UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

TOBIAS VALENTIM DE MACEDO JUNIOR

DESENVOLVIMENTO DE APLICATIVO MÓVEL PARA SISTEMA OPERACIONAL iOS EM LINGUAGEM SWIFT PARA LEITURA DE CONSUMO DE ENERGIA DE TOMADA INTELIGENTE EM TEMPO REAL

FORTALEZA, CE

2017

TOBIAS VALENTIM DE MACEDO JUNIOR

DESENVOLVIMENTO DE APLICATIVO MÓVEL PARA SISTEMA OPERACIONAL iOS EM LINGUAGEM SWIFT PARA LEITURA DE CONSUMO DE TOMADA INTELIGENTE EM TEMPO REAL

Monografia apresentada ao Curso de Engenharia Elétrica da Universidade Federal do Ceará, como requisito parcial à obtenção do título de graduado em Engenharia Elétrica

Orientador: Prof. Dr. Luiz Henrique Silva Colado Barreto.

FORTALEZA

2017

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

M125d Macedo, Tobias.
Desenvolvimento de aplicativo móvel para sistema operacional iOS em linguagem Swift para leitura de consumo de tomada inteligente em tempo real / Tobias Macedo. – 2017.
110 f. : il.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Tecnologia, Curso de Engenharia Elétrica, Fortaleza, 2017.
Orientação: Prof. Dr. Luiz Henrique Silva Colado Barreto.

1. Apple. 2. Swift. 3. Aplicativo. 4. IoT. 5. Tomada Inteligente. I. Título.

CDD 621.3

TOBIAS VALENTIM DE MACEDO JUNIOR

DESENVOLVIMENTO DE APLICATIVO MÓVEL PARA SISTEMA OPERACIONAL iOS EM LINGUAGEM SWIFT PARA LEITURA DE CONSUMO DE TOMADA INTELIGENTE EM TEMPO REAL

Esta monografia foi julgada adequada para a obtenção do grau de Graduado em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Aprovada em: ___/___/_____.

BANCA EXAMINADORA

_____
Prof. Dr. Luiz Henrique Silva Colado Barreto (Orientador)
Universidade Federal do Ceará (UFC)

_____
Eng. Ícaro Jonas Batista
Universidade Federal do Ceará (UFC)

_____
Prof. Dr. Walter da Cruz Freitas Júnior
Universidade Federal do Ceará (UFC)

AGRADECIMENTOS

A Deus.

Aos meus pais Tobias e Isabel e à minha família pelo incentivo e por toda a estrutura fornecida durante meus anos de vida.

À minha namorada Mariana Pinheiro pela paciência e companheirismo ao longo deste percurso, apoio durante o curso e pela revisão deste trabalho.

Ao Prof. Dr. Luiz Henrique Silva Colado Barreto, pela excelente orientação tanto durante a execução da monográfica quanto durante o curso. A convivência nesse período foi de grande aprendizado.

À minha amiga Elisa Zandoná por todo o apoio e incentivo, principalmente durante a execução desse trabalho.

Ao meu amigo Dalmo Mendes pelo aporte pessoal e profissional desde o colégio.

A todos os demais amigos e aos companheiros de curso Clayton, Matheus Nogueira, Felipe Porto, Matheus Jonathan, Enzo, Fábio, Raimundo Vidal, Herivelton, Letícia, Vitória e Edmundo pela amizade, descontrações e tempo de estudo durante todo o curso

Aos participantes da banca examinadora Walter da Cruz Freitas Júnior e Ícaro Jonas Batista pelo tempo, pelas valiosas colaborações e sugestões.

“Minhas coisas favoritas na vida não custam dinheiro. É realmente claro que o recurso mais precioso que temos é o tempo”. (Steve Jobs)

RESUMO

Este trabalho apresenta o desenvolvimento de um aplicativo móvel que gerencia tomadas elétricas via Wi-Fi. São apresentadas breves introduções às linguagens de programação utilizadas: Swift e PHP. Juntamente com a apresentação da linguagem PHP, é apresentado o banco de dados utilizado pelo aplicativo e os scripts utilizados como o meio intermediário entre servidor e aplicativo. A apresentação do aplicativo se divide em duas partes: Interface Homem-Máquina e Conexão com a internet. Na primeira parte, são apresentadas as interfaces presentes no aplicativo, como o usuário pode alterá-las e como essas alterações modificam os dados presentes no aplicativo. Na segunda parte são apresentadas as funções que se conectam aos scripts em PHP. O Aplicativo é voltado para o ecossistema iOS, programado em linguagem Swift. Ele se conecta ao servidor, que consiste em três tabelas MySQL, através de códigos em linguagem PHP. As principais funções da conexão são: autenticação, identificação e monitoramento de dados. O aplicativo se mostrou de interface simples, com baixo consumo de memória RAM e baixo consumo de dados.

Palavras-chave: Apple. Swift. Aplicativo. IoT. Tomada Inteligente.

ABSTRACT

This paper presentes the development of a mobile application that manages power outlets via Wi-Fi. Brief introduction to both Swift and PHP programming languages are presented. Along with PHP language introduction, the database used by the application and PHP scripts used as the intermediate between server and application are presented. The App presentation is divided in two parts: Human Machine Interface and internet connection. On the first part, the interface present in the application is presented., along with the possible user changes and how these changes modify the data inside the application. On the second part, the functions responsible for the internet connection of the application are explained. The App Works with operational system iOS, and it is programmed in Swift languagem. It connects to the server, which consists in three MySQL tables, through PHP language code. The most important functions of the connection are: authentication, identification and data monitoring. The app has a simple interfaces, low RAM consumption and low data consumption

Keywords: Apple. Swift. App. IoT. Smart Power Outlet.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ABNT	Associação Brasileira de Normas Técnicas
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
ID	Identificação
IoT	Internet of Things
iOS	iPhone Operational System
JSON	JavaScript Object Notation
MVC	Model-View-Controller
PHP	PHP: Hypertext Preprocessor
TCP	Transmission Control Protocol
IU	Interface do Usuário
URL	Uniform Resource Locator

SUMÁRIO

1. INTRODUÇÃO

1.1. Objetivos

Este trabalho tem como objetivo apresentar um projeto de aplicativo móvel para aparelhos que tem como sistema operacional *iOS*. Este aplicativo terá como principal função gerenciar a utilização de tomadas elétricas conectadas a uma mesma rede *Wi-Fi*.

1.2. Motivação

Com o desenvolvimento das redes Wi-Fi, e sua grande expansão desde o início do Século XXI, cada vez mais aparelhos são desenvolvidos com conexão sem fio à internet. Inicialmente, a tecnologia Wi-Fi era destinada apenas a computadores, mas com a popularização dos *smartphones* e a subsequente redução de preço das placas de circuito Wi-Fi fez com que a internet passasse a ser implementada cada vez mais em aparelhos domésticos.

Atualmente, com essa implementação, a gestão dos recursos se tornou mais simples e atualizada. O nome que se dá a esse processo, de obter dados e gerenciar objetos físicos, é de Internet das Coisas (IoT).

A partir desse princípio, foi desenvolvido o projeto da Tomada Inteligente, que, através da conexão à internet e de sensores de corrente, envia os dados de consumo a um servidor. Esses dados são facilmente acessáveis via interface Web, facilitando o controle do consumo pelo consumidor.

Essa tomada em desenvolvimento é uma alternativa de menor custo e maior simplicidade com foco no mercado brasileiro, em que este conceito é pouco explorado devido ao alto custo de importação e implementação dos produtos existentes no mercado, como o *WeMo switch*, da empresa *Belkin International Inc.*

Com a popularização dos *smartphones* citada acima, se torna muito mais simples para o usuário o acesso desses dados via aplicativos móveis. Plataforma como o *iOS* e o *Android* estão cada vez mais difundidas no Brasil, e a expectativa é que no final de 2017 haja um *smartphone* ativo por habitante.

Aliado a esse crescimento de aparelhos conectados à internet, o contínuo aumento do custo com energia elétrica faz com que seja cada vez mais importante ao consumidor o controle do consumo.

1.3. Metodologia

O trabalho será dividido em duas partes principais: a interface homem-máquina e a troca de dados com o servidor

A primeira parte será focada em apresentar a interface homem-máquina, ou seja, o que o usuário terá acesso para leitura ou alteração.

A segunda parte então explicitará como o dado que estava armazenado no servidor será analisado pelo programa para então chegar ao utilizador do aplicativo.

1.4. Estrutura do Trabalho

Na seção 1, são apresentados objetivos, motivação e metodologia utilizados no desenvolvimento do trabalho

Na seção 2, é apresentada uma breve introdução à linguagem de programação *Swift*, explicando sua arquitetura e suas principais funções

Na seção 3, é explicada a interface homem-máquina, como o usuário pode interagir com o programa e como essa interação afetará o aplicativo.

Na seção 4, é feita uma análise do banco de dados utilizado, em seguida uma introdução à linguagem de programação PHP, mostrando os pontos principais utilizados nesse trabalho. Finalmente, são apresentados os scripts em PHP utilizados

Na seção 5, são apresentadas as interações do aplicativo com o servidor que contém os dados que serão apresentados.

Na parte final serão apresentados os resultados obtidos e as conclusões, bem como as possibilidades de evolução e desenvolvimento.

2. CAPÍTULO 01: INTRODUÇÃO À LINGUAGEM SWIFT

A linguagem Swift é uma linguagem relativamente nova em comparação com as principais linguagens presentes no mercado. Segundo Matt Neuburg (2016, p. xi):

> Em 2 de junho de 2014, o keynote da Apple WWDC terminou com um anúncio chocante: "Nós temos uma nova linguagem de programação". Isso veio como uma grande surpresa para a comunidade de desenvolvedores, que estava acostumada com Objective-C, warts, dentre outras, e duvidavam que a Apple seria capaz de aliviá-los do peso do legado venerável dessas linguagens. (MATT NEUBURG, 2016)

Como apresentado acima, para o desenvolvimento de aplicativos para iOS, a linguagem mais comum era o Objective-C. O Swift foi uma evolução dos conceitos dessa linguagem, pois o Objective-C era apenas uma adaptação da linguagem C. Isso causava uma linguagem que não era totalmente voltada a objetos, possuindo muitos valores escalares. Além disso, a sintaxe do Objective-C era complicada, e a correção de erros era mal executada pelos programas.

O Swift foi pensado em aprimorar estes pontos, se apoiando em seis pontos principais:

- Orientação ao objeto: é uma linguagem puramente orientada ao objeto. Todos os elementos em Swift devem ser considerados objetos;
- Clareza: é uma linguagem de fácil leitura. Sua sintaxe é limpa, consistente e explícita;
- Segurança: o Swift reforça que o programador seja bem descritivo em sua linguagem, de maneira que hajam menos ambiguidades ao explicitar diretamente o tipo de cada objeto reverenciado;
- Economia: apesar de bastante descritiva, o Swift é uma linguagem de poucos tipos e funcionalidades internas a ela. As demais funcionalidades devem ser providas diretamente pelo programa;
- Gerenciamento de memória: o Swift gerencia a memória automaticamente, não sendo necessário por parte do programador a necessidade de realizar o gerenciamento dentro do código da aplicação;
- Compatibilidade com Cocoa: o API Cocoa é escrito em C e Objective-C. Swift é desenvolvido para interagir com esse API.

Mesmo possuindo várias diferenças com o Objective-C, o Swift interage muito bem com essa linguagem, fazendo com que o desenvolvimento de um aplicativo possa ter ambas as linguagens sem que haja maiores problemas na transição entre elas.

Nas seções seguintes, serão apresentadas as principais funcionalidades presentes no desenvolvimento do aplicativo, como a arquitetura usada no desenvolvimento, os tipos de dados, as funções e o API Charts, utilizado no desenvolvimento de gráficos.

2.1. A Arquitetura *Model-View-Controller* (MVC)

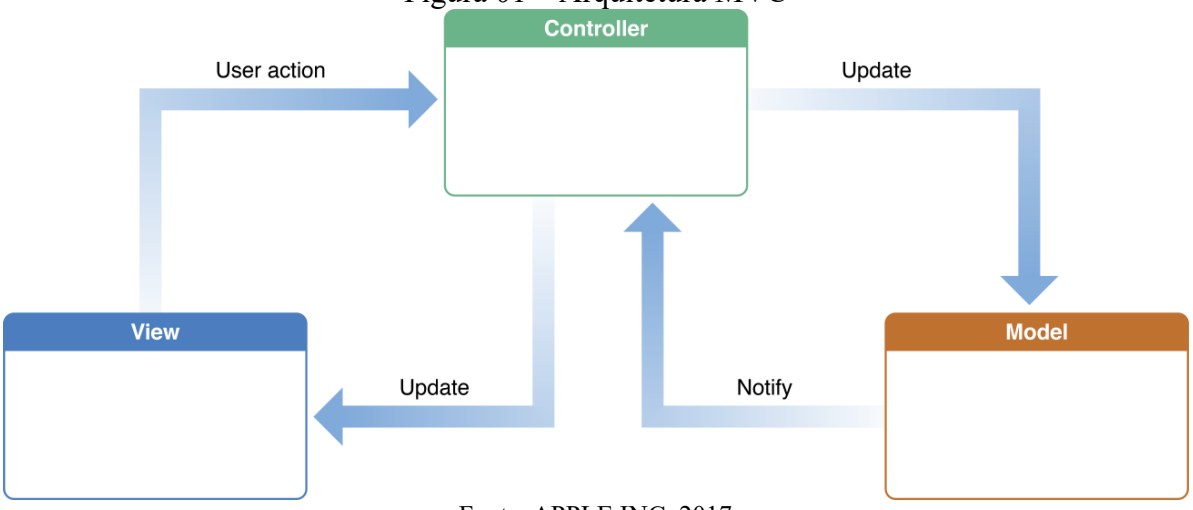
De acordo com The Swift Documentation (Apple Inc., 2017):

> A arquitetura *Model-View-Controller* (MVC) atribui três funções aos objetos de uma aplicação: *Model*, *View* ou Controller. Esse padrão não define apenas as funções que os objetos têm em uma aplicação, ele define o modo como objetos se comunicam entre eles. Cada um dos três tipos de objetos é separado por fronteiras imaginárias e se comunicam com os objetos de tipos diferentes por essas fronteiras. A coleção de objetos de um certo tipo de MVC em uma aplicação é geralmente referido como camada – por exemplo, camada *Model*. (APPLE INC, 2017)

O MVC é de grande importância para o desenvolvimento de um aplicativo em linguagem Swift. Primeiramente porque organiza o funcionamento do aplicativo e torna mais fácil e compreensível o código.

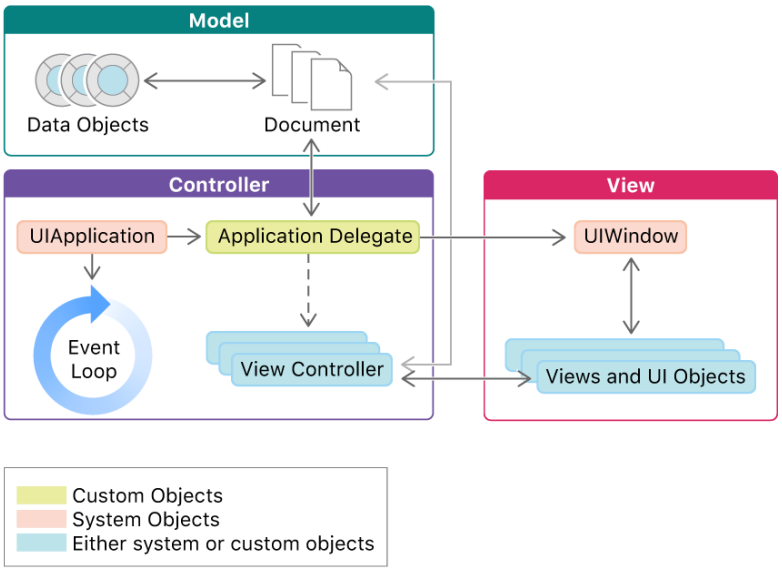
Além disso, a arquitetura facilita a programação da comunicação entre os diversos objetos. Como mostrado na Figura 01, o *Controller* geralmente se comunica tanto com o *Model*, quanto com o *View*. Já a comunicação entre *Model* e *View* praticamente inexiste, devido ao fato de ser necessário o processamento dos dados e ações feitas pelo usuário.

Figura 01 – Arquitetura MVC



Fonte: APPLE INC, 2017

Figura 02 – Arquitetura MVC – Tipos de objetos presentes em cada camada



Fonte: APPLE INC, 2017

Nas próximas subseções, serão explicadas com mais detalhes as funções de cada uma das camadas e como elas se comunicam entre si.

2.1.1. Model

Segundo The Swift Documentation (Apple Inc., 2017):

> Objetos de Modelo encapsulam os dados específicos de uma aplicação e definem a lógica e computação que manipulam e processam os dados. [...] um objeto de modelo pode ter diversos relacionamentos com outros objetos, e às vezes, a camada *Model* efetivamente consiste em um ou mais grafos. (APPLE INC., 2017)

Diante dessa concepção, observa-se que a camada Model tem como principal função armazenar os dados presentes no aplicativo. Esses dados, em sua maioria, fazem parte dos dados persistentes da aplicação, pois, ao serem guardados como variáveis nessa camada, só podem ser alterados utilizando os controladores.

Dessa forma, praticamente não há comunicação entra a camada Model e a camada View, pois a base da ideia dessa camada consiste em ter seus dados atualizados via controlador e notificá-lo da mudança.

Em Swift, os principais objetos da camada Model são *classes* (classes) e *structs* (estruturas). Esses objetos possuem muitas semelhanças, como a capacidade de o programador

definir as propriedades desses itens de maneira que eles consigam guardar os dados e utilizá-los na hora necessária. Além disso, ambos são capazes de definir métodos para aumentar sua funcionalidade. A definição desses métodos é simples: funciona como definir os métodos de qualquer tipo de variável. As classes possuem mais funcionalidades que as estruturas, como por exemplo: classes possuem a capacidade de herdar características de outras classes.

Esses objetos apenas definem como serão guardados os dados recebidos pela aplicação, então não há responsabilidade da camada Model em analisar esses dados.

2.1.2. View

De acordo com The Swift Documentation (Apple Inc., 2017):

> Objetos da camada *View* são objetos de uma aplicação que os usuários podem ver. Um objeto de *View* sabe como se desenhar e pode responder às ações dos usuários. O maior objetivo desses objetos é mostrar os dados presentes na camada *Model* do aplicativo e permitir a edição desses dados. Apesar disso, os itens da camada *View* tipicamente são desacoplados dos objetos da camada *Model*. (APPLE INC., 2017)

Estes objetos são programados geralmente para reuso e modificação constante, de maneira a prover uma consistência aos aplicativos que o usam. As bibliotecas mais utilizadas de objetos dessa camada são a *UIKit* e *AppKit*.

No aplicativo apresentado nas seções futuras, a biblioteca de objetos utilizada será a UIKit, então será dado um enfoque maior a essa biblioteca e seus objetos.

2.1.1.1 Biblioteca UIKit

A biblioteca UIKit permite ao usuário criar uma interface minimalista, gráfica e voltada ao usuário. Ao observar a Figura 02, esta biblioteca se encontra nos objetos de UI e "*views*" da camada *View*. Ela é voltada para os aplicativos que tem como sistema operacional iOS e tvOS. Esta biblioteca tem como principais funções:

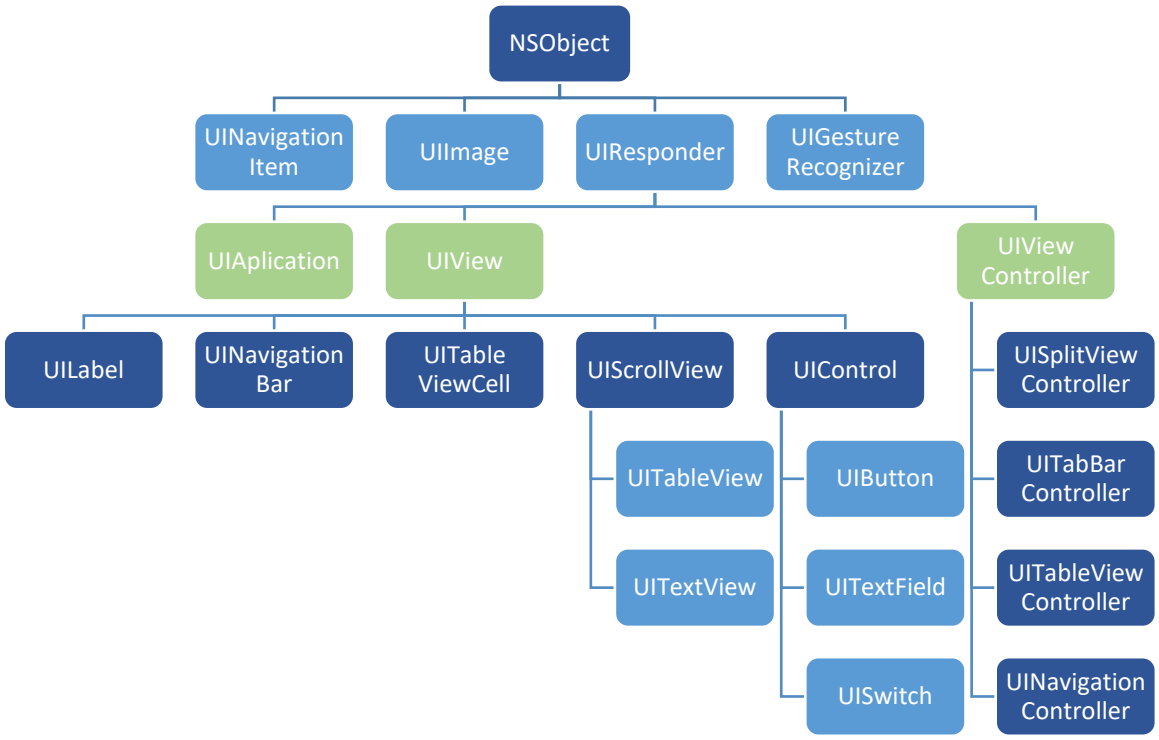
- Apresenta a arquitetura de visualização e janela para a implementação da interface;
- A capacidade de gerenciar eventos de maneira que seja possível utilizar multi-toque e outros tipos de entrada em um aplicativo;
- Provê o ciclo principal de maneira a gerenciar as interações entre usuário, sistema e aplicativo.

Em todos os aplicativos que se utilizam dessa biblioteca, é mandatória a presença de "*App icons*", que consiste no ícone único do aplicativo, que está presente principalmente na tela raiz do celular, também podendo estar em outras telas, como a tela de configurações, de modo a diferenciar o aplicativo dos demais para o usuário acessá-lo facilmente. Além disso, há necessidade da *Storyboard*, que é o conjunto de interfaces que serão apresentadas e utilizadas pelo usuário.

Quando o usuário clicar no ícone do aplicativo na tela raiz, ele deve imediatamente ir para a tela de abertura do aplicativo, de modo que o usuário saiba que o aplicativo foi aberto.

Na Figura 03, estão presentes as principais classes desta biblioteca e também as classes mais utilizadas pelo aplicativo desenvolvido. A figura também demonstra a hierarquia entre as classes:

Figura 03 – Principais classes da biblioteca UIKit



Fonte: FINALIZE.COM, 2017

Estas estruturas serão devidamente apresentadas e desenvolvidas no Capítulo 02, onde serão também exemplificadas dentro do programa.

2.1.3. Controller

A camada *Controller* (controlador) é o elo de ligação entre as camadas *Model* e *View*. É nesta camada que estão presentes as principais funções do programa. Existem dois tipos de objetos controladores: os de coordenação e de mediação. Os controladores de mediação estão presentes apenas em programação para computadores, então o foco será dado nos controladores coordenadores. O controlador então gerencia as funções, atuando principalmente em:

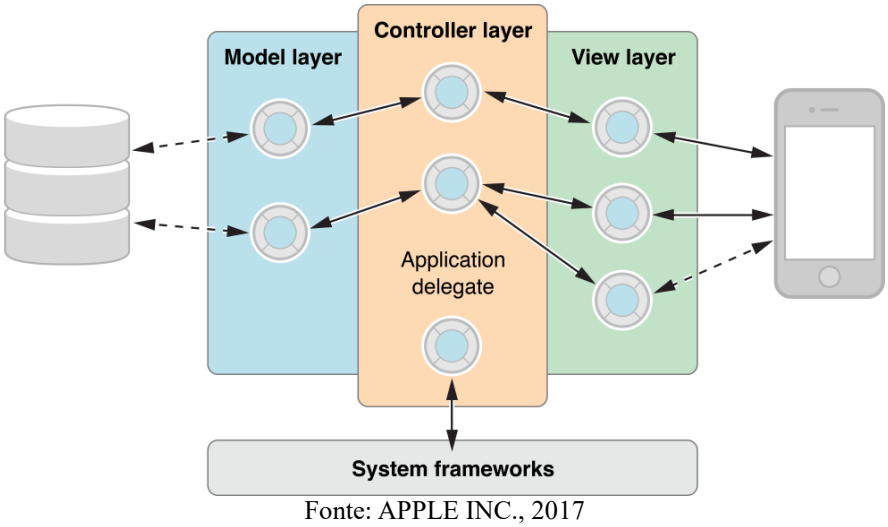
- Observar as notificações e responder às mensagens delegadas;
- Responder às mensagens de ação;
- Estabelecer conexões entre objetos e fazer tarefas de configuração;
- Gerenciar o ciclo de vida dos objetos da aplicação.

De acordo com The Swift Documentation (Apple Inc., 2017):

> Um objeto da Camada Controller atua como um intermediário entre os objetos da camada View e Model. Objetos de Controller são conduítes os quais objetos View aprendem sobre as mudanças nos objetos Model e vice-versa. Objetos de Controller também pode fazer tarefas de configuração e coordenação e gerenciar os ciclos de vida de outros objetos. (APPLE INC, 2017)

Os objetos dessa camada geralmente atuam seguindo o seguinte ciclo: eles interpretam ações do usuário nos objetos da camada *View* e comunicam essas ações aos dados da camada *Model*, alterando-os. Quando estes dados são alterados, eles informam ao *Controller* as mudanças, para que este mude os objetos então apresentados ao usuário.

Figura 04 – Camada Controller



Fonte: APPLE INC., 2017

Em programação para iOS, o controlador coordenador está presente como um "*View Controller*". Ele consiste em um bloco de texto de programação que envolverá todos os processos de uma certa interface. Como visto no tópico 2.1.2.1, existem cinco tipos de *View Controllers*, sendo que quatro deles são subclasses de *UIViewController*, ou seja, a raiz de todos é igual, mudando apenas detalhes e instâncias de acordo com a necessidade. Todos estes controladores possuem um arquivo de *View* anexo e eles controlam a apresentação e as transições desta interface.

Existem também as barras de ação, que são a barra de navegação e a barra de guia, que também são controladas por seus respectivos *View Controllers*. Também é responsabilidade do controlador monitorar os avisos de baixa memória e pela rotação do aplicativo de acordo com a rotação do aparelho.

2.2. Tipos de valores e variáveis em Swift

Como observado nas seções anteriores, a linguagem Swift foi projetada para ser uma linguagem simples, sólida e bastante descritiva. O primeiro passo para realizar operações na linguagem consiste na declaração de símbolos. Estes símbolos são declarados seguindo quatro passos simples:

- Definir a mutabilidade do símbolo, utilizando o termo *let* para constantes e *var* para variáveis. Constante podem ter seu valor atribuído apenas uma vez durante a compilação, enquanto variáveis podem ter seus valores alterados diversas vezes;
- Declarar o nome do símbolo;
- Declarar o tipo do símbolo, podendo ser um inteiro (*int*), um número decimal (*float, double*), um aglomerado de caracteres (*string*), um vetor (*array*), dentre outros tipos de valores. Isso previne erros, já que com o tipo explicitado, a função deste dado pode ser desenvolvida da maneira correta;
- Definir se o símbolo possui valor inicial ou não. Caso este símbolo tenha valor inicial definido, mas não seu tipo, será inferido um tipo de acordo com o dado recebido.

Assim, uma declaração completa desse símbolo deve seguir essas diretrizes como mostrado abaixo:

let meaningOfLife: Int = 42

Dessa forma, foi declarada uma constante “meaningOfLife” de valor 42.

A conversão de valores é bastante simples. Por exemplo, para converter uma certa variável inteira (*int*) para um número decimal (float), basta que coloque o tipo final da *conversão* e o nome do símbolo em parêntese. Utilizando o exemplo anterior, seria: *float(meaningOfLife)*. Dessa forma, a constante “meaningOfLife” teria seu valor convertido para um número decimal.

2.2.1. Optionals

Optional é um tipo de declaração de variável que significa que certo símbolo pode não conter valor nenhum (nesse caso chamado de “*nil*”). Caso um certo símbolo não seja declarado como *Optional* e possua este valor nulo, ao ser utilizado pelo programa ele causará o fechamento repentino (mais conhecido como “*crash*”). Justamente para evitar este tipo de situação em uma linguagem voltada para o desenvolvimento de aplicativos, essa opção foi criada.

No desenvolvimento, o valor retornado de uma variável *Optional* não é o mesmo de uma variável normal. Para isso, é importante que o desenvolvedor faça o desempacotamento (*unwrap*) desse símbolo. A maneira mais simples de fazer esta ação é colocando um ponto de exclamação no final do nome do símbolo ao utilizá-lo.

2.2.2. Strings

Uma *String* é uma coleção ordenada de caracteres. Uma variável que tem como valor uma string é facilmente reconhecida pela presença de aspas duplas em seu valor (por exemplo “*Hello World*”). Por ser uma coleção de caracteres, pode-se iterar facilmente os caracteres de uma String com um ciclo *for*.

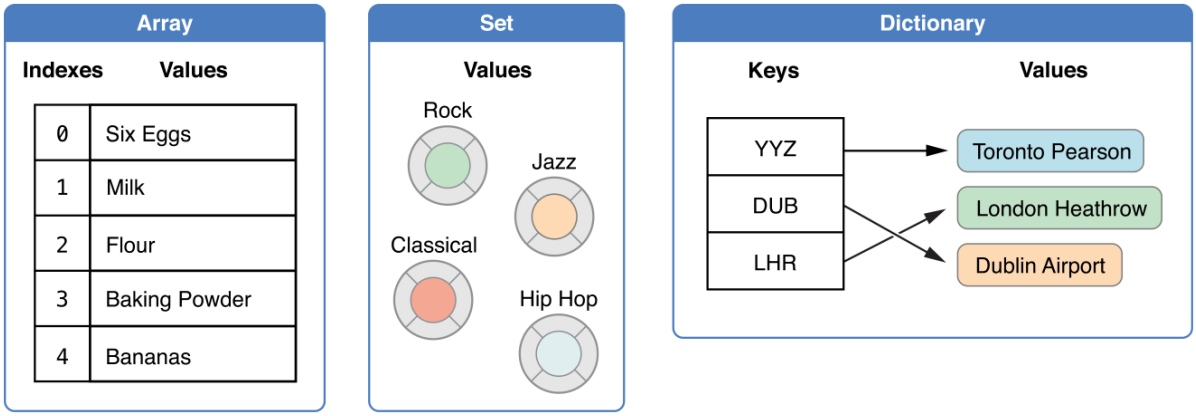
Além disso, *Strings* seguem o padrão e podem ser mutáveis ou não de acordo com as declarações podendo ter seus valores anexados a outras strings utilizando operadores simples.

2.3. Tipos de coleções

Existem três tipos primários de coleções em linguagem Swift: *Arrays* (vetores), *Sets* (conjuntos) e *Dictionaries* (dicionários). Além desses tipos, existem os *Tuples* que consistem

em colocar diversos valores atrelados a um certo símbolo. Nesta seção, serão explicados brevemente o funcionamento e as aplicações de cada um desses tipos de coleções.

Figura 05 – Tipos de coleções



Fonte: APPLE INC., 2017

2.3.1. Tuples

Um Tuple é uma coleção ordenada de valores em um determinado símbolo. Ele é declarado da seguinte maneira: após a declaração do nome da variável, coloca-se em parênteses o tipo de valores que o Tuple terá em cada um de seus elementos. Esses elementos de um tuple podem ser nomeados individualmente. Um exemplo de aplicação desse método é mostrado abaixo:

var local = (bairro: "Aldeota", cidade: "Fortaleza", estado: "Ceará", pais: "Brasil", continente: "América")

Dessa forma, foi obtido um Tuple contendo cinco Strings e cada uma dessas strings teve seu valor ligado a um determinado nome. Ou seja, ao utilizar "*local.bairro*", o valor recebido deverá ser "Aldeota". Também pode-se obter o mesmo resultado ao se referir ao valor pela sua posição dentro do Tuple.

2.3.2. Arrays

Array, também conhecido como vetor, é um tipo de coleção que armazena os dados de um mesmo tipo uma lista ordenada. A declaração de um *array* vazio consiste em escrever entre colchetes o tipo de valor armazenado seguido por parênteses. Um vetor pode conter vários inteiros, *strings* ou até mesmo *tuples*, sendo considerado um vetor de múltiplas dimensões.

Para acessar os dados de um *array*, utiliza-se o nome seguido da posição desse dado, iniciando de zero. Dessa forma, é possível iterar um vetor facilmente com o ciclo *for*.

Um vetor possui diversas funções inerentes a ele. Essas funções podem ser facilmente aplicadas utilizando o nome da *array*, ponto final e em seguida o nome da função. Algumas dessas funções serão exemplificadas abaixo:

- *Count*: conta o número de elementos presentes em um vetor;
- *Append*: anexa um valor ao vetor, ou seja, insere em sua última posição;
- *Insert*: insere um valor no vetor em uma determinada posição, fazendo com que os seguintes a ele sejam deslocados uma posição;
- *Remove*: semelhante ao *insert*, porém com a função de remover o valor de uma certa posição;
- *Sort e Sorted*: a função *sort* ordena o vetor de acordo com uma condição estabelecida no código. A diferença para a função *sorted* é que nesta é criado um novo *array* para armazenar estes valores ordenados;
- *Map:* Retorna um novo vetor com as regras definidas por esta função aplicadas a ele.

2.3.3. Sets

Um *set*, assim como um *array*, é um conjunto de elementos que obrigatoriamente é do mesmo tipo. A diferença entre os dois se dá pelo fato de o *set* não ser uma coleção ordenada. A grande vantagem dessa coleção sobre um vetor consiste no fato de que as funções de um set são mais eficientes que as do vetor caso a ordem dos elementos não seja de grande importância. Além disso, o set é utilizado caso seja necessário garantir que o elemento apareça apenas uma vez em sua coleção.

2.3.4. Dictionaries

Um dicionário consiste em uma coleção não ordenada que se divide em duas partes: *keys* (chaves) e *values* (valores). As chaves de um dicionário funcionam como um set: possuem obrigatoriamente o mesmo tipo. Já os valores podem ser completamente diferentes uns dos outros.

O que torna este tipo de coleção único é o fato de que cada chave aponta diretamente para seu valor. Então, o dicionário tem sua principal utilização em condições que é necessário achar um valor pelo seu identificador único, assim como em um dicionário gramatical.

A declaração de um *dictionary* vazio consiste em escrever entre colchetes o tipo de valor de sua chave, seguido por dois-pontos (“: ”), escrevendo então o tipo de valor de seu *value*, que pode ser do tipo *Any*, significando então que pode ter qualquer tipo de valor.

O tipo de iteração que ocorre em um dicionário também é feito através do ciclo *for*, porém neste caso é um ciclo especial chamado de *for-in*, em que, no lugar de utilizar uma variável que crescerá de valor, este ciclo iterará por cada uma das chaves do dicionário, fechando-se quando o dicionário tiver sido iterado por completo.

2.4. Fluxo de Controle

O fluxo de controle é o cérebro do desenvolvimento de um programa, seja ele um aplicativo de computador, móvel, ou até mesmo uma página da Web. A programação é desenvolvida para ser lida em sucessão. Apenas através de loops e condicionais, o programa pode operar da maneira mais correta e eficiente.

Dessa forma, serão explicadas nesta seção cinco funções de fluxo de controle que serão mais importantes no desenvolvimento de aplicativos em linguagem *Swift*: os ciclos *for* e *while*, e os condicionais *if*, *switch* e *guard*. Essas funções são bastante parecidas com as funções homônimas em linguagem C, o que facilita o entendimento. A grande diferença dessas funções nas duas linguagens é que no *Swift* não é necessária a presença de parênteses quando declarada a condição.

2.4.1. Ciclo “for”

O ciclo *for*, é conhecido em linguagem Swift também como ciclo “*for-in”*. Esse ciclo é bastante similar ao ciclo de mesmo nome em linguagem C. Em linguagem *Swift*, pode-se iterar tanto uma variável auxiliar uma quantidade de vezes, como pode-se iterar elementos de um conjunto.

Um exemplo de iteração de elementos é a iteração pelos dados de um dicionário. Através desse ciclo, pode-se acessar todos os dados de um dicionário, mesmo estes estando desordenados.

2.4.2. Ciclo "while"

O ciclo *while* executa um bloco de código até que a condição explicitada nele seja atingida. Existe também o ciclo *"repeat-while"*, que é muito próximo, mas tem como principal diferença o teste da condição após a ação ser realizada. O ciclo *while* testa a condição antes de realizar o bloco de código.

2.4.3. Condicional "if"

O condicional *if* é o tipo de declaração mais simples em um programa, mas também é uma das mais importantes. Este condicional testa uma afirmação e executa o bloco de código se esta afirmação for verdadeira. O condicional if normalmente tem associado a ele um condicional *"else"*, que traduzido para português significa "senão". Então em uma associação *"if-else"*, sempre um bloco de código condicional será executado.

Além da utilização de tratar condicionais em um programa, o *if* pode ser utilizado para desempacotar um *Optional*. Para isso, declara-se o condicional, seguido pela declaração de um símbolo, um sinal de igual ("=") e um valor. Esse tipo de sintaxe na declaração é chamado de *"conditional binding"* (ligação condicional). Dessa forma, ele testará o valor de um *Optional* e, caso ele não seja nulo, será desempacotado, este valor será associado ao símbolo e o bloco será executado com este valor de variável.

2.4.4. Condicional "switch"

O condicional *switch* considera um valor em sua declaração e testa este valor de acordo com várias declarações de valores. Então, executa o código de valor correspondente. Toda declaração de *switch* possui diversos valores possíveis. Esse método é uma simplificação do condicional *if*, para que não sejam utilizadas diversas linhas de código declarando *"if-else"*.

Além da declaração de valores diferentes, pode-se declarar um *underscore* ("_") de maneira que todos os valores sejam absorvidos e o código seja executado. Como o dado é analisado de acordo com a ordem de declaração, geralmente o *underscore* é deixado por último, de maneira que todas as possibilidades sejam testadas.

2.4.5. Condicional "guard"

O *guard*, assim como o *if*, testa uma condição e executa o código caso a condição seja verdadeira. Uma declaração de *guard* obrigatoriamente possui uma declaração de *else*. Além disso, ao ser executada, a condição de *else* deve sempre sair do fluxo do condicional *guard*. Isso pode ser realizado utilizando algum dos comandos como return, break ou então fazendo com que uma função seja executada.

A principal utilização de guard é quando deseja-se sair de uma função antes do código ser lido por completo, ou seja, pode ser utilizado para evitar fechamentos repentinos do programa ou quando um valor não está dentro dos limites esperados.

Caso a condição seja verdadeira, o condicional else é ignorado e o código continua a ser executado normalmente.

2.5. Funções

Funções são fatores importantíssimos na camada Controller. Elas são importantíssimas na análise dos dados presentes na aplicação. No livro "The Swift Programming Language" (APPLE INC, 2014), há uma breve explicação da definição das funções na linguagem:

> Funções são pedaços de código autônomos que executam uma tarefa específica. Você dá um nome a uma função que identifica o que ela faz e esse nome é utilizado para "chamar" a função para executar a tarefa quando necessária. (APPLE INC, 2014)

Para declarar uma função, primeiramente deve-se usar a palavra "*func*" seguida do nome da função. Em seguida, pode-se colocar entre parênteses valores que a função utiliza com entradas. Estes valores são chamadas de parâmetros.

As funções não são obrigadas a possuir parâmetros, mesmo assim devem possuir parênteses. Além disso, a função pode retornar ao programa com um determinado valor. Esse valor pode ser uma mensagem ou estar inserido em uma variável.

A declaração de um parâmetro se divide em três partes: um rótulo, para facilitar o entendimento conforme o programa vai se desenvolvendo, o nome do parâmetro e o tipo de variável. O rótulo pode ser omitido caso não seja necessário.

Além disso, as funções podem retornar valores para a aplicação de modo que a aplicação receba estes dados. Na declaração, deve-se escrever uma flecha (“->”) após a declaração de parâmetros seguida do tipo de valor retornado pela função.

Uma função que possui um valor retornado deve possui uma declaração com “return” seguida pelo valor executado. Essa declaração também finaliza a execução da função. Uma função pode ter mais de uma declaração de retorno, mas caso o código que suceda essa declaração nunca seja utilizado, o compilador gerará uma advertência.

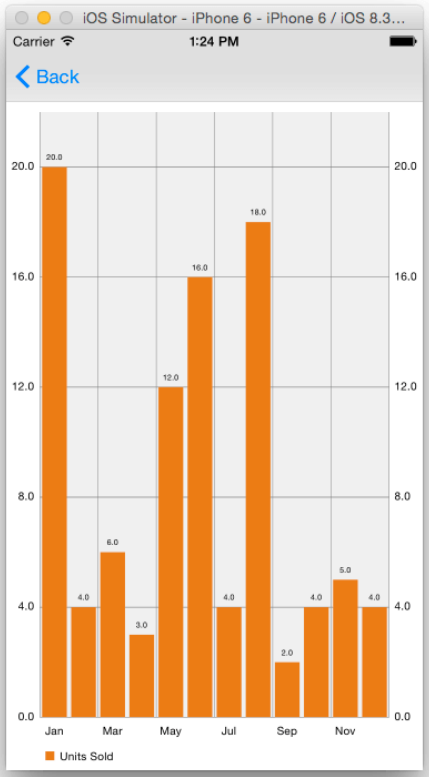
2.5. O API Charts

Uma Interface de programação de aplicações, em inglês *Application Programming Interface*, é um conjunto de comandos, funções, protocolos e objetos que são utilizados em linguagem de programação para desenvolver programas. Um exemplo principal de um API presente no aplicativo é o próprio iOS API. Esta interface possui todos os elementos presentes em um aplicativo móvel: detecção de entrada *touchscreen*, rotação, teclado virtual, barra de pesquisa e barra de guia. Além disso, ele possui as ferramentas para interação com o hardware do aparelho, como interação com câmera, caixas de som e microfones.

Diante dessa breve explicação, no aplicativo de monitoramento desenvolvido, foi utilizado um API chamado de Charts. Este API foi desenvolvido por Daniel Cohen Gindi e Philipp Jahoda e tem como principal função prover ferramentas para a fácil criação e gerenciamento de gráficos. Um exemplo de gráfico está presente na Figura 06. Como pode-se perceber, é uma interface de utilização muito amigável. Essa ferramenta é capaz de produzir os mais diversos gráficos, como gráficos de barras, linhas ou de pizza.

O API possui licença do tipo Apache, permitindo o uso e distribuição do código fonte em aplicações *open source* ou proprietárias. É necessário, no entanto, a inclusão do aviso de *copyright* e um aviso legal para a utilização do API.

Figura 06 – Gráfico de barras desenvolvido com a API



Fonte: ECHESSA, 2017.

Além de produções de gráficos simples, a ferramenta também é capaz de fundir gráficos, de modo que as comparações sejam mais acessíveis. Também podem ser feitas animações, linhas de limite e o gráfico gerado também pode ser salvo caso haja um botão que habilite essa função.

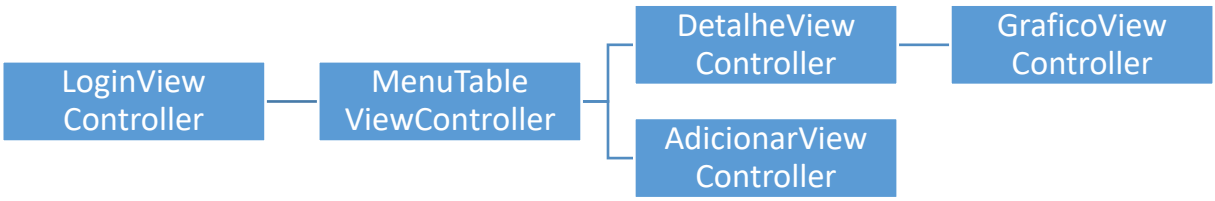
3. CAPÍTULO 02: INTERFACE HOMEM-MÁQUINA

Neste capítulo, será apresentada a Interface Homem-Máquina do aplicativo. Serão demonstrados todos os objetos utilizados no programa que de alguma maneira se conectam ao usuário.

Além disso, serão apresentadas as ligações entre as diferentes interfaces *ViewController*, chamadas de *segues*, que farão tanto a transição de interfaces quanto a transição de dados entre os *ViewController.*

A apresentação de cada uma das interfaces presentes no programa será feita primeiramente com a exposição da interface que o usuário visualizará. Em seguida, haverá uma explicação de como essa interface se conecta ao código de programação. No final da apresentação da interface serão mostradas as possíveis ligações com outras telas do programa, e que dados serão repassados entre telas.

Figura 07 – Esquemático das interfaces apresentadas e suas conexões



Fonte: o próprio autor

3.1. Tela de Login *"LoginViewController"*

A tela inicial do aplicativo consiste em uma tela simples de *login*. Esta tela possui vários elementos para interação com o usuário. Estes elementos são apresentados e numerados na figura 08. A tela de Login além de ser a tela inicial do aplicativo, é a maneira com que o usuário pode se identificar ao sistema.

Esta interface consiste em um *View* comum, embutido em um *NavigationController*. A interface simples fica conectada à interface de navegação presente em um *NavigationController*. Dessa maneira, a barra de navegação ficará presente durante toda a apresentação de interface do programa. Em algumas interfaces, outros itens de navegação podem estar presentes além da etiqueta com o nome da tela apresentada.

Figura 08 – Elementos da Tela de Login

Fonte: o próprio autor

Os elementos presentes na tela têm as seguintes funções:

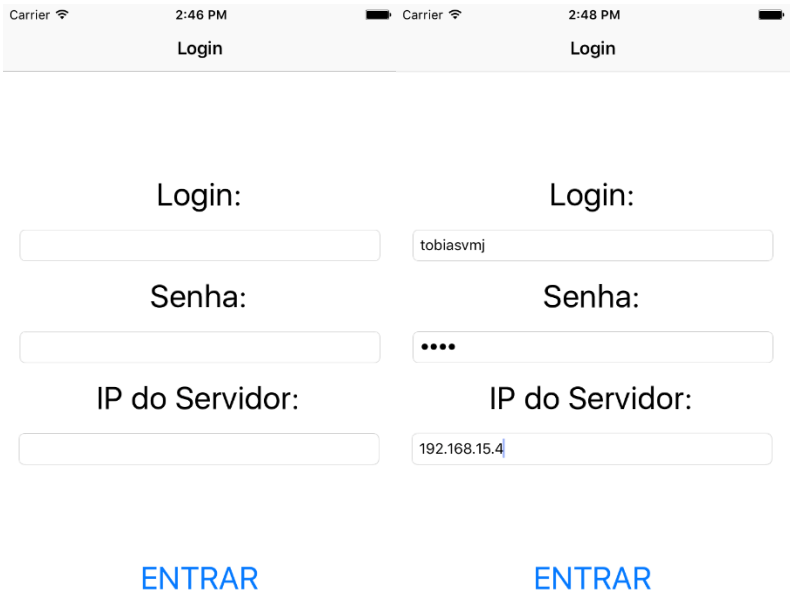
1. *Navigation Bar*: é a barra de navegação principal do programa. Estará presente durante toda a apresentação do aplicativo e possui uma etiqueta que nomeia a interface apresentada ao usuário;
2. *UILabel*: consiste em um texto de cor vermelha, que só será apresentado caso os dados de nome de usuário e senha não estejam presentes no servidor. Na ocorrência dessa situação, essa etiqueta apresentará o texto “Usuário e/ou Senha incorretos;
3. *UITextField*: é o local onde o usuário poderá digitar sua identificação. Ao clicar na caixa de texto, o teclado virtual do telefone aparecerá e o usuário poderá digitar o texto. Essa característica é comum a todos os *UITextField* presentes na aplicação;
4. *UITextField*: similar ao apresentado no item 3, porém, nessa caixa de texto o usuário digitará sua senha. Por ser um dado secreto, foi ativada um item de interface que torna o texto seguro. Assim, o texto não será apresentado

conforme o usuário digita. O que aparecerá na caixa de texto serão apresentados círculos pretos (“●”) em que o usuário poderá ver apenas a quantidade de caracteres digitados, mas não o conteúdo;

5. *UITextField*: também similar aos itens 3 e 4, porém nesse caso o usuário digita a informação do endereço de IP do servidor que será conectado ao aplicativo;
6. *UIButton*: é um botão que, ao usuário clicá-lo, ele fará dentro do código as validações de usuário e senha e, caso sejam falsas, acionará a etiqueta do item 2. Caso as informações sejam verdadeiras, o aplicativo fará a ligação dessa interface com a interface de menu presente na seção 3.2. Mais detalhes sobre essa ligação são explicados na seção 3.1.1.

Com esses elementos, a interface final da tela de autenticação de *login* é obtida, conforme mostrado na Figura 09.

Figura 09 – Interface de Login apresentada ao usuário



Fonte: o próprio autor

O usuário, de modo a prosseguir com o programa deve fazer quatro interações com a interface: três para a digitação de dados e uma delas pressionando o botão que autenticará os dados informados. Caso os dados informados estejam incorretos, aparecerá uma mensagem ao

usuário como mostrado na Figura 10. Ele então deverá reescrever os dados preenchidos até que estes estejam iguais a algum dos usuários presentes no servidor.

Figura 10 – Apresentação de etiqueta de erro ao ocorrer uma falha na autenticação de dados

Fonte: o próprio autor

No código de programação do aplicativos, as três caixas de texto e a etiqueta que apresenta o erro de usuário ou senha aparecem como *Outlets*. Um *outlet* é uma conexão de um objeto da camada *View* com a camada *Controller*. Com essa conexão presente, os dados inseridos e as alterações feitas pelo usuário podem então ser então informadas às funções que dependem dessas informações.

```
@IBOutlet weak var login: UITextField!
@IBOutlet weak var senha: UITextField!
@IBOutlet weak var ip: UITextField!
@IBOutlet weak var Errorlabel: UILabel!
var ipi: String?
```

Como pode-se observar pelo código acima, a declaração de um *Outlet* é muito similar à declaração de uma variável comum. Essas variáveis funcionam a partir da biblioteca *UIKit*, e a maioria funciona como classes ou estruturas. Além dos Outlets declarados, há a declaração da variável ipi, que consiste em uma *optional string* e armazenará o endereço de IP do servidor para transmiti-lo aos demais *ViewControllers*.

Em Controladores de um aplicativo iOS, existem funções padrão que sempre estarão presentes. Uma dessas funções interage diretamente com o usuário: a função “viewDidLoad”. Dentro desse bloco de função deve estar presente todas os dados necessários para inicializar a interface.

```
override func viewDidLoad() {
    super.viewDidLoad()
    Errorlabel.text = ""
}
```

Nesse caso, a única ação necessária antes do carregamento da página é que, ao apresentar a interface para o usuário, não houvesse nada escrito na etiqueta de erro de autenticação.

O último objeto da interface presente no texto de programação é uma ação. A ação é uma função, então ela executa um bloco de código ao ser ativada. Além disso, a ação pode ter sido enviada por algum objeto, que seria um parâmetro da função. Nesse caso, a ação é criada pelo botão “ENTRAR”, logo, ao pressionar esse botão, o bloco de código será executado.

```
@IBAction func Entrar(_ sender: Any) {
    getDadosUser()
}
```

O bloco de código presente é bem simples e consiste apenas em chamar outra função dentro do código. Essa função é responsável por retirar os dados do servidor e testá-los com os dados inseridos pelo usuário. Ela será mais detalhada no capítulo 4, assim como as demais funções que se conectam ao servidor.

3.1.1. Ligação: “LoginSegue”

A tela de login possui apenas uma conexão com outra interface. Essa conexão se dá através de um segue que se chama “*LoginSegue*”. A conexão pode se dar entre um objeto de um *View* e uma interface *ViewController* ou entre duas interfaces *ViewController*. No caso deste aplicativo, todas as conexões se deram entre interfaces *ViewController*.

Dentro das funções padrões de ViewControllers, existe a função *"prepare(for segue:)"*. Esta função atua tratando e enviando dados à próxima interface para que esta os utilize. O bloco dessa função em "*LoginViewController*" é mostrado abaixo:

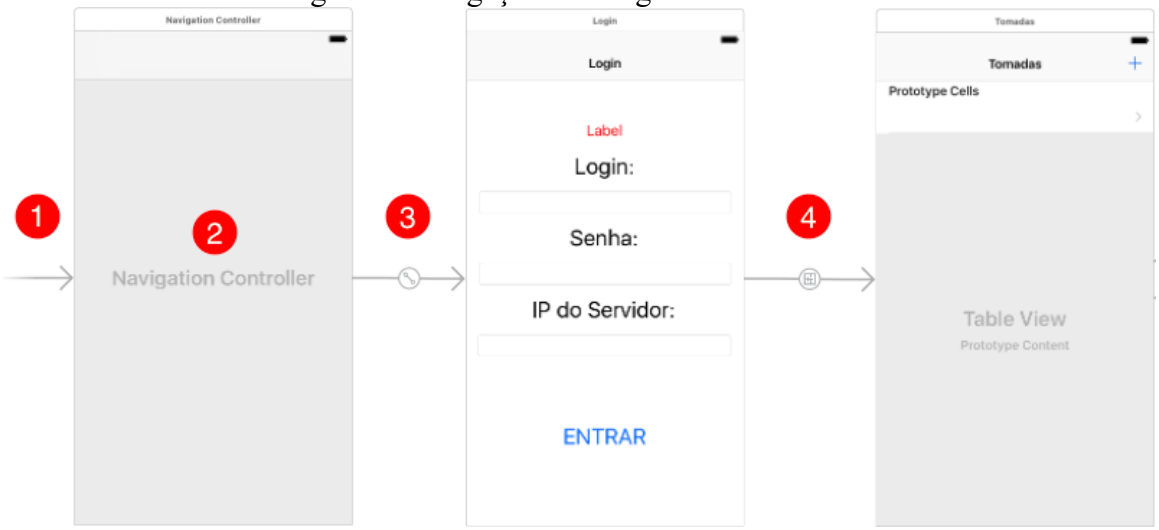
Dentro das funções padrões de ViewControllers, existe a função *"prepare(for segue:)"*. Esta função possui dois parâmetros: o tipo de segue utilizado e quem acionou a função. Ela atua tratando e enviando dados à próxima interface para que esta os utilize. O bloco dessa função em "*LoginViewController*" é mostrado abaixo:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if (segue.identifier == "LoginSegue") {
        let VC = segue.destination as! MenuTableViewController
        VC.ipi = ipi
    }
}
```

Na primeira linha dentro da função, ocorre um teste do identificador do segue. Caso o teste seja verdadeiro, sendo o segue acionado "LoginSegue", declara-se que uma variável "VC" é o destino do segue, sendo o nome do destino "MenuTableViewController". Por último, uma das variáveis dentro da interface de destino recebe o valor da variável "ipi" de "LoginViewController". Essa variável possui o endereço de IP do servidor conforme escrito pelo usuário, para que as demais interações do programa com o servidor possam acontecer. Essa variável será transmitida por praticamente todas as interfaces do programa.

Na Figura 11 estão presentes as ligações estre a tela de Login e as demais telas. Estas ligações estão divididas e numeradas em quatro partes:

Figura 11 – Ligações de LoginViewController



Fonte: o próprio autor

Os elementos presentes na tela têm as seguintes funções:

1. Esta seta significa que o *ViewController* relacionado a ela é a raiz do programa. A tela inicial do programa seria então o *Navigation Controller*;
2. *Navigation Controller*: A tela de *login* está embutida em um *Navigation Controller*, de modo que a navegação dentro do aplicativo seja permitida. Este controlador não possui um *View*, mas ele funciona como um contâiner de *ViewControllers*: ele mantém controle sobre os demais *ViewControllers* da aplicação, fazendo uma mistura do conteúdo presente nele com os demais. Dessa forma, tanto a navegação quanto a barra de navegação serão elementos presentes em todos os *ViewControllers* dentro do *Navigation Controller*;
3. *Relação "root view controller"*: através do *Navigation Controller*, essa relação, faz com que todos os *ViewControllers* à direita do *Navigation* possuam as características de navegação dele;
4. *LoginSegue*: é a representação gráfica, feita no *Storyboard*, da ligação presente entre a tela de *login* e o menu de tomadas.

3.2. Tela de Menu *"MenuTableViewController"*

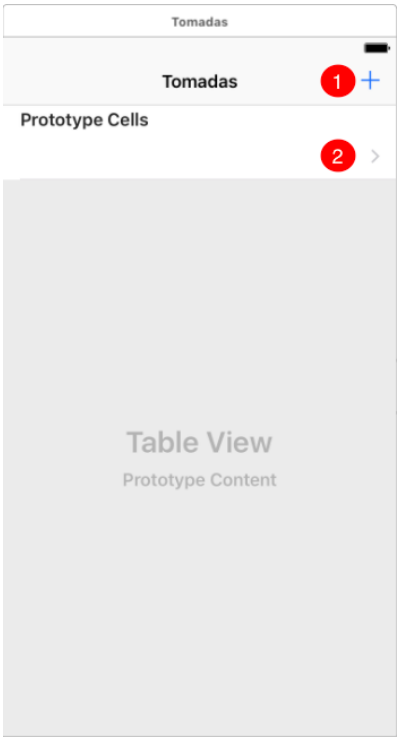
A próxima interface a ser analisada consiste na tela de menu do programa. Essa tela funciona como elo entre as demais funcionalidades do aplicativo. É nesta tela em que será

possível acessar a tela de adicionar tomadas e também acessar os detalhes de consumo de cada tomada.

Esta interface é um *TableViewController*: é uma subclasse do ViewController, de fácil organização, em que se divide em células. Um exemplo clássico de um *TableViewController* consiste no próprio menu de configurações do iPhone, em que várias células, com vários tipos de valores atrelados a ela se unem em uma interface de fácil manuseio para o usuário.

Existem dois tipos de *TableViewController*: o estático, em que as células presentes são configuradas pelo programador e se apresentam sempre da mesma maneira ao usuário e o dinâmico, em que células podem ser adicionadas ou retiradas e aparecem conforme rotinas estabelecidas pelo programador. Nesse caso, para ter o controle de adicionar as tomadas desejadas, foi utilizado um controlador dinâmico, que adiciona novas células de acordo com a adição de novas tomadas.

Figura 12 – Elementos de MenuTableViewController



Fonte: o próprio autor

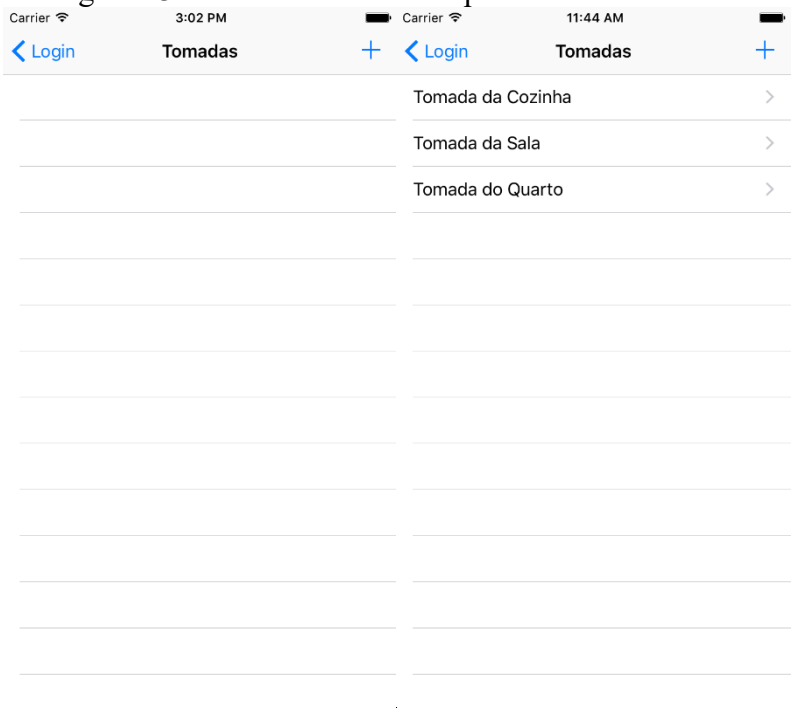
Esta interface possui apenas dois elementos em que possuem interação com o usuário. Além desses elementos, a etiqueta presente na navegação continua presente, como explicado na seção 3.1.

1. *UINavigationButton*: Este botão é a ligação entre a tela de menu e a tela de adicionar novas tomadas, por isso ele possui como etiqueta o símbolo de adição (“+”). Esta ligação será mais detalhada na seção 3.2.1;

2. *UITableViewCell*: A célula é o elemento principal do TableViewController. Ela proporciona organização e facilidade de manuseio pelo usuário. Esta é uma célula bem simples, que possui apenas uma etiqueta que terá o mesmo nome da tomada ligada a ela. Ao ser pressionada, a célula também funciona como um segue que direciona o usuário para a tela de detalhes da tomada escolhida.

Na figura 13, estão representadas a interface do sistema na primeira visualização da tela e também após alterações feitas pelo usuário.

Figura 13 – Interface de Menu apresentada ao usuário



Fonte: o próprio autor

A configuração de um TableViewController, dentro de um arquivo de classe de Swift, deve ser feita através de várias funções que pertencem ao TableViewController. Primeiramente deve-se analisar quais variáveis serão utilizadas pela classe que possuirão algum de tipo de interação com o usuário.

```
var tomadas = [String]()
var codigos = [String]()
var soids = [String] ()
var newTomada: String = ""
var newCod: String = ""
var newId: String = ""
var valueToPass:String!
var valueToPass2:String!
var valueToPass3:String!
var ipi:String!
```

Nesse caso, apenas duas variáveis possuirão essa característica: o vetor de tomadas, em que o nome de todas as tomadas presentes no programa ficará armazenado e auxiliará na nomeação das células e uma string, que armazenará o nome da tomada recentemente adicionada e inserirá esse valor ao vetor mencionado anteriormente.

As demais variáveis serão utilizadas nas ligações diretas ou indiretas realizadas pelo menu. Dentre estas, destacam-se os vetores “codigos” e “soids”, que assim como o vetor “tomadas”, armazenarão informações importantes sobre as tomadas cadastradas no aplicativos.

Em seguida, deve-se obter a função “viewDidLoad” do programa. Nesse caso, como as atualizações de UI estão diretamente conectadas a outros ViewControllers, não há nenhum carregamento inicial de dados para o usuário.

```
override func viewDidLoad() {
    super.viewDidLoad()
}
```

Em seguida, é feita a configuração do próprio ViewController. Duas funções simples realizam essa configuração: a primeira define o número de seções presentes no controlador e a segunda define o número de linhas, ou seja, o número de células presentes na aplicação.

```
override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
}

override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
    return tomadas.count
}
```

No código presente acima, a função de número de seções tem como parâmetro apenas o próprio *TableView* e retorna um valor inteiro. Como possui apenas uma seção de células no programa, o valor retornado é 1.

A segunda função tem dois parâmetros: o *TableView* e o número de linhas em uma seção, que consiste em um valor inteiro. Essa função retorna um valor inteiro que, para essa aplicação, será o número de elementos presentes no vetor “tomadas”.

É descrita em código a rotina responsável por nomear as células com o nome das tomadas respectivas a ela. Esta função ainda está dentro das funções de TableView.

```
   override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
       let cell = tableView.dequeueReusableCell(withIdentifier: "tomadaCell", for: indexPath)
       cell.textLabel!.text = tomadas[indexPath.row]
       return cell
   }
```

Essa rotina consiste em uma função de dois parâmetros que retorna uma UITableViewCell. Estes parâmetros consistem no TableView a ser modificado e no endereço (index) da linha a ser alterada pelo programa.

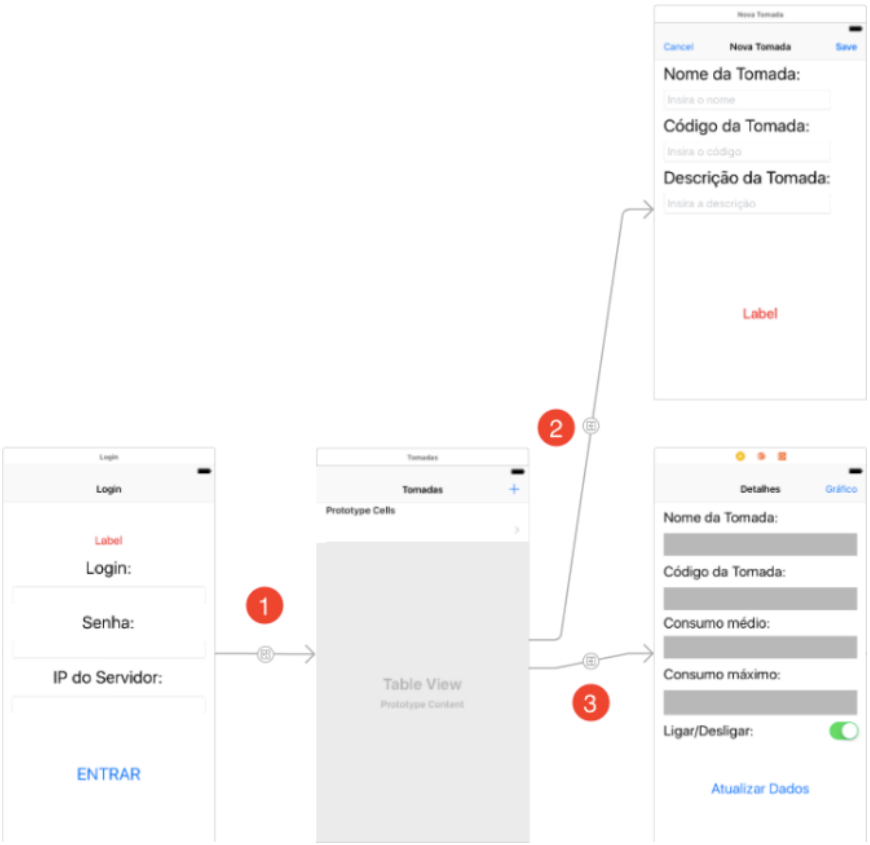
Uma vez que é definida a função, é definida uma constante dentro da instância “cell”, que representará a célula a ter o seu texto modificado. Nesse caso foi utilizada a função dequeueReusableCell, que retorna uma célula de acordo com o identificador e a posição desta célula.

Esta célula retornada, tem então seu texto alterado para o valor presente no vetor de tomadas de acordo com a linha que terá seu valor adicionado. Por exemplo, ao adicionar a terceira linha ao TableView, o nome da célula será correspondente à terceira posição no vetor de tomadas.

3.2.1. Ligações de “MenuTableViewController”

A interface de menu possui três ligações diretas e duas ligações indiretas. As ligações diretas estão presentes na Figura 14, enquanto as indiretas são feitas pela interface de adição de novas tomadas e serão detalhadas nas seções 3.3.1.1 e 3.3.1.2.

Figura 14 – Ligações de MenuTableViewController



Fonte: o próprio autor

1. *LoginSegue*: Conexão entre a tela de Login e a tela de Menu. Esta ligação foi detalhada em 3.1.1;
2. *AdicionarSegue*: Ao pressionar o botão com o símbolo de adição, esse segue é realizado. É a ligação entre a tela de menu e a tela de adição de novas tomadas;
3. *DetalheSegue:* Ligação mais complexa presente no aplicativo. Ao pressionar uma das células, o controlador deve identificar a célula pressionada para realizar esse segue.

3.2.1.1. Ligação: "AdicionarSegue"

O segue apresentado tem como função ligar a tela de menu à tela de adicionar uma nova tomada. Diferente do segue apresentado em 3.1.1, este é ligado diretamente ao botão de adicionar, então sua ativação é direta, não sendo necessário linhas de código para a mudança na interface.

Por ser raiz de duas ligações, a função “prepare(for segue:)” é dividida também em duas partes. Para facilitar a compreensão, a função será dividida e apresentada em ambas as seções com sua ligação correspondente.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if (segue.identifier == "AdicionarSegue") {
        let VC = segue.destination as! AdicionarViewController
        VC.ipi = ipi
    }
}
```

Nesse caso, a função de preparar para a apresentação da nova interface é bastante similar à apresentada em 3.1.1. Apenas a string com o valor do endereço de IP é repassada à próxima interface.

3.2.1.2 Ligação: “DetalheSegue”

Como a tela de detalhe possui todas as informações referentes à tomada escolhida, o controlador deve fazer uma análise de qual célula foi selecionada. Essa análise é realizada através de mais uma das funções de TableView. Por conta da necessidade dessa análise, é necessário que o segue seja feito entre os ViewControllers, não sendo ligado a nenhum objeto.

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    let indexPath = tableView.indexPathForSelectedRow!
    let currentCell = tableView.cellForRow(at: indexPath)! as UITableViewCell
    valueToPass = currentCell.textLabel?.text
    valueToPass2 = codigos[(indexPath.row)]
    valueToPass3 = soids[(indexPath.row)]
    performSegue(withIdentifier: "DetalheSegue", sender: self)
}
```

Através do código apresentado acima, é possível verificar a utilização de vários valores declarados no início da seção 3.2. É possível verificar que a função possui como parâmetro, assim como todas as funções de *TableView* presentes até agora, o próprio *View*, além do endereço da célula pressionada pelo usuário.

Dentro da rotina então ocorrem a declaração de duas constantes que armazenam o endereço da célula selecionada e a própria célula. O código então passa a utilizar as variáveis auxiliares “valueToPass” para definir os valores de nome, código e número de identificação da tomada escolhida. Para finalizar a função é feito o *segue* entre a tela de menu e a tela de detalhes. A função de preparo está descrita abaixo:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
   if (segue.identifier == "DetalheSegue") {
         let VC = segue.destination as! DetalheTomadaViewController
         VC.ipi = ipi
         VC.name = valueToPass
         VC.cod = valueToPass2
         VC.soid = valueToPass3
      }
}
```

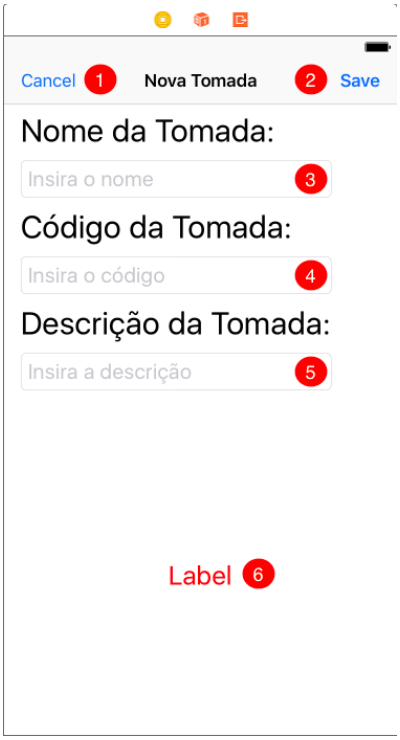
Da mesma maneira como foi explicado nas seções anteriores, é testado o segue que foi selecionado, em seguida, definido o *ViewController* destino do *segue*. Nesse caso, para que haja validação das conexões com o servidor, são repassados ao próximo ViewController os valores do endereço de IP do servidor, nome, código e identificação da tomada, para que, ao se conectar com o servidor, haja validação dos dados presentes nesse servidor.

3.3. Tela de Adicionar Nova Tomada *“AdicionarViewController”*

Como explicado na seção anterior, ao pressionar o botão presente em *TableViewController* com o símbolo de adição, a interface de adição de uma nova tomada ao aplicativo é apresentada. Esta interface também consiste em um *View* simples, com vários elementos de interação com o usuário. Ela possui muitas semelhanças com a tela de *login*, pelo fato de ambas possuírem caixas de texto para inserção de dados que necessitem de validação por parte do servidor, além de ambas possuírem etiquetas de erro de validação. A principal diferença entre elas é o tipo de dado inserido e a ligação delas com outras interfaces.

Na figura 15, estão numerados todos os elementos presentes na interface de adicionar novas tomadas.

Figura 15 – Elementos de AdicionarViewController



Fonte: o próprio autor

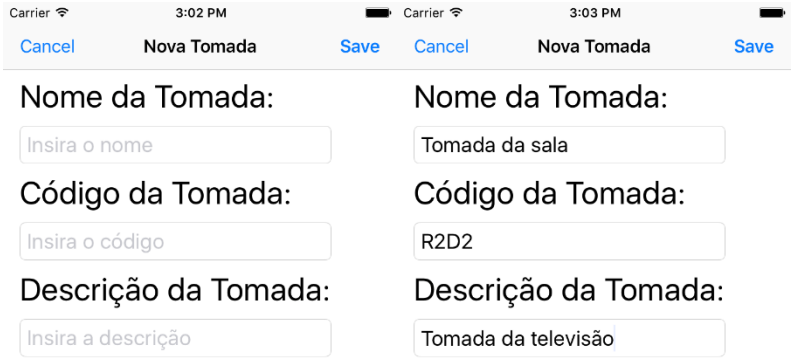
Os elementos presentes na tela têm as seguintes funções:

1. *NavigationBarButton*: Este botão, que possui a etiqueta "Cancel" ligada a ele, é utilizado juntamente com um "*unwind segue*", para cancelar imediatamente a interface de adicionar uma nova tomada e voltar para a interface de menu;
2. *NavigationBarButton*: Diferentemente do primeiro botão, o botão com a etiqueta "Save" tem como função chamar a função de validação do código da tomada, além da função para alterar o nome da tomada no servidor e então fazer outro *"unwind segue"* para voltar à tela de menu das tomadas;
3. *UITextField*: Esta caixa de texto é preenchida pelo usuário com o nome desejado para a tomada relacionada ao código tenha. Consequentemente, essa informação digitada será repassada ao servidor;
4. *UITextField*: O código da tomada, que será digitado pelo usuário nesta caixa de texto, será responsável por identificar e validar os dados preenchidos pelo usuário. Ele funciona como uma identificação da tomada e é único dentro do servidor;

5. *UITextField*: Similar aos itens 3 e 4, é uma caixa de texto onde o usuário poderá digitar a descrição da tomada, caso ele deseje mais detalhes sobre a tomada adicionada;
6. *UILabel*: Etiqueta de erro similar à presente na seção 3.1. Ao testar a validação de uma tomada, caso o resultado seja falso, a etiqueta receberá o seguinte texto: “Código não encontrado” e então o usuário terá que corrigir o seu texto de maneira que o código seja validado.

A interface final mostrada ao usuário é representada pela Figura 16.

Figura 16 – Interface de Adicionar Tomada apresentada ao usuário



Fonte: o próprio autor

Esta interface se conecta ao código através de quatro *outlets* e um ação. Os quatro outlets estão mostrados abaixo e representam as três caixas de texto e a etiqueta de erro. Esses objetos se assemelham muito com os objetos apresentados em 3.1.

```
@IBOutlet weak var tomadanome: UITextField!
@IBOutlet weak var tomadacod: UITextField!
@IBOutlet weak var tomadadesc: UITextField!
@IBOutlet weak var codigonaorec: UILabel!
var codigoarray = [String]()
var codigo: String = ""
```

O conteúdo presente dentro da classe dos outlets de *UITextField* será utilizado pelo aplicativo para troca de dados com o servidor.

Além dos quatro *Outlets*, também foram declaradas algumas variáveis. Nesta seção serão explicadas apenas duas delas, que de alguma forma interferem na execução da interface. A primeira delas é um vetor de *Strings*, que será de suma importância para a validação do código digitado, podendo interferir tanto na execução do segue quanto na apresentação da mensagem de erro. A segunda variável é uma variável auxiliar que armazena apenas uma *string*. A terceira variável está presente em todos os arquivos de código do programa que tem algum tipo de interação com a internet e armazena o endereço de IP do servidor.

Da mesma forma que na tela de Login, o UILabel só será apresentado em caso de erro de validação. Nos demais momentos da execução do aplicativo, o conteúdo dele será vazio. Isso é determinado pela função viewDidLoad, que tem como única função deixar essa etiqueta vazia para que o usuário não tenha a interface poluída.

```
override func viewDidLoad() {
    super.viewDidLoad()
    codigonaorec.text = ""
}
```

Ao preencher os dados necessários e pressionar o botão “Save”, o programa executará uma ação, chamada de “saveTest”.

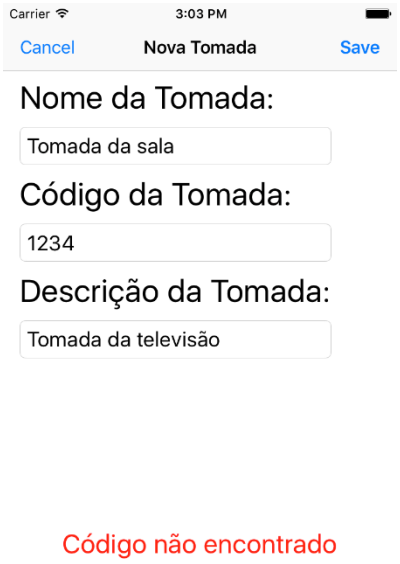
```
@IBAction func saveTest(_ sender: Any) {
    lerDataaparelhos()
    for i in 0...(codigoarray.count-1){
        if tomadacod.text == codigoarray[i] {
            mudarnomedesc()
            codigonaorec.text = ""
            codigo = codigoarray[i]
            performSegue(withIdentifier: "savesegue", sender: self)
        }
            codigonaorec.text = "Código não encontrado"
    }
}
```

Dentro da ação “saveTest”, a primeira ação do programa é executar a função “lerDataaparelhos”. Esta função será melhor descrita na seção 5.2, e terá a função de ler os dados presentes no servidor e armazenar o dado de código em um vetor chamado “codigoarray”.

O texto digitado pelo usuário será então testado com os códigos armazenados nesse vetor, através de um ciclo *for* pelo vetor “codigoarray”, e, se algum dos testes for verdadeiro, a função “mudarnomedesc” que também será melhor apresentada na seção 5.2, será executada. Esta função muda os dados presentes no servidor pelos dados inseridos pelo usuário. Em seguida o código de erro é apagado, caso ele tenha sido apresentado antes. A variável auxiliar “código” recebe então o valor do vetor codigoarray na posição em que o teste foi verdadeiro. Finalmente o segue é executado e a interface de menu é novamente apresentada com uma nova célula de tomada.

Caso os testes falhem, a última linha de código é executada, ou seja, é apresentada uma mensagem ao usuário de “Código não encontrado” através do *UILabel*. A execução dessa linha resulta na interface mostrada na Figura 17.

Figura 17 – Mensagem de erro apresentada ao usuário



Fonte: o próprio autor

3.3.1. Ligações de "AdicionarViewController"

As duas ligações de "AdicionarViewController" são indiretas e já foram mencionadas em 3.2.1. As ligações indiretas são chamadas de *"unwind segues"*. Essas ligações também podem ser chamadas de saídas de um ViewController. Elas não estão presentes graficamente no Storyboard, podendo ser visualizadas apenas dentro do código e na visualização de objetos do Xcode.

A grande diferença entre essas ligações e os segues normais é que estas podem se conectar entre qualquer um dos Storyboard, mesmo fora da ordem do segues. No caso deste aplicativo, o *"unwind segue"* é utilizado para voltar para a interface de menu. A ordem natural do programa seria a ordem da figura 14, em que os segues são unidirecionais e se dividem de acordo com o objeto pressionado na interface de menu. O *unwind segue* permite a saída dessa ordem, voltando para a tela de menu e escolhendo outra opção. Uma das características desse segue é que as ações relativas a ele estão presentes na interface de destino. Ou seja, os blocos de código presentes nesta seção estarão no arquivo de MenuTableViewController.

Os *unwind segues* descritos abaixo se ligam diretamente com os botões presentes na barra de navegação. A diferença principal entre eles é que em um deles não ocorre nenhuma ação específica, apenas a mudança de interface. O outro segue executa um bloco de códigos dentro de seu ViewController.

3.3.1.1. Ligação: "saveUnwindSegue"

O bloco de código de um *"unwind segue"* é apresentado da mesma maneira que uma ação. Nesse caso, seguindo a ideia do programa, ao validar os dados e realizar o segue, o programa deve então adicionar os dados inseridos em suas variáveis e adicionar uma nova célula que conterá o nome da tomada adicionada.

```
    @IBAction func save(segue:UIStoryboardSegue) {
        let AdicionarVC = segue.source as! AdicionarViewController
        newTomada = AdicionarVC.name
        tomadas.append(newTomada)
        newCod = AdicionarVC.codigo
        codigos.append(newCod)
        newId = AdicionarVC.idd
        soids.append(newId)
        tableView.beginUpdates()
        tableView.insertRows(at: [IndexPath(row: tomadas.count-1, section:0)],
with: .automatic)
        tableView.endUpdates()
    }
```

Na primeira linha do bloco, é declarada uma constante que é o ViewController de fonte (source) do segue. Nesse caso, a fonte é o AdicionarViewController. Em seguida, as variáveis auxiliares "newTomada", "newCod" e "newId" recebem os respectivos valores de nome, código e número de identificação da tomada escolhida. Então, cada um desses valores novos será anexado ao respectivo vetor que armazena cada um desses dados.

Em seguida, será realizada uma atualização do *TableView*. As três últimas linhas tratam dessa atualização. A primeira linha autoriza o início da atualização. A segunda linha, através de "*insertRows*", fará a inserção de uma linha nova. Esta função de TableView possui o parâmetro de endereço da linha a ser inserida. No caso, a última linha é na posição do número de tomadas menos um e na única seção do TableView (seção 0). Finalmente, as atualizações são encerradas através de "*endUpdates*". Através das funções descritas em 3.2 e 3.3, o programa consegue inserir novas tomadas em seu TableView.

3.3.1.2. Ligação: "cancelUnwindSegue"

A ação de cancelar é uma das ações mais simples presentes no aplicativo. Ela tem como função interromper a execução de AdicionarViewController e voltar à tela de menu, sem que nenhum teste de dados seja executado.

```
    @IBAction func cancel(segue:UIStoryboardSegue) {
    }
```

Este bloco de ações é vazio pelo fato de não ser necessária nenhuma ação relativa do controlador entre a camada View e a camada Model. A única mudança com essa ação é na interface do programa.

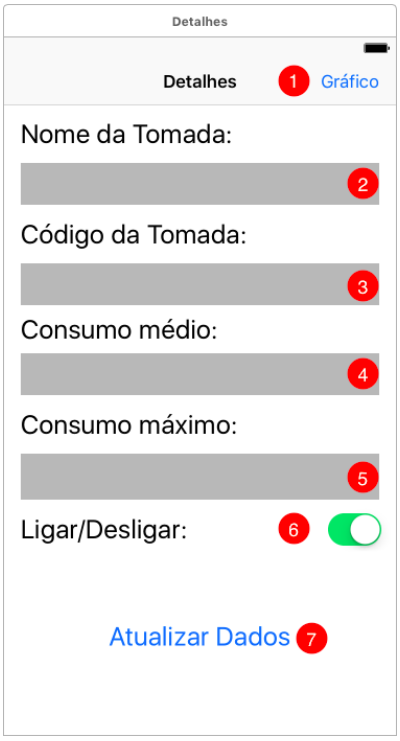
3.4. Tela de Detalhes da Tomada *"DetalheTomadaViewController"*

A interface de detalhes da tomada deve possuir todos os dados necessários para que o usuário consiga identificar a tomada escolhida, além de apresentar ao usuário o consumo máximo instantâneo e o consumo médio. Para controlar o funcionamento da tomada, o usuário deve poder mudar o estado de funcionamento da tomada através dessa interface.

Esses dados são apresentados ao usuário através de *UILabels*. Alguns desses dados são internos ao programa, pois já foram retirados do servidor em interfaces anteriores e outros dados serão retirados do servidor nessa interface.

Esse ViewController consiste em um simples *View*, assim como nas seções 3.1 e 3.3. Os elementos estão apresentados na figura 18.

Figura 18 – Elementos de DetalheTomadaViewController



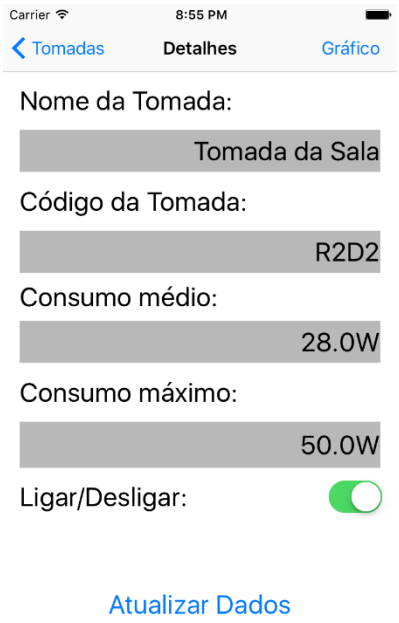
Fonte: o próprio autor

Os elementos presentes na tela têm as seguintes funções:

1. *NavigationBarButton*: Este botão, que possui a etiqueta “Gráfico” ligada a ele, é o botão responsável pela ativação do segue “DetailSegue”, que faz a ligação entre a tela de detalhes e a interface do gráfico de consumo do aplicativo;
2. *UILabel*: Essa etiqueta apresentará ao usuário o nome da Tomada que foi escolhida. Esse dado é interno ao programa e é recebido de acordo com a célula escolhida em MenuTableViewController;
3. *UILabel*: O código da tomada escolhida será apresentado ao usuário através desse UILabel. Esse dado também é interno ao programa e foi recebido da mesma maneira que o item 2;
4. *UILabel*: Etiqueta responsável por apresentar o consumo médio em Watts da tomada. Esse dado é recebido através do servidor, que recebe todos os dados de consumo instantâneo da tomada. O aplicativo então calcula a média e aplica esse valor à etiqueta;
5. *UILabel*: Similar ao item 4, essa etiqueta mostrará o Consumo instantâneo máximo em Watts da tomada. Este dado também é recebido através do servidor e a mesma função responsável por calcular a média também fornece o valor máximo. Esta função será mais detalhada em 5.4;
6. *UISwitch*: Um UISwitch é um botão que possui um valor booleano atrelado a ele. Essa chave controla o estado da tomada. Ao estar virada para a direita com a parte esquerda em coloração verde, como na figura 18, significa que a chave está com o seu valor igual a “verdadeiro” (1 em binário);
7. *UIButton*: Este botão tem como função executar novamente as funções de recebimento de dados e atualização da interface. Com isso, caso algum dado tenha sido adicionado ao servidor durante a utilização, esse botão pode ser pressionado, fazendo com que os dados novos sejam apresentados.

O usuário terá a representação desses elementos executado como na Figura 19.

Figura 19 – Interface de detalhes apresentada ao usuário



Fonte: o próprio autor

Os quatro UILabels e o UISwitch mencionados acima são conectados ao controlador através de *outlets*. A chave assume o valor booleano de acordo com o status presente no servidor.

```
@IBOutlet weak var nameLabel: UILabel!
@IBOutlet weak var codLabel: UILabel!
@IBOutlet weak var consumomedLabel: UILabel!
@IBOutlet weak var consumomaxLabel: UILabel!
@IBOutlet weak var status: UISwitch!
var name : String!
var cod : String!
var consumomed : Double?
var consumomax : Double?
var ipi:String!
```

Além dos *outlets* utilizados, a função possui a variável de endereço de IP e quatro dados importantes para a interface de usuário. As variáveis “name”, “cod”, “consumomed” e “consumomax” são responsáveis por armazenar os valores que serão repassados às etiquetas, e então atualizarão a interface. Como o usuário deve receber esses dados assim que a interface

for visível, todas as funções de cálculo e apresentação de dados serão executadas na função “viewDidLoad”.

```
override func viewDidLoad() {
    super.viewDidLoad()
    requestDataconsumo()
    consumocalc()
    verificarstatus()
}
```

Esse bloco então executa três funções. A função “requestDataconsumo” fará a requisição com o servidor necessária para que o controlador receba os dados via internet. Com os dados recebidos, a função de cálculo de consumo é executada. Dentro dessa função, o vetor contendo os dados de consumo instantâneo é destrinchado para obter os valores máximo instantâneo e médio. Essa função executa outra função chamada “updateData”, descrita abaixo.

```
private func updateData() {
    nameLabel.text = "\(name!)"
    codLabel.text = "\(cod!)"
    consumomedLabel.text = "\(String(describing: consumomed!))W"
    consumomaxLabel.text = "\(String(describing: consumomax!))W"
}
```

Essa função é responsável por atualizar o texto das etiquetas com os valores recebidos pelo programa tanto no segue quanto nas funções de cálculo de consumo. Dessa forma, os valores dos *outlets* são atualizados e apresentados ao usuário.

A última linha da função “viewDidLoad” executa a função “verificarstatus”. Essa função se conecta com a internet para verificar o valor da variável de status da tomada escolhida no servidor, e este valor é recebido pelo *outlet* da chave presente na interface. Assim o estado da chave no momento da visualização é o exato estado da tomada no servidor.

Por fim, o botão com o texto “Atualizar Dados”, é responsável por executar uma ação no sistema conforme o bloco de texto abaixo.

```
@IBAction func AtualizarDados(_ sender: Any) {
    requestDataconsumo()
    consumocalc()
}
```

Essa ação executa as mesmas funções presentes em “viewDidLoad” para que os dados presentes sejam atualizados tanto na camada Model, ou seja, nas variáveis declaradas, quanto na camada View, na interface ao usuário apresentada.

3.4.1. Ligação: “DetailSegue”

A tela de detalhe possui duas ligações diretas: a ligação com a tela de menu, onde ela receberá os dados de nome, código, número de identificação da tomada escolhida conforme mencionado na seção 2.2.1 e a ligação com a tela de gráfico, conforme mostrado na Figura 20.

Figura 20 – Ligações de DetalheViewController

Fonte: o próprio autor

Nesta seção, será detalhada a ligação DetailSegue, entre a interface de detalhes e interface de gráfico de consumo. Como esse segue consiste em uma ligação simples entre o botão presente na barra de navegação e “GráficoViewController”, a função de código a ser analisada deve ser a função “prepare(for segue:)” de DetalheViewController.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let vc = segue.destination as? GraficoViewController {
        vc.consumo = consumoarray2
        vc.datames = dataarray2
    }
}
```

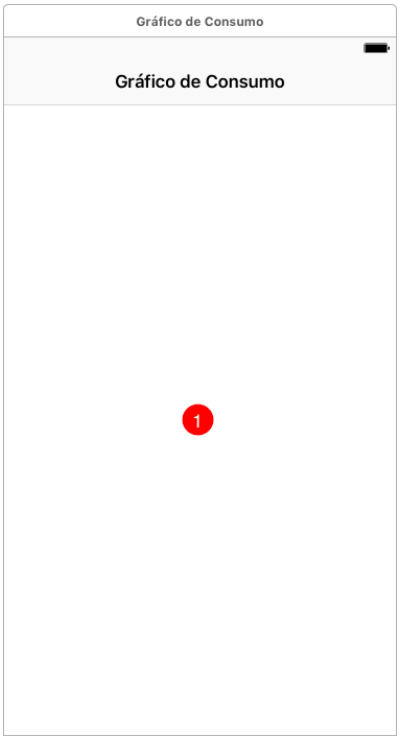
Nesse caso, pela interface possuir apenas um segue, não é necessário o teste de identificador do segue, já que este sempre será DetailSegue. O condicional *if* está presente para desempacotar a variável "vc" que é o controlador de destino da ligação: "GráficoViewController".

Esse controlador recebe os vetores de dados que contém os valores instantâneos de consumo (consumoarray2) e data de submissão do dado (dataarray2). Estes dados serão utilizados na montagem do gráfico de consumo instantâneo.

3.5. Tela de Gráfico de Consumo *"GráficoViewController"*

A última interface que o usuário pode acessar é a interface de gráfico de consumo. Ela consiste em um simples View. Nesse View, é utilizado o API Charts, mostrado na seção 2.5. Dessa forma, a tela possui apenas um elemento mostrado na Figura 21.

Figura 21 – Elementos de GráficoViewController



Fonte: o próprio autor

1. *View*: Através dessa View, foi inserido o código de configuração do API Charts para apresentar ao usuário o gráfico de consumo.

A configuração do gráfico se é dividida em quatro partes. A primeira parte consiste na escolha do gráfico utilizado. Essa escolha é feita através da configuração do *outlet*. Como os dados são medidos de maneira discreta e para tornar a interface do gráfico mais amigável ao usuário, foi escolhida a opção de gráfico de barras, similar ao gráfico presente na Figura 06.

```
@IBOutlet weak var barView: BarChartView!
var consumo = [Double]()
var consumousavel = [Double]()
var datames = [String]()
var datahora = [String]()
var ultimadata: String!
var dataaux: String!
```

A *View* é ligada ao código e definida como um "BarChartView", ou seja, um gráfico de barras. Em seguida são definidas as variáveis contendo os dados que serão apresentados no gráfico. As variáveis "consumo" e "datames" estão presentes no segue mostrado em 3.4.1 e armazenam os dados de consumo e data enviados por "DetalheViewController". A variável "ultimadata" armazenará o valor da última data presente no vetor "data". Com esse dado presente, as variáveis "consumousavel" e "datahora" possuem os valores de consumo e de horário das medições.

Assim como na seção anterior, é necessário que o gráfico seja apresentado ao usuário no momento em que a interface for carregada. Para isso, todas as funções de dados do gráfico são executadas dentro da função viewDidLoad.

```
override func viewDidLoad() {
    super.viewDidLoad()
    getultimadata()
    testusableData()
    updateChartWithData()
}
```

Na primeira linha do bloco de código, é executada a função "getultimadata". Essa função é reponsável por retirar de dentro do vetor de datas, uma string com o valor do último dia em que alguma medição foi realizada.

O dado que contém a data e a hora da medição é uma *string* com a seguinte característica "AAAA-MM-DD 00:00:00". Então, para ser obtida a *string* apenas com a data da última medição deve-se obter apenas os dez primeiros caracteres presentes nesse dado, sendo apresentada a forma final "AAAA-MM-DD".

```
func getultimadata() {
    ultimadata = datames.last!
    let index = ultimadata.index(ultimadata.startIndex, offsetBy: 10)
    ultimadata = (ultimadata.substring(to: index))
}
```

A função é executada na seguinte sequência: a variável “ultimadata” recebe o último valor presente no vetor de datas enviado através de “DetalheViewController”. Em seguida, é declarada uma constante “index”, responsável por obter o endereço do décimo caractere dentro da *string* “ultimadata”. Finalmente, é utilizada a função *substring* dentro da classe *string*. Sua utilização é bem clara: a variável “ultimadata” recebe uma parte de sua própria string, até o endereço especificado por “index”, ou seja, até o décimo caractere.

Com a última data de medição recebida, “getultimadata” é finalizada e a próxima linha da função “viewDidLoad” é executada. A função “testuableData” testará todos os dados do vetor “datames” para saber quais desses dados também possuem a última data de medição.

```
func testusableData() {
    for i in 0..<datames.count {
        let index = datames[i].index(datames[i].startIndex, offsetBy: 10)
        dataaux = (datames[i].substring(to: index))
        if dataaux == ultimadata {
            consumousavel.append(consumo[i])
            let start = datames[i].index(datames[i].startIndex, offsetBy: 11)
            let end = datames[i].index(datames[i].endIndex, offsetBy: -6)
            let range = start..<end
            datahora.append(datames[i].substring(with: range))
        }
    }
}
```

A primeira linha da função consiste em um ciclo *for* que percorrerá todas os endereços presentes no vetor “datames”. Um processo similar ao executado na função “getultimadata” é realizado nas próximas duas linhas: cada elemento do vetor “datames” tem seus dez primeiros caracteres retirados e a variável auxiliar “dataaux” os recebe, para então ter seu valor testado com “ultimadata”. Caso o teste seja falso, o ciclo for se reinicia na próxima iteração. Caso o teste seja verdadeiro, o bloco de código dentro do condicional *if* é executado.

O vetor “consumousavel” anexa o valor do elemento do vetor “consumo” no endereço em que o teste é verdadeiro. O próximo passo é retirar de um elemento de data, a hora da medição. A hora consiste nos caracteres 12 e 13 do valor de data de medição. Para isso, é definido um alcance (*range*) da *string*. Esse alcance consiste em iniciar a *substring* a partir do endereço 11, e finalizá-la a 6 caracteres do fim da *string*. A variável range é definida a partir desses endereços e por fim, é anexado ao vetor “datahora” o valor dos dois caracteres de hora no endereço em que o teste foi verdadeiro.

Com o fim de todas as iterações do ciclo for, a camada *Model* possui então todos os dados necessários para execução do gráfico: os valores e os horários na última data, e o valor da última data. Com esses valores presentes, a próxima linha da função “viewDidLoad” é executada.

A função “updateChartWithData” é responsável por inserir no gráfico todos os valores necessários para a sua apresentação ao usuário.

```
  func updateChartWithData() {
      var dataEntries: [BarChartDataEntry] = []
      for i in 0..<consumousavel.count {
         let dataEntry = BarChartDataEntry(x: Double(i), y: Double(consumousavel[i]))
         dataEntries.append(dataEntry)
      }
      barView.xAxis.valueFormatter = IndexAxisValueFormatter(values:datahora)
      barView.xAxis.granularity = 1
      let chartDataSet = BarChartDataSet(values: dataEntries, label: "Consumo instantâneo
em \(ultimadata!) (W)")
      let chartData = BarChartData(dataSet: chartDataSet)
      barView.data = chartData
      barView.chartDescription?.text = ""
  }
```

O início da função consiste em uma declaração de uma variável “dataEntries”, que é um vetor de “BarChartDataEntry”, ou seja armazena todas as entradas de dados do gráfico. Em seguida, ocorre um ciclo *for* para determinar quais são as entradas do gráfico. É declarada uma constante dataEntry, que consiste em uma entrada de dados do gráfico. Essa constante possui duas dimensões, o valor do eixo “x”, que nesse caso assume apenas o valor da variável auxiliar “i”, e o valor do eixo “y”, que recebe o dados presente no endereço “i” do vetor “consumousavel”. Em seguida, esses dados são anexados ao vetor declarado na primeira linha.

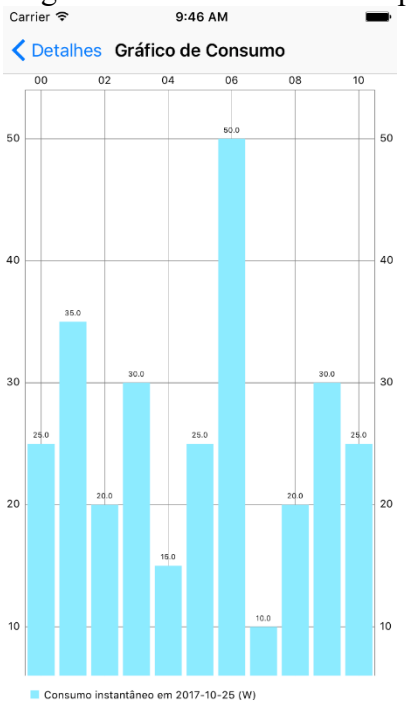
Após as iteraçõe do ciclo *for*, todos os dados do eixo “y” estão ajustados para apresentação. A função BarChartDataEntry não permite ao usuário a utilização de strings em

seu conteúdo, portanto, para definir os valores do eixo “x”, foi utilizada um dos elementos presentes em “BarChartView”. Ao adicionar “valueFormatter”, pode-se indexar qualquer tipo de dado ao eixo “x”. A próxima linha tem o elemento “granularity” recebendo o valor 1. Isso acontece para que a visualização do gráfico seja mais amigável, pois essa função omite alguns dados dos eixos, caso seja possível, tornando a interface menos poluída.

Em seguida, é declarada uma constante “chartDataSet”. Essa constante armazenará tanto os valores dos eixos do gráfico, como a etiqueta que mostra o que esses valores representam, facilitando o entendimento ao usuário. Nesse caso, a etiqueta mostra o seguinte texto “Consumo instantaneo em \(ultimadata!) (W)". A variável “ultimadata” está presente dentro desse texto de maneira que ao apresentar na interface, o valor da última data de medição seja visível ao usuário.

Em seguida, são feitas declarações de modo que os valores sejam repassados ao gráfico. Uma vez que todos esses dados estejam presentes no gráfico, pode-se visualizar a interface como na Figura 22.

Figura 22 – Interface do gráfico de consumo diário apresentada ao usuário



Fonte: o próprio autor

4. CAPÍTULO 03: O BANCO DE DADOS

Nesse capítulo, serão apresentados: a estrutura do banco de dados e os scripts responsáveis por enviar os dados do servidor ao aplicativo.

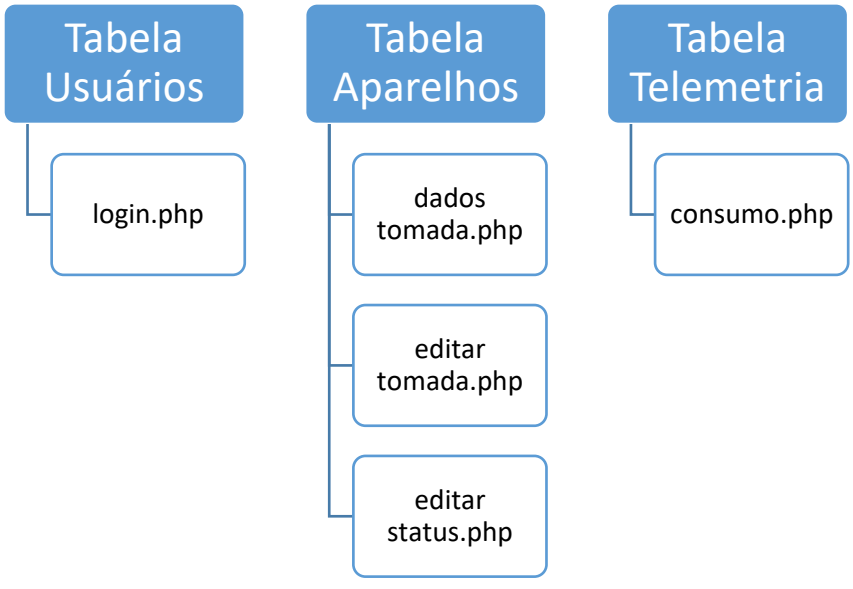
O banco de dados é dividido em três tabelas interligadas: Tabela Usuários, Tabela Aparelhos e Tabela Telemetria. Essas tabelas são utilizadas respectivamente para armazenar os dados referentes aos usuários que tem acesso ao servidor, as tomadas cadastradas nesse servidor e as medições realizadas por essas tomadas.

Existe uma interconexão entre a tabela Usuários e a tabela Aparelhos que identifica o usuário que possui determinada tomada. Também existe uma conexão entre as tabelas Aparelhos e Telemetria responsável por definir qual tomada realizou determinada medição.

O aplicativo necessita dos dados presentes no servidor para realizar diversas tarefas, como: autenticação de usuário, adição de novas tomadas, apresentação dos dados de consumo ao usuário, dentre outras aplicações. A conexão direta entre servidor e aplicativo seria muito complexa, o que geraria um programa que consumiria muita memória e dados. Para simplificar esse processo, foram utilizados códigos em linguagem PHP que funcionam como intermediários entre servidor e aplicação.

A segunda parte do capítulo é destinada a dar uma introdução à linguagem PHP, com uma breve explicação dos elementos utilizados nos scripts. Esses scripts serão então detalhados na terceira parte do capítulo. A figura 23 apresenta a relação dos códigos apresentados com as tabelas utilizadas.

Figura 23 – Relação entre Tabela MySQL e código em PHP



Fonte: o próprio autor

4.1. Características do Banco de dados

O banco de dados utilizado foi desenvolvido por Ícaro Jonas em "Tomada Inteligente Baseada em Internet das Coisas (IoT) com Leitura de Energia em Tempo Real" [3].

Nesse banco de dados estão presentes três tabelas:

1. Tabela Usuários: Utilizada por "LoginViewController" para autenticar os dados de usuário e senha inseridos. As características da Tabela estão resumidas na Tabela 01. Esta tabela apresenta um inteiro com características de incremento, utilizado como chave única de contagem de usuários, e quatro "varchar", elemento de tabelas MySQL que se assemelha com uma *string*. Esses elementos consistem respectivamente no nome próprio do usuário, apelido utilizado para login, senha e o e-mail;

Tabela 01 – Estrutura da tabela usuários.

#	Nome	Tipo	Nulidade	Atributos
1	id	INT	Não	Primary Key, AUTO_INCREMENT
2	nome	VARCHAR	Não	
3	nick	VARCHAR	Não	
4	senha	VARCHAR	Não	
5	email	VARCHAR	Não	

Fonte: JONAS (2016)

2. Tabela Aparelhos: Contém as informações das tomadas cadastradas no servidor. Esta tabela é utilizada por "AdicionarViewController" para a verificação dos dados das tomadas adicionadas ao aplicativo e também utilizada por "DetalheTomadaViewController" para determinar o estado de funcionamento da tomada. A tabela, assim como no item 1, possui uma coluna de "id" para identificar unicamente a linha, possui a coluna "id_usuario" que indica qual "id" da tabela Usuários pode acessar a tomada. Além desses dois elementos, existem quatro elementos que descrevem a tomada: "nome", "soid", "descrição" e "status". A coluna "status" possui o tipo "TINYINT", que representa um booleano em código MySQL;

Tabela 02 – Estrutura da tabela aparelhos.

#	Nome	Tipo	Nulidade	Atributos
1	id	INT	Não	Primary Key, AUTO_INCREMENT
2	id_usuario	VARCHAR	Não	Foreign Key
3	soid	VARCHAR	Não	
4	nome	VARCHAR	Não	
5	descricao	VARCHAR	Sim	
6	status	TINYINT	Sim	

Fonte: JONAS (2016)

3. Tabela Telemetria: Dividida em cinco colunas, a tabela Telemetria contém os valores das medições realizadas pelos sensores da tomada, sendo utilizada por "DetalheTomadaViewController" assim como a data em que foram registradas as medições. As duas primeiras colunas são similares ao item 2: uma coluna para identificação única do registro e outra para identificação da tomada que teve seus dados registrados. As três outras colunas contêm os dados de tensão, corrente e data de submissão dos dados.

Tabela 03 – Estrutura da tabela telemetria.

#	Nome	Tipo	Nulidade	Atributos
1	id	INT	Não	Primary Key, AUTO_INCREMENT
2	id_aparelho	INT	Não	Foreign Key
3	corrente	FLOAT	Não	
4	tensao	INT	Não	
5	data_submissao	DATETIME	Não	DEFAULT: CURRENT_TIMESTAMP

Fonte: JONAS (2016)

A partir das três tabelas apresentadas acima, é possível validar todos os dados necessários no aplicativo. Entre Banco de dados e aplicativo existe um intermediário, que é responsável por tratar o dado bruto presente na tabela e enviar ao aplicativo de uma maneira que o dado possa ser lido. Esse intermediário consiste nos scripts em linguagem PHP que serão explicados nas seções seguintes.

4.2. Introdução à linguagem PHP

Na programação em linguagem Swift, existe a possibilidade de obter dados diretamente de um banco de dados *MySQL*. Porém, este processo é bastante complexo e aumentaria bastante o tamanho e a complexidade do aplicativo.

Para simplificar este processo, são utilizados arquivos em linguagem PHP que atuam como intermediários entre os arquivos do aplicativo e o banco de dados. Uma requisição em Swift de um arquivo PHP pode levar apenas três linhas de código, o que reduz a dimensão do código programado em Swift.

O PHP é umas das linguagens de programação Web mais utilizadas e documentadas atualmente. Em PHP Documentation (ThePHP Group, 2017), está presente a sua origem:

> O PHP como é conhecido hoje, é na verdade o sucessor para um produto chamado PHP/FI. Criado em 1994 por Rasmus Lerdof, a primeira encarnação do PHP foi um simples conjunto de binários Common Gateway Interface (CGI) escrito em linguagem de programação C. Originalmente usado para acompanhamento de visitas para seu currículo online, ele nomeou o conjunto de scripts de "Personal Home Page Tools" mais frequentemente referenciado como "PHP Tools". [...] (The PHP Group, 2017)

Atualmente, a versão utilizada é o PHP 7, lançado em 2015. Em 20 anos de desenvolvimento, o PHP obteve uma grande evolução, possuindo um motor chamado “Zend Engine” como provedor de recursos.

Esta seção será focada em apresentar os elementos da linguagem que foram utilizados durante o desenvolvimento do aplicativo. Então serão explicados a conexão com o banco de dados, a retirada e exibição dos dados, além do modelo de armazenamento utilizado.

4.2.1. Conexão com os servidores

O primeiro passo na obtenção dos dados de uma tabela MySQL consiste em realizar uma conexão com o servidor, em seguida se ligar ao banco de dados. Para realizar essa conexão, é necessário utiliza a função “myqsli::__construct”.

Essa função possui até seis parâmetros:

1. Hospedeiro (host): Pode ser um nome de hospedeiro ou um endereço IP. O valor default dessa variável é “localhost”, nome utilizado pelos servidores em geral para o acesso;

2. Nome de usuário (username): O nome de usuário utilizado para entrar no servidor. Em servidores MySQL, o nome de usuário predefinido é “root”, podendo ser alterado pelo usuário;
3. Senha (passwd): Senha do banco de dados MySQL. Caso não seja fornecida, ela fará o teste com esse valor vazio, que é o valor predefinido pelo servidor MySQL;
4. Nome do Banco de Dados (dbname): Em servidores MySQL, podem existir diversos bancos de dados contendo diversas informações para finalidades diferentes. Por isso, é necessário especificar o banco de dados utilizado;
5. Porta (port): Contém o número da porta que tentará conexão com o servidor;
6. Soquete (socket): Especifica o soquete que deve ser utilizado na conexão.

Essa função retorna um objeto que representa a conexão com o servidor.

A declaração de “msqli::__construct” consiste em definir uma variável que armazenará esse objeto retornado da conexão. Geralmente, adiciona-se um condicional *if* assim que a conexão é realizada, que testa se houve um erro, para então agir nesse caso.

4.2.2. Requisição “POST”

Durante a execução do aplicativo, será necessário enviar algumas informações ao servidor, de modo a atualizar nomes de tomadas, descrição e estado de funcionamento. Para que esse dado chegue ao arquivo PHP executado, ele é enviado como um vetor via HTTP.

Uma das maneiras que a página executada em código PHP possa receber e tratar esse dado, é através de uma requisição “POST”. Esse método não é exclusividade da linguagem PHP, pois é um método do protocolo HTTP. O dado enviado é encapsulado em uma string. Um exemplo de um possível dado enviado pelo programa é “a=valor”. Mais de uma variável pode ser enviada em uma mesma *string*, basta adicionar um “e” comercial (&) à *string* e seguir com a declaração das variáveis enviadas

Para o código Web tratar esse dado, deve-se declarar uma variável que recebe o valor desejado dentro da *string* enviada à página. Então dentro da requisição deve-se colocar a identificação da variável enviada. Seguindo o exemplo do parágrafo anterior, uma requisição post para obter “valor” seria “$_POST[a]”.

4.2.3. Seleção de dados em tabelas MySQL

Após a criação da conexão com o banco de dados, é necessário selecionar os dados que serão utilizados dentro desse banco de dados. Um código em PHP é capaz de inserir, modificar, deletar ou até mesmo fazer apenas a leitura dos dados em uma tabela MySQL.

O primeiro passo é decidir o que o código deve fazer. Define-se uma variável que vai receber a requisição desejada. Em seguida, utiliza-se “INSERT” para inserir novos dados, “UPDATE” para atualizar a tabela, “DELETE” para excluir dados e “SELECT” para fazer apenas a leitura.

Durante a execução do aplicativo, serão utilizadas apenas as funções “UPDATE” para atualizar o nome, descrição e estado da tomada e a função “SELECT” para fazer a leitura dos dados presentes nas tabelas.

A sintaxe de “UPDATE” é dividida em três partes: logo após a declaração de “UPDATE”, deve-se dizer o nome da tabela. Posteriormente, o valor que será alterado deve ser declarado recebendo o novo valor precedido de “SET”. Para finalizar a atualização deve-se dizer a condição que definirá qual linha será atualizada precedida de “WHERE”.

Diferentemente da sintaxe anterior, “SELECT” possui uma declaração mais simples. Seguido de “SELECT” deve-se informar quais colunas serão lidas. Em seguida declara-se “FROM” com o nome da tabela que terá seus dados lidos.

Para armazenar os dados da tabela no código, deve-se iterar a variável que recebe a requisição através da função “fetch_assoc”, que retorna o valor de uma linha da tabela selecionada como um *array*.

4.2.4. Arrays

Um *Array* em PHP consiste em um conjunto de dados ordenado. Ele pode ser tratado como qualquer conjunto de dados: vetor, dicionário, *set*. Geralmente, um *array* em PHP se assemelha com um dicionário em Swift, ou seja, possui um valor relacionado a uma chave. A diferença entre essas duas coleções de dados é que em PHP, um *Array* pode ter chaves com tipos de valores diferentes.

O PHP faz coerções com a chave. Caso o valor de uma chave seja um *String* que contém um número inteiro, um *float* ou um valor booleano, estes valores serão devidamente convertidos para um inteiro (int). Esse tipo de conversão deve ser um fator importante na

declaração de um *array*, pois uma chave pode apontar para apenas um valor, então ao declarar chaves que possuem valores iguais com tipos diferentes de variáveis, os valores dessas chaves podem ser sobrescritos.

A declaração de um *array* vazio consiste em declarar o nome da variável que a recebe, seguida do símbolo de igual (“=”), em seguida, deve-se escrever “array()”.

4.2.5. Codificação JSON

No momento da troca de dados entre página e servidor, o servidor deve receber esse dado em um formato válido. O formato mais simples para o armazenamento é o formato de texto. A formatação em JSON é basicamente textual. Ela pode ser uma coleção de pares ou uma lista ordenada de valores. Essa formatação é absorvida tanto pelo servidor quanto pelo aplicativo, o que a torna ideal para o uso nesse tipo de aplicação.

Essa sintaxe de objeto possui as vantagens de ser um tipo de dados leve, de fácil compreensão e independente de uma linguagem. Com isso, o JSON é utilizado tanto para exibir os dados retirados do servidor na linguagem PHP quanto para a linguagem Swift converter os dados em texto para o objeto desejado.

Para converter um objeto, na linguagem PHP, em texto JSON, deve-se utilizar a função “json_encode()”. Entre parênteses deve estar presente o nome do objeto convertido.

4.3. Requisições PHP

Nessa seção, serão analisadas cinco requisições em linguagem PHP utilizando os conceitos apresentados em 4.1 e 4.2. Essas requisições se dividem em dois tipos: requisições de aquisição de dados, representadas por “login.php”, “dadostomada.php” e “consumo.php”, e requisições de edição de dados, representadas por “editartomada.php” e “editarstatus.php”.

Os scripts de aquisição de dados utilizam-se de requisições “SELECT” para selecionar os dados. Esses dados selecionados são então tratados pelo programa e transmitidos pela página.

Os scripts de edição de dados fazem o processo de maneira inversa: primeiramente eles tratam o dado recebido de maneira que ele seja utilizável pelo código PHP para então utilizar a requisição “UPDATE” para editar o dado presente na tabela MySQL.

4.3.1. Requisição "login.php"

Essa requisição tem como função enviar para o aplicativo os dados de nome de usuário e senha para que ele trate e faça a autenticação desses dados. Primeiramente ocorrerá uma conexão com a base de dados, em seguida os dados necessários da tabela serão retirados para então enviá-los ao aplicativo.

Para abrir a conexão com o servidor, deve-se utilizar a função "mysqli::__construct", exemplificada em 4.2.1.

```
$servername = "localhost";
$username = "root";
$password = "";
$dbname = "iphone";
$conn = new mysqli($servername, $username, $password, $dbname);
if ($conn->connect_error) {
   die("Connection failed: " . $conn->connect_error);
}
```

Primeiramente, definiram-se quatro variáveis contendo os dados necessários para realizar a conexão. Então foi criada a variável "conn" que receberá o objeto que representa a conexão com o servidor. Finalmente, é feito um teste para verificar se houve algum erro na conexão. Caso não haja erros, o script prossegue. Esse bloco de código estará presente em todas as requisições PHP utilizadas pelo aplicativo.

Com a conexão realizada, é necessário então definir os dados que serão retirados dentro da tabela "usuários", que contém os dados necessários para a realização da autenticação do *login*.

```
$sql = "SELECT id, nick, senha FROM usuarios";
$result = $conn->query($sql);
$rownum = count($result);
$arrayusuario = array();
```

Os dados necessários par autenticação são os dados da coluna "id", para organizar o vetor, e as colunas "nick" e "senha", que contém os dados relevantes para a autenticação.

Em seguida, a variável “result” contém os dados de uma “consulta” da tabela, feita pela função “query”. Essa consulta retornará os dados presentes na tabela que ainda necessitarão de tratamento. A variável “rownum” conta o número de linhas presentes na seleção de dados e a última variável “arrayusuario” é o vetor que armazenará os dados que serão enviados ao aplicativo.

Por fim, deve-se realizar o tratamento dos dados recebidos por “result” e enviá-los.

```
if ($result->num_rows > 0) {
    while($row = $result->fetch_assoc()) {
        for ($i=0;$i<$rownum;$i++){
        $arrayusuario[$row["id"]] = $row;
        }
    }
} else {
    echo "0 results";
}
echo json_encode($arrayusuario);
```

Esse tratamento é realizado iterando a variável “result” por todas as linhas do programa. Essas iterações são realizadas através dos ciclos “*while*” e “*for*”. Durante cada ciclo, uma das linhas da tabela é inserida dentro do vetor “arrayusuario”, com a chave desse vetor sendo o número da linha. Caso não haja nenhuma linha na tabela MySQL, o programa deve então apresentar a mensagem “0 results”.

Ao final dos ciclos, vetor “arrayusuario” será então codificado em JSON para então ser apresentado na execução do script. Esse valor seguido da expressão “echo”, é o valor transmitido pelo script.

4.3.2. Requisição “dadostomada.php”

Muito similar à requisição presente em 4.3.1, essa requisição é responsável por fornecer ao programa os dados necessários para a validação da tomada adicionada em “AdicionarViewController”. Outra utilização desse script é fornecer ao programa o estado da chave presente em “DetalheTomadaViewController”.

Essa requisição também é dividia em três partes: conexão com o servidor, consulta de dados e tratamento dos dados recebidos. A primeira parte é realizada de maneira idêntica à conexão em 4.3.2.

A grande diferença entre as requisições está no dado requerido da tabela. Enquanto na seção 4.3.1, os dados de usuário e senha são requeridos, no script “dadostomada”, são selecionados “id” para organização, “id_usuario”, “soid”, “nome”, “descrição” e “status”. Apenas a variável status é utilizada em “DetalheTomadaViewController”, enquanto as outras são utilizadas por “AdicionarViewController”.

```
$sql = "SELECT id, id_usuario, soid, nome, descricao, status FROM aparelhos";
$result = $conn->query($sql);
$rownum = count($result);
$arraydados = array();
```

Finalmente, ocorre a iteração e transmissão dos dados também de maneira idêntica à seção 4.3.1.

```
if ($result->num_rows > 0) {
    while($row = $result->fetch_assoc()) {
        for ($i=0;$i<$rownum;$i++){
        $arraydados[$row["id"]] = $row;
        }
    }
} else {
    echo "0 results";
}
echo json_encode($arraydados);
```

Nesse caso, a variável iterada, codificada e transmitida, é “arraydados”, que contém todos os dados necessários para a validação da tomada.

4.3.3. Requisição “consumo.php”

Por também ser uma requisição de aquisição de dados, esse script é muito similar aos scripts presentes em 4.3.1 e 4.3.2 Realiza-se a conexão, em seguida aquisição de dados para então transmiti-los. A conexão também é idêntica à presente na seção 4.3.1.

Os dados requeridos nesse caso serão os dados da tabela de telemetria. Essa tabela, que contém os dados medidos de tensão e corrente, será fundamental para execução de “DetalheTomadaViewController” e “GraficoViewController”.

```
$sql = "SELECT id, id_aparelho, corrente, tensao, data_submissao FROM telemetria";
$result = $conn->query($sql);
$rownum = count($result);
$arrayconsumo = array();
```

Nesse caso, os dados selecionados são “id” para organização do vetor, “id_aparelho” para validação e seleção dos dados utilizados e as colunas “corrente”, “tensão” e “data_submissao” para apresentação dos dados no programa.

Por fim, ocorre a iteração e transmissão dos dados, com a única diferença de que o vetor se chama “arrayconsumo”.

```
if ($result->num_rows > 0) {
    while($row = $result->fetch_assoc()) {
        for ($i=0;$i<$rownum;$i++){
        $arrayconsumo[$row["id"]] = $row;
        }
    }
} else {
    echo "0 results";
}
echo json_encode($arrayconsumo);
```

4.3.4. Requisição “editartomada.php”

Esse código tem como função atualizar o servidor com nome e descrição da tomada inseridos pelo usuário em “AdicionarViewController”. O script pode ser dividido em duas partes: a conexão com o servidor, idêntica à realizada em 4.3.1, e o recebimento e tratamento dos dados.

O código recebe os dados via HTTP e os trata através da requisição “POST”. Em seguida, os dados recebidos são utilizados tanto na validação da linha a ser alterada quanto na própria alteração do banco de dados.

No bloco de código acima, é realizado um teste para o método de requisição. Caso seja “POST”, o bloco de código dentro do condicional é executado. As três primeiras linhas tratam a *string* enviada pelo aplicativos, retirando os valores das variáveis “a”, “b” e “c”, que correspondem ao nome da tomada, código da tomada e descrição da tomada, respectivamente.

```
if($_SERVER['REQUEST_METHOD']=='POST'){
   $tomadaName = $_POST['a'];
   $tomadacodigo = $_POST['b'];
   $tomadadesc = $_POST['c'];
   $sql = "UPDATE aparelhos SET nome='$tomadaName', descricao='$tomadadesc'
WHERE soid='$tomadacodigo'";
        if ($conn->query($sql) === TRUE) {
        $response = "Alterações realizadas com sucesso";
        } else {
        $response = "Erro na atualização: " . $conn->error;
        }
}
echo json_encode($response);
```

Em seguida, é utilizado o método “UPDATE” para editar os dados presentes na tabela “aparelhos”, nas colunas “nome” e “descrição”, de acordo com a condição em que a linha possua o mesmo valor na coluna código.

Finalmente, é realizada uma consulta ao banco de dados, para validar as alterações realizadas e, caso essa consulta retorne com o valor verdadeiro, a variável “response” recebe “Alterações realizadas com sucesso”. Caso o teste retorne com o valor falso, é retornado uma *string* com “Erro na atualização”, seguido do erro.

Ao final do condicional, a resposta da consulta ao servidor é enviada ao aplicativo.

4.3.5. Requisição “editarstatus.php”

Essa requisição também consiste em uma requisição de edição de dados. Nesse caso, o script atualiza o servidor com o estado de funcionamento da tomada. O código presente é bastante similar ao apresentado em 4.3.4, porém ainda mais simples, pois existe apenas uma variável a ser atualizada.

O processo de conexão com o servidor é análogo ao apresentado em 4.3.1, enquanto o processo de recebimento e tratamento de dados é semelhante ao apresentado em 4.3.2, com diferenças apenas pontuais.

```
if($_SERVER['REQUEST_METHOD']=='POST'){
   $tomadastatus = $_POST['a'];
   $tomadacodigo = $_POST['b'];
   if ($tomadastatus == “1”){
   $sql = "UPDATE aparelhos SET status=1 WHERE soid='$tomadacodigo'";
   } else {
   $sql = "UPDATE aparelhos SET status=0 WHERE soid='$tomadacodigo'";
   }
   if ($conn->query($sql) === TRUE) {
      $response = "Alterações realizadas com sucesso";
   } else {
      $response = "Erro na atualização: " . $conn->error;
   }
}
echo json_encode($response);
```

Primeiramente, são recebidas as variáveis “a” e “b”, via requisição “POST”, da *string* enviada pelo aplicativo. Essas variáveis contém o estado de funcionamento da tomada e o código. Em seguida, é realizado um teste do valor da variável “tomadastatus”. Como essa variável representa uma *string* e, na tabela, a coluna status é um valor booleano, é preferível realizar esse teste e enviar um valor inteiro na função “UPDATE”.

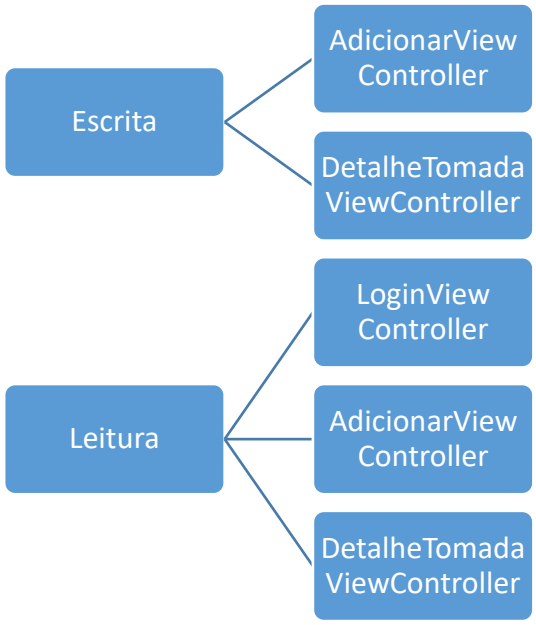
Se o valor recebido for “1”, envia-se esse valor ao banco de dados. Caso contrário, envia-se o valor “0” ao banco de dados. Em seguida, é realizada a consulta e verificado se as alterações foram realizadas no servidor, de maneira idêntica ao apresentado em 4.3.4.

5. CAPÍTULO 04: INTERAÇÕES DO APLICATIVO COM AS REQUISIÇÕES PHP

No capítulo 3, foram apresentados o servidor e o intermediário entre aplicativo e servidor. Nesse capítulo, o foco principal será a forma com que o aplicativo interage com os scripts em PHP. Esse capítulo também funciona como complemento ao capítulo 02, já que serão apresentadas as demais funções presentes nos "ViewControllers".

Assim como nos scripts, existem dois tipos principais de funções apresentadas: funções de leitura, que necessitam apenas receber dados para funcionar e funções de escrita, que enviam dados ao servidor. As funções de mesmo tipo se assemelham bastante entre si. Na figura 20, estão apresentados os tipos de funções que estão presentes nos "ViewControllers". Diferente do capítulo 02, este capítulo terá apenas três "ViewControllers": "LoginViewController", "AdicionarViewController" e "DetalheTomadaViewController", já que apenas eles possuem algum tipo de conexão com o servidor.

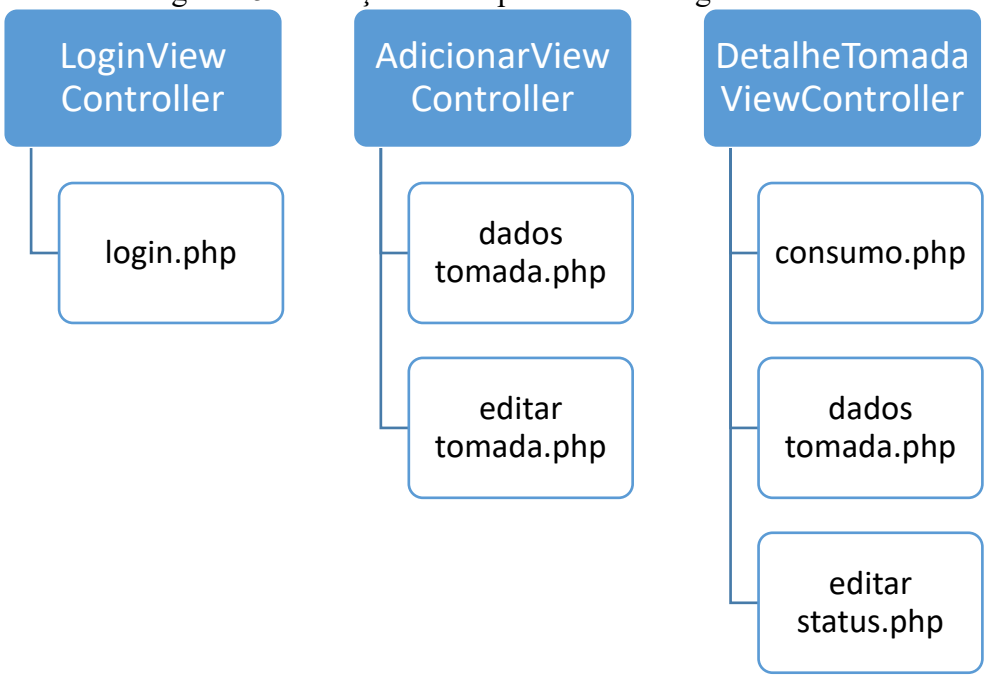
Figura 24 – Relação de escrita e leitura dos ViewController



Fonte: o próprio autor

Também é importante observar com quais scripts o aplicativo se conecta. É possível visualizar isso na Figura 25. Esses scripts não enviam os dados ao aplicativo prontos para serem utilizados. É necessário que dentro das funções de conexão haja um tratamento dos dados de acordo com sua utilização.

Figura 25 – Relação entre aplicativo e código em PHP



Fonte: o próprio autor

5.1. Funções de "LoginViewController"

Esse controlador possui função apenas de leitura de dados do servidor. Como mostrado em 4.1, a ação ligada ao botão "ENTRAR" chama a função "getDadosUser". Essa função é a responsável por se conectar ao script "login.php". Ela fará a leitura de todos os dados necessários e então validará esses dados com os dados preenchidos pelo usuário para que a entrada no aplicativo seja realizada. Primeiramente serão definidas as variáveis utilizadas nesse processo.

```
var ipi: String?
var reqlogin = URL(string:"")
var logindic = [String:Any]()
var usuario: String!
var password: String!
```

Novamente, a variável "ipi" aparece, pois ela será responsável por inserir na URL o endereço de IP do servidor. A variável "reqlogin" é responsável por armazenar a URL utilizada por login.php. O dicionário "logindic" armazenará todos os dados recebidos na tabela e as

variáveis “usuário” e “password” serão utilizadas na iteração dos elementos do dicionário e validação dos dados.

```
    private func getDadosUser() {
        ipi = ip.text!
        reqlogin = URL(string:"http://\(ipi!)/login.php")
        let urlContents = try? Data(contentsOf: reqlogin!)
        let data = urlContents
        var jsondados = try! JSONSerialization.jsonObject(with: data!, options: []) as!
[String:Any]
        for (key,_) in jsondados{
            logindic = jsondados[key] as! [String : Any]
            usuario = logindic["nick"] as! String
            password = logindic["senha"] as! String
            if usuario == login.text && password == senha.text {
                performSegue(withIdentifier: "LoginSegue", sender: self)
                return
                }
        }
        Errorlabel.text = "Usuário e/ou senha errados"
    }
```

Primeiramente, dentro do bloco de código é definida a URL de “login.php”. Essa URL é definida através do endereço de IP provido pelo usuário e da *string* “http://\(ipi!)/login.php”. Essa *string* possui “\(ipi!)”, que é o modo de se inserir o valor de uma variável dentro de uma *string*.

Em seguida é utilizada a função “try” para testar se é possível retirar dados da URL apresentada. Esses dados são enviados à constante “urlContents” que então repassa à constante “data”. Até esse momento, os dados recebidos são objetos codificados em JSON brutos. Para tratar os dados, é necessário utilizar outra vez a função “try” para realizar uma serialização da constante “data”. A variável “jsondados” recebe então os dados JSON tratados como um dicionário de chave “String” e valores “Any”. O modelo do dado recebido está presente na Tabela 04.

Tabela 04 – Estrutura do dicionário “jsondados”

Chave	**Valor**	
	Chave	Valor
1	id	-
	nome	-
	senha	-

Fonte: elaborada pelo autor.

Como é possível observar, “jsondados” é um dicionário que armazena outros dicionários. Para que seja possível acessar os dados presentes em cada um desses dicionários armazenados, é necessário iterar “jsondados” através de um ciclo “*for*”.

Dentro desse ciclo, os valores de “jsondados” são enviados à variável “logindic”. Essa variável então receberá um dicionário com três chaves “id”, “nome” e “senha”. Os valores referentes às chaves “nome” e “senha” são recebidos pelas variáveis “usuario” e “password”.

Ocorre então um teste de comparação entre os valores digitados pelo usuário e as variáveis “usuario” e “password”. Caso ambos os testes sejam verdadeiros, o segue “LoginSegue”, presente em 2.1.1 é realizado e em seguida é utilizado um *return* para sair da função. Caso o teste seja falso, outra iteração do ciclo “*for*” é realizada.

Se, após todas as iterações serem realizadas, não houve nenhuma correspondência no condicional “if”, a mensagem na etiqueta “ErrorLabel” é modificada para “Usuário e/ou senha errados” para notificar o usuário.

5.2. Funções de “AdicionarViewController”

Como demonstrado em 3.2, este “ViewController” tem como função adicionar novas tomadas ao menu “MenuTableViewController”. Para isso, são necessários dois processos: leitura dos dados para verificação do código e atualização das colunas “nome” e “descricao” da respectiva tomada. Primeiramente, deve-se demonstrar as variáveis utilizadas em ambos os processos.

```
var reqread = URL(string:"")
var codigo: String = ""
var idd: String = ""
var codigoarray = [String]()
var tomadadic = [String:Any]()
var ipi:String!
```

Praticamente todas essas variáveis serão utilizadas no processo de leitura de dados. Apenas a variável “ipi”, com utilização idêntica em 5.1, é utilizada em ambos os processos. As variáveis “codigo” e “idd” armazenarão respectivamente o código correto digitado pelo usuário e o “id” da tomada selecionada. O vetor “codigoarray” é utilizado para armazenar os códigos das tomadas presentes no banco de dados. O dicionário “tomadadic” tem função análoga ao

dicionário presente em 5.1, funciona como variável auxiliar para facilitar o acesso aos dados retirados do banco de dados.

Diferente de “LoginViewController”, a função de leitura não possui validação de dados internamente. A validação dos dados seguida da realização do *segue* é feita pela ação “saveTest”, explicada detalhadamente em 3.2.

```
    private func lerDataaparelhos() {
        reqread = URL(string:"http://\(ipi!)/dadostomada.php")
        let urlContents = try? Data(contentsOf: reqread!)
        let data = urlContents
        var jsondados = try! JSONSerialization.jsonObject(with: data!, options: []) as!
[String:Any]
        for (key,_) in jsondados{
            tomadadic = jsondados[key] as! [String : Any]
            codigo = tomadadic["soid"] as! String
            codigoarray.append(codigo)
            if tomadacod.text == codigo{
                idd = tomadadic["id"] as! String
            }
        }
    }
```

Essa função recebe os dados do script “dadostomada.php” de maneira idêntica à função “getDadosUser”, que recebe os dados de “login.php”, presente em 5.1. O dicionário recebido tem sua estrutura demonstrada na Tabela 05.

Tabela 05 – Estrutura do dicionário “jsondados”

Chave	**Valor**	
	Chave	Valor
1	id	-
	id_ususario	-
	soid	-
	nome	-
	descricao	-
	status	-

Fonte: elaborada pelo autor.

Em “AdicionarViewController”, o único dado necessário para validação é o dado da coluna “soid” da tabela Aparelhos. Então, “jsondados” será iterada pelo ciclo “for” para que o vetor “codigoarray” receba todos os códigos através da variável auxiliar “código”. Ocorre um

teste de validação dentro do ciclo de maneira que o “id” da tomada que possui o mesmo código com o digitado seja armazenado. Após essas iterações, o bloco para de ser executado e retorna para a ação “saveTest”.

Caso algum dos resultados do teste de validação da coluna “soid” com o código digitado seja verdadeiro, a função “mudarnomedesc” é executada. Essa função altera valores presentes no banco de dados conforme os dados preenchidos pelo usuário.

```
   private func mudarnomedesc() {
      let request = NSMutableURLRequest(url: NSURL(string:
"http://\(ipi!)/editartomada.php")! as URL)
      request.httpMethod = "POST"
      let postString = "a=\(tomadanome.text!)&b=\(tomadacod.text!)&c=\(tomadadesc.text!)"
      request.httpBody = postString.data(using: String.Encoding.utf8)
      let task = URLSession.shared.dataTask(with: request as URLRequest) {
         data, response, error in
         if error != nil {
            print("error=\(String(describing: error))")
            return
         }
         let responseString = NSString(data: data!, encoding: String.Encoding.utf8.rawValue)
         print("post: \(String(describing: responseString))")
      }
      task.resume()
   }
```

Na primeira linha da função é realizada uma requisição utilizando a função “NSMutableURLRequest”. Essa requisição é direcionada para a URL "http://\(ipi!)/editartomada.php". Em seguida é escolhido o método de requisição HTTP a ser utilizado. Nesse caso, como determinado em 4.2.2 e demonstrado em 4.3.4, é utilizado o método “POST”. Em seguida, é definida a *string* que será enviada. Essa *string* enviará os valores preenchidos pelo usuário nas caixas de texto presentes na interface.

Em seguida, o corpo da mensagem HTTP consiste nos dados presentes na *string* enviada. Em seguida é criada uma “tarefa”, que realizará o envio dos dados para o script em PHP. Essa função “URLSession.shared.dataTask” tem como parâmetro a URL descrita na primeira linha. Ao executar essa função, é criado um novo elemento dentro da função: uma instância, ou seja, é criado um elemento dentro do código que funcionará apenas naquele momento. Todos as mudanças realizadas em instâncias não reverberam para toda a extensão da aplicação.

Os parâmetros dessa instância são “data”, que consiste nos dados retornados pelo servidor, “response”, que contém dados importantes para a análise da resposta do servidor, e “error”, que pode conter, caso haja um erro, a descrição do erro ocorrido.

O primeiro parâmetro a ser testado é “error”. Caso tenha havido um erro na execução do script, será mostrado em um dado interno, a descrição desse erro e então a função é terminada com um “return”.

Caso esse erro seja nulo, os dados da resposta são codificados em uma *string* para então serem mostrados internamente. Em seguida, a tarefa é finalizada. Caso todos os passos tenham sido realizados com sucesso, os valores da coluna “nome” e “descricao” presentes no banco de dados serão alterados na linha que possui o mesmo código que o preenchido.

5.3. Funções de “DetalheTomadaViewController”

Nessa seção, serão apresentadas três funções de conexão com o servidor semelhantes às funções apresentadas em 5.2. Nesse “ViewController” existem duas funções de leitura e apresentação de dados e uma função de escrita de dados. Além dessas funções será apresentada uma função auxiliar de cálculo de consumo, que depende diretamente dos dados enviados pelo servidor.

Como existe uma grande quantidade de dados necessários para apresentação desse “ViewController”, existe também uma grande quantidade de variáveis utilizadas nas funções.

```
var ligado:Int = 1
var reqconsumo = URL(string:"")
var reqread = URL(string:"")
var consumomed : Double?
var consumomax : Double?
var consumoarray = [(id: Double, cons: Double)]()
var consumoarray2 = [Double]()
var dataarray = [(iden: Double, data: String)]()
var dataarray2 = [String]()
var sum : Double?
var soid : String!
var soid2 : String!
var ipi: String!
var reqread = URL(string:"")
var codigo:String?
var onoff:String?
var tomadadic = [String:Any]()
```

A função de leitura de dados de consumo utiliza o dicionário “tomadadic”, que é utilizado no tratamento dos dados recebidos, e três variáveis de medições: “reqconsumo”, que contém a URL utilizada no consumo, “consumoarray” e “consumoarray2”, vetores de *tuples* e *doubles*, respectivamente. O primeiro vetor recebe os dados desordenados de um dicionário e é utilizado para ordenar e então repassar os dados de consumo ordenados ao segundo vetor. Além disso, existem duas variáveis referentes às datas de medição: “dataarray” e “dataarray2”. Essas variáveis atuam de maneira idêntica aos vetores de consumo utilizando os dados de data de submissão das medições.

Essa função também utiliza as variáveis “soid” e “soid2” para realizar testes de validação dos dados recebidos.

As variáveis “sum”, “consumomed” e “consumomax” são utilizadas pela função auxiliar de cálculo de consumo para apresentar respectivamente: soma dos valores de consumo, consumo médio e consumo máximo instantâneo.

Na função de validação do estado da chave presente na interface, é utilizada a variável “reqconsumo” para armazenar a URL utilizada no processo. Após o recebimento de dados, a variável “onoff” armazena o estado atual. Finalmente, no processo de envio da mudança de estado, é utilizada a variável “ligado” nos testes.

A primeira função receberá os dados das medições da tomada, para organizar e tratar esses dados de maneira que o consumo seja apresentado. Essa função é chamada “requestDataconsumo”.

Essa função é requisitada em dois momentos pelo aplicativo. A primeira execução ocorre antes da interface final ser apresentada, para que o usuário possa visualizar os dados recebidos pelo programa. Ela pode ser executada novamente, caso o usuário pressione o botão “Atualizar dados”. Ela receberá os dados do servidor e realizará o cálculo novamente.

```
    private func requestDataconsumo() {
        reqconsumo = URL(string:"http://\(ipi!)/consumo.php")
        let urlContents = try? Data(contentsOf: reqconsumo!)
        let data = urlContents
        let jsonconsumo = try! JSONSerialization.jsonObject(with: data!, options: [])
as! [String:Any]
        for (key,_) in jsonconsumo{
            var tomadadic = [String:Any]()
            tomadadic = jsonconsumo[key] as! [String : Any]
            soid2 = tomadadic["id_aparelho"] as! String
            if soid == soid2 {
                let tensao = tomadadic["tensao"] as! String
                let corrente = tomadadic["corrente"] as! String
                let consumo = Double(tensao)!*Double(corrente)!
                let id = tomadadic["id"] as! String
                let data = tomadadic["data_submissao"] as! String
                dataarray.append((Double(id)!,data))
                consumoarray.append((Double(id)!,consumo))
            }
        }
        consumoarray.sort{$0.0 < $1.0}
        consumoarray2 = consumoarray.map{$0.cons}
        dataarray.sort{$0.0 < $1.0}
        dataarray2 = dataarray.map{$0.data}
    }
```

Nas primeiras linhas da função, é realizada a conexão com a URL "http://\(ipi!)/consumo.php" e os dados enviados pelo servidor são recebidos. O formato dos dados recebidos pela constante “jsonconsumo” está presente na Tabela 06.

Tabela 06 – Estrutura do dicionário “jsondados”

Chave	**Valor**	
	Chave	Valor
1	id	-
	id_aparelho	-
	corrente	-
	tensao	-
	data_submissao	-

Fonte: elaborada pelo autor.

De forma similar às seções 5.1 e 5.2, a constante “jsonconsumo” é um dicionário de dicionários. Para que os valores sejam retirados, é necessário que “jsonconsumo” seja iterada e seus valores sejam repassados ao dicionário “tomadadic”.

A variável “soid2” recebe o valor referente à chave “id_aparelho” de “tomadadic”. Em seguida é realizado um teste condicional utilizando as identificações das tomadas na medição. Caso o teste seja verdadeiro, ou seja, a coluna “id_aparelho” possua o mesmo valor do “id” da tomada analisada, são definidas quatro variáveis: “tensao”, “corrente”, “id” e “data”, que receberão respectivamente os valores de tensão, corrente, “id” e data de submissão da linha analisada. Os valores de “tensao” e “corrente” são então multiplicados e a constante “consumo” recebe o valor dessa multiplicação. Em seguida, o vetor “dataarray” tem anexado em uma posição os valores de “id” e “data”. O vetor “consumoarray” passa pelo mesmo processo de anexação dos valores de “id” e “consumo”. Para facilitar a organização, todos os valores numéricos, que foram recebidos durante o processo como *string*, foram convertidos em *double*.

Após a realização de todas as iterações, é utilizada a função de vetores *sort* para classificar em ordem crescente de “id”, o vetor “consumoarray”. Em seguida, utilizando a função de vetores *map*, a variável “consumoarray2” recebe os valores de consumo ordenados. Processo semelhante ocorre com os valores de data de submissão, que são ordenados e enviados à variável “dataarray2”. Esses vetores de consumo e data de submissão, quando ordenados, serão de grande importância na execução de “GraficoViewController”, como demonstrado em 3.5.

Em seguida, os dados de consumo necessitam ser tratados novamente para que o usuário possa visualizar o valor médio e o valor máximo instantâneo. Esse processo é realizado utilizando a função “consumocalc”.

```
private func consumocalc() {
    sum = consumoarray2.reduce(0, +)
    consumomed = sum!/Double(consumoarray2.count)
    consumomax = consumoarray2.max()
    updateData()
}
```

Na primeira linha da função, a variável “sum” recebe o resultado da soma de todos os valores de “consumoarray2”. Para isso, é utilizada a função *reduce*, que reduz todos os valores presentes em um valor apenas, somando-os. Em seguida, “consumomed” recebe esse valor da soma dividido pela quantidade de elementos presente em “consumoarray2”,

caracterizando o cálculo da média aritmética. Em seguida é utilizada a função *max*, que calcula o maior valor presente no vetor e envia à variável “consumomax”. Para finalizar a execução da função, é executado “updateData”, que atualiza a interface apresentada ao usuário com os valores calculados nessa função.

A segunda função é ligada diretamente à chave que determina o estado da tomada. Sua execução é bastante similar à função “lerDataaparelhos” apresentada em 5.2, e tem o intuito de verificar o estado na chave dentro do servidor para que esse valor seja repassado ao usuário.

```
    private func verificarstatus() {
        reqread = URL(string:"http://\(ipi!)/dadostomada.php")
        let urlContents = try? Data(contentsOf: reqread!)
        let data = urlContents
        var jsondados = try! JSONSerialization.jsonObject(with: data!, options: []) as!
[String:Any]
        for (key,_) in jsondados{
            tomadadic = jsondados[key] as! [String : Any]
            codigo = tomadadic["soid"] as? String
            if codLabel.text == codigo!{
                onoff = tomadadic["status"] as? String
                if Double(onoff!) == 1{
                    status.isOn = true
                } else {
                    status.isOn = false
                }
            }
        }
    }
```

A conexão com o servidor e recepção de dados é idêntica à apresentada em 5.2. Da mesma maneira, a estrutura de “jsondados” está presente na Tabela 05. A maior diferença entre as funções está no tratamento dos dados recebidos. Os dados importantes na verificação do estado são as colunas “soid” e “status” da tabela “aparelhos”.

Com a iteração dos dados de “jsondados”, é repassado à variável “codigo”, o valor da coluna “soid” da linha analisada. Em seguida é realizado um teste para saber se a variável “codigo” possui o mesmo valor que o presente na interface. Caso o teste retorne verdadeiro, a variável “onoff” recebe o valor da coluna “status” da linha analisada, Em seguida é realizado um teste para determinar o estado da chave presente na interface. Se o valor de “onoff” é 1, a chave se apresentará ligada, caso contrário, ela se apresentará desligada.

A terceira função de conexão com o servidor, na verdade é uma ação ligada à “UISwitch”, e é responsável por o verificar a mudança no estado da chave na interface, para então se conectar ao servidor e repassar essa mudança à coluna “status” na tabela “aparelhos” do banco de dados.

```
  @IBAction func mudarstatus(_ sender: Any) {
     let request = NSMutableURLRequest(url: NSURL(string: "http://\(ipi!)/editarstatus.php")!
as URL)
     request.httpMethod = "POST"
     if status.isOn == false {
        ligado = 0
     } else {
        ligado = 1
     }
        let postString = "a=\(String(ligado))&b=\(cod!)"
        request.httpBody = postString.data(using: String.Encoding.utf8)
        let task = URLSession.shared.dataTask(with: request as URLRequest) {
           data, response, error in
           if error != nil {
              print("error=\(String(describing: error))")
              return
           }
           let responseString = NSString(data: data!, encoding: String.Encoding.utf8.rawValue)
           print("post: \(String(describing: responseString))")
        }
        task.resume()
  }
```

As requisições realizadas são similares às requisições presentes na função “mudarnomedesc”. É realizada uma requisição através da URL “http://\(ipi!)/editarstatus.php”, utilizando o método “POST”. Em seguida é realizado um teste do estado atual da chave, para que seja repassado o novo valor da chave para o servidor utilizando a *string* “postString”. Além desse valor de estado, é repassado dentro da *string* o valor do código da tomada. Em seguida ocorre a instância devido à utilização de “URLSession.shared.dataTask”. Dentro dessa instância, serão recebidos os valores referentes à conexão com o servidor. Caso não ocorra nenhum erro e os valores sejam, o programa deve receber a mensagem interna “Record updated successfully”.

6. CONCLUSÃO E TRABALHOS FUTUROS

6.1. Conclusão

No desenvolvimento desse trabalho, foi concebido um aplicativo móvel baseado em linguagem Swift, para sistema operacional iOS, para monitoramento de uma tomada inteligente via Wi-Fi.

Sua criação pode ser dividida em quatros partes: introdução à linguagem Swift, Interface Homem-Máquina, características do banco de dados e dos scripts em PHP e conexão do aplicativo com os scripts.

Na primeira parte, foram desenvolvidos os conceitos da linguagem de programação utilizada de maneira concisa, de modo que o entendimento do código apresentado nos demais capítulos seja facilitado.

Em seguida, são apresentadas as interfaces presentes no aplicativo, os elementos presentes nessas interfaces, como o usuário interage com esses elementos e como essas interações afetam os dados presentes no aplicativo, através dos blocos de código chamados “ViewControllers”.

As duas últimas partes são dedicadas à conexão com a internet. A terceira parte mostra o lado do servidor, como os dados estão armazenados e como eles são retirados do servidor. Na parte final são apresentadas as demais funções do aplicativo, relacionadas à conexão com a internet. As mensagens trocadas utilizam-se do protocolo HTTP via TCP, o que garante velocidade e segurança.

Na execução do que foi demonstrado, é notável a simplicidade da interface apresentada. Além de simples, também é eficiente: são apresentadas ao usuário apenas as informações necessárias para a utilização do aplicativo.

Além disso, o aplicativo consume pouca memória RAM, o consumo médio do aplicativo foi de 30MB durante a execução. Considerando os aparelhos recentes mais básicos da Apple: iPhone 7 e 8, que possuem 2GB de memória RAM, o consumo chega a 1,46% da memória do celular.

A análise de rede do programa também mostrou um consumo de dados baixo. Em um ciclo do aplicativo: autenticação de usuário, adição de nova tomada e apresentação dos dados dessa tomada, o aplicativo utilizou-se apenas de 6kB de dados. Isso faz com que, mesmo em conexões ruins, o aplicativo funcione de maneira rápida.

6.2. Trabalhos Futuros

- *Migração do Aplicativo para outros sistemas operacionais:* apesar do iOS ser um sistema bastante popular, atualmente existe o Android, plataforma mais aberta, que está presente em uma quantidade mais diversa de aparelhos, fazendo com que possua maior base instalada de usuários. Além disso, outros sistemas operacionais estão em desenvolvimento e podem se popularizar no futuro, fazendo com que seja importante a presença do aplicativos nesses ecossistemas;
- *Incremento e Atualização do Aplicativo*: a linguagem de programação Swift está em constante desenvolvimento, por isso é importante que o aplicativo sempre esteja atualizado tanto com o desenvolvimento da linguagem quanto com o desenvolvimento dos aparelhos que utilizam essa linguagem. É importante também que o aplicativo siga o desenvolvimento da própria tomada inteligente: conforme novas funcionalidades no hardware da tomada surjam, é necessário que o software apresente essas funcionalidades;
- *Autenticação de usuário via TouchID e FaceID*: os aparelhos mais recentes da Apple apresentam o TouchID, que é a capacidade de reconhecer o usuário via identificação biométrica. Além disso, uma nova forma de reconhecimento de usuário foi apresentada: o FaceID, que consiste no reconhecimento facial para autenticação de usuário. Essas funcionalidades geram mais segurança e podem ser adicionadas na autenticação de usuário do aplicativo sem que haja um grande ônus de performance.

REFERÊNCIAS

APPLE INC. **The Swift Documentation**, 2017. Disponível em: <https://developer.apple.com/library/content/navigation/>. Acesso em 03/11/2017.

FINALIZE.COM. UIKit Class Hierarchy Chart, 2017. Disponível em: <https://finalize.com/2012/12/14/uikit-class-hierarchy-chart/>. Acesso em 03/11/2017.

JONAS, Ícaro. **Tomada Inteligaente Baseada em Internet das Coisas (IoT) com Leitura de Energia em Tempo Real**. 2016. 39 f. Trabalho de conclusão de curso (Monografia) – Curso de Engenharia Elétrica, Universidade Federal do Ceará, Fortaleza, Ceará. 2016.

NEUBURG, Matt. **iOS Programming Fundamentals with Swift**: Swift, Xcode and Cocoa Basics. 3ª. ed. O'Reilly: 2016

APLLE INC. **The Swift Programming Language (Swift 4)**, 4ª. ed. Apple Inc.: 2014

HOFFMAN, Jon. **Mastering Swift 3**. 1ª. ed. Packt: 2016

THE PHP GROUP. PHP **Documentation**, 2017. Disponível em: <http://php.net/docs.php/>. Acesso em 03/11/2017.

UNIVERSIDADE FEDERAL DO CEARÁ. Biblioteca Universitária. **Guia de normalização de trabalhos acadêmicos da Universidade Federal do Ceará**. Fortaleza, 2013.

ECHESSA, Joyce. How to use iOS Charts API to Create Beautiful Charts in Swift, 2015. Disponívem em <https://www.appcoda.com/ios-charts-api-tutorial/>. Acesso em: 03/11/2017

APÊNDICE A – CÓDIGO DE “LOGINVIEWCONTROLLER”

```
//
//  LoginViewController.swift
//  Tomada2
//
//  Created by Tobias Macedo on 03/10/17.
//  Copyright © 2017 Tobias Macedo. All rights reserved.
//

import UIKit

class LoginViewController: UIViewController {


    @IBOutlet weak var login: UITextField!
    @IBOutlet weak var senha: UITextField!
    @IBOutlet weak var ip: UITextField!
    @IBOutlet weak var Errorlabel: UILabel!
    @IBOutlet weak var IpErrorLabel: UILabel!
    var ipi: String?
    var reqlogin = URL(string:"")
    var logindic = [String:Any]()
    var usuario: String!
    var password: String!


    override func viewDidLoad() {
        super.viewDidLoad()
        Errorlabel.text = ""
        IpErrorLabel.text = ""
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }


    // MARK: - Navigation

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if (segue.identifier == "LoginSegue") {
            let VC = segue.destination as! MenuTableViewController
            VC.ipi = ipi
        }
    }
```

```
    // MARK: - Actions
    @IBAction func Entrar(_ sender: Any) {
        getDadosUser()
    }

    // MARK: - PHP Requests

    private func getDadosUser() {
        ipi = ip.text!
        reqlogin = URL(string:"http://\(ipi!)/login.php")
        let urlContents = try? Data(contentsOf: reqlogin!)
        let data = urlContents
        var jsondados = try! JSONSerialization.jsonObject(with: data!, options: []) as!
[String:Any]
        for (key,_) in jsondados{
            logindic = jsondados[key] as! [String : Any]
            usuario = logindic["nick"] as! String
            password = logindic["senha"] as! String
            if usuario == login.text && password == senha.text {
                performSegue(withIdentifier: "LoginSegue", sender: self)
                return
                }
        }
        Errorlabel.text = "Usuário e/ou senha errados"
    }
}
```

APÊNDICE B – CÓDIGO DE “MENUTABLEVIEWCONTROLLER”

```
//
//  MenuTableViewController.swift
//  TomadaTCC
//
//  Created by Tobias Macedo on 14/09/17.
//  Copyright © 2017 Tobias Macedo. All rights reserved.
//

import UIKit

class MenuTableViewController: UITableViewController {

    var tomadas = [String]()
    var codigos = [String]()
    var soids = [String] ()
    var newTomada: String = ""
    var newCod: String = ""
    var newId: String = ""
    var name : String = ""
    var valueToPass:String!
    var valueToPass2:String!
    var valueToPass3:String!
    var ipi:String!

    @IBAction func cancel(segue:UIStoryboardSegue) {

    }
    @IBAction func cancel3(segue:UIStoryboardSegue) {

    }

    @IBAction func save(segue:UIStoryboardSegue) {
        let AdicionarVC = segue.source as! AdicionarViewController
        newTomada = AdicionarVC.name
        tomadas.append(newTomada)
        newCod = AdicionarVC.codigo
        codigos.append(newCod)
        newId = AdicionarVC.idd
        soids.append(newId)
        tableView.beginUpdates()
        tableView.insertRows(at: [IndexPath(row: tomadas.count-1, section: 0)],
with: .automatic)
        tableView.endUpdates()
    }

    override func viewDidLoad() {
        super.viewDidLoad()
    }
```

```
    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }

    // MARK: - Table view data source
    override func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
        return tomadas.count
    }

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "tomadaCell", for:
indexPath)
        cell.textLabel!.text = tomadas[indexPath.row]
        return cell
    }


    override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
        let indexPath = tableView.indexPathForSelectedRow!
        let currentCell = tableView.cellForRow(at: indexPath)! as UITableViewCell
        valueToPass = currentCell.textLabel?.text
        valueToPass2 = codigos[(indexPath.row)]
        valueToPass3 = soids[(indexPath.row)]
        performSegue(withIdentifier: "DetalheSegue", sender: self)
    }

    // MARK: - Actions
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if (segue.identifier == "DetalheSegue") {
            let VC = segue.destination as! DetalheTomadaViewController
            VC.ipi = ipi
            VC.name = valueToPass
            VC.cod = valueToPass2
            VC.soid = valueToPass3
          }
        if (segue.identifier == "AdicionarSegue") {
            let VC = segue.destination as! AdicionarViewController
            VC.ipi = ipi
        }
    }
}
```

APÊNDICE C – CÓDIGO DE “ADICIONARVIEWCONTROLLER”

```
//
//  AdicionarViewController.swift
//  Tomada2
//
//  Created by Tobias Macedo on 14/09/17.
//  Copyright © 2017 Tobias Macedo. All rights reserved.
//

import UIKit

class AdicionarViewController: UIViewController, UITextFieldDelegate {

    @IBOutlet weak var tomadanome: UITextField!
    @IBOutlet weak var tomadacod: UITextField!
    @IBOutlet weak var tomadadesc: UITextField!
    @IBOutlet weak var codigonaorec: UILabel!
    @IBOutlet weak var saveButton: UIBarButtonItem!

    var name: String = ""
    var reqread = URL(string:"")
    var codigo: String = ""
    var idd: String = ""
    var codigoarray = [String]()
    var tomadadic = [String:Any]()
    var ipi:String!

    override func viewDidLoad() {
        super.viewDidLoad()
        codigonaorec.text = ""
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }

    //MARK - Navigation

    override func prepare(for segue: UIStoryboardSegue, sender: Any!) {
        if segue.identifier == "savesegue" {
            name = tomadanome.text!
        }
    }
```

```
    //MARK - Actions

    @IBAction func saveTest(_ sender: Any) {
        lerDataaparelhos()
        for i in 0...(codigoarray.count-1){
            if tomadacod.text == codigoarray[i] {
                mudarnomedesc()
                codigonaorec.text = ""
                codigo = codigoarray[i]
                performSegue(withIdentifier: "savesegue", sender: self)
            } else {
                codigonaorec.text = "Código não encontrado"
            }
        }
    }

    //MARK - Private Funcs

    private func lerDataaparelhos() {
        print(ipi)
        reqread = URL(string:"http://\(ipi!)/dadostomada.php")
        let urlContents = try? Data(contentsOf: reqread!)
        let data = urlContents
        var jsondados = try! JSONSerialization.jsonObject(with: data!, options: []) as!
[String:Any]
        for (key,_) in jsondados{
            tomadadic = jsondados[key] as! [String : Any]
            codigo = tomadadic["soid"] as! String
            codigoarray.append(codigo)
            if tomadacod.text == codigo{
                idd = tomadadic["id"] as! String
            } else {
                print("INCORRETO")
            }
        }
    }
```

```
    private func mudarnomedesc() {
        let request = NSMutableURLRequest(url: NSURL(string:
"http://\(ipi!)/editartomada.php")! as URL)
        request.httpMethod = "POST"
        let postString =
"a=\(tomadanome.text!)&b=\(tomadacod.text!)&c=\(tomadadesc.text!)"
        request.httpBody = postString.data(using: String.Encoding.utf8)
        let task = URLSession.shared.dataTask(with: request as URLRequest) {
            data, response, error in
            if error != nil {
                print("error=\(String(describing: error))")
                return
            }
            let responseString = NSString(data: data!, encoding:
String.Encoding.utf8.rawValue)
            print("post: \(String(describing: responseString))")
        }
        task.resume()
    }
}
```

APÊNDICE D – CÓDIGO DE “DETALHETOMADAVIEWCONTROLLER”

```
//
//  DetalheTomadaViewController.swift
//  TomadaTCC
//
//  Created by Tobias Macedo on 15/09/17.
//  Copyright © 2017 Tobias Macedo. All rights reserved.
//

import UIKit

class DetalheTomadaViewController: UIViewController {


    @IBOutlet weak var nameLabel: UILabel!
    @IBOutlet weak var codLabel: UILabel!
    @IBOutlet weak var consumomedLabel: UILabel!
    @IBOutlet weak var consumomaxLabel: UILabel!
    @IBOutlet weak var status: UISwitch!

    var ligado:Int = 1
    var reqconsumo = URL(string:"")
    var name : String!
    var cod : String!
    var consumomed : Double?
    var consumomax : Double?
    var consumoarray = [(id: Double, cons: Double)]()
    var tensaoarray = [Double]()
    var correntearray = [Double]()
    var consumoarray2 = [Double]()
    var dataarray = [(iden: Double, data: String)]()
    var dataarray2 = [String]()
    var sum : Double?
    var i : Double = 0
    var soid : String!
    var soid2 : String!
    var ipi: String!
    var reqread = URL(string:"")
    var tomadadic = [String:Any]()
    var codigo:String?

    override func viewDidLoad() {
        super.viewDidLoad()
        requestDataconsumo()
        consumocalc()
        verificarstatus()
    }
```

```
    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }


    // MARK: - Navigation

    // In a storyboard-based application, you will often want to do a little preparation before
navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if let vc = segue.destination as? GraficoViewController {
            vc.consumo = consumoarray2
            vc.datames = dataarray2

        }
    }

    //MARK - PHP Requests

    private func verificarstatus() {
        reqread = URL(string:"http://\(ipi!)/dadostomada.php")
        let urlContents = try? Data(contentsOf: reqread!)
        let data = urlContents
        var jsondados = try! JSONSerialization.jsonObject(with: data!, options: []) as!
[String:Any]
        for (key,_) in jsondados{
            tomadadic = jsondados[key] as! [String : Any]
            codigo = tomadadic["soid"] as? String
            if codLabel.text == codigo!{
                onoff = tomadadic["status"] as? String
                if Double(onoff!) == 1{
                    status.isOn = true
                } else {
                    status.isOn = false
                }
            }

        }
    }
```

```
    private func requestDataconsumo() {
        reqconsumo = URL(string:"http://\(ipi!)/consumo.php")
        let urlContents = try? Data(contentsOf: reqconsumo!)
        let data = urlContents
        let jsonconsumo = try! JSONSerialization.jsonObject(with: data!, options: []) as!
[String:Any]
        for (key,_) in jsonconsumo{
            var tomadadic = [String:Any]()
            tomadadic = jsonconsumo[key] as! [String : Any]
            soid2 = tomadadic["id_aparelho"] as! String
            if soid == soid2 {
                let tensao = tomadadic["tensao"] as! String
                let corrente = tomadadic["corrente"] as! String
                let consumo = Double(tensao)!*Double(corrente)!
                let id = tomadadic["id"] as! String
                let data = tomadadic["data_submissao"] as! String
                dataarray.append((Double(id)!,data))
                consumoarray.append((Double(id)!,consumo))
            }
        }
        consumoarray.sort{$0.0 < $1.0}
        consumoarray2 = consumoarray.map{$0.cons}
        dataarray.sort{$0.0 < $1.0}
        dataarray2 = dataarray.map{$0.data}
    }

    //MARK - Actions

    @IBAction func mudarstatus(_ sender: Any) {
        let request = NSMutableURLRequest(url: NSURL(string:
"http://\(ipi!)/editarstatus.php")! as URL)
        request.httpMethod = "POST"
        if status.isOn == false {
            ligado = 0
        } else {
            ligado = 1
        }
            let postString = "a=\(String(ligado))&b=\(cod!)"
            request.httpBody = postString.data(using: String.Encoding.utf8)
            let task = URLSession.shared.dataTask(with: request as URLRequest) {
                data, response, error in
                if error != nil {
                    print("error=\(String(describing: error))")
                    return
                }
                let responseString = NSString(data: data!, encoding:
String.Encoding.utf8.rawValue)
                print("post: \(String(describing: responseString))")
            }
            task.resume()
    }
```

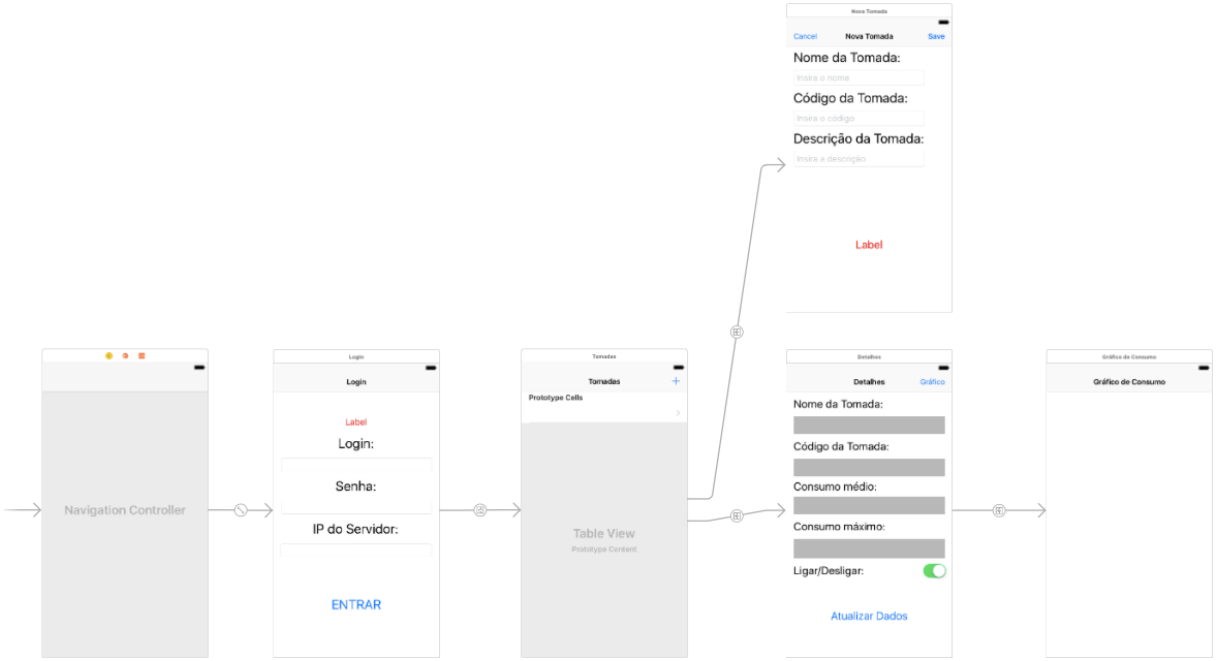
```
    @IBAction func AtualizarDados(_ sender: Any) {
        consumoarray.removeAll()
        dataarray.removeAll()
        requestDataconsumo()
        consumocalc()
    }

}
```

APÊNDICE E – CÓDIGO DE “GRAFICOVIEWCONTROLLER”

```
//
//  GraficoViewController.swift
//  TomadaTCC
//
//  Created by Tobias Macedo on 16/09/17.
//  Copyright © 2017 Tobias Macedo. All rights reserved.
//

import UIKit
import Charts

class GraficoViewController: UIViewController {

    @IBOutlet weak var barView: BarChartView!
    var consumo = [Double]()
    var consumousavel = [Double]()
    var datames = [String]()
    var datahora = [String]()
    var ultimadata: String!
    var dataaux: String!

    override func viewDidLoad() {
        super.viewDidLoad()
        getultimadata()
        testusableData()
        updateChartWithData()
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }

    func getultimadata() {
        ultimadata = datames.last!
        let index = ultimadata.index(ultimadata.startIndex, offsetBy: 10)
        ultimadata = (ultimadata.substring(to: index))
    }
```

```
    func testusableData() {
        for i in 0..<datames.count {
            let index = datames[i].index(datames[i].startIndex, offsetBy: 10)
            dataaux = (datames[i].substring(to: index))
            if dataaux == ultimadata {
                consumousavel.append(consumo[i])
                let start = datames[i].index(datames[i].startIndex, offsetBy: 11)
                let end = datames[i].index(datames[i].endIndex, offsetBy: -6)
                let range = start..<end
                datahora.append(datames[i].substring(with: range))
            }
        }
    }

    func updateChartWithData() {
        var dataEntries: [BarChartDataEntry] = []
        for i in 0..<consumousavel.count {
            let dataEntry = BarChartDataEntry(x: Double(i), y: Double(consumousavel[i]))
            dataEntries.append(dataEntry)
        }
        barView.xAxis.valueFormatter = IndexAxisValueFormatter(values:datahora)
        barView.xAxis.granularity = 1
        let chartDataSet = BarChartDataSet(values: dataEntries, label: "Consumo instantâneo
em \(ultimadata!) (W)")
        let chartData = BarChartData(dataSet: chartDataSet)
        barView.data = chartData
        barView.chartDescription?.text = ""
    }

}
```

APÊNDICE F – ESQUEMÁTICO “STORYBOARD” DO APLICATIVO



APÊNDICE G – CÓDIGO DE “LOGIN.PHP”

```
<?php
$servername = "localhost";
$username = "root";
$password = "";
$dbname = "iphone";
$conn = new mysqli($servername, $username, $password, $dbname);
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$sql = "SELECT id, nick, senha FROM usuarios";
$result = $conn->query($sql);
$rownum = count($result);
$arrayusuario = array();


if ($result->num_rows > 0) {
    while($row = $result->fetch_assoc()) {
        for ($i=0;$i<$rownum;$i++){
        $arrayusuario[$row["id"]] = $row;
    }
    }
    ;
} else {
    echo "0 results";
}
echo json_encode($arrayusuario);
?>
```

APÊNDICE H – CÓDIGO DE “DADOSTOMADA.PHP”

```
<?php
$servername = "localhost";
$username = "root";
$password = "";
$dbname = "iphone";
$conn = new mysqli($servername, $username, $password, $dbname);
if ($conn->connect_error) {
   die("Connection failed: " . $conn->connect_error);
}

$sql = "SELECT id, id_usuario, soid, nome, descricao, status FROM aparelhos";
$result = $conn->query($sql);
$rownum = count($result);
$arraydados = array();


if ($result->num_rows > 0) {
   while($row = $result->fetch_assoc()) {
      for ($i=0;$i<$rownum;$i++){
      $arraydados[$row["id"]] = $row;
      }
   }
} else {
   echo "0 results";
}
echo json_encode($arraydados);
?>
```

APÊNDICE I – CÓDIGO DE “CONSUMO.PHP”

```
<?php
$servername = "localhost";
$username = "root";
$password = "";
$dbname = "iphone";
$conn = new mysqli($servername, $username, $password, $dbname);
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$sql = "SELECT id, id_aparelho, corrente, tensao, data_submissao FROM telemetria";
$result = $conn->query($sql);
$rownum = count($result);
$arraytensao = array();


if ($result->num_rows > 0) {
    while($row = $result->fetch_assoc()) {
        for ($i=0;$i<$rownum;$i++){
        $arraytensao[$row["id"]] = $row;
            }
    }
} else {
    echo "0 results";
}
echo json_encode($arraytensao);
?>
```

APÊNDICE J – CÓDIGO DE “EDITARTOMADA.PHP”

```
<?php
$servername = "localhost";
$username = "root";
$password = "";
$dbname = "iphone";
$conn = new mysqli($servername, $username, $password, $dbname);
if ($conn->connect_error) {
   die("Connection failed: " . $conn->connect_error);
}

if($_SERVER['REQUEST_METHOD']=='POST'){
   $tomadaName = $_POST['a'];
   $tomadacodigo = $_POST['b'];
   $tomadadesc = $_POST['c'];
   $sql = "UPDATE aparelhos SET nome='$tomadaName', descricao='$tomadadesc'
WHERE soid='$tomadacodigo'";
        if ($conn->query($sql) === TRUE) {
        $response = "Record updated successfully";
        } else {
        $response = "Error updating record: " . $conn->error;
        }
}
echo json_encode($response);
?>
```

APÊNDICE K – CÓDIGO DE “EDITARSTATUS.PHP”

```
<?php
$servername = "localhost";
$username = "root";
$password = "";
$dbname = "iphone";
$conn = new mysqli($servername, $username, $password, $dbname);
if ($conn->connect_error) {
   die("Connection failed: " . $conn->connect_error);
}

if($_SERVER['REQUEST_METHOD']=='POST'){
   $tomadaName = $_POST['a'];
   $tomadacodigo = $_POST['b'];
   $tomadadesc = $_POST['c'];
   $sql = "UPDATE aparelhos SET nome='$tomadaName', descricao='$tomadadesc'
WHERE soid='$tomadacodigo'";
          if ($conn->query($sql) === TRUE) {
          $response = "Record updated successfully";
          } else {
          $response = "Error updating record: " . $conn->error;
          }
}
echo json_encode($response);
?>
```