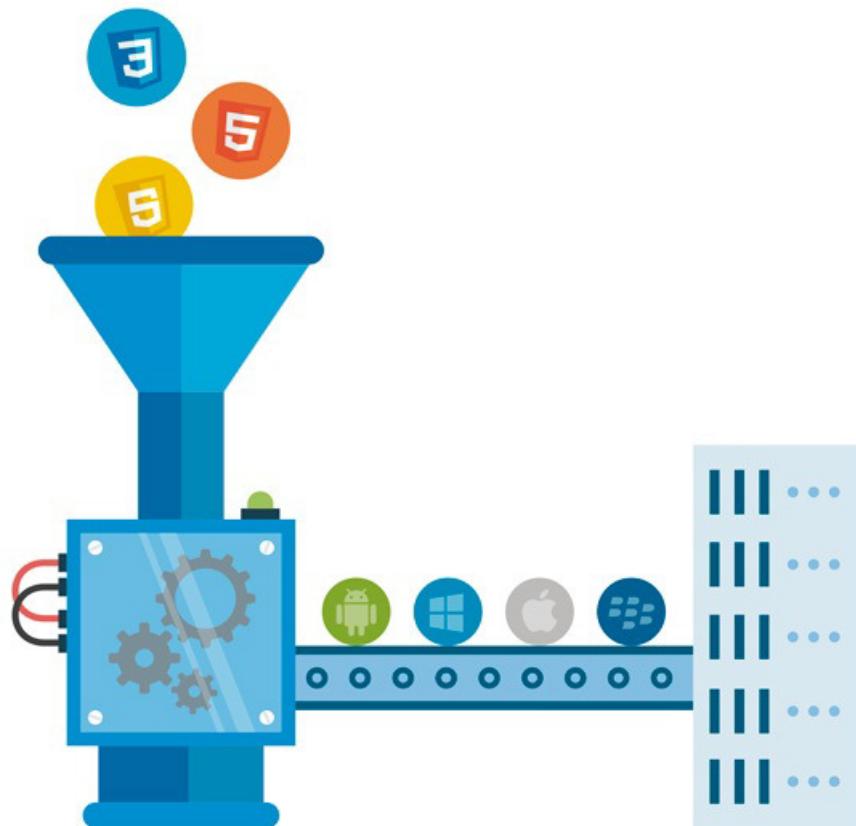


# Ionic Framework

Construa aplicativos para todas as  
plataformas mobile



Casa do  
Código

ADRIAN GOIS

© Casa do Código

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

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

*Edição*

Adriano Almeida

Vivian Matsui

*Revisão*

Bianca Hubert

Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

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

[www.casadocodigo.com.br](http://www.casadocodigo.com.br)



## **SOBRE O GRUPO CAELUM**

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura ([www.alura.com.br](http://www.alura.com.br)), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum ([www.caelum.com.br](http://www.caelum.com.br)), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

# ISBN

Impresso e PDF: 978-85-5519-288-3

EPUB: 978-85-5519-289-0

MOBI: 978-85-5519-290-6

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

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

## AGRADECIMENTOS

A realização deste projeto tornou-se possível e real com o apoio e confiança de algumas pessoas. Faço questão de deixar aqui o meu agradecimento a elas.

Em primeiro lugar, quero agradecer ao Adriano Almeida pela oportunidade e confiança na minha obra. À Vivian Matsui que foi fundamental para aprender e continuar aprendendo os macetes de uma boa escrita.

Agradeço à minha família pela paciência e compreensão incondicional, principalmente à minha filha Letícia por ter "cedido" o tempo em que poderíamos estarmos juntos, como também seu colo para descanso e relaxamento em momentos de estresse.

Este parágrafo não é um simples agradecimento, mas um intenso agradecimento e dedicação à memória de meus pais, principalmente minha mãe, falecida no mesmo ano de publicação, por ter deixado o legado de uma boa educação e ter me apoiado e incentivado a estudar TI para seguir a profissão onde me sinto realizado e feliz.

# O AUTOR



Figura 1: Adrian Gois

Em 1995 recebi de presente de minha querida e saudosa mãe meu primeiro computador com o SO Windows 95: um ITAUTEC equipado com um processador *Pentium* que fez me apaixonar à primeira vista pela tecnologia. A partir daí, comecei a ingressar no mundo da internet desde já, pesquisando como programar aquela máquina.

Fiz assim meus primeiros programas na linguagem **Clipper** e, logo em seguida, comecei a estudar *HTML* e *JavaScript*. Depois disto, nunca mais deixei de lado o fascínio por computadores. Assim me tornei bacharel em Ciência da Computação pela UNIFACS/Salvador-BA, fundei a empresa ABG TI SOLUTIONS (<http://www.abgsolucoes.com.br>), onde obtive experiência em Desenvolvimento de Sistemas em grandes empresas multinacionais, passando por linguagens como *Java*, *JavaScript*, *C*, *C++*, *Delphi*, entre outras que permeiam o mundo do desenvolvimento.

Hoje lidero uma equipe de TI em uma fábrica de *software* com arquiteturas diversas, passando por .NET até *Microsoft SharePoint*, com contratos em empresas de grande e médio porte. Desenvolvo projetos de aplicativos híbridos, utilizando a tecnologia deste livro.

# PREFÁCIO

Já parou para imaginar, enquanto você lê esta pequena frase, quantos *smartphones* estão sendo utilizados no mundo? Pois bem! Segundo a *TeleGeography*, em 2015 tínhamos 7,1 bilhões de chips ativos no mundo, ou seja, quase a quantidade de habitantes do globo terrestre. Com isso, podemos inferir sobre o grande potencial de mercado que é o de *smartphones* e telefonia.

Com a invenção dos *smartphones*, surgiram tanto grandes como descartáveis ideias de aplicativos. No início, tudo era muito restrito ao desenvolvimento destas ideias em uma plataforma e, a partir de seu sucesso ou não, a migração do código para outras plataformas, principalmente na dupla *iOS versus Android*.

Por conta disto, surgiram os *frameworks* para compilação híbrida de aplicativos sendo desenvolvidos em uma única linguagem. Hoje, sabe-se que isso é possível com a mesclagem de *HTML, JavaScript* e *CSS*, basicamente.

Dentro destes *frameworks*, o foco desta obra é *Ionic framework*. Nele é possível utilizar componentes responsivos e atrativos para o desenvolvimento dos aplicativos, com a possibilidade de compilação e fácil instalação nas mais diversas plataformas – seja *Windows Phone, Android, iOS, BlackBerry* etc.

O livro é dividido em 10 capítulos, nos quais inicialmente é explanado sobre a instalação e preparação do ambiente de desenvolvimento e, em seguida, um capítulo dedicado a explicar os comandos básicos do *framework*. Posteriormente, vamos evoluindo na criação e entendimento dos artefatos do aplicativo e,

a cada novo capítulo, temos novas inserções de componentes.

Ao final, dedico dois capítulos a recursos extras, como utilização da câmera do dispositivo, consumo de serviços do *firebase* do Google, entre outros recursos avançados.

Esta leitura levará você em uma viagem ao mundo do desenvolvimento de aplicativos, sem se preocupar com a plataforma nativa à qual ele será destinado, tornando as coisas mais fáceis para manutenção e evolução.

## Público-alvo

Este livro é direcionado aos desenvolvedores que já tenham, pelo menos, uma base em *JavaScript*, *HTML5* e *CSS*. No decorrer do livro, mesmo o leitor não tendo experiência com as tecnologias que permeiam o *Ionic*, como *AngularJS* e *Cordova/Phonegap*, pretendo fazê-lo alcançar um nível de conhecimento básico sobre todas elas.

É aconselhável também uma base de conhecimento em lógica de programação, para que se evolua no desenvolvimento de um *Caso de uso* que utilizaremos como exemplo para os capítulos. Sendo assim, se já programou nessa tríade (*HTML*, *JS* e *CSS*) e quer iniciar no mundo de aplicativos com o *Ionic Framework*, não se preocupe, você está no lugar certo.

## Código-fonte

Durante a leitura deste livro, desenvolveremos um aplicativo denominado Cardápio Móvel e seu código fonte pode ser clonado através do link: <https://github.com/adriangois/codigo-livro>

[ionicframework](#).

# Sumário

<b>1 Introdução</b>	<b>1</b>
1.1 Instalações	6
1.2 Conclusão	7
<b>2 Iniciando nossa aplicação</b>	<b>9</b>
2.1 Command Line Interface e seus comandos básicos	11
2.2 Criando os artefatos	13
2.3 Testando a aplicação	14
2.4 Estrutura de pastas	15
2.5 Adicionando a plataforma Android	16
2.6 Compilando e instalando o aplicativo	17
2.7 Metendo a mão na massa	20
2.8 Conclusão	25
2.9 Para saber mais	25
<b>3 Conhecendo os nossos arquivos</b>	<b>27</b>
3.1 Um pouco de MVC	27
3.2 Separando o joio do trigo	31
3.3 O cardápio e o MVC	34

3.4 Conclusão	39
3.5 Para saber mais	40
<b>4 Completando o menu</b>	<b>41</b>
4.1 Estados	41
4.2 Conclusão	45
4.3 Para saber mais	46
<b>5 Detalhando os itens</b>	<b>47</b>
5.1 O protótipo	47
5.2 Camada de serviços	49
5.3 Criando os arquivos	50
5.4 Ordem na casa	53
5.5 Criando a tela genérica	55
5.6 Passando um parâmetro	59
5.7 Recuperando o parâmetro	61
5.8 Criando o serviço de detalhamento	61
5.9 Detalhando o item	65
5.10 Conclusão	68
5.11 Para saber mais	68
<b>6 Fazendo pedidos</b>	<b>69</b>
6.1 Os requisitos do aplicativo	69
6.2 Uma forma de guardar nossa sessão	71
6.3 Hora de injetar o value	73
6.4 Guardando os pedidos	76
6.5 Conclusão	83
6.6 Para saber mais	84

Casa do Código	Sumário
<b>7 A bandeja</b>	<b>85</b>
7.1 Elementos do modal	85
7.2 Vamos ter retrabalho	96
7.3 Editar os itens da bandeja	99
7.4 A função de confirmar edição	103
7.5 Exclusão de itens	105
7.6 Confirmando os pedidos	106
7.7 Conclusão	108
7.8 Para saber mais	109
<b>8 A conta</b>	<b>110</b>
8.1 Construção do template	110
8.2 Vamos dividir a conta	120
8.3 Pagar ou pedir?	121
8.4 Conclusão	135
8.5 Para saber mais	136
<b>9 Usando a câmera</b>	<b>137</b>
9.1 Instalação do plugin	137
9.2 Construção de artefatos	138
9.3 A câmera	141
9.4 Conclusão	146
9.5 Para saber mais	146
<b>10 Recursos</b>	<b>147</b>
10.1 O Firebase	147
10.2 Consumindo serviços	148
10.3 O cardápio modificado	152

---

<a href="#">10.4 ionicList</a>	154
<a href="#">10.5 O ionicActionSheet</a>	157
<a href="#">10.6 Dialog do ngCordova</a>	159
<a href="#">10.7 Conclusão</a>	161
<a href="#">10.8 Para saber mais</a>	161

Versão: 20.9.30

## CAPÍTULO 1

# INTRODUÇÃO

Com o nascimento dos dispositivos inteligentes, surgiu uma grande quantidade de Sistemas Operacionais. Com isso, surgia também a dificuldade em padronizar uma linha de desenvolvimento de aplicações que fossem portáveis ou multiplataforma.

O Java ME, lançado em 1999, foi a plataforma criada da subdivisão do Java 2 (Java 1.2). Com o objetivo de integrar dispositivos limitados em termo de *hardware*, veio como promessa da já citada padronização, visto que todos os aplicativos rodariam em uma Máquina Virtual (*VM* – *Virtual Machine*), não importando em qual sistema/dispositivo estaria hospedado. Tudo isso provido pela portabilidade da tão crescente tecnologia Java.

Com o passar do tempo, em junho de 2007, a **Apple Inc** viria a anunciar o primeiro *SmartPhone* com seu iOS como Sistema Operacional Móvel. Consequentemente, o grupo Google criou o Sistema Operacional Android para dispositivos, baseado no núcleo do Linux. Além destes, outros nasceram no mesmo contexto, como por exemplo, Windows Phone e o Blackberry.

Com essa variedade de Sistemas Operacionais em uso no mundo, o leitor deve estar imaginando a dificuldade de agilizar o

processo de desenvolvimento e compilação dos aplicativos, a fim de torná-los portáveis para as distintas plataformas, sem que fosse necessário reprogramar na linguagem de programação nativa de cada dispositivo.

Uma luz no fim do túnel veio com o surgimento do HTML5, em que se criou uma segunda camada, que seria o intermediário entre a linguagem do sistema nativo (iOS, Android, Windows Phone, BlackBerry etc.) e a aplicação, construída basicamente em HTML5, CSS3 e JavaScript. Esta camada seria responsável pela compilação final do aplicativo, provendo à camada *view* (JavaScripts, CSS e HTML) os recursos da plataforma nativa (dispositivo para o qual estará sendo compilado).

Surgiu também o PhoneGap com o propósito de facilitar a vida de muitos desenvolvedores, disponibilizando acesso aos diversos *hardwares* (câmeras, GPS, acelerômetros) nos variados dispositivos móveis. Ou seja, tudo o que estaria disponível somente via linguagem nativa agora é possível acessar por meio de chamadas na linguagem JavaScript.

A imagem a seguir ilustra como funciona, na prática, o PhoneGap. Nele, a entrada é de um conjunto de artefatos na tríade já citada (HTML, CSS e JavaScript), passando por um *build* que gera no final o aplicativo na plataforma escolhida.

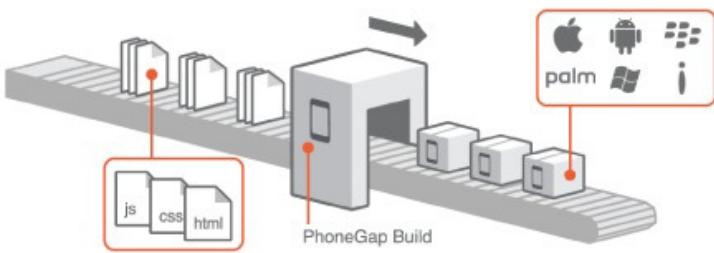


Figura 1.1: Processo de build do PhoneGap

Apesar de o PhoneGap não ser o principal foco do livro, à medida que formos implementando nosso caso de uso (um aplicativo de Cardápio Móvel que explicaremos mais à frente), nos familiarizaremos com ele utilizando alguns de seus *plugins*.

Dentro de todo este contexto e aprimorando mais o processo de desenvolvimento, em 2013 nasceu o *Ionic Framework*, mesclando outros frameworks já existentes, como o PhoneGap citado anteriormente e *AngularJS*. O Ionic é um *open source SDK* que usa um conceito chamado *native-feeling mobile apps*, o que quer dizer desenvolvimento de aplicativos móveis com tecnologias web como, HTML, CSS e *JavaScript*.

O foco do framework é o *front view* dos aplicativos, ou seja, ele fornece componentes para o desenvolvimento da interface dos aplicativos, não sendo um substituto do PhoneGap (que foca no acesso aos recursos), por exemplo. Na prática, o *Ionic* nos fornece uma gama de componentes para serem utilizados na *view*, fazendo o controle através do *AngularJS* e, por fim, é compilado através do *PhoneGap*. Esta é a junção de vários frameworks para um fim

específico.

No progresso de nossa leitura, vamos ver que, para estender a capacidade do aplicativo, usaremos o Cordova/Phonegap, que possibilita o acesso aos recursos do *hardware*, através da instalação de *plugins*. Com isto, nos preocupamos em construir, planejar e executar as tarefas por camadas, separando *views*, *controllers* e *models*. Isto é possível por causa da integração do Ionic com o *AngularJS*.

Veremos através dos capítulos a construção de uma aplicação de exemplo, a que daremos o nome de Cardápio Móvel, em que o leitor participará do planejamento (através de protótipos), construção, teste e implantação. Veremos também o aplicativo consumindo serviços, através de simulações de *webservices* e, no final, não mais uma simulação, mas o consumo de um serviço na nuvem através do *firebase* que será explicado no capítulo *Recursos*.

A aplicação será baseada na seguinte situação: construiremos um aplicativo que nos forneça uma lista de produtos, com descrição, foto e preço dele. Esta aplicação seria uma forma de o proprietário de um restaurante disponibilizar, através de um serviço hospedado na nuvem, os produtos para os seus cardápios.

Além disto, o aplicativo deve permitir que os usuários façam pedidos, colocando-os na bandeja, similar a um **carrinho de compras** muito utilizado nos *e-commerce*s. Ao confirmar a bandeja, o pedido será enviado para o setor responsável que atenderá o consumidor. O usuário também terá a possibilidade de pagar a conta parcial ou total, e/ou pedi-la que seja encerrada e entregue na mesa.

Nas interfaces da aplicação, serão apresentados alguns componentes CSS do framework, bem como sua extensão AngujarJS, que fornece uma rica biblioteca de JavaScript para que toda a mágica funcione.

A cada nova funcionalidade, apresentaremos um protótipo do que se deseja alcançar ao final da construção. Isto facilitará a compreensão da leitura do capítulo, e se consagrará quando pusermos em prática o conhecimento adquirido, chegando assim ao produto planejado.

O HTML5 chegou e, ao que parece, chegou para ficar como grande padronizador, sendo a linguagem de interfaces. Assim, ele deixa os recursos mais complexos e que estejam fora do contexto de UI (Interface de Usuário), para outras tecnologias e padrões. Vamos explorar este recurso ao máximo e nos familiarizar, pois, por ser o futuro (alguns já o consideram presente), vamos nos preparar agora.

Através dessa sopa de frameworks (*Ionic*, *AngularJS* e *Phonegap/Cordova*) é possível construir um aplicativo portável e funcional, não se preocupando com a sua linguagem de programação nativa, mas sim em desenvolver o visual e a lógica numa linguagem simples e universal na linguagem da web, o HTML.

Para entender melhor como surgiu a ideia de se mesclar tudo em um framework, criar aplicações rápidas e ter a facilidade de compilar para as variadas plataformas existentes, utilizando HTML5, JavaScript e CSS3, basicamente, vamos meter a mão na massa e instalar as ferramentas necessárias para iniciarmos nossa aplicação de exemplo.

Nos concentraremos em uma instalação ambiente Windows + Android. Isso não impede que o leitor possa trabalhar em outro ambiente e gerar a aplicações para iOS, Windows Phone ou qualquer outro de sua preferência, visto que este é exatamente o propósito do Ionic Framework.

## 1.1 INSTALAÇÕES

Para iniciarmos o desenvolvimento de nossa aplicação de exemplo, devemos dispor das seguintes ferramentas:

- Ionic Framework CLI (até a edição deste livro, a versão mais nova era a 1.3 ou Beta 2, mas vamos trabalhar com o 1.3);
- Java Development Kit - JDK (7 ou superior);
- Android SDK;
- NodeJS (até a edição deste livro, a versão mais nova era a 4.4.4);
- Sublime Text (até a edição deste livro, a versão mais nova era a 3).

Vamos lá?!

1. Comece baixando a JDK no site da Oracle, em <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>. Recomendamos que baixe a versão mais nova. Até a edição deste livro era a 8u102.
2. Tudo pronto para o Android SDK? Faça o download e

instale em <https://developer.android.com/studio/index.html>.

3. Agora vamos ao NodeJS, em <https://nodejs.org/en/download/>. Instale e vamos partir para nossas dependências.
4. Depois de instalado o NodeJS, abra a linha de comando do Windows, do Linux ou do Mac OS, e instale o Cordova e o Ionic, com o seguinte comando: `npm install -g cordova ionic` .

## 1.2 CONCLUSÃO

Se tudo ocorreu bem, já estamos preparados para criarmos nossa primeira aplicação. No próximo capítulo, aprenderemos mais sobre a linha de comando do *Ionic* e, consequentemente, sobre o comando para gerar uma aplicação que servirá de base em todo o nosso livro. Vamos à diversão!

### Para saber mais

1. Sobre as instalações do *JDK* e *Android SDK*, acesse <http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html>. E para o *Developer Android* em <https://developer.android.com>.
2. Você encontra um guia completo para instalação destas dependências em <http://cordova.apache.org/docs/en/5.1.1/guide/platforms/android/index.html>.
3. **AngularJS** – <https://angularjs.org>.

4. Phonegap – <http://phonegap.com>.
5. Ionic Framework – <http://ionicframework.com>.

## CAPÍTULO 2

# INICIANDO NOSSA APLICAÇÃO

Para fins didáticos, vamos conhecer os principais componentes do framework e aplicá-los na prática. Portanto, pensaremos em um caso de uso que vamos construir e evoluir ao longo dos capítulos. Ao final deste livro, teremos um aplicativo que nos fornecerá um cardápio de um determinado estabelecimento, com os seguintes requisitos:

1. Permitirá a listagem e o detalhamento de cada produto existente, assim como fazer o pedido determinando sua quantidade.
2. Fazer pedidos através desta listagem e permitir confirmar ou editar o pedido; tudo isto similar a um carrinho de compras de um *e-commerce*.
3. Pagar e/ou pedir a conta de forma parcial ou total.

Neste capítulo, conheceremos os comandos básicos do CLI (*Command Line Interface*), que é a linha de comando do framework. Começaremos executando o comando responsável pela criação de toda a estrutura do nosso caso de uso.

Posteriormente, executaremos a aplicação no *browser* para os

testes e, em seguida, adicionaremos uma plataforma (em nosso caso Android), fazendo o *build* e *deploy* em um dispositivo móvel real.

O protótipo inicial deste aplicativo está ilustrado na figura a seguir, na qual se deseja construir toda estrutura e navegação através do menu lateral.

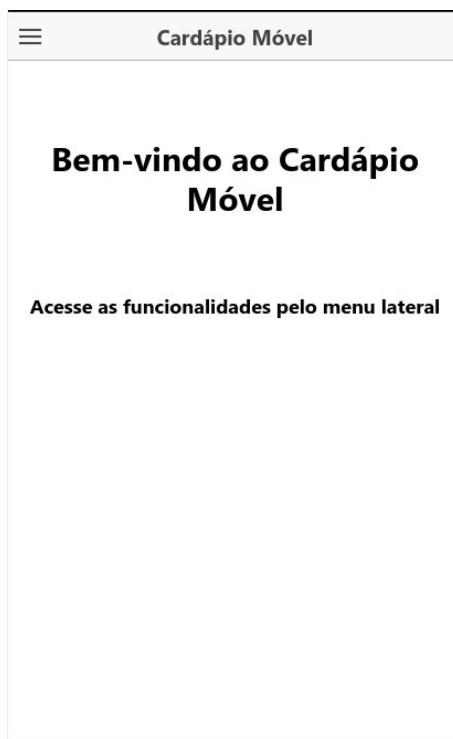


Figura 2.1: Protótipo deste capítulo – Construção da estrutura do Cardápio Móvel

Preparado para uma viagem sem volta? Vamos à prática do framework.

## 2.1 COMMAND LINE INTERFACE E SEUS COMANDOS BÁSICOS

O *Ionic CLI* é a ferramenta do framework que permite executar uma boa quantidade de comandos. Vamos nos concentrar em alguns deles, nos que usaremos com mais frequência. Ao decorrer do livro, aprenderemos mais sobre estes comandos.

O Ionic facilita o processo de criação inicial do desenvolvimento, passando pelos testes em browser, sem a necessidade de instalar o aplicativo em um dispositivo. Isso exceto em casos extremos, como a utilização de APIs para os recursos da plataforma em questão.

Para a construção dos códigos, usaremos o editor de texto *Sublime Text 3*, mas nada impede que o leitor utilize qualquer outra ferramenta de edição de sua preferência. Existe um leque de opções de editores que deixarei como dica na seção **Links úteis - Editores**, no final deste capítulo. Espero que o leitor se sinta à vontade para escolher o que mais lhe satisfaz. Essa é uma das vantagens do desenvolvimento híbrido, em que o desenvolvedor não fica atrelado a uma IDE de cada linguagem nativa.

Com o Ionic instalado, abriremos a linha de comando e digitaremos `ionic`. Isto nos dará uma lista de comandos que o framework suporta, como mostra a seguinte figura:

```
[|/|] [.-\|-] [|/] [C] CLI v1.7.14
Usage: ionic task args
=====
Available tasks: (use --help or -h for more info)
  start ..... Starts a new Ionic project in the specified PATH
  serve ..... Start a local development server for app dev/testing
  platform ..... Add platform target for building an Ionic app
  run ..... Run an Ionic project on a connected device
  emulate ..... Emulate an Ionic project on a simulator or emulator
  build ..... Locally build an Ionic project for a given platform
  plugin ..... Add a Cordova plugin
  resources ..... Automatically create icon and splash screen resources (beta)
```

Figura 2.2: Alguns comandos do CLI

Observe que a figura não ilustra todos os comandos, mas apenas os que achamos relevantes para iniciarmos os trabalhos. Mas não se preocupe, ao longo do livro, quando necessário, utilizaremos outros comandos, explicando detalhadamente.

O framework contém alguns *templates* já prontos, em que é gerado todo artefato apenas com um comando. Se, por exemplo, você desejar construir um aplicativo com *tabs* de navegação, poderia gerá-lo com o seguinte comando: `ionic start [nome_de_sua_aplicação] tabs` .

Veja a seguir três templates básicos que podem ser úteis em muitas aplicações, como também podem ser customizados pelo desenvolvedor.

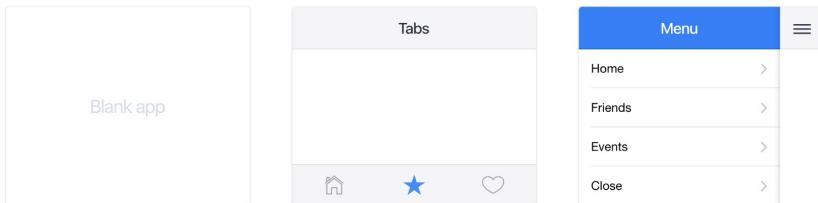


Figura 2.3: 3 templates prontos que são gerados pelo framework com o comando ionic start

Os comandos para geração desses templates são, respectivamente:

1. `ionic start [nome_aplicação] blank` : um template sem menu, sem rodapé ou cabeçalho;
2. `ionic start [nome_aplicação] tabs` : um template com rodapé e cabeçalho com um título;
3. `ionic start [nome_aplicação] sidemenu` : um template com um menu lateral e cabeçalho com um título.

Simples, não é?!

## 2.2 CRIANDO OS ARTEFATOS

Em nosso caso, para criarmos os artefatos da aplicação de exemplo deste livro, vamos executar o comando `ionic start cardapio-movel sidemenu`. Este comando cria uma pasta chamada `cardapio-movel` no diretório que você estiver executando.

O `sidemenu` no comando fará o framework construir um aplicativo com um menu lateral, conforme ilustrado no protótipo. Após a execução deste comando, navegue sobre a pasta criada para ir se familiarizando com os artefatos criados.

## 2.3 TESTANDO A APLICAÇÃO

Prontinho! Agora que foram gerados os artefatos, na mesma pasta execute o comando `ionic serve`. Com ele, sua aplicação já estará sendo executada em seu navegador padrão, no endereço `http://localhost:8100/`.

Teste e navegue sobre o menu. Brinque um pouco e conheça sua nova aplicação.

Caso queira executar em um navegador específico, que não seja o padrão, o comando é: `ionic serve -w [nome_do_navegador]`. Por exemplo, no Google Chrome, você pode utilizar `ionic serve -w chrome`. Sugiro que use este browser e abra-o em modo desenvolvedor (`Ctrl + Shift + i`), em seguida utilize o `Toggle device mode`, simulando um dispositivo, clicando no ícone a seguir:

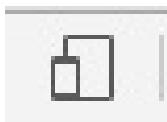


Figura 2.4: Estrutura de pastas

Com isto, você será capaz de ter uma visualização bem próxima de como o aplicativo ficará quando for instalado no dispositivo móvel.

Muito bem! Você já está com a estrutura da sua aplicação montada e testada. Agora vamos conhecer um pouco do que foi gerado por trás deste comando tão poderoso.

## 2.4 ESTRUTURA DE PASTAS

A estrutura de suas pastas deve ter sido gerada como ilustra a seguinte figura:

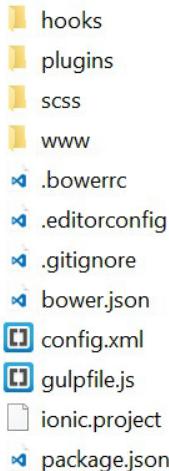


Figura 2.5: Estrutura de pastas

Por enquanto, vamos nos concentrar em dois arquivos principais que estão no diretório raiz: `package.json` e `config.xml`. O `package.json` é o arquivo *npm* (<https://npmjs.org>), onde são configuradas as dependências do seu projeto, suas versões, entre outras informações referentes a elas.

O `config.xml` diz respeito às configurações do aplicativo, bem como orientação de tela (horizontal, vertical ou ambas), nome do aplicativo, resources, versão e permissões. Este arquivo é a base para o `AndroidManifest.xml`, no caso de o build ser feito em Android, por exemplo.

Não vamos mexer nesses arquivos por enquanto. Mas é de

suma importância o conhecimento da existência deles, assim como também é importante entender a estrutura das pastas geradas:

1. `www` : contém os templates em HTML, arquivos JavaScript, CSS e imagens, ou seja, praticamente onde você vai trabalhar.
2. `plugins` : contém os plugins que foram gerados e serão configurados ao longo do desenvolvimento.
3. `scss` : uma sintaxe para o CSS, para quem quer implementar nessa estrutura. Vamos trabalhar apenas com CSS.
4. `hooks` : não nos aprofundaremos no entendimento deste diretório, pois não vamos usá-lo. Basta apenas saber que ele é gerado para customizações de comandos do *Cordova*.

Com nossa estrutura montada e aplicação rodando, vamos adicionar ao projeto uma plataforma. Como havia dito no início deste livro, o nosso *build* será para a plataforma Android, portanto, a partir de agora, todos os comandos serão executados em ambiente Windows e a geração do aplicativo será baseada no Android.

Isto não impede que o leitor opte por outra plataforma, caso sinta-se à vontade. Faça à sua maneira.

## 2.5 ADICIONANDO A PLATAFORMA ANDROID

Depois de construída a aplicação, é preciso adicionar a plataforma que se deseja compilar. Como executaremos o aplicativo na plataforma Android, vamos aprender a gerar um

.apk , e em seguida fazer um deploy em algum dispositivo desta plataforma.

Para isto, devemos rodar o seguinte comando: `ionic platform add android` . Rode e observe o que acontece. A seguir, explicaremos com mais detalhes.

## Nova estrutura de pastas

Se tudo correr bem, sua estrutura ganhou uma nova pasta (`platforms/android`) . Caso você tenha executado o comando `ionic platform add ios` , da mesma forma o framework criará a plataforma iOS (`platforms/ios`) .

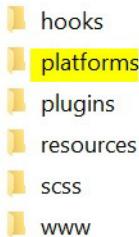


Figura 2.6: Pasta adicionada para a plataforma Android

À medida que formos avançando, teremos mais conhecimentos sobre esta estrutura e seus respectivos arquivos. Explicaremos cada um deles de forma clara e técnica.

## 2.6 COMPILANDO E INSTALANDO O APPLICATIVO

Agora que criamos e adicionamos uma plataforma ao aplicativo, vamos testá-lo em uma plataforma móvel. O Ionic,

como já foi citado neste livro, é totalmente compatível com o Phonegap/Cordova. No processo de compilar para nossa plataforma Android, o Phonegap/Cordova gera toda a estrutura de classes em *Java*, configurações ( *AndroidManifest.xml* ), ícones, e todos os recursos necessários para que nossa aplicação execute como esperado.

Por padrão, os dispositivos com Android não estão habilitados para executar um ambiente de desenvolvimento. Entretanto, não há motivo para se preocupar, pois esta é uma tarefa muito fácil e vamos lhe instruir em relação a isto.

Executaremos em um celular Motorola G3, com a versão do Android 6.0, mas isto não impede de ser instalado e executado em outro dispositivo. Esta informação é apenas para fins didáticos, já que seguiremos alguns comandos para preparação do ambiente no dispositivo. O Android segue sempre este padrão de desbloqueio de ambiente desenvolvedor.

Para isto, é necessário seguir os passos:

1. Abra a configuração do sistema em **Configurar** ;
2. Vá até o último item **Sobre o telefone** e clique;
3. Desça até **Número da versão** e clique 3 vezes, até aparecer a seguinte mensagem: *Faltam 4 etapas para você se tornar um desenvolvedor*. Continue clicando até aparecer: *Agora você é um desenvolvedor*.

Pronto! Agora você já deve visualizar a opção Configurar > Programador no seu dispositivo. Conecte um cabo USB no computador e em seu dispositivo, e vamos configurar algumas opções de desenvolvimento.

1. Se a opção Programador estiver como Desativado , passe para Ativado ;
2. Selecione Permanecer ativa para que a sua tela não bloquee enquanto estiver com o cabo USB plugado;
3. Mais abaixo, encontre a opção Depuração USB e deixe habilitada. Se aparecer a mensagem pedindo permissão de depuração, clique em OK .

A seguir, uma ilustração desta configuração:

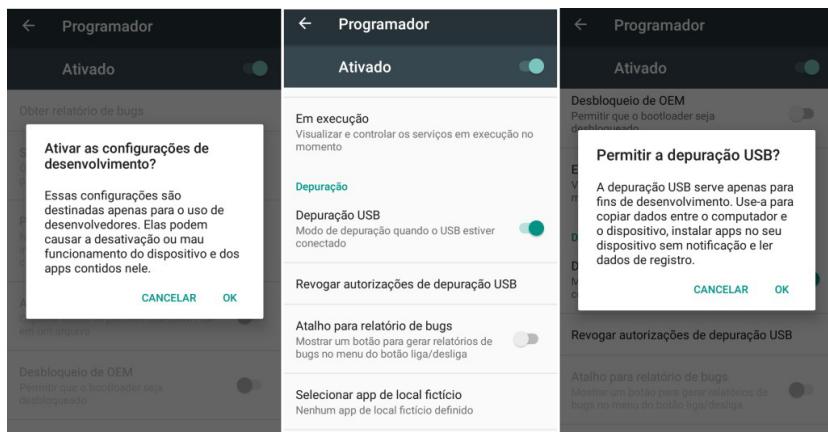


Figura 2.7: Configuração de desenvolvedor

Agora a festa vai começar. Você só precisa executar um comando para que seu aplicativo seja compilado e instalado. Abra a linha de comando e execute `ionic run` .

Durante a execução deste comando, pode aparecer uma confirmação de depuração USB com a seguinte mensagem: *A impressão digital da chave RSA deste computador é: [aqui a chave RSA]*. Selecione a opção Sempre permitir a partir deste computador e clique em OK .

Se tudo correr bem, você terá ao final o aplicativo rodando em seu dispositivo como ilustrado na figura a seguir. Teste, brinque e avalie.

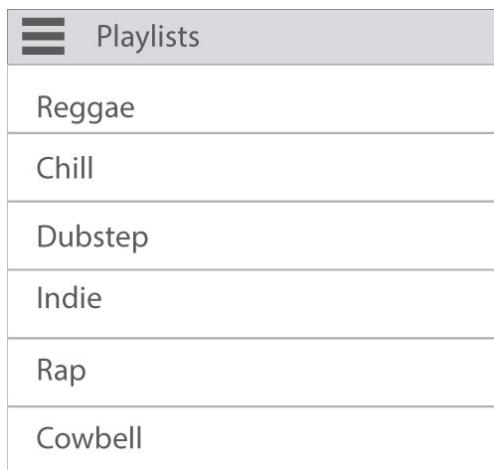


Figura 2.8: Layout inicial no dispositivo

Vamos deixá-lo parecido com nosso protótipo?

## 2.7 METENDO A MÃO NA MASSA

Agora vamos conhecer um pouco nosso arquivo `app.js` que

fica em `/cardapio-movel/www/js`. Ele é responsável pela criação, registro e recuperação de módulos do AngularJS.

Não se preocupe em entender a fundo todos os conceitos deste arquivo neste momento, pois, à medida que formos avançando em nosso caso de uso, aprofundaremos cada vez mais. O que o leitor deve entender neste momento é que neste arquivo estarão as rotas de nossa aplicação, configurações para estas rotas e inicialização da aplicação.

Como nosso intuito é "enxugar" este layout inicial, criado pelo próprio framework, para que fique bem próximo ao nosso protótipo, vamos direto ao ponto.

Toda aplicação tem um ponto de partida, ou seja, no `start` o app será direcionado para onde o desenvolvedor configurou como tela de início. No nosso caso, está sendo uma lista de músicas, a `/templates/playlists.html`. Isto foi gerado na configuração do módulo do Ionic, chamado `$urlRouterProvider` que faz a chamada à função `otherwise([rota])`, direcionando para esta interface como padrão. Veja a seguir a linha que vamos modificar:

```
$urlRouterProvider.otherwise('/app/playlists');
```

Esta linha está simplesmente informando ao framework que sua página inicial é a `/app/playlist`. O leitor pode estranhar a forma de referenciar a página aqui. Na verdade, o `$urlRouterProvider` não referencia a página diretamente, mas sim um `$stateProvider` pré-configurado.

Preste atenção no trecho do código a seguir, que se encontra no `app.js`. Observe que existem dois estados (`state`) configurando `url`, `menu`, `template` e `controller`. O nome

destes estados são `app/browse` e `app.playlist`, que foram passados como primeiro parâmetro em sua configuração.

```
$stateProvider
...
.state('app/browse',
{
    url: '/browse',
    views: {
        'menuContent': {
            templateUrl: 'templates/browse.html'
        }
    }
})
.state('app/playlists',
{
    url: '/playlists',
    views: {
        'menuContent': {
            templateUrl: 'templates/playlists.html',
            controller: 'PlaylistsCtrl'
        }
    }
})
...
...
```

Agora ficou fácil, não é?! Vamos modificar no `$urlRouterProvider` para que inicie nossa aplicação, chamando o estado `app/browse` em vez de `app.playlists`. Para isto, deixe a linha do `$urlRouterProvider` da seguinte forma:

```
$urlRouterProvider.otherwise('/app/browse');
```

Observe que o nome do estado é `app/browse`, mas a forma de passar esse parâmetro ao `$urlRouterProvider.otherwise` é em formato *URL*. Portanto, substituímos o `.` (ponto) por uma `/` (barra). A partir de agora, assimile esta regrinha básica para chamadas dos estados. Vamos precisar dela no decorrer do desenvolvimento.

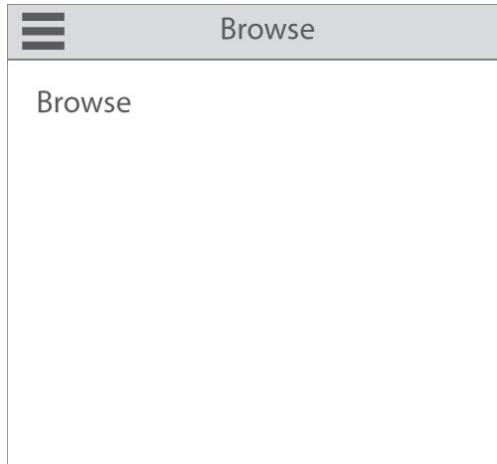


Figura 2.9: Direcionando para página inicial browse

Quase tudo parecido com nosso protótipo, conforme imagem anterior. Agora vamos apenas ajustar os textos para que tudo fique conforme planejamos. Afinal, você **não quer** desapontar o pessoal de design que lhe ofereceu o protótipo da tela, não é mesmo?!

Primeiro, construiremos dois estilos no nosso arquivo CSS que se encontra em `/www/css/style.css`, com a finalidade de centralizar os textos que constituem a primeira tela. Abra este arquivo e vamos editá-lo da seguinte forma:

```
.txt-centralizado{  
    text-align: center;  
}  
.txt-espaco{  
    margin: 70px 0 70px 0 !important;  
}
```

Agora é o momento de modificar o arquivo `/www/templates/browse.html`, alterando o título para Cardápio Móvel e inserindo nossos dois labels ( Bem-vindo ao

Cardápio Móvel e Acesse as funcionalidades pelo menu lateral ).

O título deste estado deve ser alterado diretamente na tag do framework `ion-view` . Depois removeremos a tag `<h1>Browse</h1>` e substituímos por duas com as classes do arquivo de estilo que construímos anteriormente. Nossa código no arquivo ficará assim:

```
<ion-view view-title="Cardápio Móvel">
  <ion-content>
    <h2 class="txt-centralizado txt-espaco">Bem-vindo ao Cardápio Móvel</h2>
    <h4 class="txt-centralizado">Acesse as funcionalidades pelo menu lateral</h4>
  </ion-content>
</ion-view>
```

Quase tudo pronto! Só precisamos centralizar o título. Para isto, deveremos alterar uma configuração do framework através de uma função chamada da API `$ionicConfigProvider` . Após o carregamento da aplicação, o framework passa pelo ciclo de configurações por esta função: `angular.module('starter', ['ionic', 'starter.controllers']).config(function($stateProvider, $urlRouterProvider))` .

Injetaremos a nossa API nesta função e informaremos a posição do título. Feito isto, nosso código ficará como a seguir:

```
...
.config(function($stateProvider, $urlRouterProvider,$ionicConfigProvider) {
  $ionicConfigProvider.navBar.alignTitle('center');
...

```

Sugiro que vá se familiarizando com o termo *injetar*, pois será bastante usado no decorrer do livro, já que é a base da utilização dos módulos do AngularJS em nosso framework. *Injetar* refere-se à injeção de dependências feita pelo framework por meio da passagem de parâmetro, como está sendo feito com o `$ionicConfigProvider` no código anterior.

## 2.8 CONCLUSÃO

Neste capítulo, aprendemos alguns comandos básicos do framework que são essenciais para a construção de uma aplicação básica, bem como adição de plataformas, compilação e empacotamento para posterior instalação em seus respectivos dispositivos. Aprendemos também a configurar alguns módulos e, com isto, direcionar a aplicação para um estado/template inicial e posicionar o título dos templates através da API do módulo `$ionicConfigProvider`.

No próximo capítulo, aprenderemos mais sobre os arquivos gerados, principalmente sobre os *controllers* no arquivo `www/js/controllers.js`. Organizaremos estes *controllers* para que a aplicação fique mais modularizada e, consequentemente, facilite eventuais manutenções.

## 2.9 PARA SABER MAIS

- |        |                              |   |
|--------|------------------------------|---|
| 1. API | <b>\$ionicConfigProvider</b> | - |
|--------|------------------------------|---|

- [http://ionicframework.com/docs/api/provider/\\$ionicConfigProvider/](http://ionicframework.com/docs/api/provider/$ionicConfigProvider/).
2. **AngularUI Router** – <https://github.com/angular-ui/ui-router>.

## Links úteis – Editores

1. **Sublime Text** – <https://www.sublimetext.com>
2. **Atom** – <https://atom.io>
3. **Brackets** – <http://brackets.io>
4. **Visual Code Studio** – <https://code.visualstudio.com>

## CAPÍTULO 3

# CONHECENDO OS NOSSOS ARQUIVOS

Este capítulo vai descrever um pouco sobre os arquivos que serão utilizados para a implementação das funcionalidades do nosso aplicativo.

Teremos um conhecimento mais abrangente do que são *controllers* e suas responsabilidades no *front-view* da aplicação, as rotas criadas para a transição entre as telas do aplicativo, a separação dos artefatos em pastas organizadas e instalações de novos *plugins*.

Ao final, o leitor estará mais apto para adicionar as funcionalidades do aplicativo, consumindo retornos de serviços (em nosso caso, será um JSON simulando serviços) e executando tarefas como transições entre telas, preenchimento de campos e validações de formulários, bem como alguns componentes próprios do *framework* para serem aplicados na marcação HTML.

### 3.1 UM POUCO DE MVC

Uma aplicação *web* é constituída basicamente de alguma linguagem de programação, seja no *back-end* ou *front-end*, e uma

representação desses dados. A representação, em seu estado final, sempre será um HTML padrão, interpretado por todos os browsers do mercado.

Com o surgimento dessas aplicações (*World Wide Web*) e a necessidade de se separar o código do negócio (Java, .Net, PHP, JavaScript) da representação visual (HTML), surgiu um padrão de arquitetura de software que separa a aplicação em 3 camadas (*Model, View e Controller*), o qual foi abreviado para MVC.

O AngularJS é um framework que segue este conceito (*Model-View-Controllers*, em português Modelo-Visão-Controles). Com ele, é possível manipular a DOM do HTML5 por meio dos *controllers*, utilizando um evento denominado *Data Binding*. Este é exatamente a forma de atualizar a Visão (View) toda vez que o modelo (Model) é alterado, e vice-versa. Esta interação é feita através do *controller* de cada aplicação.

Em resumo, o papel de cada camada em nosso framework será descrito a seguir:

1. **View ou Visão:** o HTML, ou seja, cada tela do aplicativo que será escrita nesta linguagem;
2. **Controller ou Controle:** *JavaScripts* responsáveis por receber solicitações, através de funções, e delegar estas chamadas a um serviço remoto ou executar alguma regra ou ação, como por exemplo, uma inserção no banco de dados local (SQLite);
3. **Model ou Modelo:** responsável por fazer a ponte entre o HTML e o *controller*, executando o *Data binding*, ou seja, a atualização constante entre estas duas camadas. O modelo em AngularJS é um objeto cuja referência é o `$scope`.

Não se preocupe, pois, na prática, vamos explanar melhor esta arquitetura através de nosso caso de uso, onde estes conceitos (MVC) serão aplicados.

A figura a seguir ilustra bem o que é o *Data Binding* do AngularJS:

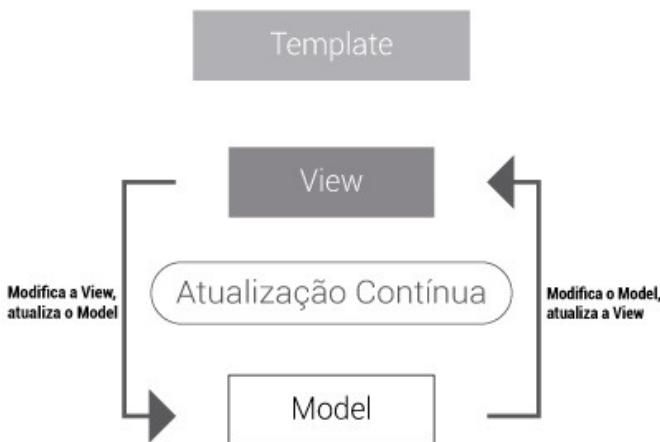


Figura 3.1: Data Binding do AngularJS

Vemos que a *view* é a camada logo abaixo do *template* (em nosso caso, um arquivo HTML). Na verdade, deve-se entender que a *view* é de fato o nosso arquivo HTML, que sofre atualizações contínuas do Modelo, fazendo também o caminho contrário, atualizando o modelo da aplicação através de formulários, inputs, ações de botões e tudo mais que seja capaz de enviar através do HTML.

A seguir, veja um exemplo de marcação HTML com AngularJS, em que se percebe um modelo chamado `yourName`. Este recebe no controller um valor e atualiza automaticamente na view em `<h1> Hello {{yourName}} </h1>`:

```
1.  <!doctype html>
2.  <html ng-app>
3.    <head>
4.      <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/
angular.min.js"></script>
5.    </head>
6.    <body>
7.      <div>
8.        <label>Name:</label>
9.        <input type="text" ng-model="yourName" placeholder="Enter a name
here">
10.       <hr>
11.       <h1>Hello {{yourName}}!</h1>
12.     </div>
13.   </body>
14. </html>
```

Figura 3.2: AngularJS

Na figura a seguir, o leitor tem uma visão do conceito de MVC aplicado ao AngularJS, onde as *views* (HTML, CSS) interagem com os usuários e se responsabilizam por atualizar o modelo ou o *controller*, através de chamadas *JavaScript*. Observe o AngularJS fazendo o meio dos caminhos entre o modelo e a *view*. Além disto, percebe-se também que a responsabilidade de uma chamada externa, um *webservice* por exemplo, é do *controller*.

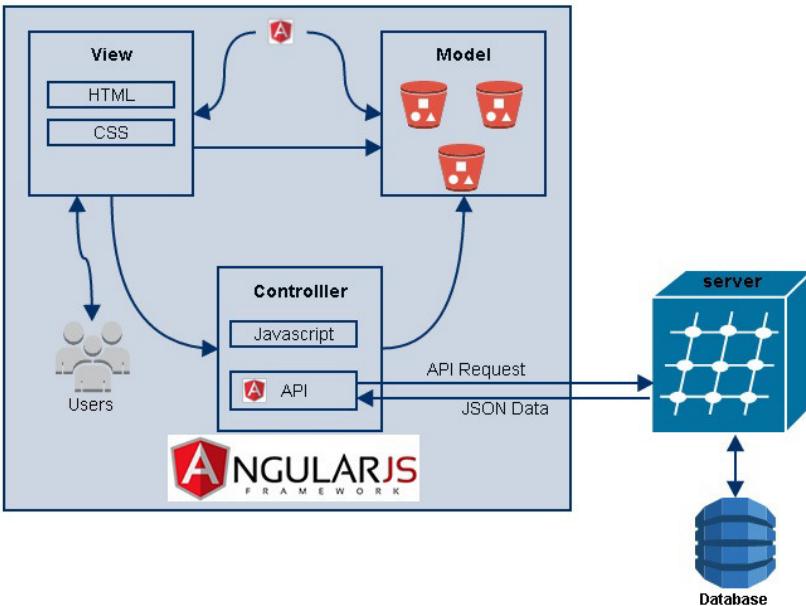


Figura 3.3: MVC

O *Ionic v1*, por padrão, cria apenas um *controller* por aplicação. Porém, isso não nos impede de separar em arquivos diferentes para um melhor entendimento e manutenção.

Nosso aplicativo possui, na pasta `www/js`, dois arquivos: `app.js` e `controllers.js`. Vamos focar no `controllers.js`, visto que já explicamos o `app.js` no capítulo anterior.

Vamos organizar as coisas?

## 3.2 SEPARANDO O JOIO DO TRIGO

Vamos criar um controller no nosso aplicativo para cada funcionalidade. Para isto, começaremos modificando o nome da

chamada de nossa primeira funcionalidade (Listar Produtos). Abra o arquivo `www/templates/menu.html` e localize a seguinte linha de código:

```
...
<ion-list>
  <ion-item menu-close ng-click="login()">
    Login
  </ion-item>
  <ion-item menu-close href="#/app/search">
    Search
  </ion-item>
  <ion-item menu-close href="#/app/browse">
    Browse
  </ion-item>
  <ion-item menu-close href="#/app/playlists">
    Playlists
  </ion-item>
</ion-list>
...
```

Observe que a transição entre as telas do aplicativo, pelo menu, são feitas conforme a marcação `<ion-item menu-close href="#">`. Em nosso aplicativo, o cardápio ficará dividido em 4 categorias (bebidas, petiscos, sucos e conta). Portanto, vamos construir todas as configurações deste menu para chamar cada tela referente à sua respectiva funcionalidade.

Desta maneira, nosso arquivo de menu ficará como a seguir:

```
...
<ion-item menu-close href="#/app/bebidas">
  Bebidas
</ion-item>
<ion-item menu-close href="#/app/petiscos">
  Petiscos
</ion-item>
<ion-item menu-close href="#/app/sucos">
  Sucos
</ion-item>
```

```
<ion-item menu-close href="#/app/conta">
    Conta
</ion-item>
```

Perceba que ainda temos um trabalho de tradução para fazer. O título do menu está como *left* e deveremos substituí-lo por "Menu". Para isto, abra o arquivo `www/templates/menu.html`, para editar a seguinte linha:

```
...
<ion-side-menu side="left">
    <ion-header-bar class="bar-stable">
        <h1 class="title">Left</h1>
    </ion-header-bar>
...

```

Ela ficará assim:

```
...
<ion-side-menu side="left">
    <ion-header-bar class="bar-stable">
        <h1 class="title">Menu</h1>
    </ion-header-bar>
...

```

Com estas modificações, temos um menu seguindo a nossa ideia de cardápio. Veja a seguir:

Menu	☰
Bebidas	
Petiscos	
Sucos	
Conta	
	Aces

Figura 3.4: MVC

Agora que temos um menu, vamos dar a ele mais vida, ou seja, torná-lo funcional. Para isto, teremos que configurá-lo para chamar nossas telas que conterão nossos produtos.

Eles poderão ser carregados através de um *webservice*. Porém, como nosso foco não será explanar sobre a criação destes serviços, faremos isto em forma de *mocks*, ou seja, objetos simulados de forma controlada, para que tenhamos uma ideia mais concreta dessas listas. Será uma simulação de um retorno dos *webservices* em formato de JSON.

*Webservices* são componentes que permitem que as aplicações, mesmo com tecnologias diferentes, se comuniquem através de uma rede por padrões predeterminados entre as duas partes: consumidor (aplicação que consome o serviço) e o provedor (aplicação que fornece o serviço).

Vamos começar a brincar?!

### 3.3 O CARDÁPIO E O MVC

Faremos agora todo o processo de chamada de uma tela do nosso aplicativo utilizando toda a técnica de MVC. Nesta seção, o leitor vai entender como tudo isto funciona de uma forma bem prática.

Nosso primeiro passo será criar um estado no nosso arquivo `www/js/app.js`, onde estará configurada a chamada para nossa

*view* `www/templates/bebidas.html` . Esta será controlada por nosso *controller* `www/js/bebidas-controller.js` e terá como modelo nosso objeto `$scope` .

Observe que usamos de propósito a sopa de letrinhas MVC para frisar que nosso framework é baseado neste modelo, como já foi citado em nossa introdução.

## A view

Para criarmos nossa *view*, que nada mais é do que um arquivo HTML, e representar nossa tela no aplicativo, deveremos construir a listagem de bebidas do nosso cardápio. Para isto, crie um arquivo `www/templates/bebidas.html` . Em seu conteúdo, coloque o seguinte código:

```
<ion-view title="Bebidas">
  <ion-content>
    <ion-list>
      <ion-item ng-repeat="bebida in bebidas" href="#/app/
bebida/{{bebida.id}}">
        {{bebida.nome}} - R$ {{bebida.preco}}
      </ion-item>
    </ion-list>
  </ion-content>
</ion-view>
```

Perceba que o HTML contém componentes ou marcações que são próprias do *framework* (Ionic), como `<ion-view` , `<ion-content>` , `<ion-list>` e `<ion-item>` . Estes componentes fazem toda a diferença quando renderizadas no *browser* dos dispositivos, construindo uma interface elegante.

Mais adiante, veremos em que formatos esta marcação ficará e explicaremos melhor cada um deles com seus respectivos atributos

e diretivas ( `ng-repeat` , `title` etc). Para isto, vamos criar nossa rota para esta *view* e nosso *controller* responsável por ela.

## O Controller

Nosso *controller* será o *JavaScript* que proverá a troca (*data binding*) de informação entre a camada *view* (HTML) e o *model* (modelo). Portanto, vamos criar o seguinte arquivo: `www/js/bebidas-controller.js`.

Com o arquivo salvo, construiremos um *controller* que será composto de uma codificação simples:

```
app.controller('BebidasCtrl', function($scope){  
    //Mock de bebidas  
    $scope.bebidas = [  
        {'nome': 'Vinho Tinto Brasil', 'id' : 01, 'preco' : '40,00'},  
        {'nome': 'Cerveja Especial', 'id' : 02, 'preco' : '23,00'},  
        {'nome': 'Whisky 18 anos', 'id' : 03, 'preco' : '200,00'}  
    ]  
});
```

O que temos nesse código é a definição do *controller* para ser referenciado pelo objeto `BebidasCtrl`. Observe isto no primeiro parâmetro passado para a função `app.controller()`. Ele será carregado quando for chamada a *view* correspondente e executará a construção de um array de **bebidas** com 3 objetos.

Observe que este array foi atribuído a um objeto referenciado por `$scope`. Preste atenção neste objeto e não se preocupe, na seção seguinte falaremos dele.

Note também o comentário acima do array de bebidas, que informa que isto é um mock ( `//Mock de bebidas` ), ou seja, um objeto que simulará uma chamada a um *web service*.

Objetos *mock* são simulações de objetos, criados para testar aplicações, quando não há a possibilidade de obtê-los de um ambiente real. Em nosso caso, como foge do contexto criar uma aplicação que proverá um *webservice* para consumirmos no aplicativo, simularemos todos os serviços através dos *mocks*.

## O model

Lembra de quando pedi para o leitor observar o objeto `$scope` na seção anterior? Agora vamos aprender mais sobre ele.

Esta referência está diretamente ligada ao *model* do MVC. O AngularJS usa-o justamente para fazer aquela **atualização contínua**, ilustrada na figura sobre o *data binding*. Através deste objeto, o HTML `www/templates/bebidas.html` é capaz de interagir e exibir a lista em:

```
<ion-item ng-repeat="bebida in bebidas" href="#/app/bebida/{{bebida.id}}">
    {{bebida.nome}} - R$ {{bebida.preco}}
</ion-item>
```

Note que, na *view*, não foi preciso referenciar o objeto (`$scope`), porém, no *controller*, isso é necessário. Agora vamos configurar a chamada para a tela desta funcionalidade.

## As rotas

Toda configuração de tela no nosso aplicativo é tratada como um estado (`state`). Para que nossa tela de bebidas seja

apresentada ao clicar no menu, deveremos criar uma rota em nosso arquivo `www/js/app.js`. A seguir, veja um exemplo de nosso *state* para o template que criamos no item anterior deste capítulo:

```
.state('app.bebidas' , {  
    url : '/bebidas',  
    views: {  
        'menuContent' : {  
            templateUrl : 'templates/bebidas.html',  
            controller : 'BebidasCtrl'  
        }  
    }  
})  
  
Crie este estado entre .state('app') e  
.state('app.search').
```

A ordem não afeta o funcionamento, mas para que façamos tudo no mesmo padrão, esta é apenas uma sugestão de organização.

Prontinho, vamos explicar nosso *state*. Simplesmente, dissemos que, ao invocar a URL `/app/bebidas` (URL no *state*), estamos dizendo que devemos carregar a *view* `templates/bebidas.html` e, juntamente com ela, o *controller* `www/js/bebidas-controller.js`.

Execute a aplicação com comando `ionic serve` em seu CLI, demonstrado no capítulo anterior. Então, clique no menu Bebidas e observe tudo acontecendo. A tela a seguir deve ser exibida em seu browser:



Figura 3.5: Tela Bebidas

Não foi fácil? Neste momento, você deve estar ansioso para instalar o aplicativo em um celular (ou qualquer outro dispositivo móvel), e observar como estão ficando os trabalhos até aqui. Pois bem! Vá em frente. Execute o comando já citado no capítulo anterior através do CLI, `ionic run`, e curta seu funcionamento.

## 3.4 CONCLUSÃO

Neste capítulo, foi explanado uma visão geral sobre MVC, *mocks* e toda a estrutura que o *Ionic Framework* cria para que seja possível a construção de aplicativos nesta arquitetura. Vimos que através de estados (`.state`) no arquivo de configuração `www/js/app.js`, é possível montar uma transição de tela, configurando o *controller* responsável por ela, como também carregar seu template ou *view* (HTML).

No próximo capítulo, concluiremos nosso menu, chamando todas as funcionalidades de nosso aplicativo. Também aprenderemos um pouco mais sobre os componentes do

*framework*. Veremos que existem formas muito simples de apresentar listas, botões, inputs e labels.

### 3.5 PARA SABER MAIS

1. **SQLite** – <http://ngcordova.com/docs/plugins/sqlite>;
2. **Mock** – [https://pt.wikipedia.org/wiki/Objeto\\_Mock](https://pt.wikipedia.org/wiki/Objeto_Mock);
3. **Data Binding** – <https://angularjs.org/>

## CAPÍTULO 4

# COMPLETANDO O MENU

Neste capítulo, concluiremos o menu principal para chamar as telas de listagem dos produtos contidos no cardápio. Todo conceito aprendido até aqui servirá como base para o melhor entendimento do framework, principalmente no que diz respeito às configurações dos *states* e construções dos *controllers* e *views*.

## 4.1 ESTADOS

Como vimos no capítulo anterior, toda "mágica" de transição de funcionalidades é feita através da configuração de um estado (`.state`), também chamado de rotas. Isso acontece no arquivo `www/js/app.js`, onde é necessário informar o *template*, *controller* e construir as *tags* que vão renderizar nossa listagem fornecida pelos *mocks*.

Se acaso tentarmos executar qualquer outra parte do cardápio – através do menu – que não seja "bebidas", o aplicativo não direcionará para as telas correspondentes pelo seguinte motivo óbvio: ainda não construímos os artefatos e configurações necessárias para que isto aconteça.

Por isso, não se desespere! Preparado para construir todas as chamadas do menu principal da aplicação? Então, vamos lá!

```
Abra o arquivo www/js/app.js e adicione um novo estado  
(petiscos) logo abaixo da nossa última configuração  
.state('app.bebidas'... :  
...  
.state('app.petiscos', {  
    url: '/petiscos',  
    views : {  
        'menuContent' : {  
            templateUrl : 'templates/petiscos.html',  
            controller : 'PetiscosCtrl'  
        }  
    }  
}  
...  
...
```

Criado o estado app.petiscos , se tentarmos abrir esta tela no aplicativo, teremos um erro 404 do protocolo HTTP, ou seja, não será encontrado o HTML referente e mapeado em templateUrl . A seguir, veja o erro que pode ser verificado no console do navegador:



```
✖ ▶ GET http://localhost:8100/templates/petiscos.html      ionic.bundle.js:25000  
404 (Not Found)
```

Figura 4.1: 'Erro 404'

Vamos então construir nossa tela de petiscos?

É muito simples, pois é bastante parecida com a de bebidas que criamos no capítulo anterior. Vamos fazer apenas as alterações necessárias.

Comece criando o seguinte arquivo www/templates/petiscos.html e, em seguida, faremos a seguinte codificação:

```

<ion-view title="Petiscos">
  <ion-content>
    <ion-list>
      <ion-item ng-repeat="petisco in petiscos" href="#/ap
p/petisco/{{petisco.id}}">
        {{petisco.nome}} - R$ {{petisco.preco}}
      </ion-item>
    </ion-list>
  </ion-content>
</ion-view>

```

Veja que houve poucas mudanças em relação ao arquivo `www/templates/bebidas.html`. Só modificamos o título da tela, o nome do array em `<ion-item ng-repeat...>`, e a referência para cada objeto deste array (`petisco`).

Tente abrir esta tela do aplicativo e observe a saída de log no console do seu navegador. Você vai verificar que algo está faltando. Consegue identificar qual arquivo deixou de ser carregado pelo aplicativo?

```

✖ ▶ Error: [ng:areq] Argument 'PetiscosCtrl' is not a function, got undefined ionic.bundle.js:26794
http://errors.angularjs.org/1.5.3/ng/areq?
p0=PetiscosCtrl&p1=not%20a%20function%2C%20got%20undefined
  at ionic.bundle.js:13438
  at assertArg (ionic.bundle.js:15214)
  at assertArgFn (ionic.bundle.js:15224)
  at $controller (ionic.bundle.js:23373)
  at Object.self.appendViewElement (ionic.bundle.js:59900)
  at Object.render (ionic.bundle.js:57893)
  at Object.init (ionic.bundle.js:57813)
  at Object.self.render (ionic.bundle.js:59759)
  at Object.self.register (ionic.bundle.js:59717)
  at updateView (ionic.bundle.js:65398)

```

Figura 4.2: Erro de controller

Simplesmente referenciamos o nosso *controller* no estado com a variável `PetiscosCtrl`, porém, não criamos e nem carregamos este arquivo no nosso aplicativo. Precisamos criá-lo e, em seguida, construir nosso *mock*, já discutido no capítulo anterior – ou seja,

uma simulação de um objeto que poderia ser o consumo de um serviço que estaria na nuvem, por exemplo.

O *mock* (objeto) de petiscos teria como atributos um nome, um identificador e o seu preço. Exibiremos o nome e preço na tela principal, em forma de *grid*, e usaremos o identificador para carregar os detalhes de cada petisco.

Com isto, vamos simular que, em um banco de dados, temos uma tabela, com os petiscos e seus atributos: id, nome, imagens, descrição e preço. Como estamos sempre falando em consumo de serviços, vamos imaginar um *webservice* no lugar desta tabela.

Inicialmente, vamos construir o *mock* simples e evoluirmos aos poucos para melhor entendimento. Construindo nosso *controller*, nosso arquivo `petiscos-controller.js` em `www/js/` ficará da seguinte forma:

```
app.controller('PetiscosCtrl', function($scope){  
  
    $scope.petiscos = [  
        {'nome': 'Filé ao molho madeira', 'id' : 01, 'preco' : '60,00'  
    },  
        {'nome': 'Camarão ao alho', 'id' : 02, 'preco' : '21,00'},  
        {'nome': 'Lagosta assada', 'id' : 03, 'preco' : '100,00'}  
    ]  
  
});
```

Mas isto basta? Não! Ainda temos de referenciar este *JavaScript* no nosso HTML principal, ou seja, no `index.html`. Faça a inclusão da seguinte *tag*:

```
...  
<script src="js/petiscos-controller.js"></script>  
...
```

Agora sim será possível carregar nossa tela de petiscos. Peça uma cerveja especial e, de preferência, para acompanhar um camarão ao alho, pois o preço está bastante atrativo. Bom apetite!

≡	Petiscos
	Filé ao molho madeira - R\$ 60,00
	Camarão ao alho - R\$ 21,00
	Lagosta assada - R\$ 100,00

Figura 4.3: Nova tela – Petiscos

Deixaremos para o leitor a responsabilidade de criar a tela de sucos do aplicativo. Siga o mesmo raciocínio e construa esta funcionalidade antes de prosseguir com a leitura. Isto reforçará o aprendizado e, quando perceber, já estará construindo estados, *views* e *controllers* de forma automática.

## 4.2 CONCLUSÃO

Até aqui, configuramos todo *menu* do aplicativo, exceto *Conta*. Este deixaremos para o final do livro.

No próximo capítulo, vamos permitir que o usuário navegue melhor no cardápio, clicando em cada item em que terá uma breve descrição e uma foto, além de disponibilizar um botão para efetuar o pedido. Como esta demanda, acrescentaremos novos atributos nos *mocks* para representar o detalhamento do item.

## 4.3 PARA SABER MAIS

1. **Controllers**                  em                  **AngularJS**                  -  
<https://docs.angularjs.org/guide/controller>

## CAPÍTULO 5

# DETALHANDO OS ITENS

Neste capítulo, vamos construir novas telas para o aplicativo, onde será possível visualizar detalhes sobre cada item do cardápio, como imagem, descrição, além de ser permitido efetuar pedidos. Para isto, vamos pôr em prática um conceito muito utilizado em desenvolvimento de *softwares* que é o reaproveitamento. Perceba que a tela de detalhes será igual para todos os itens, por isto, vamos planejar uma tela que seja reutilizada.

## 5.1 O PROTÓTIPO

Pensando em uma tela genérica para nosso aplicativo, chegaremos a algo parecido com o protótipo da figura a seguir:

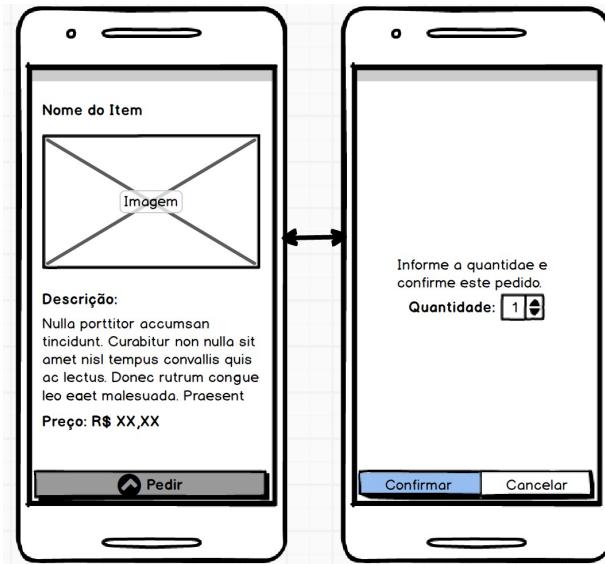


Figura 5.1: Protótipo

Observe que carregaremos de cada item um título, uma imagem, uma breve descrição e seu valor (preço). Caso o usuário opte por efetuar o pedido, o aplicativo direcionará para uma página (ou modal) de confirmação, onde ele poderá desistir, ou informar a quantidade e confirmar.

Quantas cervejas você pedirá com a turma do trabalho sexta-feira à noite? Acho que o *input* de quantidade veio a calhar neste momento, não é mesmo?!

Agora que temos uma ideia, vamos colocá-la em prática. Pense também sobre um serviço que seja reaproveitado para que não fiquemos duplicando os *mocks* em cada tela. A seguir, vamos aprender a organizar estes *mocks*, simulando melhor uma chamada a um serviço. Para isso, faremos uma outra camada na nossa

aplicação.

## 5.2 CAMADA DE SERVIÇOS

Já vimos como criar arquivos *JavaScript* que têm a função de *controllers*. Da mesma forma, o *Ionic* permite criar serviços (não confunda com *webservices*) que são *scripts* reutilizáveis dentro de uma aplicação. Isto elimina a replicação de código e centraliza funcionalidades em camadas, como demonstra a figura a seguir.

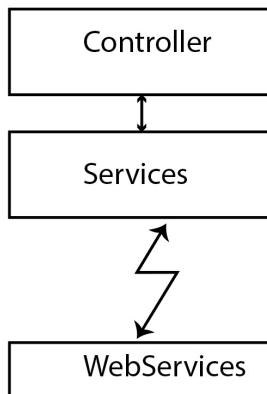


Figura 5.2: Camadas

Isto ilustra que os *controllers* passarão agora a fazer chamadas aos *services* que, por sua vez, consomem *webservices* fora da aplicação. Na prática, em nosso app, isto não ocorrerá, pois vamos apenas simular o consumo destes serviços através dos *mocks*.

Entretanto, se houvesse a necessidade de se implementar *webservices*, estes estariam em algum servidor na nuvem. E nossos *scripts* (serviços), dentro da aplicação, seriam os responsáveis por consumi-los.

## 5.3 CRIANDO OS ARQUIVOS

Se você fosse pensar na arquitetura da seção anterior, faria um arquivo (*service*) que centralizasse todas as chamadas aos *webservices*, ou vários arquivos, um para cada funcionalidade? Depende, não é mesmo? Mas, depende de quê? Do quanto complexa será a aplicação.

Em nosso caso, faremos apenas um arquivo, pois é um aplicativo simples, pequeno e de fácil manutenção. Mas, se por acaso você estivesse trabalhando em uma aplicação maior, em que o escopo fosse uma infinidade de funcionalidades, a melhor opção seria dividir em vários *services*, pois faria jus à máxima da Ciência da Computação: **dividir para conquistar**. Esta seria a melhor solução.

Portanto, vamos iniciar criando o arquivo da camada de *services*. Crie para isto a pasta `www/js/services`. Preste atenção que criamos uma pasta para organizar nossos *services*, porém teremos na aplicação apenas um arquivo. Sugeri no plural apenas com o propósito didático para demonstrar que tudo pode ser organizado por pastas. Na seção seguinte, vamos organizar da mesma forma todos os nossos *controllers*.

Agora vamos criar nosso arquivo cujo nome será `cardapio-services.js`, dentro de nossa pasta `www/js/services`, que ainda não será funcional, pois não vai dispor de nenhum código. Apenas vamos criá-lo para usar como piloto de nosso propósito. Ele ficará da seguinte forma:

```
app.service('CardapioServices', function(){  
});
```

Observe que estamos dando a este objeto a referência `CardapioServices`, que usaremos logo em seguida em nossos `controllers`.

Criado o arquivo, precisamos injetar. Lembra-se deste termo quando sugeri que se familiarizasse com ele, lá no capítulo *Iniciando nossa aplicação?* Pois é! Vamos agora injetar nosso `service` no `controller`, abrindo nosso arquivo `www/js/petiscos-controller.js` e inserindo a referência do `service` dentro da `function`. Ela ficará da seguinte forma:

```
app.controller('PetiscosCtrl', function($scope, CardapioServices)
{
    ...
}
```

Agora tente chamar a tela de petiscos no menu do aplicativo e observe o log no console do navegador. A seguir, veja o que deve ter acontecido com seu aplicativo:

```
✖ ► Error: [$injector:unpr] Unknown provider: CardapioServicesProvider <- PetiscosCtrl
http://errors.angularjs.org/1.5.3/$injector/unpr?
p0=CardapioServicesProvider%20%3C-%20CardapioServices%20%3C-%20PetiscosCtrl
    at http://localhost:8100/lib/ionic/js/ionic.bundle.js:13438:12
    at http://localhost:8100/lib/ionic/js/ionic.bundle.js:17788:19
    at Object.getService [as get]
( http://localhost:8100/lib/ionic/js/ionic.bundle.js:17941:39 )
    at http://localhost:8100/lib/ionic/js/ionic.bundle.js:17793:45
    at getService
( http://localhost:8100/lib/ionic/js/ionic.bundle.js:17941:39 )
    at injectionArgs
( http://localhost:8100/lib/ionic/js/ionic.bundle.js:17965:58 )
    at Object.instantiate
( http://localhost:8100/lib/ionic/js/ionic.bundle.js:18007:18 )
    at $controller
( http://localhost:8100/lib/ionic/js/ionic.bundle.js:23412:28 )
    at Object.self.appendViewElement
( http://localhost:8100/lib/ionic/js/ionic.bundle.js:59900:24 )
    at Object.render
( http://localhost:8100/lib/ionic/js/ionic.bundle.js:57893:41 ) <ion-nav-view
name="menuContent" class="view-container" nav-view-transition="ios">
```

Figura 5.3: Erro de injeção

O que o *Ionic* está nos informando é que você injetou

`CardapioServices` em `PetiscosCtrl`, ou seja, em nosso controller, porém ele ainda é desconhecido. Sabe o porquê disto? Simplesmente precisamos importar o arquivo *JavaScript* em nosso `index.html`. Vamos lá?!

Abra o arquivo principal `index.html` e insira abaixo da declaração do *service* de nosso cardápio.

```
<script src="js/services/cardapio-services.js"></script>
```

Prontinho! Agora o erro não ocorrerá mais, porém, ainda precisamos simular os *webservices* e invocá-los nos *controllers*. Para isto, crie todos os *mocks* que usaremos em `www/js/services/cardapio-services.js`. Lembre-se de que todos os nossos itens têm como atributos o nome, um identificador (*id*) e seu preço. Para isto, criamos os *mocks* em formato de *array*, como no código a seguir:

```
app.service('CardapioServices', function(){
    //Mock Bebidas
    var bebidas = [
        {'nome':'Vinho Tinto Brasil','id':01,'preco':'40,00'},
        {'nome':'Cerveja Especial','id':02,'preco':'23,00'},
        {'nome':'Wisky 18 anos','id':03,'preco':'200,00'}
    ];
    //Mock Petiscos
    var petiscos = [
        {'nome':'Filé ao molho madeira','id':01,'preco':'60,00'},
        {'nome':'Camarão ao alho','id':02,'preco':'21,00'},
        {'nome':'Lagosta assada','id':03,'preco':'100,00'}
    ];
    //Mock Sucos
    var sucos = [
        {'nome':'Cajá','id':01,'preco':'3,00'},
        {'nome':'Açaí','id':02,'preco':'4,00'},
        {'nome':'Cacau','id':03,'preco':'5,00'}
    ];
});
```

```
this.getPetiscos = function(){
    return petiscos;
}

this.getBebidas = function(){
    return bebidas;
}

this.getSucos = function(){
    return sucos;
}

});

});
```

Observe que transferimos todos os *mocks* para nosso *service* e criamos chamadas às suas funções pelo modificador `this`. Desta forma, não precisamos mais destes *mocks* nos controllers, ficando assim mais próximo de uma aplicação real, na qual invocaremos sempre os *webservices* através desta última camada.

Para que tudo fique em seu lugar, vamos modificar o *controller* `petiscos-controller.js`, removendo o *mock* de petiscos e chamando-o por meio do *service*. Observe como ficará nosso arquivo final:

```
app.controller('PetiscosCtrl', function($scope, CardapioServices)
{
    $scope.petiscos = CardapioServices.getPetiscos();
});
```

Deixo aqui mais um desafio ao leitor de transferir as chamadas ao serviço correspondente para o controller de bebidas e sucos. No final, peça um suco de cacau, pois precisamos refrescar um pouco nossa mente para organizar nossa aplicação na próxima seção.

## 5.4 ORDEM NA CASA

Anteriormente, criamos a pasta `www/js/services`, onde organizamos nosso arquivo da camada de serviço. Seguindo o mesmo raciocínio, vamos organizar nossa camada de *controller*. Fazendo isto, você estará facilitando a manutenção, permitindo ao desenvolvedor a rápida localização dos arquivos referentes a cada camada e funcionalidade.

Para isto, crie a pasta `controllers` em `www/js`, e transfira para ela os arquivos `controllers.js`, `bebidas-controller.js`, `petiscos-controller.js` e `sucos-controller.js`. Mas isto não basta; lembre-se de nossa importação de *scripts* no arquivo `index.html`, onde esta aponta para `www/js`. Veja a seguir:

```
...
<!-- your app's js -->
<script src="js/app.js"></script>
<script src="js/controllers.js"></script>

<script src="js/bebidas-controller.js"></script>
<script src="js/petiscos-controller.js"></script>
<script src="js/sucos-controller.js"></script>
<script src="js/detalhar-controller.js"></script>
...

```

Modifique-o para que refencie os arquivos na nova pasta criada, ficando da seguinte forma:

```
...
<!-- your app's js -->
<script src="js/app.js"></script>
<script src="js/controllers/controllers.js"></script>
<script src="js/controllers/bebidas-controller.js"></script>
<script src="js/controllers/petiscos-controller.js"></script>
<script src="js/controllers/sucos-controller.js"></script>

<script src="js/services/cardapio-services.js"></script>
...

```

Observe que transferimos também um arquivo que ainda nem alteramos, o `controllers.js`. Este *controller* é o responsável por carregar todos os outros *controllers*, pois é ele quem instancia a variável `app` usada por todos os outros arquivos, inclusive os *services*. Por isto, este é o primeiro controller a ser carregado no arquivo `index.html`.

Se esta sequência for alterada, o aplicativo não funcionará, porque o HTML é carregado sequencialmente de cima para baixo.

Depois de transferido para a nova estrutura de pastas, abra o arquivo `controllers.js` e edite-o, deletando tudo o que há nele, deixando apenas a função de carregamento dos outros *controllers*. Isso porque usaremos para isto a variável `app`, que está sendo instanciada nele, pegando a referência de `angular.module()`, isto é, o módulo principal do AngularJS.

```
var app = angular.module('starter.controllers', []);  
  
app.controller('AppCtrl', function($scope, $ionicModal, $timeout)  
{  
    //Nenhuma function  
});
```

Muito bem! Tudo organizado agora, vamos fazer o detalhamento de cada item, como planejado no início deste capítulo.

## 5.5 CRIANDO A TELA GENÉRICA

Até aqui, só vimos um protótipo de nossa tela de detalhes dos itens. Vamos construir utilizando os componentes CSS do *Ionic*, de forma que fique bem próxima ao nosso protótipo. O pessoal de *design* vai adorar saber que você foi fiel ao protótipo.

Crie o seguinte arquivo na sua pasta `www/templates/detalhe-item.html`. Nele insira o seguinte código e vamos analisá-lo:

```
<ion-view>
  <ion-content>
    <div class="card">
      <div class="item">
        <p>Título do Item</p>
        
        <p>Descrição:</p>
        <div class="item item-text-wrap">
          <p>
            Aqui, alguma descrição.
          </p>
        </div>
        <p>Preço: R$ XX,XX</p>
      </div>
    </div>
    <div class="item">
      <div class="bar bar-footer">
        <button class="button icon-right ion-checkmark butto
n-positive button-full">
          Pedir
        </button>
      </div>
    </div>
  </ion-content>
</ion-view>
```

Calma! Não se desespere! Vamos tentar entender ao máximo o que está acontecendo nesta tela.

Vejamos o resultado deste código e, logo em seguida, a explicação da *view*.



Figura 5.4: Pagina de detalhes

Lembra da nossa introdução, onde há uma explanação que o *Ionic* é o framework focado no *front view* da aplicação. Pois bem, aqui ficará bem claro que usamos muitos componentes prontos em CSS, facilitando assim a vida do desenvolvedor de aplicativos. Vamos explicar o código anterior e entender cada componente nele.

No início, temos uma `div card`. Os *cards* são retângulos com sombreamentos bastante utilizados em aplicativos. Eles dão destaque para diversos conteúdos, sejam estes textos, imagens,

botões ou forms.

Em nossa tela, observe que o título do item, a imagem, a descrição e o preço encontram-se dentro do nosso *card*, e isto dá uma aparência agradável ao aplicativo, além de destacar o conteúdo, separando-o do botão de ação **Pedir**.

Observe que cada parte do conteúdo foi separado pelas marcações `<div>`, com a classe de estilo `item`. Este dá o destaque tracejando cada elemento, como os traços abaixo de "Descrição" e acima de "Preço".

Por fim, o botão `Pedir` que contém 5 classes. Na sequência, temos a classe `button` informando ao *framework* que isto é um botão, com um ícone posicionado à direita (`icon-right`) e com símbolo de um check (`ion-checkmark`).

Existem muitos ícones já prontos para serem utilizados no *Ionic*. Estes podem ser encontrados em <http://ionicons.com>.

A classe `button-positive` configura apenas a cor do botão para azul e, logo em seguida, temos uma classe chamada `button-full`. Esta determina que o botão seja da largura da tela do dispositivo, ajustando ao máximo. Sem esta classe, o botão se ajustaria apenas ao *label*.

Por fim, este botão está dentro de uma `<div>` que determina o rodapé da tela em forma de barra (`bar` e `bar-footer`).

Fácil, não é?! Mas é preciso decorar todas estas classes? Não!

Sempre que for preciso, podemos consultar os componentes do *Ionic* em <http://ionicframework.com/docs/components/>.

## 5.6 PASSANDO UM PARÂMETRO

Já imaginou como faremos para transitar da tela principal de cada seção do cardápio para o detalhamento dos itens? Lembre-se de que cada item tem um identificador e, anteriormente, citamos que usaremos dois *webservices* neste contexto: um para carregar a lista de itens, e outro para o seu detalhamento. Portanto, poderemos imaginar que, ao clicar em um item, enviaremos o seu *id* para que o serviço retorne os seus detalhes.

Vamos ver agora como passar parâmetros entre as telas e recuperá-los nas transições de nosso aplicativo. Para isto, precisamos construir um *controller* em `www/js/controllers` , e configurá-lo no `www/js/app.js` , mapeando sua rota através de um estado ( `.state` ).

Portanto, crie um arquivo e dê a ele o nome `detalhar-controller.js` , salvando-o na pasta `www/js/controllers` . Em seguida, dê a ele uma referência denominada '`DetalharCtrl`' .

```
app.controller('DetalharCtrl', function($scope){  
})
```

Lembre-se de que esta referência será utilizada em nosso arquivo de configuração, onde criaremos o estado para a transição entre a tela principal e os itens que serão detalhados. Para isto, no arquivo `www/app.js` , logo abaixo do último estado que configuramos ( `app.petiscos` ), crie um estado que terá como diferencial a passagem de parâmetro na URL. Veja como ele ficará:

```
...
.state('app.detalhar', {
    url: '/detalhar/:itemID',
    views : {
        'menuContent' : {
            templateUrl : 'templates/detalhe-item.html',
            controller : 'DetalharCtrl'
        }
    }
})
...

```

Observe que, na URL, fizemos a passagem do parâmetro `id` do item. Este será utilizado para carregar o detalhamento dele, independente de qual seja. Assim, permitirá que a nossa tela, de fato, torne-se genérica, isto é, seja reaproveitada em todos os itens disponíveis.

Altere o arquivo `www/templates/petiscos.html` para que chame a URL passando o parâmetro `id` do item. Para isto, modifique o atributo `href` da lista para que passe o identificador de cada objeto carregado na `grid`.

```
...
<ion-item
    ng-repeat="petisco in petiscos"
    href="#/app/detalhar/{{petisco.id}}">
...

```

Agora tente clicar nos petiscos para ver a tela sendo executada. Se tudo ocorreu bem, você visualizará a tela genérica em todos eles. Apesar de já estar passando o parâmetro na URL, como pode ser observado no campo de endereço do seu browser, nada está sendo feito com ele. Na seção seguinte, vamos aprender a recuperar este parâmetro e carregar apenas o detalhamento do item referente a ele.

## 5.7 RECUPERANDO O PARÂMETRO

Como já foi explanado na seção anterior, precisamos recuperar o parâmetro `itemID` na tela de detalhamento, para que possamos consumir o serviço que retorna estes detalhes. Para isto, o *Ionic Framework* utiliza o objeto `$stateParams`. Você precisa apenas injetar este objeto em seu *controller* para que possa utilizá-lo. Vamos fazer uma experiência baseada nisto?

Crie um objeto `item` dentro do controle `DetalharCtrl`, recupere o `itemID` e atribua a este objeto. Em seguida, na tela genérica de detalhamento de itens, exiba este valor. Para isto, nosso código deverá ficar da seguinte forma:

```
app.controller('DetalharCtrl', function($scope, $stateParams){  
    $scope.item = {};  
    $scope.item.id = $stateParams.itemID;  
})
```

Agora você tem um problema para resolver. Depois de ter passado o parâmetro para tela seguinte e recuperá-lo dentro de seu *controller*, como você conseguirá exibi-lo na sua *view*? Você se lembra do *data binding* citado em nossa explicação sobre AngularJS? Pois bem, esta é a grande mágica ou magia, como queira chamá-la, do nosso *framework*. Constantemente, toda atualização feita no *controller* é exibida na *view*.

Faça esta experiência e veja na prática o `id` sendo exibido, o que significa que foi recuperado do evento *click* da tela anterior. Para exibi-lo, basta inserir em sua *view* uma referência ao objeto `item`, desta maneira: `{{item.id}}`.

## 5.8 CRIANDO O SERVIÇO DE

## DETALHAMENTO

Construiremos nesta seção um simulador de *webservice*, que retornará os atributos de detalhes dos itens do cardápio para que sejam exibidos na tela genérica. Para isto, vamos imaginar que cada item deve conter um título, uma imagem, uma descrição e um preço. Tudo isso foi esboçado na tela do protótipo do detalhamento do item, exibida neste capítulo.

Agora que conhecemos todos os atributos do nosso objeto, vamos codificá-lo para que o serviço retorne-os, simulando um acesso ao banco de dados e fazendo uma consulta pelo id de cada item.

Como você pode observar, nosso parâmetro de passagem é o `id`, portanto, deve estar claro que nosso *webservice* receberá o identificador do item. Com este parâmetro, o serviço consultará um array de itens, onde deverá comparar cada identificador e retornar o objeto referente a ele.

Esta não é a melhor forma de se localizar um objeto em uma aplicação, mas estamos simulando com *mocks*, e não com bancos de dados reais. Se fosse dessa forma, estariámos passando para o serviço o `id` do item e este retornaria o item referente, através de uma consulta ao banco com um `SELECT`. Em nosso caso, simularemos percorrendo a lista dos *mocks* até encontrar o item desejado.

Para isto, vamos utilizar o mesmo arquivo que criamos os

serviços, ou seja, nosso `CardapioServices` que se encontra na pasta `www/js/services`. Considerando que cada item será comparado pelo `id`, vamos modificar nossos arrays de itens para que eles tenham um identificador único, simulando assim uma tabela de itens em um banco de dados real. Para isto, vamos alterar todos os `ids` a partir dos petiscos, onde receberá os `ids` 3, 4, 5 e assim sucessivamente.

Além disto, devemos acrescentar a imagem e descrição de cada item. Neste livro, alteraremos apenas o cardápio de bebidas, deixando a cargo do leitor fazer a alteração dos demais itens, a título de exercício e prática.

Com isto, nosso serviço passará a ter o seguinte conteúdo:

```
//Mock Bebidas
var bebidas = [
    {'nome': 'Vinho Tinto Brasil', 'id': 1, 'preco': '40,00',
     'imagem': 'img/garrafas.png', 'descricao': 'Vinho tinto seco, da
     região Sul do país.'},
    {'nome': 'Cerveja Especial', 'id': 2, 'preco': '23,00',
     'imagem': 'img/cerveja.png', 'descricao': 'Cerveja elaborada art
     esanalmente com um toque frutado.'},
    {'nome': 'Wisky 18 anos', 'id': 3, 'preco': '200,00',
     'imagem': 'img/wisky.png', 'descricao': 'Wisky de fabricação Rus
     sa.'}
];
//Mock Petiscos
var petiscos = [
    {'nome': 'Filé ao molho madeira', 'id': 4, 'preco': '60,00'},
    {'nome': 'Camarão ao alho', 'id': 5, 'preco': '21,00'},
    {'nome': 'Lagosta assada', 'id': 6, 'preco': '100,00'}
];
//Mock Sucos
.
.
.
```

O que modificamos? Como foi descrito, alteramos os *ids*, a partir do array de petiscos, para que tenham um identificador único. Já no array de bebidas, adicionamos uma descrição e um caminho para uma imagem que deve estar na pasta `img`.

Caso o leitor prefira, pode baixar nossa imagem clonando o seguinte repositório: <https://github.com/adriangois/livro-ionicframework>. Todo nosso código estará disponível neste repositório e poderá ser baixado pelo leitor.

Mas o que falta? Um método que recebe um *id* e retorne o item referente. Para isto, vamos criar a seguinte função *JavaScript*:

```
this.getDetalheItem = function(id, callback){  
  
    bebidas.forEach(function(bebida){  
        if(bebida.id == id){  
            callback(bebida);  
        }  
    });  
  
    petiscos.forEach(function(petisco){  
        if(petisco.id == id){  
            callback(petisco);  
        }  
    });  
  
    sucos.forEach(function(suco){  
        if(suco.id == id){  
            callback(suco);  
        }  
    });  
}  
  
callback = function(item){  
    return item;  
}
```

Observe que percorremos todas as listas do cardápio, através

da função `forEach` do *JavaScript*. Assim que encontramos um item com o identificador igual ao que foi passado, retornaremos através de uma função de *callback*.

Não se desespere! *Callback* é uma função em *JavaScript*, em que o interpretador executa assim que finaliza todo processamento da função principal. Ou seja, por ser uma linguagem assíncrona, o *JavaScript* executaria os 3 `forEach`s em paralelo, e nosso retorno não seria satisfatório. Fizemos isto para que, ao final do processamento, seja chamada a função retornando nosso item.

## 5.9 DETALHANDO O ITEM

Quase tudo pronto, restam-nos dois detalhes para que nosso serviço seja chamado e os itens sejam exibidos:

a) Modificar nosso *controller* de detalhamento (`DetalharCtrl`) para injetarmos `CardapioServices`. Também executaremos a chamada ao serviço `getDetalheItem`, passando o *id* e uma função (citada acima) de *callback*, recebendo um item:

```
app.controller('DetalharCtrl',
  function($scope, $stateParams, CardapioServices){
    $scope.item = {};
    CardapioServices.getDetalheItem($stateParams.itemID,
      function(item){
        $scope.item = item;
      });
})
```

b) Modificar nossa *view* (`detalhe-item.html`) para que ela receba os atributos do objeto passado:

```
<ion-view>
  <ion-content>
    <div class="card">
```

```

<div class="item">
    <h1>{{item.nome}}</h1>
    <!--  -->
    <div class="card">
        
    </div>
    <p>Descrição:</p>
    <div class="item item-text-wrap">
        <p>
            {{item.descricao}}
        </p>
    </div>
    <p>Preço: R$ {{item.preco}}</p>
    </div>
</div>
<div class="item">
    <div class="bar bar-footer">
        <button class="button icon-right ion-checkmark button
-positive button-full">Pedir</button>
    </div>
</div>
</ion-content>
</ion-view>

```

Customizamos duas classes no CSS da aplicação em [www/css/style.css](http://www/css/style.css) : centralizar-imagem e margem-imagem . Elas foram criadas para centralizar a imagem do item e inserir uma margem abaixo para organização dos elementos na tela. Vejamos a seguir o código CSS:

```
.
.centralizar-imagem{
    margin-left: 20%;
}

.margem-imagem{
    padding:1em;
}
```

A partir daqui, vamos utilizar muito essas classes, principalmente a `margem-imagem`.

Feito isto, façamos um teste clicando no cardápio de bebidas e, em seguida, detalhando cada um dos 3 itens listados nele. Veja o resultado na figura:



Figura 5.5: Detalhamento de item

## 5.10 CONCLUSÃO

Neste capítulo, vimos como passar parâmetros de uma tela para outra e criar serviços. Com isto, aprendemos a detalhar todos os itens utilizando uma premissa muito importante em programação: reaproveitamento. Criamos apenas uma tela genérica, e nela detalhamos todos os itens, independente de qual cardápio ele faça parte.

No próximo capítulo, vamos criar nossa lista de pedidos e enviar para o serviço, adicionando todos os valores em uma conta.

## 5.11 PARA SABER MAIS

### 1. Cards

<http://ionicframework.com/docs/components/#cards>

### 2. Listas

<http://ionicframework.com/docs/api/directive/ionList>

## CAPÍTULO 6

# FAZENDO PEDIDOS

Neste capítulo, vamos dar vida ao botão `Pedir` da tela de detalhes do item, para que assim possamos simular um carrinho de compras dos sistemas de *e-commerce*. O fluxo funciona da seguinte maneira: o cliente faz o pedido, informando a quantidade e ele será adicionado à bandeja do sistema. Será possível adicionar vários itens à bandeja antes de confirmar os pedidos.

Ao clicar na bandeja, após estes pedidos, o cliente poderá editar ou confirmá-los. Ao confirmar a lista de pedidos, estes serão enviados ao setor responsável pelo atendimento e a lista retornará vazia, mas adicionando os pedidos na conta do usuário.

## 6.1 OS REQUISITOS DO APLICATIVO

Vamos raciocinar como funcionaria um cardápio eletrônico em sua forma de efetuar os pedidos. Omitiremos a funcionalidade de logar, pois somente assim seria possível identificar a mesa que estará fazendo as solicitações, através do aplicativo. Imagine que nosso cardápio já terá um identificador guardado em sua sessão (logado).

Agora vamos avaliar como um usuário faria seu pedido a um garçom humano após verificar as informações do cardápio. Este

provavelmente o chamaria e pediria um item de cada vez e com sua quantidade. Por exemplo: "*Garçom! Por favor, traga-me duas cervejas especiais e um filé ao molho madeira*". Ao final, o garçom encerra, confirmando o pedido e emitindo uma anotação ao setor responsável por despachá-los.

Em nosso caso, vamos imaginar que, ao clicar no botão Pedir do detalhamento de cada item, um modal para informar a quantidade se abrirá e, ao confirmar, ele entrará em nosso "carrinho de compras". Para confirmar os pedidos, será preciso clicar no "carrinho" e, em seguida, enviar todos os pedidos para o servidor.

Tudo isto já foi desenhado em nosso protótipo no capítulo anterior. Precisamos apenas definir como se comportará nosso "carrinho". Por uma decisão coerente, a partir de agora, chamaremos de **bandeja** e ela estará posicionada no lado superior direito. Veja a seguir um protótipo desta função.



Figura 6.1: Bandeja

Observe que, simulando uma bandeja, o ícone no canto superior direito sugere que temos pedidos e, ao clicar neste ícone, a tela será direcionada para uma listagem das solicitações. O próximo passo do fluxo é a confirmação do pedido para que sejam enviados ao setor responsável (servidor) e limpeza da bandeja, retornando para a tela inicial do aplicativo.

Agora que temos os requisitos da funcionalidade, vamos implementar. A proposta é continuarmos fiéis ao protótipo que o pessoal do *design* nos passou. Eles vão adorar e agradecer.

## 6.2 UMA FORMA DE GUARDAR NOSSA SESSÃO

Para que nosso aplicativo simule que o usuário se encontra logado com um identificador único, vamos manter um número

que será usado para localizar a mesa do pedido e uma lista que será nossa variável para guardar nossa bandeja. Para isto, precisamos entender um novo conceito em AngularJS: os *values*.

Toda aplicação web construída em AngularJS é formada por objetos que colaboram entre si para executar as funcionalidades. Estes objetos são criados e conectados através do conceito já citado: **injeção** dos serviços.

O AngularJS dispõe de um serviço chamado *Value Recipe*. Vamos utilizar este serviço para guardar os valores em diversos lugares de nosso aplicativo. Fazendo analogia com qualquer linguagem de programação, é como se guardássemos um objeto em uma variável global e, quando necessário, recuperássemos estes valores.

Para esta finalidade, usaremos o *value* para guardar nosso identificador da mesa. A seguir, criamos um *script* chamado `www/js/services/sessao.js`.

```
angular.module('starter.controllers').value('Sessao', {  
    mesaID: "123",  
    bandeja: []  
});
```

Observe que, no lugar do *controller* ou *service*, instanciamos um *value*. Isto diz ao *framework* que este *script* nada mais é do que um *contêiner* usado para guardar algumas constantes e variáveis. Agora é só importá-lo no `index.html` e ele estará pronto para ser injetado nos *controllers* da aplicação. Vamos fazer isto?

...

```
<!-- Cabeçalho do index.html -->  
<script src="js/services/sessao.js"></script>  
</head>
```

...

Prontinho! Agora você já pode injetar dentro de qualquer *controller* ou *service*. Mas qual o mais indicado para isto? Veremos a seguir.

## 6.3 HORA DE INJETAR O VALUE

Já que construímos uma simulação da sessão em nossa aplicação, podemos analisar que, para a utilização desses valores (mesa e bandeja), devemos deixá-los guardados em algum *controller* que esteja o tempo todo instanciado. Queremos dizer com isto que, dependendo da configuração de sua aplicação, cada vez que você chama uma nova tela, o *controller* da tela anterior sai da memória e entra o novo em seu lugar, e assim sucessivamente.

E agora?! Onde guardarei minha sessão já que isto acontece?

Vamos recapitular a seguinte estrutura de nossa pasta `www/templates` : temos nela alguns HTMLs (como por exemplo, `bebidas.html` e `detalhe-item.html` ) e nossa aplicação tem um layout com um menu lateral. Este menu é mantido pelo `menu.html` , ou seja, este é o *template* onde todas as páginas citadas são inseridas.

Seguindo a linha deste raciocínio, vamos ao arquivo `www/js/app.js` e vejamos o que ele diz sobre meu `menu.html` .

```
...
stateProvider
  .state('app', {
    url: '/app',
    abstract: true,
    templateUrl: 'templates/menu.html',
    controller: 'AppCtrl'
```

```
})  
...
```

Observe no código que o *controller* do menu é o `AppCtrl`. Com isto, conclui-se que ele sempre estará em memória durante a execução da nossa aplicação. Isso porque, mesmo quando você carrega telas diferentes em sua aplicação, o menu lateral estará lá.

Assim, injetaremos o *script value*, configurado anteriormente, em nosso *controller*. Para isto, basta declarar, em nosso `www/js/controllers/controllers.js`, um parâmetro na *function* chamado `Sessao`, ou seja, a instância de nosso *script value*.

Além disto, devemos declarar uma variável de escopo (`$scope`) para a bandeja de nosso cardápio. Portanto, nosso código ficará como a seguir:

```
var app = angular.module('starter.controllers', []);  
  
app.controller('AppCtrl', function($scope, Sessao, $ionicModal, $timeout) {  
    $scope.bandeja = Sessao.bandeja;  
});
```

Observe que injetamos '`Sessao`' baseado no nome que você deu à instância do *Value Recipe*. Isto é, o nome que o AngularJS disponibiliza para ser injetado nos *controllers* e *services* é o mesmo que você declara na assinatura do objeto `'.value([nome_da_instância])'`.

Quase tudo pronto, vamos agora obedecer ao pessoal de design

que construiu nosso protótipo e inserir um ícone com um contador de produtos, que também servirá de link de direcionamento para a listagem de pedidos em nossa bandeja.

Para isto, usaremos dois componentes fornecidos pelo *framework*: um botão decorado com um ícone, e um elemento chamado *badge*, que nada mais é do que um contador destacado de vermelho no topo da aplicação.

A seguir, veja a modificação de nosso `www/templates/menu.html` com o código devidamente comentado:

```
<ion-side-menus enable-menu-with-back-views="false">
  <ion-side-menu-content>
    <ion-nav-bar class="bar-stable">
      <ion-nav-back-button>
      </ion-nav-back-button>
      <ion-nav-buttons side="left">
        <button class="button button-icon button-clear ion-navicon" menu-toggle="left">
        </button>
      </ion-nav-buttons>

      <!-- Modificação da posição do badge. A melhor prática diz
      que este trecho deve estar no arquivo style.css,
      mas para efeito didático, colocamos dentro do html. -->
      <style>
        .button .badge {
          top: -36px;
        }
      </style>
      <ion-nav-buttons side="right">
        <!-- Botão que direcionará para a listagem de pedidos. -->
        <button class="button button-icon button-clear ion-information-circled">
          <!-- Badge com número de pedidos -->
          <span ng-show="true" class="badge badge-assertive">{{bandeja.length}}</span>
```

```
        </button>
    </ion-nav-buttons>
</ion-nav-bar>
<ion-nav-view name="menuContent"></ion-nav-view>
</ion-side-menu-content>
...

```

Preste atenção nos comentários, onde criamos um botão com a classe `button-icon` e `ion-information-circled`. Estes componentes criam um botão com o ícone de um círculo. Dentro deste, foi inserida a tag `<span>` do HTML com as classes `badge` e `badge-assertive` (na cor vermelha) para informar o número de pedidos.

Não deixe de observar também que a quantidade está sendo construída através do tamanho (`length`) do array `bandeja` e executando o *data-binding* do AngularJS `{{bandeja.length}}`. O resultado disto pode ser visto na seguinte figura.

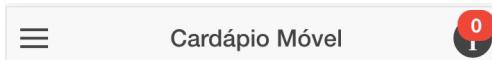


Figura 6.2: Badge da bandeja

## 6.4 GUARDANDO OS PEDIDOS

Bom trabalho até aqui! Construímos nosso layout e já temos onde guardar nossos pedidos. Agora precisamos construir uma função que executaremos quando o cliente selecionar um item do menu e clicar no botão `Pedir`. Além disto, ele deve informar a quantidade para que seja calculado o valor real em sua conta, e os itens sejam fornecidos de forma correta.

Lembra da tela de detalhamento de pedidos que foi planejada com o protótipo e posteriormente construída? Pois bem! No

capítulo anterior, aprendemos como passar parâmetros entre as telas, construímos nossa tela genérica de detalhes dos itens através deste parâmetro, mas deixamos de lado a ação do botão `Pedir`.

Não tem problema! Vamos construir aqui esta ação baseada no protótipo que nos emitirá um modal para informar a quantidade e confirmar o pedido. Posteriormente, assim que confirmado o pedido, ele será inserido na bandeja e terá um comportamento similar ao carrinho de compras de qualquer sistema de *e-commerce*, conforme já comparamos anteriormente neste capítulo.

Agora vamos lembrar de uma máxima que escutamos desde pequenos: "tenho uma boa e uma má notícia para você. Qual você quer ler primeiro?". Não fugindo à regra, o leitor provavelmente pensou: "A má primeiro, por favor!". Pois bem! Vamos seguir nesta sequência.

A má é que o *Ionic* não oferece nativamente um componente que precisamos usar: o *number picker*. Este tem a característica de um *input* de números através de botões incrementando/decrementando estes valores, conforme visto em nosso protótipo. Ao menos em sua versão 1, o *framework* não nos dá isso prontinho para utilizar.



Figura 6.3: Number Picker

Agora vamos ler a boa: utilizaremos um componente de terceiros, instalando-o através do comando `bower`. Para entendermos o que é este comando, precisamos relembrar da instalação de nosso ambiente no primeiro capítulo, onde usamos o

comando `npm install -g cordova ionic`.

Tanto o `bower` (<https://bower.io/>) quanto o `npm` (<https://www.npmjs.com/>) são gerenciadores de pacotes *JavaScript*, em que você pode reutilizar, compartilhar e encontrar componentes de outros desenvolvedores. É uma espécie de repositório. Para que você possa utilizar o `bower`, será necessário instalá-lo pelo `npm` com o seguinte comando `npm install -g bower`. Após a instalação, execute também:

```
bower install angular-number-picker --save
```

Viu?! Além da boa e da má notícia, existe uma outra: a excelente! O componente será instalado com apenas um comando.

Observe a instalação e, quando finalizar, o seu componente estará prontinho para ser usado em `www/lib/angular-number-picker`. Agora vamos importá-lo em nosso projeto no arquivo `www/index.html`. Para isto, insira o código do script no `index.html`, como tem feito até aqui:

```
...
<script type="text/javascript" src="lib/angular-number-picker/dist/angular-number-picker.min.js"></script>
...
```

Chegou a vez de injetarmos este componente em `www/js/app.js`, para que ele seja utilizado em toda aplicação. Com isto, o módulo ficará com os seguintes parâmetros:

```
angular.module('starter',
  ['ionic',
   'starter.controllers',
   'ngNumberPicker'])
...
```

Agora que você já tem noção do que vai construir, sugiro que

volte ao arquivo `www/templates/detalhe-item.html` e localize nosso botão `Pedir`. Observe que ele está lá, sem vida, sem ação. Vamos dar a ele vida e ação, pois cor ele já tem.

Precisamos construir um modal que ficará escondidinho (`hide`) até o momento que o usuário clicar em `Pedir`, conforme visto no protótipo do capítulo anterior. Para isto, adicione o seguinte código antes do botão:

```
<ion-view>
  <ion-content>
    <!-- Modal -->
    <script id="my-modal.html" type="text/ng-template">
      <ion-modal-view>
        <ion-content>
          <div class="card">
            <div class="item txt-centralizado">
              <p>Informe a quantidade</p>
              <p>e confirme este pedido.</p>
              <br />
              <h2>{{item.nome}}</h2>
              <p><h-number value="input.quant" min="1" max="100" step="1"></h-number></p>
            </div>
            <div class="margem-imagem"></div>
          </div>
          <div class="bar bar-footer">
            <div class="button-bar">
              <button class="button icon-right ion-checkmark button-positive" ng-click="pedir()">Pedir</button>
              <button class="button icon-right ion-checkmark button-assertive" ng-click="fechaModal()">Cancelar</button>
            </div>
          </div>
        </ion-content>
      </ion-modal-view>
    </script>
    <!-- Fim/Modal -->
    ...
  </ion-content>
</ion-view>
```

Observe que, mesmo antes de construirmos as funções para

fechar o modal e confirmar o pedido - `fechaModal()` e `pedir()` -, já estamos prevendo e colocando-as no `ng-click` dos botões. Analise o código e verifique também que criamos uma `div` com a `class button-bar`, na qual teremos o seguinte efeito:



Figura 6.4: Button bar

Criamos assim uma barra com dois botões alinhados. A class `button-assertive` e `button-positive` determinam as cores dos botões.

Analisando mais o código, note que reutilizamos uma classe criada no início de nosso livro (a `txt-centralizado`) para centralizar o conteúdo de nosso modal na `div` que encapsula o texto e o *number picker*. Além disto, veja o uso do componente que baixamos com o bower . O `<h-number>` recebe um `value` que deve estar declarado em nosso *controller*. Vamos ver este valor mais adiante quando o codificarmos.

Refrescando nossa memória, lembre-se de que o *controller* deste *template* é o `DetalharCtrl`, declarado no arquivo `www/js/detalhar-controller.js` . Abra-o, pois é nele que vamos atuar a partir de agora.

Vamos construir uma função para fazer a chamada através de nosso botão. Para isto, adicionaremos as seguintes funções de nosso modal:

```
...
$scope.input = {};
$scope.input.quant = 0;
```

```
$ionicModal.fromTemplateUrl('my-modal.html', {
    scope: $scope,
    animation: 'slide-in-up'
}).then(function(m){
    $scope.modal = m;
});

$scope.abreModal = function(){
    $scope.modal.show();
}

$scope.fechaModal = function(){
    $scope.modal.hide();
}
...
...
```

Agora observe que estamos usando um objeto como referência para o nosso modal: `$ionicModal`. Nele inserimos o template construído em `www/templates/detalhe-item.html`, e informamos parâmetros como animação (*animation*) abrindo a variável de escopo (`$scope`) do AngularJS a partir do topo da aplicação.

Além disto, abriremos o modal através de `$scope.modal.show()` e fecharemos através de `$scope.modal.hide()`. Mas lembre-se: para que tudo seja executado corretamente, precisaremos ter o objeto `$ionicModal` também corretamente inserido no contexto.

Vamos fazer agora um exercício de memória? Qual recurso nos permite inserir este componente no contexto da aplicação? Claro! Através da injeção. Eu sabia que você se lembraria. Também tenho certeza de que você se antecipou e a assinatura de seu *controller* ficou assim:

```
...
app.controller('DetalharCtrl',
```

```
function($scope,
        $stateParams,
        CardapioServices,
        Sessao,
        $ionicModal){
...
}
```

Observe também o início do código onde declaramos o `$scope.input.quant` para que seja utilizado pelo componente `<h-number>`, por meio do parâmetro `value="input.quant"`. Desta forma, estaremos guardando o valor para inserir em nossa bandeja.

Tudo pronto, faça um teste e verá que o modal abrirá, porém o botão `Pedir` não confirmará ainda o pedido. O botão de cancelar fechará o modal e o usuário voltará à tela do produto.

Vamos então construir nossa função para adicionar o pedido na bandeja. Para isto, precisamos fazer uma função que adicione o item em uma lista com 2 parâmetros importantes: o objeto `item` e a quantidade deste item.

Após este processo, o pedido deve ser inserido na bandeja e o modal deverá ser fechado, retornando à tela anterior. Vamos ver como ficará esta função no `www/js/detalhar-controller.js` ?

```
...
$scope.inserirPedido = function(){
    Sessao.bandeja.push({item: $scope.item, quantidade : $scope.i
nput.quant});
    $scope.modal.hide();
    $scope.input.quant = 1;
}
```

Verifique no código a utilização da função `push` do JavaScript que insere o item e a quantidade que está no `$scope` . Colocamos a `quantidade` com um valor inicial igual a 1, pois assim ficará

quando voltarmos a abrir este modal. Faça isto também na função `fechaModal()`, pois impede do usuário aumentar a quantidade e desistir no meio do processo, assim cancelando, e retornar com o valor guardado.

Bem! Tudo prontinho, agora é só testar e observar que o seu aplicativo já está inserindo o item na bandeja, pois o ícone já incrementa a quantidade no topo da aplicação, como ilustra a figura a seguir:



Figura 6.5: Bandeja incrementando

## 6.5 CONCLUSÃO

Aqui vimos como criar um modal para inserir um objeto em um array JavaScript, aproveitando o valor (quantidade) informado nele, e com isto guardá-lo na sessão do aplicativo. Vimos também como instalar um componente de terceiros através do gerenciador de pacotes JavaScript `bower` e como usá-lo.

No próximo capítulo, aprenderemos a gerenciar os pedidos da bandeja, excluindo, alterando quantidades e visualizando seus valores.

## 6.6 PARA SABER MAIS

1. **Bower** – <https://bower.io>
2. **Npm** – <https://www.npmjs.com>
3. **Angular Number Picker** –  
<https://github.com/leftstick/angular-number-picker>

## CAPÍTULO 7

# A BANDEJA

Neste capítulo, vamos planejar e implementar a visualização da bandeja com a possibilidade de editar os itens e também confirmar o pedido para que seja enviado ao setor responsável por receber e encaminhar ao garçom.

Vimos no capítulo anterior que a bandeja é similar ao carrinho de compras dos *e-commerces*. Portanto, o processo consiste em inserir pedidos dentro deste carrinho, em nosso caso na bandeja, e posteriormente ser possível visualizar e editar as quantidades destes itens.

Para isto, vamos imaginar uma tela com uma lista onde teremos a quantidade pedida, o item e um valor - produto da multiplicação da quantidade pelo preço. Usaremos um novo modal que abrirá com o toque no botão que representará nossa bandeja, no topo superior direito do aplicativo. Nele implementaremos ações para a lista de itens. Vamos ao trabalho?!

## 7.1 ELEMENTOS DO MODAL

Como já vimos no capítulo anterior, precisamos "esconder" o modal em algum template e mostrá-lo assim que o usuário executar determinada função responsável por abri-lo. O modal

também será um template, ou seja, um template escondido (inserido) em um template. Parece confuso?!

Na prática, veremos como isto funciona. Existem várias formas de construí-lo, inclusive em arquivos separados para melhor reaproveitamento. Porém, em nosso caso, faremos dentro do `www/templates/menu.html`, pois este modal será muito específico, e provavelmente não seria reutilizado em nossa aplicação. Até porque, dentro do template de menu, ele estará disponível para abertura em qualquer cenário. Basta tocar no botão da bandeja quando ele tiver um pedido ou mais.

Pensando no protótipo da bandeja apresentado no capítulo anterior, teríamos este presentinho do pessoal do design:

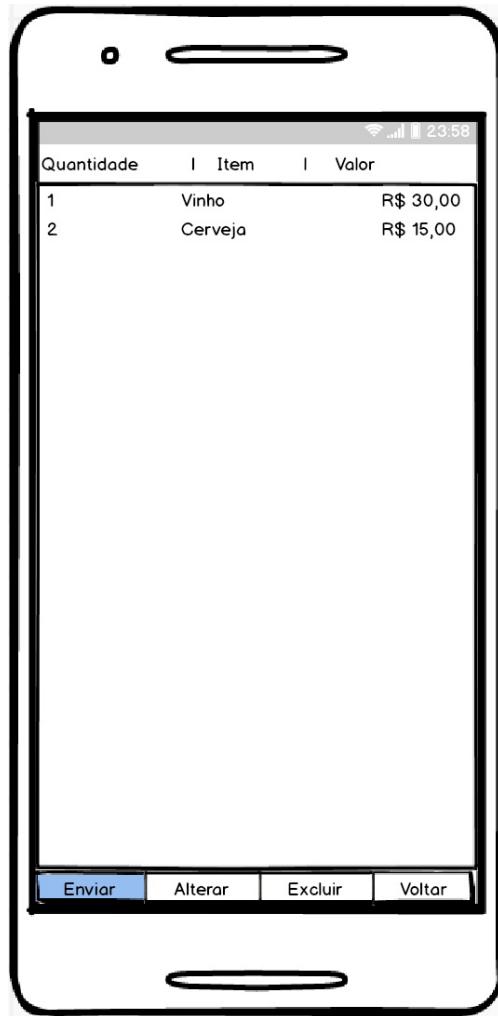


Figura 7.1: Protótipo

Observe a lista na qual cada elemento tem uma quantidade, que foi determinada no modal aberto pela ação `Pedir`. Lembra-se dela? Ao efetuar um pedido, o usuário informa quantos itens devem conter nele através de um *number picker*. Além disto, a lista

traz um título do item e o valor deste pedido, ou seja, uma multiplicação do preço do item com a quantidade.

Logo abaixo, teremos 4 botões para as ações: **Enviar** a bandeja para efetuar o pedido, **Alterar** o pedido selecionado, **Excluir** um item selecionado da bandeja sem efetuar o pedido e **Voltar** fechando o modal e permanecendo os pedidos na bandeja. Mas como faremos isto na prática utilizando o *Ionic*? No papel, está fácil, não é, pessoal do *design*?

Aproveitando para filosofar um pouco, todo aprendizado tem sua origem na ignorância. Não sabemos ainda como construir este modal, mas pode ter certeza de que, ao final deste capítulo, você vai olhar para trás e pensar que valeu a pena. A partir daí, a ignorância passou a ser aprendizado, e isto provavelmente vai acrescentar muito em sua vida profissional.

Para início de conversa, vamos aos nossos componentes prontos e, como diria meu avô, "mastigados" do *framework*. O arquivo `www/templates/menu.html`, a partir da tag `</ion-side-menu>`, ficará assim:

```
<!-- Modal -->
<script id="bandeja.html" type="text/ng-template">
  <ion-modal-view>
    <ion-content>
      <div class="card">
        <div class="item txt-centralizado">
          <!-- Colunas -->
          <div class="row">
            <div class="col">
              Quantidade
            </div>
            <div class="col">
              Item
            </div>
            <div class="col">
```

```

        Valor
    </div>
</div>
<!-- Loop em Lista -->
<ion-list ng-repeat="it in bandeja">
    <!-- Radio -->
    <ion-radio>
        <div class="row">
            <div class="col">
                {{it.quantidade}}
            </div>
            <div class="col">
                {{it.item.nome}}
            </div>
            <div class="col">
                {{it.valor}}
            </div>
        </div>
    </ion-radio>
</ion-list>
<br />
</div>
<div class="margem-imagem"></div>
</div>
<div class="bar bar-footer">
    <div class="button-bar">
        <button class="button button-positive"><i class="ion-ios-checkmark-outline"></i></button>
        <button class="button button-dark"><i class="ion-edit"></i></button>
        <button class="button"><i class="ion-ios-trash-outline"></i></button>
        <button class="button button-assertive" ng-click="fecharModal()"><i class="ion-ios-undo-outline"></i></button>
    </div>
</div>
</ion-content>
</ion-modal-view>
</script>
<!-- Fim/Modal -->
```

Não se desespere! Como já foi dito, estes componentes foram feitos para você, desenvolvedor de aplicativos, utilizar à vontade.

Basta entendê-los e, por isso, agora vamos explicar.

O modal contém uma lista que será exibida com o componente `<ion-list>`. O *loop* na lista é feito através do atributo `ng-repeat` deste componente. Observe que fizemos um *loop* em `bandeja` que estará definido no *controller*, atribuindo cada elemento deste *array* em uma variável chamada `it`. De cada elemento, daremos um *print* através das chaves duplas ( `{ { } }` ), que vimos em capítulos anteriores, o qual utiliza o *data-binding* do AngularJS para sincronizar as atualizações desta variável.

Temos também um componente novo nesta iteração e acima dela, que são as colunas de um *grid*. Observe a seguir como se comportam essas *grids* responsivas no *Ionic*.

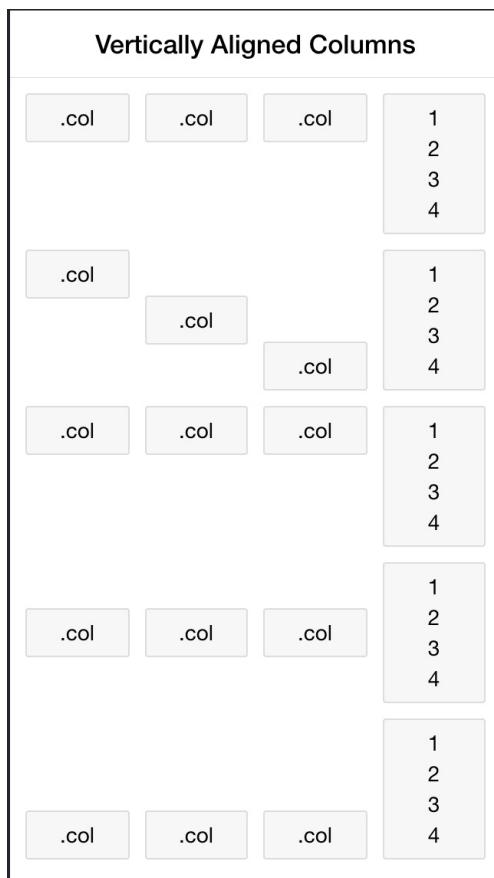


Figura 7.2: Sistema de Grids

Não vamos nos aprofundar em grids do framework, mas o importante é saber que elas existem. A forma mais simples de organizar conteúdos com ela é construindo uma linha com a class css `row` e inserir colunas com a class css `col`.

Para aprofundar mais sobre as grids, a documentação do *framework* é **vasta**, em

<https://ionicframework.com/docs/components/#grid>. Por enquanto, vamos focar apenas nesta forma simples de organizar os componentes.

Um outro componente novo é o `radio button` e ele foi declarado em `<ion-radio>`. Cada linha do objeto bandeja estará envolvida em um `radio`, ou seja, uma seleção única. Isto foi planejado por causa da funcionalidade de editar o item pedido, alterando sua quantidade. O usuário selecionará um item para edição e acionará o botão `Alterar` para que seja possível efetuar, de fato, a edição do item selecionado.

O último elemento novo em nossa codificação é o `<i>`, dentro de cada tag `<button>`. Ela determina que o botão tenha uma imagem, e não um label como de praxe. A imagem será declarada por cada `class`. São elas: `ion-ios-checkmark-outline`, `ion-edit`, `ion-ios-trash-outline` e `ion-ios-undo-outline`. O próprio *framework* disponibiliza uma grande quantidade de ícones que podem ser pesquisados e utilizados, em <http://ionicons.com>.

Bem! Até aqui, só vimos como criar nosso modal, mas ainda não o chamamos em botão algum, entretanto já sabemos onde vamos executar esta ação, não é mesmo?! Para isto, vamos criar uma função JavaScript em nosso `www/js/controllers.js`.

Lembre-se de que este *controller* contém um objeto simulando nossa sessão, usado para guardar os itens pedidos em nossa bandeja (`Sessao.bandeja`). Abaixo dele, vamos inserir nosso código para declarar, abrir e fechar o modal do arquivo `www/templates/menu.html`.

...

```

$ionicModal.fromTemplateUrl('bandeja.html', {
    scope: $scope,
    animation: 'slide-in-up'
}).then(function(m){
    $scope.modal = m;
});

$scope.abreModal = function(){
    $scope.modal.show();
}

$scope.fechaModal = function(){
    $scope.modal.hide();
}
...

```

Observe que o template do modal é o `bandeja.html` , o mesmo declarado na tag `<script id="bandeja.html" type="text/ng-template">` do `www/templates/menu.html` . Pronto! Agora só precisamos colocar a ação de exibi-lo.

Vale lembrar que o ícone representante da bandeja está envolvido por um botão, até então sem ação. É justamente nele que faremos a ação de abrir o modal, através da diretiva `ng-click` , já vista antes.

Ainda assim, vamos explicar melhor o que são essas diretivas. Elas são extensões do HTML, providas pelo *framework* AngularJS. Elas permitem que tenhamos novos comportamentos, ou seja, o desenvolvedor poderá construir suas diretivas ou utilizar as já existentes do AngularJS. Veja a seguir o ícone da bandeja sendo envolvido por um `<button>` :

```

...
<!-- Botão que direcionará para a listagem de pedidos. -->
<button class="button button-icon button-clear ion-informatio
n-circled">
    <!-- Badge com número de pedidos -->
    <span ng-show="true" class="badge badge-assertive">{{bandej

```

```
a.length}}</span>
</button>
...
```

Nele é possível observar duas características do AngularJS: uma diretiva `ng-show` e um *data-binding* através do `{{bandeja.length}}`. O *data-binding* estará sendo atualizado constantemente, de acordo com a quantidade de itens inseridos no array `bandeja`. Inicialmente, estará com 0 (zero) e será incrementado de acordo com a ação do usuário ao inserir ou remover itens (decrementando).

Agora vamos usar a diretiva `ng-click`, que tem o comportamento parecido com o nativo do `onclick` do HTML. Daremos a ela a função para abrir o modal toda vez que for clicada. Esta função já foi criada anteriormente neste capítulo. Lembra-se dela?

```
...
$scope.abreModal = function(){
    $scope.modal.show();
}
...
```

Enfim, nosso botão passará a ter uma diretiva `ng-click`, chamando esta função da seguinte forma:

```
...
<!-- Botão que direcionará para a listagem de pedidos. -->
<button class="button button-icon button-clear ion-informatio
n-circled" ng-click="abreModal()">
    <!-- Badge com número de pedidos -->
...

```

Clique e observe o comportamento. Consegue prever um problema aqui? Quando a "Equipe de Testes" começarem a testar

seu aplicativo, eles vão devolver um *bug* e você não terá o direito de ficar bravo com eles. O modal abre, mesmo sem existir pedido na bandeja. A parte boa desse *bug* é que você resolverá em 1 minuto e poderá ir para a copa tomar um café quentinho enquanto relaxa um pouco. Vamos à solução?

```
...
$scope.abreModal = function(){
    //Testa se existe pelo menos um item na bandeja.
    if($scope.bandeja.length > 0)
        $scope.modal.show();
}
...
```

Simplesmente isto! Ao executar a função, vamos testar se existe pelo menos um pedido inserido na bandeja através do tamanho do array. Se não existe, a função não executará nada. Caso contrário, a bandeja será aberta através da função `show` do modal.

Com esta mesma lógica e linha de raciocínio, poderemos construir um outro modal ou apenas um aviso na tela do aplicativo informando que não há item na bandeja. Mas isso fica a critério do leitor. Não faremos esta funcionalidade neste livro pelo simples motivo de que vamos apenas replicar códigos já vistos.

Ao executar a função e abrir o modal, você poderá observar que só estamos exibindo a quantidade de item que foi informada no componente *number picker* e a descrição do item. Precisamos calcular o valor total de cada pedido, multiplicando a quantidade pelo preço. Entretanto, temos um problema em nosso planejamento inicial.

Ao pensarmos em nosso *mock* criado no serviço de detalhamento dos itens no capítulo *Detalhando os itens*, esquecemos de um pequeno detalhe, mas saibam que foi

proposital. Utilizamos para o preço do item um valor do tipo `VARCHAR`, ou `string`, como quiser chamar o nosso valor alfanumérico. Isto aconteceu porque pensamos mais na forma como este valor seria apresentado em tela do que representado em um banco de dados, por exemplo.

Com isto, o JavaScript, que é uma linguagem de programação não tipada, faz a variável assumir um valor do tipo `string`, o que torna impossível multiplicá-lo por um número do tipo `number`. Para saber mais sobre tipos em JavaScript, consulte a seção *Para saber mais* no final deste capítulo.

Entretanto, o que acontece aqui é que determinamos o preço do item através de aspas. Isto faz com que o JavaScript interprete em tempo de execução que nossa variável é do tipo `string`.

## 7.2 VAMOS TER RETRABALHO

Tendo o problema em vista, assumindo nosso erro de planejamento com nosso **Gerente de Projetos**, vamos corrigir o serviço para que nos retorne o valor em formato numérico. Para isto, altere-os da seguinte forma, no arquivo `www/js/cardapio-services.js`:

```
...
//Mock Bebidas
var bebidas = [
    {'nome': 'Vinho Tinto Brasil', 'id' : 1, 'preco' : 40.00,
     'imagem' : 'img/garrafas.png', 'descricao':'Vinho tinto seco, da região Sul do país.'},
    {'nome': 'Cerveja Especial', 'id' : 2, 'preco' : 23.00,
     'imagem' : 'img/cerveja.png', 'descricao':'Cerveja elaborada artesanalmente com um toque frutado.'},
    {'nome': 'Whisky 18 anos', 'id' : 3, 'preco' : 200.00,
     'imagem' : 'img/whisky.png', 'descricao':'Whisky de fabri
```

```
cação Russa.' }  
];  
...
```

Retiramos a vírgula do preço e também as aspas. Agora a linguagem JavaScript assume que este valor é um `number`, e assim você poderá multiplicá-lo com qualquer número. Veja que só fizemos a correção no *array* de bebidas, ficando a critério do leitor a correção nos outros itens.

Agora é hora do cálculo do valor total. Vamos lá? Voltando ao arquivo `www/js/detalhar-controller.js`, na função `inserirPedido()` criada no capítulo *Fazendo pedidos*, vamos fazer o cálculo na linha que adiciona o item no *array* através da função `push()` do JavaScript.

```
...  
Sessao.bandeja.push({item: $scope.item, quantidade : $scope.input  
.quant, valor: $scope.input.quant*$scope.item.preco});  
...
```

Todo desenvolvimento de sistemas está inserido em um perigo temido pelos analistas e gestores: o retrabalho causado por um impacto em alguma falha na fase de planejamento. Com o nosso aplicativo, aconteceu exatamente isto. Corrigimos um problema, mas inserimos outros, pois esperávamos que o preço do produto chegasse, quando consultássemos o serviço, no formato "redondinho" para exibição em tela.

Entretanto, de acordo com a necessidade, vimos que é preciso um retorno numérico e cru para que sejam efetuadas operações matemáticas nele. Além disto, o preço deve representar o tipo de dados que ele realmente é: `number`.

O problema que inserimos é visível quando se direciona para

qualquer tela que exibe o preço do produto. Este estará no seguinte formato: R\$ 99.999, ou seja, com um ponto representando um valor `double` em vez do formato de nossa moeda, o Real, com a vírgula.

Calma! Não se desespere, pois o pessoal que desenvolveu o JavaScript pensou nisso e resolveu de uma forma bem discreta e eficiente este problema. Basta transformar o valor, em todas as telas em que se quer representar o preço, através de uma *string* com parâmetros que determinam a moeda.

Vamos aqui lhe apresentar a função `toLocaleString()` do JavaScript e, se quiser se aprofundar nela, veja a seção **Para saber mais** onde encontrará a especificação desta função e os códigos que podem ser usados para outras moedas.

Vamos fazer a transformação com esta função para a moeda Real brasileira e, a seguir, explicaremos o que foi utilizado para isto. Em nosso arquivo `www/templates/bebidas.html`, fizemos o seguinte:

```
...
<ion-item ng-repeat="bebida in bebidas" href="#/app/detalhar/{{bebida.id}}">
  {{bebida.nome}} - {{bebida.preco.toLocaleString('pt-BR', { style: 'currency', currency: 'BRL'})}}
</ion-item>
...
```

Veja que utilizamos como parâmetro da função `toLocaleString()` o `pt-BR`, determinando que o local é aqui, em nosso país. Em seguida, enviamos um objeto informando qual o estilo (moeda) e finalmente o formato dela (BRL). Experimente colocar EUR no lugar de BRL e observe que o formato será alterado para Euro.

Agora volte para realidade e aceite o Real como moeda. Lembre-se de fazer isto em todos os lugares que representam valores no aplicativo: tela de bebidas, petiscos e sucos, e modal da bandeja.

Tudo pronto, tente efetuar alguns pedidos e abrir a bandeja para visualizar os itens pedidos com o cálculo efetuado.

## 7.3 EDITAR OS ITENS DA BANDEJA

Para editar, daremos ao usuário apenas a ação de informar uma nova quantidade ao item. Isto é bem similar à sua inserção, em que o usuário seleciona um item no cardápio e informa a quantidade que ele deseja. Tendo em vista isto, vamos começar por esta função colocando-a no ícone `Editar`.



Figura 7.3: Ícone Editar

Construiremos logo todo JavaScript envolvido no `www/js/controller.js`. É muito simples! Para isto, basta adicionar uma ação de abrir um modal com um *number picker*, similar ao que foi construído anteriormente, quando se efetua um pedido. Vamos dar a este *template* o nome de `modal-edicao.html`. Nossa arquivo ficará assim:

```
...
$scope.input = {};
$scope.input.quant = 1;
$scope.data = {};

$ionicModal.fromTemplateUrl('modal-edicao.html', {
  scope: $scope,
```

```

        animation: 'slide-in-up'
    }).then(function(m){
        $scope.modalNumber = m;
    });

$scope.cancelarEdicao = function(){
    $scope.modalNumber.hide();
    $scope.input.quant = 1;
}

$scope.editarItem = function(){
    $scope.input.quant = Sessao.bandeja[$scope.data.item].qua
ntidade;
    $scope.modalNumber.show();
}

$scope.confirmarEdicao = function(){
    //Para fazer
}
...

```

Observe que ele será bem parecido com o modal de pedir item do capítulo anterior, porém demos a ele um nome diferente em sua referência, `modalNumber`, para que não se confunda com um modal da bandeja já construído neste *controller*. Poucas são as novidades neste código: uma função de cancelar, na qual simplesmente fecha-se o modal; e nossa função `editarItem`, que receberá um item do `radio button` selecionado. Para isto, decidimos pegar o item no array pelo seu `index`, da seguinte forma:

`Sessao.bandeja[INDEX]`

Jogaremos o valor da quantidade do item na variável `input.quant` declarada como modelo do *number picker*, e depois exibiremos o modal (`modalNumber.show()`) com este valor, dando assim a opção do usuário incrementar ou decrementar a quantidade do pedido. Logo em seguida, construímos uma função

vazia (`confirmarEdicao()`), em que faremos futuramente a ação de confirmar a alteração, ao atualizar os dados da lista.

Por enquanto, deixe-a em *stand-by* e vamos ver como passar o item selecionado para que seja possível recuperá-lo, e assim editá-lo. Preste bastante atenção à variável `$scope.data` que foi declarada no início, e usada na função `editarItem()` para recuperar o item do *array*.

Agora vamos ao código que deve ser alterado em nossa lista implementada por `radio buttons`, em `www/templates/menu.html`. Lembra-se deles? Pois vamos lhe mostrar a mágica:

```
...
<ion-list ng-repeat="it in bandeja" >
    <ion-radio ng-value="bandeja.indexOf(it)"
        ng-model="data.item">
...

```

Veja que usamos as diretivas do AngularJS, sendo que já explicamos algumas neste mesmo capítulo: `ng-value` receberá um valor. Esta será a indexação do *array* JavaScript utilizado para inserir os itens.

Lembre-se de que, em JavaScript, o *array* é inicialmente indexado pelo valor 0 (zero), portanto, o *index* dos itens segue a seguinte ordem: 0, 1, 2, 3 e assim sucessivamente. A função usada para isto é a `indexOf()` da própria linguagem. Passa-se o item como parâmetro, e assim é retornado o *index* que será atribuído à diretiva `ng-value`, como já foi dito.

A diretiva `ng-model` será nosso *data-biding* para o item, ou seja, através dela será possível recuperar o valor que foi selecionado

no radio button ao clicar no botão que contém a ação `editarItem()`. Tudo prontinho pelo lado *view* e pelo lado *controller*.

Agora vamos construir nosso modal ainda em `www/templates/menu.html`, cujo nome será `modal-edicao.html`. Abaixo do nosso último modal, insira um template como a seguir:

```
...
<!-- Modal Edição -->
<script id="modal-edicao.html" type="text/ng-template">
  <ion-modal-view>
    <ion-content>
      <div class="card">
        <div class="item txt-centralizado">
          <p>Informe a quantidade</p>
          <p>e confirme este pedido.</p>
          <br />
          <h2>{{item.nome}}</h2>
          <p><h-number value="input.quant" min="1" max="100" step="1"></h-number></p>
        </div>
        <div class="margem-imagem"></div>
      </div>
      <div class="bar bar-footer">
        <div class="button-bar">
          <button class="button icon-right ion-checkmark button-positive" ng-click="confirmarEdicao()">Confirmar</button>
          <button class="button icon-right ion-checkmark button-assertive" ng-click="cancelarEdicao()">Cancelar</button>
        </div>
      </div>
    </ion-content>
  </ion-modal-view>
</script>
<!-- Fim/Modal -->
...
```

Sem nenhum segredo ou novidade, este modal é praticamente igual ao do pedido. Alteramos apenas os nomes das funções para a

diretiva `ng-click` do botões: `confirmarEdicao()` e `cancelarEdicao()`. Execute o aplicativo, insira um pedido, abra a bandeja selecionando o pedido e clique no botão de edição. Se tudo correr bem, só precisamos construir a ação do botão `Confirmar`.

## 7.4 A FUNÇÃO DE CONFIRMAR EDIÇÃO

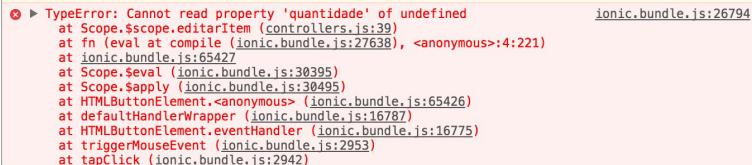
Tudo o que queremos agora é recuperar o objeto inserido no `array`, e em seguida alterar a quantidade, fazendo um novo cálculo do valor e multiplicando o preço pela quantidade. Para isto, vamos voltar ao `www/js/controller.js` e implementar a função, até então, vazia: `confirmarEdicao()`.

```
...
$scope.confirmarEdicao = function(){
    Sessao.bandeja[$scope.data.item].quantidade = $scope.input.quant;
    Sessao.bandeja[$scope.data.item].valor = $scope.input.quant *
    Sessao.bandeja[$scope.data.item].item.preco;
    $scope.modalNumber.hide();
}
...
```

Viu que não há segredo? Apenas recuperamos o item selecionado com `$scope.data.item` e fizemos as devidas operações. Primeiro inserimos a nova quantidade informada, e depois pegamos o preço do item e multiplicamos pela nova quantidade. Após isto, fechamos o modal e assim voltamos à tela anterior.

Observe a mágica do *data binding* explicado neste livro onde o próprio AngularJS é capaz de atualizar tudo, sem que haja a necessidade de se dar um *refresh*, ou recuperar os novos valores. Tudo é feito de forma transparente e contínua.

Opa! O pessoal do teste falou que existe um bug em nossa aplicação. Eles disseram que se entrar na tela para edição e clicar no botão `Editar`, sem selecionar algum item, aparecerá uma exceção no `browser` com a seguinte mensagem:



```
✖ ▶ TypeError: Cannot read property 'quantidade' of undefined
    at Scope.$scope.editarItem (controllers.js:39)
    at fn (eval at compile (ionic.bundle.js:27638), <anonymous>:4:221)
    at ionic.bundle.js:65427
    at Scope.$eval (ionic.bundle.js:30395)
    at Scope.$apply (ionic.bundle.js:30495)
    at HTMLButtonElement.<anonymous> (ionic.bundle.js:65426)
    at defaultHandlerWrapper (ionic.bundle.js:16787)
    at HTMLButtonElement.eventHandler (ionic.bundle.js:16775)
    at triggerMouseEvent (ionic.bundle.js:2953)
    at tapClick (ionic.bundle.js:2942)
```

Figura 7.4: Sem selecionar

Ok! Isso parece uma exceção nos informando que não há uma propriedade `quantidade`. Ela está indefinida. Vamos tentar contornar isto obrigando um item a ser selecionado por padrão, sempre que o modal da bandeja for aberto. Vamos eleger o item com `index 0` (zero). Basta, para isso, declarar nossa variável antes do modal `bandeja`. Na ação de abertura, diremos a ele que o `$scope.data.item` é igual a zero. Vejamos como ficará nossa alteração:

```
...
//Declaração da variável antes de ser utilizada
$scope.data = {};
$ionicModal.fromTemplateUrl('bandeja.html', {
  scope: $scope,
  animation: 'slide-in-up'
}).then(function(m){
  //Selecionar por padrão o zero
  $scope.data.item = 0;
  $scope.modal = m;
});
...
```

Agora tudo estará tranquilo com o pessoal de testes. Espero que nos dê um desconto na nossa próxima funcionalidade. Vamos construir a exclusão dos itens e aguardarmos ansiosos que tudo termine sem *bugs*.

## 7.5 EXCLUSÃO DE ITENS

Para excluir itens da lista, vamos implementar uma função que atenderá ao botão com ícone de lixeira da bandeja. Pensando em um aplicativo profissional, mais detalhado, teríamos de planejar uma tela de confirmação após o clique de exclusão. Isto poderia ser feito com um novo modal.

Faremos de uma outra forma quando chegarmos no capítulo *Recursos*, onde usaremos alguns recursos que o **Cordova** nos oferece através da instalação de *plugins*. Por enquanto, vamos abstrair uma confirmação, partindo do pressuposto que, ao clicar no excluir, o usuário saiba realmente o que está fazendo.

Utilizaremos apenas a função JavaScript `splice(pos, num)`. Ela é responsável por remover um ou mais itens em um *array*. O parâmetro `pos` determina a posição em que se deve iniciar a remoção, e o `num` determina o número de elementos a partir daquela posição. Em nosso caso, removeremos apenas o item selecionado pelo `radio button`.

Isto lembra algo? Sim! Isso mesmo! Temos a posição do item no *array* através do `$scope.data.item` de nosso `www/js/controllers.js`. Então passaremos apenas o valor fixo 1 (um) para o segundo parâmetro. Vamos lá?

...

```
$scope.removeItem = function(){
    Sessao.bandeja.splice($scope.data.item,1);
}
...
```

O código dispensa comentários, não é mesmo?! Apenas usamos o `splice(pos, num)` explicado no *array* da bandeja, removendo apenas o item que estará selecionado pela diretiva `ng-model` do `radio button`. Agora vamos chamar esta função no botão do modal a seguir:



Figura 7.5: Botão Excluir

O código em `www/templates/menu.html` ficará da seguinte forma:

```
...
<button class="button" ng-click="removerItem()><i class="ion-ios-trash-outline"></i></button>
...
```

Acho que o pessoal do teste não encontrará nenhum *bug* dessa vez. Execute os seus próprios testes.

## 7.6 CONFIRMANDO OS PEDIDOS

Já sabemos como alterar nossa bandeja, excluindo itens ou atualizando as quantidades, agora vamos aprender a confirmar. Para isto, precisamos planejar como foi feito tudo até aqui. O planejamento faz parte de qualquer processo de desenvolvimento

de software. Mesmo que você não o coloque no papel, de alguma forma, o planejamento estará implícito em sua cabeça.

Neste caso, vamos planejar implicitamente que, ao clicar no botão confirmar, os pedidos sejam enviados para um "serviço". Como não temos um serviço de fato, até agora o artifício que utilizamos para simular isto foram os *mocks*. Lembra-se deles?!

Mas os *mocks* são valores fixos e, em nosso caso, deverá ser uma variável dinâmica parecida com nossa bandeja. Vamos guardá-la também em toda aplicação, enquanto ela estiver aberta, pois simularemos que nela estará o histórico de pedidos do cliente.

Para iniciarmos, vamos aproveitar o objeto tipo *value* `Sessao`, criado no capítulo *Fazendo pedidos*, em `www/js/services/sessao.js`, e colocar esta variável. Daremos o nome de `historicoPedidos` para este *array*.

```
angular.module('starter.controllers').value('Sessao', {  
    mesaID: "123",  
    bandeja: [],  
    historicoPedidos: []  
});
```

Agora que temos um *array* mantido na sessão, toda vez que confirmarmos a bandeja, vamos adicionar todos os itens neste *array* e limpá-los da bandeja. Portanto, a ação de transferência desses *arrays* é muito simples.

Até aqui, já falei em algumas funções para trabalhar com *array* em JavaScript. Elas são muito úteis e vale a pena dar uma olhada em todas que se encontram disponíveis na sessão *Para saber mais*.

Utilizaremos mais uma: `Array.concat(array)`. Esta função simplesmente concatena um *array* que é passado como parâmetro

e retorna um novo com todos os elementos. Como nosso objetivo é justamente copiar todos os pedidos da bandeja e jogá-los em outro objeto de sessão, então ela será muito útil.

Manteremos assim o estado antigo deste *array* e criaremos sempre um novo com itens adicionados no final dele. Feito isto, deveremos limpar e fechar nossa bandeja. Para a limpeza de nossa lista, usaremos a função já conhecida *splice*. Vamos à codificação no [www/js/controllers.js](#).

```
...
$scope.historicoPedidos = Sessao.historicoPedidos;
$scope.pedir = function(){
    Sessao.historicoPedidos = Sessao.historicoPedidos.concat(Sessao.bandeja);
    Sessao.bandeja.splice(0,Sessao.bandeja.length);
    $scope.historicoPedidos = Sessao.historicoPedidos;
    $scope.modal.hide();
}
...
...
```

Analizando o código, o leitor pode verificar a junção dos *arrays* em *concat* e seu retorno para a variável *historicoPedidos*. Logo abaixo, limpamos o *array* com o *splice* começando pelo primeiro elemento (0) e finalizando em seu tamanho total (*Sessao.bandeja.length*). Em seguida, atualizamos a variável *\$scope.historicoPedidos* com o novo array para que tenhamos a referência atualizada. Por fim, fechamos a bandeja com o *hide* do modal. Fácil, não é mesmo?!

## 7.7 CONCLUSÃO

Espero que tenhamos chegado até aqui com uma boa bagagem de utilização do *Ionic framework*, onde há várias tecnologias envolvidas que, juntas, formam um paradigma inovador de

desenvolvimento de aplicativos: os Apps híbridos. Vimos um pouco de cada coisa como, JavaScript, AngularJS, CSS, HTML, Cordova etc.

No próximo capítulo, vamos aprender a simular nosso pedido de conta, fazendo a recuperação dela, e dando a possibilidade de o cliente pedir que seja finalizada ou pagá-la. Não desista, estamos chegando ao final do livro e sairemos dele com um conhecimento imensurável.

Ao final, de cortesia, demonstrarei em um capítulo especial e fora do contexto de nosso caso de uso como utilizar um recurso mais avançado: a câmera. Além disto, teremos também um capítulo dedicado a recursos avançados fornecidos pelo *framework*.

## 7.8 PARA SABER MAIS

### 1. Grids

<https://ionicframework.com/docs/components/#grid>

### 2. Ícones – <http://ionicons.com>

### 3. Typeof JavaScript – <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof>

### 4. `toLocaleString()` – [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Number/toLocaleString](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Number/toLocaleString)

### 5. Currency ISO – <https://www.currencyiso.org/en/home/tables/table-a1.html>

### 6. Operações com array em JavaScript – [https://www.w3schools.com/jsref/jsref\\_obj\\_array.asp](https://www.w3schools.com/jsref/jsref_obj_array.asp)

## CAPÍTULO 8

# A CONTA

Neste capítulo, consideraremos que o usuário precisa encerrar sua conta e pagá-la com uma integração via cartão de crédito. Visto que os garçons demoram para trazer a conta hoje em dia, isso pode ser uma boa ideia, não? Portanto, vamos tentar simular esta integração?

Para isto, vamos planejar todo fluxo de tela, como temos feito até aqui. Pediremos ao pessoal do design para que nos apresentem uma tela e veremos o que será possível fazer por eles.

## 8.1 CONSTRUÇÃO DO TEMPLATE

Avaliando os requisitos, podemos afirmar que o layout desta tela será bastante similar à lista da bandeja. Conversamos com o pessoal de design e Analistas de Requisitos, e eles chegaram a uma conclusão: "Vamos construir um rascunho e enviar para vocês desenvolverem". Assim que recebemos, tivemos uma surpresa. Não é que estávamos certos? Realmente vamos aproveitar muita coisa de lá.

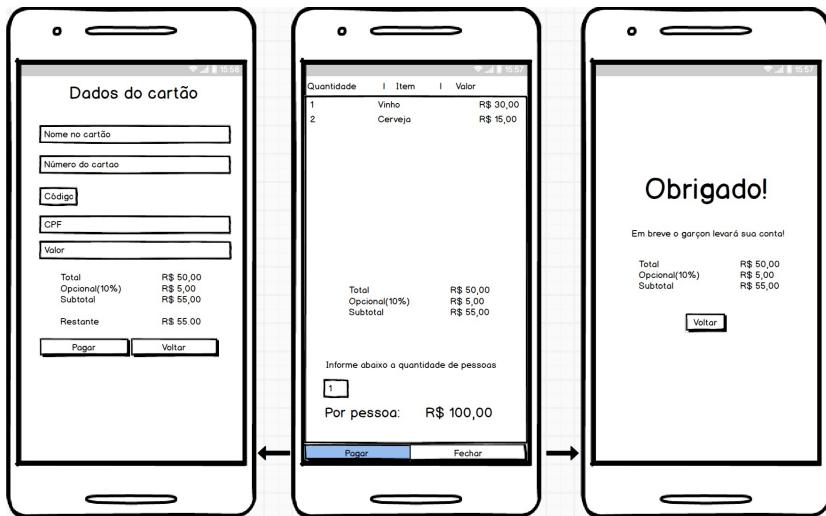


Figura 8.1: Protótipo da conta

Analisando a primeira tela (no protótipo, a segunda da esquerda para direita), verificamos que existe sim uma similaridade com a bandeja, mas com algumas particularidades: um cálculo da conta a pagar, um divisor por pessoas e uma barra de botões logo abaixo para a opção de **Pagar** ou **Fechar** a conta.

Observe as setas para entender os dois fluxos que vamos simular. No cenário em que o usuário clicar no botão de pagar, direcionaremos para uma tela de preenchimento dos dados referentes ao pagamento. Se o usuário resolver apenas fechar a conta, a tela será de agradecimento, supondo que o pedido foi enviado ao caixa para que o garçom conduza a nota até a mesa.

Munido dessas informações, vamos implementar primeiro a tela inicial. Para isto, basta aproveitar um pouco o código do modal da bandeja e ir adaptando. Mas antes, vamos relembrar do

que fizemos nos capítulos *Conhecendo os nossos arquivos* e *Completando o menu*, onde fizemos os menus, mas deixamos de lado a tela de conta. Vamos construir isto no `www/js/app.js`.

```
...
.state('app.conta', {
    url: '/conta',
    views: {
        'menuContent': {
            templateUrl: 'templates/conta.html',
            controller : 'ContaCtrl'
        }
    }
});

...
...
```

Portanto, nosso último `state` será o da conta, em que configuramos o `controller` `ContaCtrl` e o `template` `www/templates/conta.html`. Sem nenhuma novidade, mapeamos a tela da conta para ser representada pela URL `#/app/conta/`. Em nosso `www/templates/menu.html`, podemos fazer assim:

```
...
<ion-list>
    <ion-item menu-close href="#/app/bebidas">
        Bebidas
    </ion-item>
    <ion-item menu-close href="#/app/petiscos">
        Petiscos
    </ion-item>
    <ion-item menu-close href="#/app/sucos">
        Sucos
    </ion-item>
    <ion-item menu-close href="#/app/conta">
        Conta
    </ion-item>
</ion-list>
...
```

Já que declaramos o *template* e o *controller*, só nos resta construí-los. Vamos primeiro ao template. Basta reaproveitar o modal da bandeja que se encontra em `www/templates/menu.html`. Removendo as tags `<script>` e `<ion-modal-view>`, e inserindo `<ion-view>`, você não estará mais colocando um template dentro de um arquivo que contém outros *templates* - visto no capítulo anterior.

Você também não usará mais a tela em forma de modal, com todas funções JavaScript de comandos (fechar e abrir), além de ter de declarar o objeto que representa o modal com `$ionicModal.fromTemplateUrl()`. Vamos criar um template único em um HTML, e pronto. Portanto, como nossa tela não será mais um modal, vamos inserir o código na tag `<ion-view>`. O restante será a mesma coisa (por enquanto) e ficará assim:

```
<ion-view title="Conta">
  <ion-content>
    <div class="card">
      <div class="item txt-centralizado">
        <div class="row">
          <div class="col">
            Quantidade
          </div>
          <div class="col">
            Item
          </div>
          <div class="col">
            Valor
          </div>
        </div>
      </div>
      <ion-list ng-repeat="it in historicoPedidos" >
        <ion-radio ng-value="historicoPedidos.indexOf(it)">
          ng-model="data.item">
        <div class="row">
          <div class="col">
```

```
    {{it.quantidade}}
  </div>
  <div class="col">
    {{it.item.nome}}
  </div>
  <div class="col">
    {{it.valor.toLocaleString('pt-BR', { style: 'currency', currency: 'BRL'})}}
  </div>
  </div>
</ion-radio>
</ion-list>
<br />
</div>
<div class="margem-imagem"></div>
</div>
<div class="bar bar-footer">
  <div class="button-bar">
    <button class="button button-positive" ng-click="pedir()"><i class="ion-ios-checkmark-outline"></i></button>
    <button class="button button-dark" ng-click="editarItem()"><i class="ion-edit"></i></button>
    <button class="button" ng-click="removerItem()"><i class="ion-ios-trash-outline"></i></button>
    <button class="button button-assertive" ng-click="fecharModal()"><i class="ion-ios-undo-outline"></i></button>
  </div>
</div>
</ion-content>
</ion-view>
```

Desta forma, nada muda em relação à bandeja, mas não é exatamente isto que o pessoal de design pediu, não é mesmo? Precisamos adaptar algumas mudanças.

Observe que, com nosso copiar e colar, não estamos omitindo o aprendizado. Estamos fazendo isto em códigos que já foram explicados anteriormente e, a partir deles, vamos evoluir em algo que ainda não conhecemos.

Mesmo com a repetição desses códigos, seria interessante você começar a construir as *views* inserindo os componentes um a um, sem utilizar o recurso de copiar/colar. Isso ajudará a fixar o nome dos componentes e o aprendizado.

Primeiramente, vamos adaptar os botões do rodapé. Não precisamos de tantos, mas apenas dois, e com um texto em vez de ícones. Vamos ao código:

```
...
<div class="bar bar-footer">
  <div class="button-bar">
    <button class="button button-positive">Pagar</button>
    <button class="button button-assertive">Fechar</i></button>
  </div>
</div>
...
```

Veja que excluímos dois botões e alteramos os dois que sobraram: retiramos a tag *<i>* que representava os ícones para dar lugar ao texto que aparecerá no botão. Analisando-os, dá para concluir que são botões sem ação, portanto, inúteis. Precisamos entregar para eles funções que vão agregar valor à nossa funcionalidade. Para isto, mais à frente, construiremos as funções de **pagar** e **fechar** a conta.

Por enquanto, vamos deixar um pouco de lado as funções

JavaScript e nos preocupar com o formato do template, fazendo com que ele reflita a aparência do nosso protótipo. Para isto vamos construir um *controller*, pois precisamos chamar a tela e ver como está ficando. Além disto, este *controller* será usado nas operações da conta.

```
app.controller('ContaCtrl', function($scope){  
});
```

Agora inclua dois itens à bandeja, confirme os pedidos e depois clique em conta no menu. Vejamos o resultado:

Quantidade	Item	Valor
1	Vinho Tinto Brasil	R\$40,20
1	Cerveja Especial	R\$23,00

Figura 8.2: Conta sem ajustes

Ops! Algo estranho aconteceu nesta tela. A conta não deve permitir a opção de selecionar um item, pois não vamos editá-lo. Sugiro que retire o *radio button*, deixando apenas a lista de itens desta forma:

```
...  
<ion-list ng-repeat="it in historicoPedidos" >  
  <div class="row">  
    <div class="col">
```

```

        {{it.quantidade}}
    </div>
    <div class="col">
        {{it.item.nome}}
    </div>
    <div class="col">
        {{it.valor.toLocaleString('pt-BR', { style: 'currency
', currency: 'BRL'})}}
    </div>
</div>
...

```

Agora precisamos acrescentar os campos novos: total, opcional e subtotal. Além disto, colocaremos uma calculadora para dividir a conta entre a quantidade de pessoas informadas em um *input*. Em nossa *view*, usaremos novamente *grids* para organizar estes campos, seguindo a imagem do protótipo.

...

```

<div class="row">
    <h2>Total:</h2>
</div>
<div class="row">
    Opcional(10%):
</div>
<div class="row">
    <h2>Subtotal:</h2>
</div>
<br />
<div class="row">
    Informe abaixo a quantidade de pessoas
</div>
<div class="row">
    <div class="col item item-input col-25">
        <input type="number" />
    </div>
    <div class="col">
        <p><h2>Valor por pessoa:</h2></p>
    </div>
</div>
<div class="margem-imagem"></div>

```

```
<!-- Abaixo fechar o card e insere a barra de botões -->
...

```

Por enquanto, temos nossa tela seguindo o protótipo, mas sem muita função além de listar todos os pedidos que foram confirmados através da bandeja. Pegue agora uma boa xícara de café e vamos avaliar como faremos todos os cálculos que essa tela precisa.

O primeiro é total, que representa a soma de todos os pedidos. Em seguida, devemos calcular 10% sobre este total e, por fim, somar este percentual com o total para fornecer um valor ao usuário.

Estes cálculos são básicos e padrão em contas de qualquer estabelecimento. Vamos construir uma função responsável por executar esta regra de negócio de forma assíncrona, porém rápida, assim que entrarmos na tela.

Teremos de fazer uma função que execute um *loop* no *array* de pedidos, some todos os seus valores e guarde em uma variável. A partir dela, poderemos fazer os outros cálculos.

```
...
$scope.conta = [];
$scope.calcularConta = function(){
    $scope.conta.total = 0;
    $scope.historicoPedidos.forEach(function(v){
        $scope.conta.total = $scope.conta.total + v.valor;
    })
    $scope.conta.porcentagem = $scope.conta.total * 10/100;
    $scope.conta.subtotal = $scope.conta.porcentagem + $scope
    .conta.total;
}

```

Analise com atenção este código. Veja que, antes da função, declaramos um objeto `$scope.conta`. Ele servirá de base para

inserirmos os atributos `total`, `percentual` e `subtotal`. Logo abaixo vem a função propriamente dita, que chamaremos toda vez que a tela de conta for executada.

Você deve estar curioso para saber onde a chamaremos, não é mesmo? Explicaremos mais tarde, por enquanto vamos pensar apenas neste código. Veja que, na chamada, faremos o *loop* na lista de itens, somando o valor de todos os pedidos e guardando na propriedade do objeto `$scope.conta.total`.

Para isto, antes, devemos zerar este total, visto que esta função está no *controller* principal e nunca perderá seu estado. Abaixo do *loop*, temos de fato o cálculo do percentual e o subtotal. Dessa forma, vamos fazer o *data-binding* na tela em `www/templates/conta.html`, da seguinte forma:

```
...
<div class="row">
    <h2>Total: {{conta.total}}</h2>
</div>
<div class="row">
    Opcional(10%): {{conta.porcentagem}}
</div>
<div class="row">
    <h2>Subtotal: {{conta.subtotal}}</h2>
</div>
...
```

Mas e agora? Onde chamaremos nossa função?

Calma! Vamos ao `www/templates/menu.html` para utilizar um conhecimento já adquirido neste livro. O famigerado conceito de diretiva do AngularJS. No menu da conta, devemos ter o componente `<ion-item menu-close>` com o seguinte formato:

```
...
<ion-item menu-close href="#/app/conta"
```

```
    ng-click="calcularConta()">>
  Conta
</ion-item>
...
```

Vejo no código que, através da diretiva `ng-click`, chamamos a função `calcularConta()` de nosso *controller*. Isso é totalmente diferente dos outros itens de menu. O que fizemos foi apenas executar uma função JavaScript – antes de abrir nossa tela –, responsável por executar os cálculos da conta. Simples assim! Agora tudo está muito bem feito e esperando você executar alguns testes.

Eu sei que você leu este livro até aqui e chegou a este ponto sabendo muitos conceitos de uma variedade de plataformas, frameworks e linguagem de programação. Disso tudo, vimos uma característica muito legal do JavaScript, que é a forma de representar a moeda.

Eu também sei que você deve estar achando estranho que não colocamos isto nos novos campos inseridos. Não se preocupe! Foi tudo proposital, meu caro! Deixamos isto para você praticar. Faça bom uso da função `.toLocaleString('pt-BR', { style: 'currency', currency: 'BRL' })` para representar nossa moeda brasileira, e agora sim tudo ficará muito bem ajustado.

## 8.2 VAMOS DIVIDIR A CONTA

Sabe aquela hora em que um amigo diz que está indo embora e pede uma conta parcial? Após este evento, o restante vai saindo na sequência, mas sempre pedindo a conta parcial. Agora é a hora de ver quanto ficará para cada um, já que todos beberam e quem não bebeu segue a máxima: "Sentou, sorriu. A conta dividiu!". Para

facilitar a vida do pessoal, vamos implementar a calculadora na qual será possível digitar apenas uma informação: quantidade de pessoas que pagarão a conta.

Depois de todos os cálculos feitos na função anterior, fazer um cálculo de divisão ficou simples. Vamos fazer um *data-binding* da quantidade de pessoas, e assim calcular o valor da divisão na própria *view* (`www/templates/conta.html`).

```
...
<div class="row">
    <div class="col item item-input col-25">
        <input type="number" ng-model="conta.quantidadePessoa"/>
    </div>
    <div class="col">
        <p><h2>Valor por pessoa: {{conta.subtotal / conta.quantidadePessoa
            .toLocaleString('pt-BR', { style: 'currency', currency: 'BRL'})}}</h2></p>
    </div>
</div>
...
...
```

Observe a diretiva `ng-model` do *input* e veja que ela faz o papel do *data-binding* já explicado. O valor é atualizado constantemente e, na mesma *view*, pegamos este valor e dividimos através da operação entre 4 chaves (`{}{}`). Observe que, depois da divisão, utilizamos a função do JavaScript para formatar novamente em nossa moeda **Real**, com dois dígitos decimais após a vírgula.

## 8.3 PAGAR OU PEDIR?

Quando o usuário entrar na funcionalidade de conta, conforme o protótipo, ele terá de escolher entre pagar através dos dados do cartão de crédito, ou efetuar o pedido de fechar a conta. Estas duas

ações serão executadas através dos dois botões, já inseridos na tela. Para isto, precisamos construir dois modais que representarão estas duas funcionalidades e chamar de acordo com a ação escolhida.

Pensando dessa forma, vamos inserir os dois modais no mesmo *template*, `www/templates/conta.html`, como já fizemos com a bandeja. A inserção destes modais serão pela tag `<script type="text/ng-template">`, não esquecendo que cada um terá um *id*.

Vamos dividir o fonte em duas partes para não ficar muito longo. Portanto, vamos ao fonte do modal que representará o pedido de fechar a conta.

```
...
<script id="modal-fechar-conta.html" type="text/ng-template">
  <ion-modal-view>
    <ion-content>
      <div class="card">
        <br />
        <div class="row">
          <div class="col col-offset-25">
            <h1>Obrigado!</h1>
          </div>
        </div>
        <div class="margem-imagem"></div>
        <div class="txt-centralizado">Em breve o garçom levará sua conta!</div>
        <div class="margem-imagem"></div>
        <div class="row">
          <div class="col-offset-33">
            Total: {{conta.total.toLocaleString('pt-BR', { style: 'currency', currency: 'BRL'})}}
          </div>
        </div>
        <div class="row">
          <div class="col-offset-33">
            Opcional(10%): {{conta.porcentagem.toLocaleString('pt-BR', { style: 'percent', precision: 0 })}}
          </div>
        </div>
      </div>
    </ion-content>
  </ion-modal-view>
</script>
```

```

('pt-BR', { style: 'currency', currency: 'BRL'}})}}
        </div>
    </div>
    <div class="row">
        <div class="col-offset-33">
            Subtotal: {{conta.subtotal.toLocaleString('pt-BR')}}
        , { style: 'currency', currency: 'BRL'}})}}
            </div>
        </div>
        <br />
    <div class="row">
        <div class="col col-offset-33">
            <button type="button" class="button button-positive
button-block">Voltar</button>
        </div>
        <div class="col"></div>
    </div>
    </div>
</ion-content>
</ion-modal-view>
</script>
...

```

O código contém poucas novidades, mas algo novo poderá lhe chamar a atenção. Organizamos os elementos com *grids* através das classes `row` e `tag`. Existe no framework uma classe usada para deslocar uma coluna dentro de um *grid*. Essa classe é a `col-offset-xx`, em que `x` é o percentual que queremos deslocar.

A seguir, veremos uma tabela com os percentuais de deslocamento e suas respectivas classes:

Colunas offset – Nome das classes com percentual	%
.col-offset-10	10%
.col-offset-20	20%
.col-offset-25	25%
.col-offset-33	33.3333%
.col-offset-50	50%

.col-offset-67	66.6666%
.col-offset-75	75%
.col-offset-80	80%
.col-offset-90	90%

Se colocado na prática, teremos a seguir um exemplo do deslocamento.

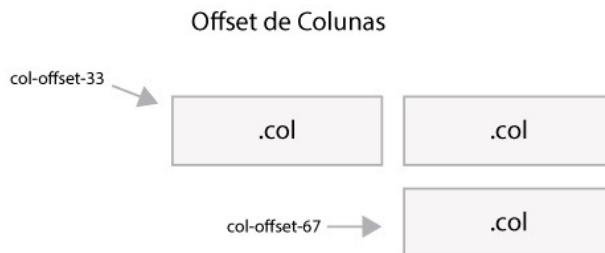


Figura 8.3: Grid com offset

Utilizamos deste artifício também para deslocar o botão Voltar do modal até o meio da tela, mas para isto, deveríamos preencher a coluna com este botão. A classe utilizada foi a `button-block`. Ela determina que o botão preencherá toda a coluna desta linha. Se fizermos dessa forma apenas, acontecerá a seguinte situação:

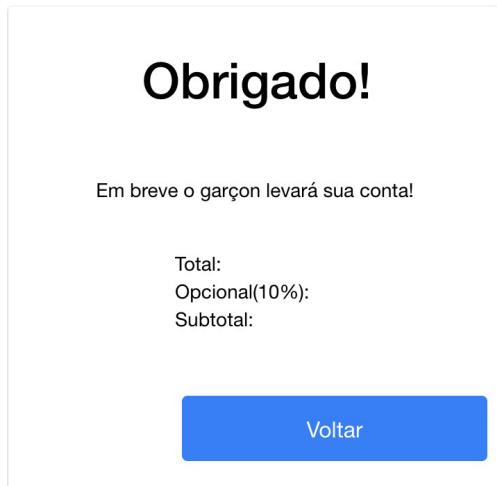


Figura 8.4: Botão deslocado

Veja que o botão de fato preencheu toda coluna a partir do `col-offset-33`, porém como não encontrou nenhuma coluna à direita dele, ele extrapolou e o efeito não ficou satisfatório para nosso *design*. Buscando uma solução inteligente, pensamos em uma coluna *fake* à direita deste botão. Dessa forma, cria-se a seguinte tag `<div class="col"></div>`, que determina que existe uma coluna, mas sem nenhum elemento nela.

Com tudo pronto para abrir o modal, só nos resta declará-lo no JavaScript do *controller* `www/js/conta-controller.js`. Também precisaremos chamar a função de abrir no botão `Figar` da tela de conta, e a função de fechar no botão `Voltar` do modal. O código da declaração ficará assim:

```
...
$ionicModal.fromTemplateUrl('modal-fechar-conta.html', {
  scope: $scope,
  animation: 'slide-in-up'
}).then(function(modal){
```

```
$scope.modalFecharConta = modal;  
});  
...
```

Tudo muito parecido por enquanto, não é mesmo? Observe que declaramos o modal que foi construído na tag `<script>` com `id` igual a `modal-fechar-conta.html`, e demos a ele o nome de `modalFecharConta` que ficará no objeto `$scope`. Agora vamos fazer algo diferente.

Lembra de quando construímos a bandeja e, além de declarar o modal, construímos também duas funções: `fechaModal()` e `abreModal()`? Esse código muitas vezes faz apenas o papel de encapsular as funções `hide()` e `show()`, que já estão prontas no framework. Inteligentemente, vamos apenas chamá-las nas diretivas `ng-click` de nossos botões em vez de sair por aí escrevendo códigos para fechar e abrir modais, espalhados por todas as classes.

Se você sentir a necessidade de criar alguma regra de negócio, validação ou até mesmo carregar alguma informação antes dos eventos de abrir e/ou fechar os modais, aconselho que construa sua função dentro do *controller*.

No botão `Fechar`, colocamos:

```
...  
<!-- Botões para fechar a conta -->  
<div class="bar bar-footer">  
  <div class="button-bar">  
    <button class="button button-positive">Pagar</button>  
    <button class="button button-assertive"  
      ng-click="modalFecharConta.show()">Fechar</i>
```

```
        </button>
    </div>
</div>
...

```

E no botão de voltar do modal:

```
...
<div class="row">
<div class="col col-offset-33">
    <button type="button"
            class="button button-positive button-block"
            ng-click="modalFecharConta.hide()">
        Voltar
    </button>
</div>
<div class="col"></div>
</div>
...

```

Agora sim! Nossa modal já pode ser testado. Adicione alguns itens na bandeja, depois confirme os pedidos e veja a simulação ocorrer como planejamos. Estamos simulando que o serviço estará guardando todos os itens confirmados e, ao final, nós teremos todo o histórico de pedidos na memória, em um caso real, em algum *webservice* na nuvem.

Então, vamos construir o nosso outro modal, o que terá a função de *pagar* a conta. Este nos dará os fundamentos necessários para a criação de *forms* e o leitor conhecerá a facilidade que o *framework* nos fornece para este componente.

Antes de começarmos, lembre-se de que tudo será montado em mais um modal. Portanto, vamos iniciar declarando-o em `www/js/conta-controller.js`, entretanto com outro nome para o *template*: `modal-pagar.conta.html`.

...

```

$ionicModal.fromTemplateUrl('modal-pagar-conta.html', {
  scope: $scope,
  animation: 'slide-in-up'
}).then(function(modal){
  $scope.modalPagarConta = modal;
});
...

```

Feito isto, vamos construir o modal dentro do mesmo arquivo da conta. Em `www/templates/modal-pagar-conta.html`, declararemos a tag `<script>` com a seguinte estrutura:

```

...
<!-- Modal pagar conta -->
<script id="modal-pagar-conta.html" type="text/ng-template">
  <ion-modal-view>
    <ion-content>
      <div class="card">
        <div class="margem-imagem"></div>
        <div class="row">
          <div class="col-offset-25"><h3>Dados do cartão</h3>
        </div>
        <div>
          <div class="row">
            <div class="col-offset-33">
              Total: {{conta.total.toLocaleString('pt-BR', { style: 'currency', currency: 'BRL'})}}
            </div>
          </div>
          <div class="row">
            <div class="col-offset-33">
              Opcional(10%): {{conta.porcentagem.toLocaleString('pt-BR', { style: 'currency', currency: 'BRL'})}}
            </div>
          </div>
          <div class="row">
            <div class="col-offset-33">
              Subtotal: {{conta.subtotal.toLocaleString('pt-BR', { style: 'currency', currency: 'BRL'})}}
            </div>
          </div>
          <br />
        <div class="row">

```

```
<div class="col col-offset-33">
    <button type="button" class="button button-positive
button-block" ng-click="modalPagarConta.hide()">Voltar</button>
</div>
<div class="col"></div>
</div>
</ion-content>
</ion-modal-view>
</script>
...
```

Por enquanto, este modal apresenta apenas os valores: total, opcional e subtotal e o botão Voltar . Fizemos isto para reaproveitar um pouco do código do modal anterior, acrescentamos um *offset* de coluna para o título de 25%, mas ainda precisamos ajustar no nosso protótipo.

Vamos começar inserindo um Pagar ao lado do Voltar . Para isto, vamos retirar a classe col-offset-33 , pois não há mais a necessidade de deslocar uma coluna com 33%. O que organizará os botões aqui serão duas colunas, lado a lado.

Poderíamos utilizar uma barra de rodapé com os botões como foi feito na primeira tela. Fizemos desta forma apenas com o intuito de aprender novos conceitos de organização dos elementos.

Veja como ficará o código dos botões a seguir:

```
...
<div class="row">
    <div class="col">
        <button type="button" class="button button-positive button-
```

```
block">Pagar</button>
  </div>
  <div class="col">
    <button type="button" class="button button-positive button-block" ng-click="modalPagarConta.hide()">Voltar</button>
  </div>
</div>
...
```

Simples, não é mesmo?! Uma linha (`row`) e duas colunas na mesma linha contendo cada uma um botão. Por enquanto, o `Pagar` está sem ação, mas daqui a pouco veremos o que vamos fazer com ele. Por agora, apenas vamos focar em construir o *form* com seus *inputs*.

No *Ionic*, existem várias classes para os formulários. Existe uma documentação ampla sobre *inputs* com *labels* ao lado, encapsulados, *placeholders*, ícones, entre outras características que podemos dar aos elementos do formulário. Neste livro, trataremos apenas dos principais, mas se o leitor quiser se aprofundar, poderá consultar a seção *Para saber mais* ao final deste capítulo.

O `input` no HTML tem um atributo chamado `placeholder` que nada mais é do que uma descrição curta sobre o dado esperado naquele campo. Por exemplo, o código `<input placeholder="Nome Completo">` se apresentaria da seguinte forma:

A screenshot of a mobile application interface showing a single input field. The field has a light gray background and contains the placeholder text "Nome Completo" in a medium-sized, dark gray font.

```
Nome Completo
```

Figura 8.5: Placeholder

Vamos distribuir nossos *inputs* em nosso *form* com os *labels* em formato de *placeholder*. Além disto, ficaria mais interessante se,

ao preencher os dados, o texto animasse e subisse ao topo do campo. Normalmente, por padrão do *placeholder*, este *label* sumiria. Todo este desejo pode ser construído com facilidade através do *Floating Labels* do *Ionic*. A seguir, veremos como ficará nosso código e, em seguida, uma explanação mais detalhada.

```
...
<div class="list">
  <label class="item item-input item-floating-label">
    <span class="input-label">Nome no Cartão</span>
    <input type="text" placeholder="Nome no Cartão">
  </label>
  <label class="item item-input item-floating-label">
    <span class="input-label">Número do Cartão</span>
    <input type="number" placeholder="Número do Cartão">
  </label>
  <label class="item item-input item-floating-label">
    <span class="input-label">Código de Segurança</span>
    <input type="number" placeholder="Código de Segurança">
  </label>
  <label class="item item-input item-floating-label">
    <span class="input-label">CPF</span>
    <input type="number" placeholder="CPF">
  </label>
  <label class="item item-input item-floating-label">
    <span class="input-label">Valor a pagar</span>
    <input type="number" placeholder="Valor a pagar">
  </label>
</div>
...
```

Cada campo é composto por um *label* , um *span* e um *input* . Todos eles encapsulados em uma *div* com a classe *list* . Os *labels* são configurados com as classes *item* , *item-input* e *item-floating-label* . Vamos explicar cada uma dela:

1. **Classe *list*** : é a forma simples de exibir uma lista. Esta é a forma mais comum de exibir conteúdos como texto simples,

botões, imagens, *inputs* e rádios.

2. **Classe `item`** : representa um novo item na lista, como se fosse uma nova linha, traçando assim uma borda para separá-los.
3. **Classe `item-input`** : configura o item para ter uma aparência de *input*.
4. **Classe `item-floating-label`** : responsável por informar ao *framework* que este item é um *Floating Label*, citado anteriormente, causando assim o efeito de "flutuar" o *placeholder* para cima do *input*. Observe a seguir uma ilustração deste efeito.

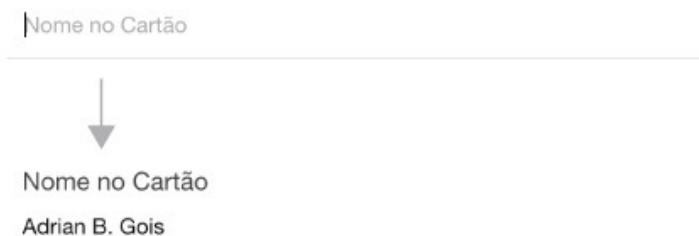


Figura 8.6: Efeito Floating Label

Não será o foco deste livro tratar de exceções ou validações de dados. Portanto, vamos considerar que o usuário vai preencher todos os dados e de forma correta. Mas fica a dica para que você pesquise sobre *validações de campos em AngularJS*, e tente fazer o caso de uso de uma forma mais elaborada. Consulte a seção **Para saber mais** deste capítulo para ver uma referência do W3schools sobre isto.

Já temos nosso *form* com uma boa aparência, agora nos resta fazer um último ajuste: aquele campo no protótipo do início deste capítulo chamado `Restante`, onde deverá ser um *data-binding* seguindo uma regra que terá como base o campo `Valor`. Na prática, isto quer dizer que, se uma pessoa na mesa paga a parte a ele destinada, o restante vai ser exibido neste campo até que a conta seja totalmente paga.

Para isto, vamos inserir primeiro na diretiva `ng-model` do campo `Valor` a variável para guardá-lo:

```
...
<label class="item item-input item-floating-label">
  <span class="input-label">Valor a pagar</span>
  <input type="number" placeholder="Valor a pagar" ng-model="conta.valorPagar">
</label>
...
```

Em seguida, inserimos a operação no campo `Restante` da seguinte forma:

```
...
<div class="row">
  <div class="col-offset-33">
```

```
        Restante: {{{conta.subtotal - conta.valorPagar).toLocaleString('pt-BR', { style: 'currency', currency: 'BRL'})}}}
    </div>
</div>
...
```

Tudo pronto, vamos apenas simular que a conta foi paga. Quando o usuário clicar no botão, devemos inserir um novo item no histórico de pedidos. Este novo item será o pagamento e terá um valor negativo, para que, ao gerar uma nova conta, o valor deste pagamento seja subtraído do total.

Para isto, vamos inserir a codificação de adicionar um novo item à lista de pedidos em `www/js/conta-controller.js`.

```
...
$scope.pagar = function(){
    var valor = -$scope.conta.valorPagar;
    Sessao
        .historicoPedidos
            .push({item: { nome : "Pagamento"}, quantidade : 1, v
alor: valor});
    $scope.modalPagarConta.hide();
}
...
```

Não devemos esquecer da injeção do objeto `Sessao` no controller:

```
app.controller('ContaCtrl', function($scope, $ionicModal, Sessao)
{...
```

Inserimos assim um valor negativo e demos o nome de `Pagamento`. Através do `push` do array, adicionamos este novo item à lista `historicoPedidos`, e depois fechamos o modal.

Agora chamaremos essa função através do botão `Pagar` e sua diretiva `ng-click`.

```
...
<div class="col">
    <button type="button"
        class="button button-positive button-block"
        ng-click="pagar()">
        Pagar
    </button>
</div>
...

```

Depois de tudo implementado, chegou a hora de testar o fluxo da aplicação. Faça um pedido, confirme-o depois de ser inserido na bandeja e acesse a conta para pagar parcial ou totalmente, como também efetuar o pedido através do botão `Fechar`. Se tudo ocorreu bem até aqui, seu aplicativo está pronto e nosso caso de uso chegou ao fim.

## 8.4 CONCLUSÃO

Neste capítulo, encerramos nosso caso de uso. Ele foi implementado através do framework para aplicações híbridas: *Ionic* versão 1. Conhecemos alguns conceitos básicos sobre toda tecnologia envolvida no *framework*, seus conceitos e peças chaves, além de utilizarmos muitos componentes CSS que o próprio *Ionic* fornece.

Temos mais dois capítulos pela frente, em que vou explanar o uso da câmera do dispositivo através da instalação do plugin do **ngCordova** (capítulo *Usando a câmera*), e falarei de mais alguns recursos que poderemos explorar em nossos aplicativos (capítulo *Recursos*).

Espero que tenha gostado do que viu até agora, e seguiremos nessa viagem até o último capítulo.

## 8.5 PARA SABER MAIS

### 1. Forms

<https://ionicframework.com/docs/components/#forms>

### 2. Lists – <https://ionicframework.com/docs/components/#list>

### 3. Validações AngularJS

[https://www.w3schools.com/angular/angular\\_validation.asp](https://www.w3schools.com/angular/angular_validation.asp)

## CAPÍTULO 9

# USANDO A CÂMERA

Neste capítulo, vamos aprender a utilizar um dos recursos mais explorados do dispositivo: a câmera. Para isto, vamos esquecer um pouco o *browser* em relação aos testes da aplicação, pois o *plugin* requer um dispositivo real, ou um emulador, para executar a aplicação corretamente utilizando a câmera.

Portanto, o recurso escolhido para testarmos será um dispositivo móvel: de preferência um celular ou um tablet com *Android*. Retornando ao capítulo *Iniciando nossa aplicação*, vimos que é preciso plugar um dispositivo na porta USB, e assim executar o comando `run` para que o framework faça o *build* da aplicação e instale (no dispositivo).

## 9.1 INSTALAÇÃO DO PLUGIN

Com a ascensão do *AngularJS* e do *Cordova*, surgiu uma coleção de *plugins* para rodar na junção destas duas plataformas. Todos eles estão no site **ngCordova**, em <http://ngcordova.com>, e até a escrita deste livro possuía mais de 70 *plugins*.

O que estes *plugins* fazem é apenas trazer para a linguagem JavaScript a possibilidade de utilizar as APIs do dispositivo, como câmeras, GPS, *filesystem*, *touch id*, entre outros. Tudo isto sendo

implementado em poucas linhas de código.

Para começar a usar estes recursos, você deve instalar o *plugin* da câmera no diretório da aplicação. Assim, use o comando `cordova plugin add cordova-plugin-camera`. Ele faz parte dos *plugins* disponíveis em ngCordova, portanto, precisamos deste recurso também instalado em nossa aplicação.

O comando utilizado para isto é o `bower install ngCordova`, ou seja, através do repositório bower instalamos o ngCordova. Por fim, vamos importá-lo na aplicação através do arquivo `www/index.html`:

```
...
<script src="lib/ngCordova/dist/ng-cordova.js"></script>
<script src="cordova.js"></script>
...
```

Certifique-se de que o `cordova.js` esteja importado também em sua aplicação, conforme linha de código anterior. O ngCordova depende desta API para funcionar.

Como nosso caso de uso do Cardápio Móvel foi concluído no capítulo anterior, consideramos que o leitor já está apto a construir uma aplicação nova, ou utilizar os conceitos aprendidos neste capítulo, reutilizando a mesma estrutura do aplicativo construído.

## 9.2 CONSTRUÇÃO DE ARTEFATOS

Até aqui, nosso livro foi baseado em um caso de uso. Todas as

funcionalidades foram baseadas em dois artefatos principais: a tela (*view*) e o *controller*. Nesta funcionalidade, não será diferente. Precisamos criar estes dois arquivos e construir a funcionalidade baseada neles.

Através do arquivo `www/js/app.js`, onde configuramos todos os *states* da aplicação, você deve configurar a chamada para nossa tela que será baseada no seguinte protótipo:

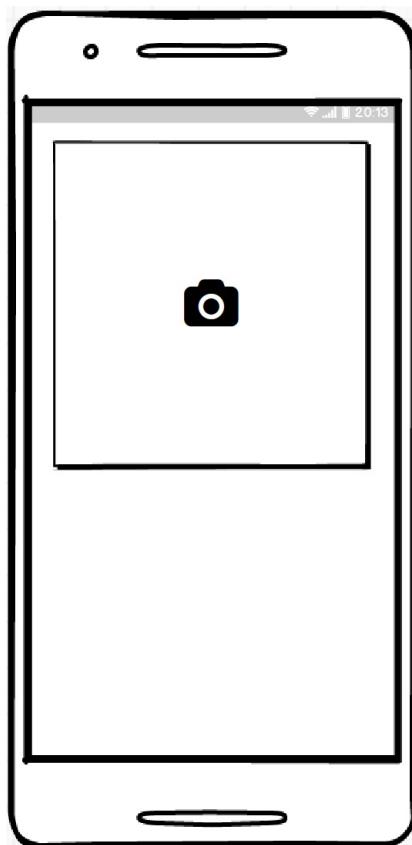


Figura 9.1: Protótipo câmera

Ufa! O pessoal do *design* pegou leve dessa vez. O que precisamos é de apenas um ícone dentro de um botão? Estamos até desconfiados deles, não é mesmo?

Bom! Então, vamos lá. Para que o ícone fique dentro de um botão tão grande assim, precisamos criá-lo maior. Isto só é possível através da manipulação via CSS. Para tal, vamos criar o estilo desta tela em nosso `www/css/style.css`, que refletirá a nossa necessidade.

```
...
.bt-camera {
    min-height: 200px;
    max-height: 200px;
    min-width: 80%;
    max-width: 80%;
}
...
```

Feito isto, estamos construindo uma classe que determina um tamanho mínimo para altura (`min-height`, `max-height`) e largura (`min-width`, `max-width`). Essa será a base de nosso botão que terá como ícone a câmera ilustrada no protótipo.

Para isto, devemos construir um arquivo que chamaremos de `www/templates/camera.html` e um estado que nos levará a um controle `www/js/camera-controller.js`. Aqui não importa como o leitor vai construir estes artefatos. Se forem feitos no projeto do Cardápio Móvel, lembre-se de que deverá fazer um novo **estado**, depois um novo menu e chamar a *view* associada a um *controller*.

Se você leu o livro até aqui, não terá dificuldade em entender isto. Caso decida fazer tudo em uma nova aplicação, a regra será basicamente a mesma. Portanto, vamos aos artefatos *controller* e

*view*, respectivamente.

```
//Controller
app.controller("CameraCtrl", function(){
});

<!-- View -->
<ion-view title="Camera">
  <ion-content>
    <div class="row">
      <div class="col col-offset-20">
        <button class="button bt-camera ion-camera">
        </button></div>
      </div>
    </ion-content>
</ion-view>
```

Observe que o *controller* foi declarado e, por enquanto, não há nenhuma função nele. A *view* tem apenas o botão que representa nosso protótipo. Crie um estado e execute para visualizar o resultado.

Por enquanto, estes artefatos podem ser testado em *browser*, pois ainda não usamos o recurso da câmera.

## 9.3 A CÂMERA

Para utilizar a câmera com o plugin instalado no início deste capítulo, o leitor precisa entender o método disponível nele, o `getPicture(options)`. Nele vamos configurar, através do parâmetro (`options`), todas as características da imagem que será capturada. Por isso é válido conhecer os parâmetros que são passados a esta função. Verifique a seguir:

Options	Type	Detail
<i>quality</i>	Número	Qualidade da imagem gravada entre 0 - 100
<i>destinationType</i>	Número	Tipo do retorno em base64 ou URL
<i>sourceType</i>	Número	Origem da imagem
<i>allowEdit</i>	Boolean	Permite edição da imagem antes de salvar
<i>encodingType</i>	Número	JPEG = 0, PNG = 1
<i>targetWidth</i>	Número	Largura da imagem (pixels). Usado com <i>targetWidth</i>
<i>targetHeight</i>	Número	Altura da imagem (pixels). Usado com <i>targetHeight</i>
<i>mediaType</i>	String	Tipo da imagem
<i>cameraDirection</i>	Número	Traseira = 0, Frontal = 1
<i>popoverOptions</i>	String	Opção que especifica a localização no iPad (somente iOS)
<i>saveToPhotoAlbum</i>	Boolean	Salvar a imagem no álbum de fotos
<i>correctOrientation</i>	Boolean	Corrigir a imagem capturada em caso de erro na orientação

As opções devem ser passadas em formato *JSON*, ou seja, um objeto neste formato. Um *JSON* nada mais é que um *JavaScript Object Notation* (Notação de Objetos JavaScript). Esta definição dispensa explicação, basta o leitor entender que os objetos JavaScript podem ser representados de várias formas, inclusive como *JSON*. Verifique no exemplo a seguir:

```
var opcoes = {
    quality: 50,
    destinationType: Camera.DestinationType.DATA_URL,
    sourceType: Camera.PictureSourceType.CAMERA,
    allowEdit: true,
    encodingType: Camera.EncodingType.JPEG,
    popoverOptions: CameraPopoverOptions,
    saveToPhotoAlbum: false,
```

```
    correctOrientation:true  
};
```

Todo JSON é formatado entre chaves, com seus parâmetros definidos no seguinte formato: [chave] : [valor] e separados por vírgula. Sabendo disto, vamos em seguida utilizar estes parâmetros. Ao clicar no ícone da câmera de nosso exemplo, o aplicativo abrirá a opção de tirar uma foto ( `sourceType` ) e a guardará em formato JPEG `encodingType`, permitindo a edição após execução ( `allowEdit` ).

Depois disto, exibiremos a foto que foi tirada e guardada em um formato `base64`. Este formato foi configurado com o parâmetro `destinationType`, atribuindo o valor `Camera.DestinationType.DATA_URL`. Caso você prefira guardar em um arquivo, poderá utilizar a seguinte opção: `destinationType: Camera.DestinationType.FILE_URI`. Para que a exibição seja possível, vamos inserir uma `div` ao `html`. Veja a seguir a codificação de nossa `view`:

```
<ion-view title="Camera">  
  <ion-content>  
    <div class="row">  
      <div class="col col-offset-20"><button ng-click="tira  
rFoto()" class="button bt-camera ion-camera" id="btCamera"><img i  
d="minhaImagem" ></button></div>  
    </div>  
  </ion-content>  
</ion-view>
```

Analisando melhor, podemos verificar a alteração do botão, em que este recebe uma diretiva `ng-click=tirarFoto()` e, logo em seguida, a `div` que receberá a imagem. Portanto, vamos preparar nosso `controller` com o seguinte código:

```
app.controller("CameraCtrl", function($scope){
```

```

var options = {
    quality: 50,
    destinationType: Camera.DestinationType.DATA_URL,
    sourceType: Camera.PictureSourceType.CAMERA,
    allowEdit: true,
    encodingType: Camera.EncodingType.JPEG,
    popoverOptions: CameraPopoverOptions,
    saveToPhotoAlbum: false,
    correctOrientation:true
};

$scope.tirarFoto = function(){
    $cordovaCamera
        .getPicture(options)
        .then(function(imageData) {
            var image = document.getElementById('minhaImagen');
            image.src = "data:image/jpeg;base64," + imageData;

        }, function(err) {
            // error
        });
    }
});
```

Observe que usamos uma variável `options` onde informamos os principais parâmetros apresentados na tabela do início desta seção. Após isto, construímos uma função que executará a chamada à câmera, passando os parâmetros e recebendo o resultado no *callback* da função chamada de `then`.

Este *callback* recebe, através de uma função, a imagem capturada na câmera, que neste caso estará em um formato `base64`. Com o resultado neste retorno, vamos disponibilizá-lo na `div` que referenciamos através do `document.getElementById('minhaImagen')`. Se executarmos esta funcionalidade no *browser*, encontraremos o seguinte erro:

```
✖ ► ReferenceError: Camera is not defined ionic.bundle.js:26794
    at new <anonymous> (http://localhost:8100/js/controllers/camera-controller.js:5:24)
    at Object.instantiate (http://localhost:8100/lib/ionic/js/ionic.bundle.js:18010:14)
    at $controller (http://localhost:8100/lib/ionic/js/ionic.bundle.js:23412:28)
    at Object.self.appendViewElement (http://localhost:8100/lib/ionic/js/ionic.bundle.js:59
900:24)
    at Object.render (http://localhost:8100/lib/ionic/js/ionic.bundle.js:57893:41)
    at Object.init (http://localhost:8100/lib/ionic/js/ionic.bundle.js:57813:20)
    at Object.self.render (http://localhost:8100/lib/ionic/js/ionic.bundle.js:59759:14)
    at Object.self.register (http://localhost:8100/lib/ionic/js/ionic.bundle.js:59717:10)
    at updateView (http://localhost:8100/lib/ionic/js/ionic.bundle.js:65398:23)
    at http://localhost:8100/lib/ionic/js/ionic.bundle.js:65382:9 <ion-nav-view
name="menuContent" class="view-container" nav-view-transition="android">
```

Figura 9.2: Erro sem câmera

Isto significa que chegamos ao nosso limite de testes de aplicações através do *browser*, ou seja, o aplicativo está tentando encontrar um recurso que não existe no navegador. Para isto, precisamos utilizar um simulador, ou debugar nossa aplicação em um dispositivo real.

Este comando foi explanado no capítulo *Iniciando nossa aplicação* e, caso o leitor não se lembre, poderemos refrescar sua memória. O comando é o `ionic run [plataforma]`, e a plataforma pode ser **Android**, **iOS** etc., dependendo de qual o desenvolvedor esteja utilizando e plugado via USB em seu computador.

Antes de executar o comando, precisamos injetar o recurso em nossa aplicação e em nosso *controller*. Abra o arquivo `www/js/app.js` e injete o `ngCordova` no módulo da aplicação assim:

```
...
angular.module('starter', ['ionic', 'starter.controllers', 'ngNumberPicker', 'ngCordova'])
...
```

Agora a assinatura do nosso *controller* receberá a injeção do *plugin* através do seguinte objeto:

```
app.controller("CameraCtrl", function($cordovaCamera, $scope){  
    ...
```

Feito isto, você já pode testar a câmera e ver o funcionamento deste *plugin*. Se tudo ocorreu bem até aqui, você verá a câmera ser chamada através do botão e, ao tirar uma foto, o aplicativo permitirá a sua edição. Posteriormente, quando confirmada, será exibida na `div` que incluímos em nossa *view*.

## 9.4 CONCLUSÃO

Este capítulo foi uma introdução à adição de um dos muitos *plugins* disponíveis no repositório **ngCordova**, e serve como base para o leitor entender como é feita uma chamada a um recurso através desta instalação. Os demais *plugins* trabalham de forma similar, e basta uma ligeira consulta na documentação do recurso para entender como será feita a chamada às suas funções.

## 9.5 PARA SABER MAIS

1. **ngCordova** – <http://ngcordova.com>
2. **Câmera** – <http://ngcordova.com/docs/plugins/camera>

# CAPÍTULO 10

# RECURSOS

Estamos chegando ao final de nosso livro, e este último capítulo será utilizado para fazermos uma miscelânea de recursos do *Ionic Framework*. Vamos consumir dados aloados na nuvem, e também uma lista mais avançada do que a vista no capítulo *A bandeja*. Conheceremos o `$ionicActionSheet`, uma espécie de painel de opções que abre com efeito *slide-up*, através do evento *click* de um botão.

Faça bom proveito destes recursos e espero que, com eles, o leitor tenha uma melhor visão e possa ampliar assim seus conhecimentos, utilizando-os em suas aplicações. Inclusive, você poderá fazer melhorias no caso de uso do **Cardápio Móvel**, construído neste livro.

## 10.1 O FIREBASE

Para consumir serviços de uma base de dados na *web*, vamos utilizar o *firebase* do Google. Este recurso é uma base que se encontra na nuvem e tem como diferencial o conceito de *real time*. Isto é, tudo o que for atualizado no banco de dados será propagado em tempo real para sua aplicação.

Por exemplo, se a aplicação executar uma leitura de um

registro no *firebase* e de alguma forma estiver "escutando" este registro, quando ele for alterado por outra aplicação, o valor será atualizado sem necessidade de um novo ciclo de consulta ao banco de dados. Para visualizar melhor este conceito, imagine um *chat* onde vários usuários enviam mensagem de forma assíncrona. Assim que um usuário grava no banco uma mensagem, todos as aplicações que estão conectadas no *firebase* receberão a atualização.

Na verdade, o *firebase* envia a atualização no sentido inverso do que acontece em uma aplicação convencional. É uma espécie de *listener*, em que a aplicação fica aguardando – sem precisar fazer consulta – qualquer alteração no banco de dados.

Obviamente não vamos nos aprofundar na criação de uma base no *firebase*, pois este não é o foco deste capítulo, no entanto, vou disponibilizar uma base pronta para ser consumida. Se quiser saber mais sobre o *firebase*, sugiro que consulte a seção *Para saber mais*. Estamos prontos?!

## 10.2 CONSUMINDO SERVIÇOS

Antes de tudo, precisamos instalar o *client* do *firebase*, através do *bower*. Em nosso caso, devemos utilizar um suporte oficial do *AngularJS* para executar as funções deste recurso. Vamos ao comando para instalação?!

```
bower install angularfire --save
```

Se você observar, serão instalados dois *plugins* em sua aplicação com este comando: o *Angularfire* propriamente dito e as APIs do *firebase* no diretório *lib* de sua aplicação. Precisamos agora

importá-los em nosso `www/index.html`, para que seja possível injetar estes artefatos na aplicação. Veja a seguir o resultado da instalação e a importação na aplicação, respectivamente:

```
angularfire#2.3.0 www/lib/angularfire
└── angular#1.5.3
    └── firebase#3.7.4
```

Figura 10.1: Conclusão da instalação

```
...
<!-- Importação no arquivo www/index.html -->
<script src="lib/angularfire/dist/angularfire.js"></script>
<script src="lib.firebaseio.firebaseio.js"></script>
...
```

Após a importação, vamos consumir três listas que estão disponíveis em <https://livro-ionic.firebaseio.com>. Se você tentar acessar esta *URL*, não conseguirá, ainda que faça login em sua conta Google. Entretanto, poderá ler o banco de dados que está na seguinte estrutura:



Figura 10.2: Lista no firebase

Esta imagem o faz recordar de alguma coisa? Se você fez o

dever de casa e leu o livro até aqui, estará se perguntando: "Será que eu posso consumir estes serviços para o meu cardápio e eliminar os *mocks* que criei? Dessa forma, estarei construindo um aplicativo mais próximo da realidade". A resposta é: **você não só pode, como deve!**

Vamos entender como tudo funcionará com o *firebase* e, em seguida, você poderá transferir todo conceito para o cardápio móvel.

Com tudo pronto, vamos inicializar o *firebase* em nossa aplicação pelo arquivo `www/js/app.js`. Este é o módulo *starter* da aplicação, ou seja, é a localização global para criar, registrar e recuperar módulos do *AngularJS*. Portanto, vamos registrar (ou inicializar) as configurações *firebase*, dentro da função `run()` e `$ionicPlatform.ready()`.

```
...
.run(function($ionicPlatform) {
  $ionicPlatform.ready(function() {
    // Hide the accessory bar by default (remove this to show the
    accessory bar above the keyboard
    // for form inputs)

    // Initialize Firebase
    var config = {
      apiKey: "AIzaSyAV6AVt9iTDoa5Rla_xbrN07EDjcRzSrmA",
      authDomain: "livro-ionic.firebaseio.com",
      databaseURL: "https://livro-ionic.firebaseio.com"
    };
    firebase.initializeApp(config);
...
}
```

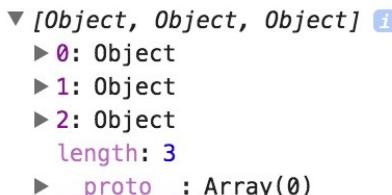
Observe que estas configurações são específicas do banco de dados que foi disponibilizado para o leitor deste livro. Portanto, tudo pode mudar de acordo com o banco que você criar. Temos aí uma chave `apiKey` (que nada mais é do que um identificador

dentro da base de dados) e, em seguida, o domínio e a *URL*.

Depois, inicializamos de fato o banco com a chamada a `initializeApp`, passando como parâmetro as configurações. Feito isto, já podemos utilizar o *firebase* dentro de nossa aplicação para consultar, setar e escutar valores na nuvem. Porém, em nosso caso, o leitor só terá a permissão para consulta. Utilize um *controller* com a seguinte chamada:

```
var ref = firebase.database()
    .ref("/bebidas")
    .once("value",function(valor){
    console.log(valor.val());
});
```

Chame este *controller* e observe o console no *browser*. Você vai observar que estão sendo retornados os três valores do *array* das bebidas, similar à figura a seguir:



The screenshot shows a browser's developer tools console. It displays an array with three elements, each represented as an object. The array is shown with a triangle icon followed by '[Object, Object, Object]'. Below it, there are three entries, each preceded by a triangle icon and labeled '0:', '1:', and '2:'. Each entry shows an object with a 'length' property of 3. At the bottom of the array, there is a entry for the prototype, preceded by a triangle icon and labeled '\_\_proto\_\_:' followed by 'Array(0)'.

Figura 10.3: Objetos retornados

Entendendo o código, temos na primeira linha uma variável que guardará a referência do banco, de acordo com as configurações que fizemos ao inicializar a aplicação (`database()`). Em seguida, demos ao `database()` a referência que queremos dentro do banco: `/bebidas`.

Chamamos assim a função `once()` com o parâmetro `value`, informando que queremos "tirar uma foto", ou seja, pegar os dados

uma só vez no *firebase*. Isto significa que o aplicativo busca na nuvem a lista de bebidas e, mesmo que ela seja atualizada, nada acontecerá mais nesta lista local até que se faça uma nova consulta.

Isto entra em contradição com o que foi dito sobre o *firebase* no início deste capítulo, não é? Sim! Mas, como estamos fazendo uma funcionalidade somente de consulta, resolvemos fazer desta forma. Caso o leitor queira aprender mais sobre as atualizações citadas, poderá se aprofundar na documentação do link disponível na seção *Para saber mais sobre o firebase*.

Agora vamos entender o que a função de *callback* faz. Ela simplesmente recebe um valor que é acessível através da função `val()`. Por fim, a função `console()` do JavaScript imprime os objetos recebidos. Teste e observe a lista sendo retornada. Feito isto, brinque um pouco com as listas, modificando apenas a referência (`ref`) com seus respectivos parâmetros: `"/petiscos"` e `"/sucos"`.

## 10.3 O CARDÁPIO MODIFICADO

Agora que você já sabe como chamar um serviço na nuvem utilizando o *firebase*, podemos modificar nosso **caso de uso** para que, em vez de utilizar *mocks*, passe a consumir serviços, de fato.

Para executar estas chamadas no Cardápio Móvel, precisamos apenas modificar o *controller* para o seguinte código:

```
app.controller('BebidasCtrl', function($scope, Sessao, CardapioSe
rvices,$firebaseObject,$ionicLoading){
$ionicLoading.show({
  template: 'Carregando...'
}).then(function(){
  $scope.bebidas  = [];
});
```

```

});;

var ref = firebase.database().ref("/petiscos").once("value", function(valor){
    $ionicLoading.hide().then(function(){
        $scope.bebidas = valor.val();
    });
});
});;

```

Ops! Algo novo aconteceu aqui. Veja que injetamos o `$ionicLoading`. Este objeto é uma tela de espera do *Ionic* que é bastante utilizada para consumo de serviços e processamentos. Ele contém duas funções básicas: `show()` e `hide()`.

A primeira exibe o template que for determinado pelo desenvolvedor. Em nosso caso, apenas exibiremos a mensagem "Carregando..." e inicializamos a variável `$scope.bebidas` vazia. Após executar o `once()` do *firebase*, note que fechamos a tela de espera com o `hide()` e referenciamos em `$scope.bebidas` o valor que foi recuperado. Assim exibiremos na *view* de nossa aplicação.

Portanto, o `$ionicLoading` é um excelente componente para consultas assíncronas pelo *Ionic*. A estratégia é simples:

1. Fazer uma consulta no *webservice*;
2. Abrir a tela de espera com o `$ionicLoading` ;
3. Enquanto o serviço não é retornado através da *promise* `once()` , a tela exibirá a mensagem de "Carregando...";
4. Ao retornar, dentro da *promise*, fechamos a tela de espera `$ionicLoading` e exibimos os dados.

**PROMISE**, segundo o site da *Mozilla Foundation*, é uma promessa em JavaScript, ou seja, um objeto usado para processamento assíncrono. Ele representa um valor que pode estar disponível agora, no futuro ou nunca. Se quiser saber mais sobre *promise*, consulte <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference>.

Agora que você entendeu como consumir as listas na nuvem, deixarei a seu critério a modificação dos outros itens, eliminando assim os *mocks* construídos em `www/js/services/cardapio-services.js`.

## 10.4 IONICLIST

Vamos agora nos aprofundar mais nas listas do *Ionic*. Este componente é bastante utilizado em aplicações nas quais há necessidade de exibir muitos elementos em um *grid*. Para isto, o *framework* nos oferece o componente `$ionicList` com muitas possibilidades de manipulações dos elementos na lista.

Lembra da nossa bandeja, na qual tínhamos a possibilidade de edição e exclusão dos itens? Pois bem! Poderíamos ter feito tudo muito mais elegante se tivéssemos iniciado o livro de trás para a frente, porém, deixei isto por último para que você aprendesse o "algo a mais" do *framework*.

Sim! Reinventamos a roda quando construímos botões para edição e remoção dos itens da lista. Além disto, o pessoal de *design*

não imaginou que nossos componentes fariam boa parte deste trabalho. Agora vamos de fato usar a "roda" mais elegante. O problema disto é convencer os *designers* a reconstruir o comportamento desta lista.

Vamos fazer como na seção anterior, em que construímos um exemplo e, a partir daí, fica a critério do leitor transferir o conceito para o Cardápio Móvel. Consumiremos o mesmo serviço disponível em <https://livro-ionic.firebaseio.com> para demonstração desta lista. Utilizaremos para isto a lista bebidas .

O *controller* conterá agora o seguinte trecho de código:

```
...
$scope.data = {
  showDelete: false
};
$scope.onItemDelete = function(item) {
  $scope.bebidas.splice($scope.bebidas.indexOf(item), 1);
}
...
...
```

O que fizemos foi declarar um objeto que receberá o estado inicial `false` . Ele será explicado e melhor entendido quando construiremos a *view*. Em seguida, temos uma função que receberá um item para ser excluído pelo `splice(index, quant)` , já explicada no capítulo *A bandeja*. Feito isto, vamos partir para a *view*:

```
...
<div class="buttons">
  <button class="button button-icon icon ion-ios-minus-outline"
    ng-click="data.showDelete = !data.showDelete"></button>
</div>
<ion-list show-delete="data.showDelete">
  <ion-item ng-repeat="bebida in bebidas" href="#/app/detalhar/
  {{bebida.id}}">
    <ion-delete-button class="ion-minus-circled"
```

```
        ng-click="onItemDelete(bebida)">
      </ion-delete-button>
      {{bebida.nome}} - {{bebida.preco.toLocaleString('pt-BR',
      { style: 'currency', currency: 'BRL'})}}
    </ion-item>
</ion-list>
...
```

Nela é possível observar um botão para exibir a lista com itens removíveis, ou seja, este botão ficará no topo da página, antes da lista. Ele tem como *data-binding* o `data.showDelete` que declaramos no *controller*, lembra?

Veja que, ao clicar neste botão com a diretiva `ng-click`, o estado da variável `showDelete` receberá a negação do valor atual. Ou seja, se for `true` se tornará `false`, e vice-versa.

Logo abaixo, a lista recebe duas novidades: `show-delete` e `<ion-delete-button>`. O `show-delete` configura a lista para abrir os botões de *delete* construídos com o `<ion-delete-button>`, dentro de cada item. Nele está configurada a exibição do ícone com o sinal de menos ( - ) que executará uma função, construída no *controller*, passando assim a bebida como parâmetro.

Na figura a seguir, é possível ver exemplo desta ação sendo executada. À esquerda, o botão de menos ainda não foi clicado e, à direita, temos o resultado do clique neste botão. Faça um teste e aproveite para modificar o Cardápio Móvel (na bandeja do capítulo *A bandeja*).



Figura 10.4: Lista turbinada

## 10.5 O IONIC ACTIONSHEET

O *Action Sheet* será nosso último recurso a ser demonstrado. Ele é apenas um painel de opções que rola de baixo para cima (*slide-up*), sem tomar a tela por completo. Com certeza, muitos já se depararam com este componente em aplicativos que compartilham algum tipo de conteúdo.

Ao clicar no recurso a ser compartilhado, este exibe uma *Action Sheet* para escolha do aplicativo. Veja a seguir seu formato:

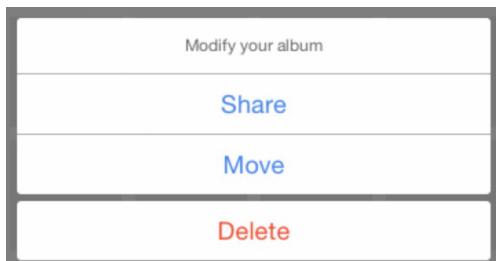


Figura 10.5: Action Sheet

A figura nos diz que este *Action Sheet* foi criado para modificar um álbum, sendo possível executar as seguintes ações: *Compartilhar*, *Mover* ou *Excluir*.

Vamos incrementar em nossa lista um botão de compartilhar.

O comportamento seguirá o seguinte fluxo:

1. Ao deslizar o dedo sobre um item da direita para a esquerda, será exibido um botão Compartilhar ;
2. Ao clicar neste botão, abrirá nosso *Action Sheet* com dois botões: WhatsApp , E-mail .

Para isto, execute na lista construída anteriormente a seguinte modificação:

```
...
<ion-list show-delete="data.showDelete">
    <ion-item ng-repeat="bebida in bebidas" href="#/app/detalhar/{{bebida.id}}">
        <ion-delete-button class="ion-minus-circled"
            ng-click="onItemDelete(bebida)">
        </ion-delete-button>
        <ion-option-button class="button-calm" ng-click="abrirActionSheet()">Compartilhar</ion-option-button>
        {{bebida.nome}} - {{bebida.preco.toLocaleString('pt-BR',
        { style: 'currency', currency: 'BRL'})}}
    </ion-item>
</ion-list>
...
```

Perceba que só colocamos um botão a mais: o `<ion-option-button>` com a diretiva `ng-click` chamando a função `abrirActionSheet()` , que construiremos no *controller* logo a seguir:

```
...
$scope.abrirActionSheet = function(){
    $ionicActionSheet.show({
        buttons: [
            { text: 'Whatsapp' },
            { text: 'Email' }
        ],
        buttonClicked: function(index) {
            return true;
        }
}
```

```
    });
}
...
```

Não se esqueça de injetar o `$ionicActionSheet` em seu *controller*. O que fizemos foi apenas configurar o nosso componente com dois botões ( `buttons` ): um WhatsApp e outro E-mail. A função `buttonClicked` é executada toda vez que algum botão, de fato, é acionado. Aqui você poderia fazer sua regra de negócio, sabendo em qual botão o usuário clicou através do `index` que é passado como parâmetro.

Tudo pronto para teste. Fica aqui o desafio para você colocar um botão ao deslizar o dedo sobre um item, onde o mesmo substitua a função de Editar do capítulo *A bandeja*, para que os itens da lista de bandeja sofram esta ação. Boa sorte!

## 10.6 DIALOG DO NGCORDOVA

Em nosso *caso de uso* do cardápio, fizemos uma bandeja, na qual o usuário adiciona vários itens antes de confirmar estes pedidos, permitindo que se volte várias vezes e edite ou remova estes itens. Ao fazer a funcionalidade de remover no capítulo *Fazendo pedidos*, deixamos de lado o modal de confirmação, ou seja, o usuário deveria clicar em remover um item. Mas antes de executar a função, o aplicativo abrirá uma tela de confirmação desta ação.

Esta função pode ser construída através da adição do *plugin dialog* do ngCordova, com o seguinte comando: `cordova plugin add cordova-plugin-dialogs` . Feito isto, é preciso refazer a função `removerItem()` do `www\js\controllers.js` .

Primeiro precisamos injetar o *plugin* recentemente instalado dentro do *controller*:

```
app.controller('AppCtrl', function($scope, Sessao,  
    $ionicModal, $cordovaDialogs) {...
```

Agora é só inserir um *dialog* na função que está sendo chamada pelo botão de remoção. Este *dialog* deverá conter dois botões (Cancelar e OK) mais uma mensagem de confirmação. Ao clicar em OK , o *dialog* deverá efetuar um teste para executar a remoção que estava sendo feita antes, ou ignorar fechando o *dialog* se acaso o usuário escolher o botão Cancelar . Veja o código a seguir:

```
...  
//Remoção  
$scope.removerItem = function(){  
    $cordovaDialogs.confirm('Deseja realmente remover este item?'  
'  
        'Remoção', ['Cancelar','OK'])  
        .then(function(buttonIndex) {  
            'Cancelar' = 0, 'OK' = 1  
            var btnIndex = buttonIndex;  
            if( btnIndex == 1){  
                Sessao.bandeja.splice($scope.data.item,1);  
            }  
        });  
...  
}
```

Analizando o código, vemos que o `$cordovaDialogs.confirm()` contém 3 parâmetros: uma mensagem, um título e um *array* de botões, respectivamente. Ao clicar em um botão, a `promise then()` pega a referência do botão (ou seja, seu *index*) e testa. Se for igual a 1, a função executa a remoção do item da bandeja. Ao contrário, se o usuário clicar em cancelar ( `index = 0` ), nada acontecerá, exceto o fechamento do *dialog*.

## 10.7 CONCLUSÃO

Bem, caros colegas, chegamos ao final deste livro! Espero ter contribuído para a disseminação do conhecimento, algo muito importante na área de tecnologia. Ao longo dos capítulos, aprendemos um pouco de cada coisa. É o que chamamos de *overview* do *framework*.

Através desta leitura, aprendemos desde a criação do ambiente e criação de nova aplicação até a utilização de recursos do dispositivo. Com isto, passamos pela maioria dos componentes usados em um aplicativo através do *Ionic Framework*, deixando assim algumas dicas para que o leitor tenha uma experiência mais aprofundada através da leitura da sua documentação oficial.

Somente a partir disto, o leitor poderá agregar mais conhecimento e experiência com esta ferramenta poderosa que é o *Ionic Framework*. Toda sua documentação está disponível e citada em cada seção **Para saber mais** dos capítulos. Portanto, é importante a consulta sempre que necessário.

Participe também das comunidades referentes ao assunto e tire dúvidas com desenvolvedores mais experientes. O meu desejo é que você tenha tido um ótimo aprendizado com este livro e que dê continuidade se aprofundando na busca de novos conhecimentos sobre o *framework*, já que ele disponibiliza uma infinidade de componentes que não foram explorados aqui.

## 10.8 PARA SABER MAIS

1. **Angularfire** – <https://github.com/firebase/angularfire>
2. **Documentação do Firebase** –

<https://firebase.google.com/docs>

3. Action Sheet -  
[http://ionicframework.com/docs/v1/api/service/\\$ionicActionSheet](http://ionicframework.com/docs/v1/api/service/$ionicActionSheet)