

## 13.7 Remoção de uma peça no servidor

Para que possamos remover uma peça de nossa base de dados centralizada, precisamos criar um novo serviço em nossa classe `PecaController.java`. Então, voltando ao STS, vamos criar o método a seguir. Veja que anotamos o método também com `@PostMapping` e que agora o argumento do método é um `@RequestBody`. Nossa única instrução é a responsável por remoção do objeto da base de dados. Novamente, informo a adoção de uma estratégia simples para o problema que estamos trabalhando. Lembre-se sempre dos problemas já comentados com objetos associados. Você pode pensar também no tratamento de possíveis exceções, como fizemos no método que recebe a peça e seu arquivo de imagem (`saveComImagem()`).

```
@PostMapping( "/removebyid" )
@Transactional

public ResponseEntity<String> removeById (@RequestBody UUID id){
    repository.delete(id);
```

```

        return new ResponseEntity<String>( "true" , new HttpHeaders() , HttpStatus.OK);
    }
}

```

Com este método implementado, podemos testá-lo no Postman, antes de enviarmos a nova implementação para o Heroku. Acesse o Postman, e vamos recuperar as peças que temos já registradas em nossa base local, que inserimos durante nossos testes com o Postman. Para isso, invoque o método `http://localhost:8080/pecas/findAll` por uma mensagem `GET`. Na parte de baixo da janela, no resultado, copie um dos valores retornados para a propriedade `pecaID`. Com este valor copiado, configure uma requisição `POST` informando, na URL, o endereço `http://localhost:8080/pecas/removebyid`. Em `Body`, marque `raw`, selecione `JSON (application/json)` e então cole, entre aspas duplas, o valor copiado anteriormente, referente ao ID de uma das peças registradas. Envie a requisição ao servidor e depois requisite novamente as peças registradas. Veja que a peça cujo ID você copiou foi retirada da base de dados.

Com o novo serviço REST implementado e testado, precisamos enviá-lo para o Heroku. Para realizarmos esta atividade, acesse pelo console a pasta de sua aplicação, tal qual já fizemos anteriormente, e execute as instruções a seguir.

```

git add .
git commit -am "Pecas removebyid"
git push heroku master

```

Realize os testes que fizemos anteriormente com o Postman para o serviço agora distribuído no Heroku. Utilize a URL `https://meucalhambeque.herokuapp.com/pecas/findAll` (via `GET`) para recuperar todas as peças registradas e a `https://meucalhambeque.herokuapp.com/pecas/removebyid` (via `POST`), configurada para enviar um ID. Você pode usar o PgAdmin também para ver a base de dados.

## 13.8 Recuperação de uma peça específica no servidor

Em nossa aplicação, não temos a necessidade de um método que recupere uma peça em específico no servidor, mas vamos implementar o código necessário, que lhe poderá ser útil em aplicações futuras. Veja no código a seguir como deve ser seu método na classe `PecaController.java`. O método utilizará o `GET` para ser invocado. Execute sua aplicação localmente e teste no Postman. A URL deverá ser semelhante a `http://localhost:8080/pecas/findbyid?id=819ad850-6c20-43f6-86e2-e311dc75d7d6`. Após o teste, realize o `commit` e `push`, mudando apenas o nome para a atualização.

Teste sua aplicação no Heroku, também com o Postman. A URL deverá ser semelhante a `https://meucalhambeque.herokuapp.com/pecas/findbyid?id=819ad850-6c20-43f6-86e2-e311dc75d7d6`. Observe a passagem do argumento por meio do símbolo de interrogação (?) seguido do nome do parâmetro e o valor que deverá ser atribuído a ele no servidor.

```

@RequestMapping( "/findbyid" )
public ResponseEntity<Peca> findById(@RequestParam( "id" ) UUID id) {
    Peca p = repository.findOne(id);
    return new ResponseEntity<Peca>(p, new HttpHeaders(), HttpStatus.OK);
}

```

## 13.9 Envio de uma peça sem imagem para o servidor

Em nosso exemplo desenvolvido nos capítulos 10 e 11 no livro e também neste documento complementar, trabalhamos um modelo de negócio onde há uma imagem e implementamos o serviço REST (neste material

complementar) e o método consumidor prevendo esta situação (no livro). Vimos que existe uma certa burocracia, em ambas implementações, quando o envio é composto por um arquivo.

Entretanto, podemos ter situações em que precisamos enviar apenas o objeto que será persistido, sem ele possuir uma imagem associada, o que é mais simples para implementar. Com vistas a subsidiar isso em aplicações futuras que você possa fazer, vamos agora implementar um serviço REST que receba apenas nosso objeto. Sendo assim, no STS, em nossa classe `PecaController.java`, implemente o seguinte método.

```
@PostMapping( "/savesemimagem" )
@Transactional

public ResponseEntity<String> saveSemImagem (@RequestBody Peca peca){

    if (peca.getPecaID() == null)
        peca.setPecaID(UUID.randomUUID());

    peca.setSincronizado( true );
    repository.save(peca);

    return new ResponseEntity<String>( "true" , new HttpHeaders(), HttpStatus.OK);
}
```

Bem mais simples, não é? Entretanto, veja que não tratei a possibilidade de exceções, apenas para minimizar o código apresentado. Lembre-se de que é sempre bom pensar nesta situação.

Você pode testar o novo serviço no Postman, tal qual fizemos para os serviços anteriores, sendo necessário que você execute sua aplicação localmente caso o teste seja local. Depois, para testar no Heroku, você precisará realizar o `commit` e `push` pelo Git. Como nosso serviço é `POST`, você precisa configurar a mensagem desta maneira e informar, em forma de JSON, o objeto que será enviado, tal qual é apresentado na sequência.

```
{
    "pecaID": "{$guid}" ,
    "nome": "Teste sem imagem" ,
    "valor": 2,
    "sincronizado": false
}
```

### 13.10 Um serviço REST que retorna uma imagem

Vamos realizar uma adaptação na classe de negócio `Peca.java`, para ter em suas propriedades a imagem associada à peça. Estando no STS, insira na classe o código a seguir. A imagem precisa ser persistida como dado binário, por isso a declaração da propriedade como um array de bytes.

```
private byte [] arquivoEmBytes;

public byte [] getArquivoEmBytes() {
    return arquivoEmBytes;
}

public void setArquivoEmBytes ( byte [] arquivoEmBytes) {
```

```

        this.arquivoEmBytes = arquivoEmBytes;
    }

```

Vamos criar um novo método para utilizarmos a funcionalidade de armazenarmos a imagem na base de dados. Depois você pode decidir em seu projeto qual é a melhor solução para ele e deixar apenas um dos métodos, lembrando que, se você for utilizar o Heroku como solução servidora, e optar por manter o arquivo fisicamente, precisará de um servidor de arquivos para manter os uploads, tal qual é informado em <https://help.heroku.com/K1PPS2WM/why-are-my-file-uploads-missing-deleted/>. Veja na sequência a listagem para o novo método. Ele fica até mais simples.

```

@PostMapping( "/saveimagemembytes" )
@Transactional

public ResponseEntity<String> saveComImagemEmBytes (@ModelAttribute Peca peca){

    if (peca.getPecaID() == null )
        peca.setPecaID(UUID.randomUUID());

    peca.setSincronizado( true );

    if (peca.getArquivo() != null ) {

        try {

            byte [] bytes = peca.getArquivo().getBytes();
            peca.setArquivoEmBytes(bytes);

            peca.setCaminhoImagem( null );

        } catch (IOException e) {

            return new ResponseEntity<String>(e.getMessage(), new HttpHeaders(), HttpStatus.OK);
        }
    } else {

        Peca pecaJaGravada = repository.findOne(peca.getPecaID());

        if (pecaJaGravada != null )
            peca.setArquivoEmBytes(pecaJaGravada.getArquivoEmBytes());
    }

    repository.save(peca);

    return new ResponseEntity<String>( "true" , new HttpHeaders(), HttpStatus.OK);
}

```

Você pode testar localmente este método em funcionamento, pelo Postman, requisitando uma mensagem tal qual fizemos para o armazenamento do arquivo em disco e verificar, pelo PgAdmin, o dado atribuído ao campo arquivoembytes da tabela de peças. Também no Postman, é possível testar a recuperação de todos os objetos armazenados, pelo serviço `findAll` ou o `findById`, mas nos cabe agora uma ponderação sobre estes serviços.

Em nossa aplicação, quando recuperarmos as peças, sejam todas ou uma única, nós faremos uso da imagem armazenada em bytes na base de dados local? Lembre-se de que quando recuperamos todos as peças queremos realizar a sincronização com a base local e estamos criando serviços REST, com as imagens armazenadas na internet justamente para não termos o peso do armazenamento destas imagens no dispositivo. Então, podemos assumir que

não queremos este dado vindo junto com a resposta dos serviços. Veja que isso é para a nossa aplicação. Em outro contexto, isso pode se tornar necessário. O que precisamos fazer? Veja o código a seguir, do método `findAll()` de nossa classe `PecaController.java`. Realizamos algumas alterações para que, a cada objeto recuperado, possamos atribuir `null` à propriedade `arquivoEmBytes`, evitando que este dado seja devolvido para nosso consumidor.

```
@RequestMapping( "/findAll" )

public ResponseEntity<List<Peca>> findAll() {

    List<Peca> list = new ArrayList<Peca>();
    repository.findAll().forEach((peca) -> {

        peca.setArquivoEmBytes( null );
        list.add(peca);
    });

    return new ResponseEntity<List<Peca>>(list, new HttpHeaders(), HttpStatus.OK);
}
```

Podemos pensar nessa mesma situação para o `findById`, como temos na listagem a seguir. São mudanças simples, estas que realizamos nestes dois serviços, mas que podem ser importantes no tráfego no momento da sincronização e recuperação de dados do servidor.

```
@RequestMapping( "/findById" )

public ResponseEntity<Peca> findById (@RequestParam( "id" ) UUID id) {

    Peca p = repository.findOne(id);
    p.setArquivoEmBytes( null );

    return new ResponseEntity<Peca>(p, new HttpHeaders(), HttpStatus.OK);
}
```

Muito bem, vamos para a criação do serviço que nos retornará uma imagem, gravada em nossa base de dados em binário. O código é simples. Recebemos um ID pela URL, de uma maneira melhor e mais natural do que vimos para o `findById`, mas preferi mostrar as duas situações para você saber que elas existem. Veja o código para este novo serviço na listagem a seguir. Observe a anotação `@GetMapping` e a definição de como o caminho final da URL deve ser composto. Note os colchetes `{}` para definir o nome da variável que receberá o ID que enviaremos. Observe também o uso de `produces` na anotação. Ele garante, para o cliente, que está enviando uma imagem.

```
@GetMapping( value = "/getimagem/{id}" , produces = MediaType.IMAGE_JPEG_VALUE )

public @ResponseBody byte [] getImagenComoBytes(@PathVariable(value= "id" ) UUID id) throws IOException {

    return repository.findOne(id).getArquivoEmBytes();
}
```

Depois de você executar sua aplicação no STS, o teste local no Postman pode ser realizado com, por exemplo, a URL `http://localhost:8080/pecas/getimagem/655b160c-21a6-4641-b78b-2684736e16e9`. Se você informar um ID válido, a imagem será exibida para você. Se nada for enviado, um erro de página não encontrada (404) será exibido e, se o valor enviado não for encontrado na base de dados, será retornado um `null`. O que acha de realizar o `commit` e `push` do Git para a aplicação no Heroku e testá-la no Postman?

## 13.11 Conclusão

Chegamos ao final deste documento. Com ele foi possível implementarmos serviços REST em Java, distribuirmos

na internet, por meio do Heroku, com persistência em base de dados do PostgreSQL e testarmos suas funcionalidades por meio do Postman. Os serviços aqui codificados têm o foco complementar para consumo pelo conteúdo dos capítulos 10 e 11 deste livro.

# Sumário

[ISBN](#)

[Agradecimentos](#)

[Sobre o autor](#)

[Prefácio](#)

[Sobre o livro](#)

[Dispositivos móveis, desenvolvimento cross-platform e o Xamarin](#)

[1.1 Os dispositivos móveis na atualidade](#)

[1.2 O desenvolvimento móvel cross-platform](#)

[1.3 O Xamarin](#)

[1.4 O foco prático deste livro com o MVVM e o Entity Framework Core](#)

[1.5 Conclusão](#)

[Xamarin — Instalação e testes](#)

[2.1 Download e instalação](#)

[2.2 Teste da instalação realizada](#)

[2.3 Visualização das páginas diretamente no Visual Studio](#)

[2.4 Xamarin Live Player](#)

[2.5 Conclusão](#)

[Tipos de páginas, layouts e alguns controles para interação com o usuário](#)

[3.1 ContentPage e o StackLayout](#)

[3.2 TabbedPage e o Grid como layout](#)

[3.3 Navegação entre páginas, ListView e ContentView](#)

[3.4 Master Detail Pages e Hamburger Menu](#)

[3.5 Conclusão](#)

[O padrão Model-View-ViewModel e o Messaging Center do Xamarin](#)

[4.1 A página de serviços oferecidos](#)

[4.2 Alteração de um serviço](#)

[4.3 Inserção de um novo serviço](#)

[4.4 Remoção de um serviço da coleção](#)

[4.5 Conclusão](#)

[Execução no dispositivo físico, SQLite e Entity Framework Core](#)

[5.1 Publicação da aplicação para um dispositivo iOS](#)

[5.2 Publicação da aplicação para um dispositivo Android](#)

[5.3 Persistência de dados de maneira física com SQLite](#)

[5.4 Dependency Service para utilizar a base de dados](#)

[5.5 A classe de modelo](#)

[5.6 A classe de contexto](#)

[5.7 A classe de manipulação de dados](#)

[5.8 A visão que lista todos os clientes](#)

[5.9 Implementação do Hamburger Menu](#)

[5.10 A página responsável pela inserção e alteração de um cliente](#)

[5.11 Remoção de cliente já cadastrado](#)

[5.12 Interação e manutenção dos dados de Serviço](#)

[5.13 Atualização da base de dados EF Core com Migrations](#)

[5.14 Manipulação da base de dados do SQLite](#)

[5.15 Conclusão](#)

[Associações, Pesquisa, DatePicker, TimePicker e ActionSheet](#)

[6.1 Classe de modelo e de acesso a dados](#)

[6.2 A listagem de atendimentos](#)

[6.3 Pesquisa por clientes para atendimento](#)

[6.4 Véículo, datas e horas de entrada, previsão e entrega](#)

[6.5 Botão e Command para realizar a gravação do atendimento](#)

[6.6 Operações com atendimentos registrados](#)

[6.7 Implementações para a consulta e alteração de um atendimento](#)

[6.8 Destaque de um atendimento finalizado pelo uso de Converters](#)

[6.9 O método para recuperar um único objeto](#)

[6.10 Conclusão](#)

#### [Associações com coleções](#)

[7.1 Classe de modelo e de acesso a dados](#)

[7.2 Acesso às visões para itens de serviço e fotos do veículo](#)

[7.3 A listagem e remoção de serviços registrados para o atendimento](#)

[7.4 A pesquisa de serviços para o atendimento](#)

[7.5 Registro dos serviços a serem realizados](#)

[7.6 Conclusão](#)

#### [Uso de câmera e álbum](#)

[8.1 Registro das fotos do veículo](#)

[8.2 Visão para registro das fotos](#)

[8.3 O acesso à câmera](#)

[8.4 Tirando foto com o Android](#)

[8.5 Problemas no caminho para a imagem gravada](#)

[8.6 O acesso ao álbum de fotos](#)

[8.7 Gravação do caminho da foto na base de dados](#)

[8.8 Remoção de objetos de associações](#)

[8.9 Conclusão](#)

#### [Listagem de fotos e manipulação de gestos](#)

[9.1 Listagem das fotos inseridas](#)

[9.2 O uso de gestos para definir funcionalidades](#)

[9.3 Operação de zoom na imagem selecionada](#)

[9.4 Remoção de uma foto da listagem](#)

[9.5 Remoção de objetos de associações](#)

[9.6 Conclusão](#)

#### [Custom renderers, login de acesso e consumo de serviços REST](#)

[10.1 A visão de Login](#)

[10.2 Renderizadores personalizados \(Custom Renderers\)](#)

[10.3 O login com o serviço REST](#)

[10.4 Xamarin Essentials](#)

[10.5 Conclusão](#)

#### [O CRUD de peças com o consumo de serviços REST](#)

[11.1 A classe Peca, com arquivo de imagem, suas visões e DAL para o REST](#)

[11.2 Envio de um objeto com uma imagem para o serviço REST](#)

[11.3 Recebendo as peças registradas no servidor](#)

[11.4 Remoção de uma peça no servidor](#)

[11.5 Recuperação de uma peça específica no servidor](#)

[11.6 Envio de uma peça sem imagem para o servidor](#)

[11.7 Um serviço REST que retorna uma imagem](#)

[11.8 Inserção e alteração para a nova estratégia](#)

[11.9 Peças que não forem sincronizadas automaticamente](#)

[11.10 Conclusão](#)

#### [Os estudos não param por aqui](#)

#### [Apêndice - Criação de serviços REST](#)

[13.1 Preparação do ambiente para o desenvolvimento REST](#)

[13.2 Criação do projeto Web no STS](#)

[13.3 Teste e deploy da aplicação para o Heroku](#)

[13.4 O serviço REST que receberá o objeto e imagem para persistência](#)

[13.5 Envio de um objeto com uma imagem para o serviço REST](#)

[13.6 Obtenção de dados remotos por meio de um serviço REST](#)

[13.7 Remoção de uma peça no servidor](#)

[13.8 Recuperação de uma peça específica no servidor](#)

[13.9 Envio de uma peça sem imagem para o servidor](#)

[13.10 Um serviço REST que retorna uma imagem](#)

### 13.11 Conclusão