PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

Uma abordagem prática ao desenvolvimento de aplicativos híbridos multiplataforma*

Article template Institute of Mathematical Sciences and Informatics

Lucas Brandão Magalhães Ricoy[1]

Resumo

O presente trabalho abordou o problema de falta de reuso de *software* em aplicações multiplataforma, focando principalmente em casos onde o mesmo *software* é reescrito diversas vezes a fim de permitir sua execução em diferentes plataformas. O trabalho teve como objetivo principal a elaboração de uma arquitetura de software que fosse capaz de gerar a partir de um mesmo código fonte compartilhado, um aplicativo híbrido multiplataforma para ser executado como um *website*, aplicativo móvel ou *desktop*. O projeto possui caráter experimental e é embasado em estudos de técnicas já estabelecidas com conceitos inerentes ao desenvolvimento de *software* baseado em componentes a fim de apresentar uma aplicação prática dos mesmos em um cenário atual. Os resultados evidenciam que o desenvolvimento de aplicativos híbridos multiplataformas com tecnologias atuais é factível apesar de as ferramentas utilizadas serem opinativas em relação ao processo em si.

Palavras-chave: Aplicativos Híbridos. Componentes. Desenvolvimento Móvel. DBC. AngularJS. Ionic. Electron.

*Artigo apresentado ao Instituto de Ciências Exatas e Informática da Pontifícia Universidade Católica de Minas Gerais como pré-requisito para obtenção do título de Bacharel em Ciência da Computação.

[1]Programa de Graduação em Sistemas de Informação da PUC Minas, Brasil – ricoy.lucas@gmail.com.

1 INTRODUÇÃO

Computadores fazem parte de nosso dia a dia de forma onipresente. Utilizamos nossos dispositivos de forma natural e em diversos formatos: celulares, *tablets*, *smartphones*, computadores de mesa, *notebooks*, etc. Esta diversidade de dispositivos é benéfica e nos oferece novos produtos e soluções a medida que as tecnologias existentes evoluem e novas surgem. Porém a cada novo dispositivo criado, a complexidade para que aplicativos específicos sejam desenvolvidos aumenta e muitas vezes, é um empecilho para sua adoção e disseminação.

Durante muito tempo, a maioria dos aplicativos desenvolvidos era destinado a uma mesma plataforma, por exemplo, computadores de mesa ou dispositivos móveis e algumas vezes para Sistemas Operacionais (SO) diferentes. Caso fosse necessário atender uma nova plataforma ou SO, o aplicativo deveria ser reescrito a partir do zero, muitas vezes em uma linguagem diferente e sem reaproveitar o código existente, para que este novo requisito fosse atendido. Tamanha complexidade dificultou que desenvolvedores mais experientes pudessem utilizar linguagens, ferramentas e técnicas já conhecidas e difundidas para produzir aplicativos, que utilizando o mesmo código, fossem aceitos em mais de uma plataforma.

Com a grande disseminação de tecnologias de computação móvel, como os *smartphones*, este problema se tornou cada vez mais crítico. De acordo com Moore, Van Baalen e Shevchenko (2015), a demanda por aplicações móveis nunca foi tão alta e os desenvolvedores vem sofrendo para criar experiências atrativas nas principais plataformas e atingir os maiores mercados da área. Isso nos mostra, que apesar de efervescente, o mercado não está preparado para atender todas as expectativas dos clientes atuais.

Para encarar este problemas, iniciativas como o Intel XDK, Apache Cordova, Adobe Phonegap e Microsoft Xamarin foram criadas visando permitir o desenvolvimento para múltiplas plataformas de uma forma expansível e customizável, com experiências e funcionalidades específicas em cada uma.

Atualmente, uma abordagem comum e que possui grande adoção é o empacotamento de *WebApps* em *containers* nativos. Esta abordagem possui duas grandes vantagens: a possibilidade de utilizar diversas bibliotecas e ferramentas já disseminadas e testadas no desenvolvimento web e a facilidade para desenvolvimento e validação do produto desenvolvido, tendo em consideração que o produto final nada mais é do que um *website* com acesso aos recursos do dispositivo por meio do *container* em que foi empacotado. Aplicativos desenvolvidos desta forma são normalmente denominados "híbridos".

Este artigo tem por objetivo principal a elaboração e formalização de uma arquitetura que permita o desenvolvimento de aplicativos híbridos multiplataforma de forma eficaz, possibilitando o uso de diversos padrões e ferramentas já existentes e o reaproveitamento do mesmo código entre plataformas. Como objetivo secundário está a apresentação de um um protótipo, que utilizando da arquitetura elaborada, será capaz de ser distribuído para sistemas *Desktop* tradicionais (Linux, Windows, Mac), dispositivos móveis (iOS, Android, Windows Phone) ou como um site na *WEB* sem alterações em suas camadas internas, mudando apenas a forma de

apresentação em cada plataforma, quando necessário.

2 REFERENCIAL TEÓRICO

Nesta seção serão apresentados fundamentos teóricos do trabalho desenvolvido.

2.1 Reutilização de Software

Desde que começamos a desenvolver *software*, estamos buscando formas mais eficientes de fazê-lo. Uma das formas mais eficientes de desenvolver *software* é não desenvolver o que já existe, mas reutilizar padrões e soluções já definidos e testados.

(PRESSMAN, 2011) diz que componentes reutilizáveis foram criados à medida que a disciplina de engenharia evoluiu, sendo essa uma prática natural em projetos do tipo, enquanto que por outro lado, componentes de *software* reutilizáveis em larga escala, apenas começaram a serem alcançados. Hoje, sistemas complexos devem ser desenvolvidos em prazos muito curtos com uma alta qualidade. Tais exigências, segundo (SOMMERVILLE, 2011), fizeram com que o desenvolvimento baseado em reuso, se tornasse o padrão para novos sistemas a partir do ano 2000. Tal cenário faz com que os programas escritos devam ser modulares e capazes de se adaptar a diferentes contextos.

A ideia da reutilização, segundo (PRESSMAN, 2011) não é nova, pois vem sendo utilizada desde os primórdios da computação, tendo sua estratégia definida por (MCILROY, 1978) há mais de 40 anos. Apesar de existente, antigamente essa rotina era realizada de modo menos organizado e não sendo tratada como princípio ativo do desenvolvimento. O contexto atual exige uma padronização e direcionamento, normalmente tentando-se maximizar a reutilização de *softwares* existentes ou planejando o desenvolvimento, com o reuso acontecendo em tês níveis distintos, definidos por (SOMMERVILLE, 2011).

1. **Reúso do sistema de aplicação:** Onde a totalidade do sistema pode ser reutilizada sem alterações em outros sistemas via configurações específicas para diferentes clientes.

2. **Reúso de Componentes:** Ao reutilizar componentes internos ou subsistemas em diferentes sistemas de informação.

3. **Reúso de objetos e funções:** Funções ou componentes que implementa uma única função, como uma função matemática ou uma classe de objetos podem ser reutilziados na forma de bibliotecas.

A construção com base em componentes, é, segundo (PRESSMAN, 2011), o caminho o qual a indústria está trilhando. Apesar da afirmação do autor ainda não possuir comprovação

ou dados para validação segundo (VALE et al., 2016), o desenvolvimento de componentes reutilizáveis vem recebendo cada vez mais adeptos segundo <PETE BACON WHITE PAPER DO GOOGLE> e é a abordagem utilizada por esse artigo.

2.2 Arquitetura de Software

A Arquitetura de *Software* é uma área da Engenharia de *Software* que visa relacionar e estruturar os componentes e como eles interagem entre si através de interfaces, expondo ao final a estrutura do sistema de uma forma abstrata. É definida por (Len Bass et al., 2012), como:

A arquitetura de um *software* de um programa ou sistema computacional é a estrutura ou estruturas de um sistema, que inclui os elementos de *software*, as propriedades externamente visíveis destes elementos, e os relacionamentos entre eles. Arquitetura é responsável pelas interfaces públicas dos elementos. Seus detalhes privados (detalhes que tenham a ver somente com a implementação interna) não fazem parte da arquitetura.

3 METODOLOGIA

Nessa seção as fases do projeto e seus respectivos objetivos são apresentados. Visando melhor separar o processo teórico da implementação prática, três etapas foram definidas e realizadas em sequência.

3.1 *Identificação do Problema*

A fim de melhor atender o objetivo deste projeto, uma clarificação e análise do estado da arte atual foram realizadas nessa etapa.

O cenário atual de desenvolvimento de *software* para dispositivos móveis foi analisado e de forma mais aprofundada, as técnicas e ferramentas utilizadas no desenvolvimento de aplicações híbridas. Foi identificado que diversos autores já haviam dissertado sobre o *status quo* e as constantes mudanças que essa área vem sofrendo ao longo dos anos e alguns, inclusive, ofereciam soluções teóricas para o problema de baixa reutilização de *software* para múltiplas plataformas (DIWAKAR, 2012; BRADLEY, 2013).

Após a confirmação do problema de baixa reutilização, foi realizada uma pesquisa das ferramentas de desenvolvimento de software multiplataforma atuais, que se encaixavam no perfil do projeto, com o objetivo de identificar e testar em cada plataforma (*web*, *mobile* e *desktop*), as opções existentes a fim de formar uma opinião técnica mais clara em relação a cada uma para possíveis utilizações no projeto.

3.2 *Especificação da Arquitetura*

Após pesquisar as ferramentas atuais, teve início a etapa de especificação da arquitetura sugerida para a resolução do problema identificado. Durante essa etapa, foi produzido um documento contendo a arquitetura sugerida utilizando uma notação de camadas simples.

As camadas finais foram definidas utilizando os conceitos do DBC em conjunto com alguns itens da filosofia Unix sugeridos por (MCILROY, 1978) e são o resultado de diversos rascunhos produzidos e testados com as ferramentas selecionadas na etapa anterior. Estes testes, em sua maioria, focavam na facilidade em que um mesmo código pudesse ser distribuído dentro dos requisitos definidos pelo projeto e a medida em que foram executados, algumas arquiteturas sugeridas foram refutadas, quando se mostravam demasiado complexas, de difícil extensão ou se provarem incapazes de solucionar o problema especificado.

De uma forma geral, o processo de especificação foi interativo e utilizou uma metodologia ágil, o que segundo (PRESSMAN, 2011) oferece uma validação mais rápida.

Após os testes iniciais, as arquiteturas que se provaram teoricamente viáveis foram alinhadas com as ferramentas selecionadas na primeira etapa para que a implementação do protótipo fosse iniciada.

3.3 *Implementação do Protótipo*

O processo de codificação se baseou nos objetivos adquiridos nas etapas anteriores para realizar a implementação de um protótipo utilizando a arquitetura sugerida. Com base nos testes realizados, foi definido que para uma experiência mais unificada as ferramentas utilizariam a mesma linguagem base e possuiriam uma interface programática similar a fim de facilitar a integração entre elas. Uma explicação detalhada da implementação pode ser encontrada em uma seção específica deste artigo.

4 TECNOLOGIAS UTILIZADAS

Nessa seção serão apresentadas as tecnologias utilizadas.

4.1 AngularJS

AngularJS é um *framework JavaScript* de código aberto que auxilia a criação de *WebApps*, oferecendo alta performance em múltiplas plataformas e é utilizado por milhões de pessoas ao redor do mundo (GOOGLE, 2016). Seu foco inicial era facilitar o desenvolvimento

e teste de aplicação *single-page* oferecendo uma arquitetura MVC[1]/ MVVM[2] para o *front-end*. Apesar de ainda cumprir muito bem esse papel, vem evoluindo e se solidificando como um dos *frameworks* mais utilizados para desenvolvimento de aplicativos, sejam estes *WEB* ou não (GOOGLE, 2016). É mantido principalmente por uma equipe do Google e conta com o apoio da comunidade *open-source*. Teve sua primeira versão lançada em Outubro de 2010 e sua próxima evolução, denominada *Angular* 2, a que iremos utilizar para a implementação desse artigo, se encontra em Beta.

Com a evolução das tecnologias disponíveis juntamente com a crescente complexidade, inerente de *WebApps* modernos, o *framework* adotou em sua nova versão uma forma mais opinativa e declarativa de desenvolvimento baseada em componentes reutilizáveis. Linguagens com mais recursos e de alto nível, com tipagem mais restritiva como *TypeScript* e *Dart*, facilitam a identificação de erros antes de transpilar o código para *JavaScript* e são consideradas alguns dos pontos mais marcantes e *disruptivos* do *Angular* 2.

4.2 Apache Cordova

O *Apache Cordova* foi criado inicialmente por integrantes da empresa *Nitobi Sofware* em um evento chamado *iPhoneDevCamp* que ocorreu na cidade de São Francisco em 2006. Na época o projeto se chamava *PhoneGap* e foi comprado pela *Adobe Systems* em 2011 (ADOBE, 2011), um ano após a *Apple* ter confirmado que o *framework* havia sido aprovado como uma forma válida de desenvolvimento de aplicativos para sua loja, a *Apple Store*. De acordo com o documento de aquisição (ADOBE, 2011), após a compra, a *Adobe* liberou o código fonte para a fundação *Apache*, porém manteve o nome *PhoneGap* como proprietário, a *Apache* então renomeou o projeto como *Apache Cordova*.

O *Cordova* possibilita que desenvolvedores construam aplicações para dispositivo móveis ao empacotar os arquivos HTML, CSS e *JavaScript* em um *container* nativo. Este *container* possui acesso a diversos recursos do dispositivo em que está sendo executado como por exemplo bibliotecas do sistema operacional e recursos de hardware e expõe uma API em *JavaScript* que pode ser consumida pela própria aplicação para acessar esses recursos, não necessitando implementações diferenciadas para plataformas específicas por parte do produto final, ficando isso a cargo da própria biblioteca de *plugins* nativos do *Cordova* (DIWAKAR, 2012).

Por utilizar esta abordagem de empacotamento, os aplicativos criados, apesar de serem distribuídos como código nativo, são considerados híbridos, pois toda a renderização da interface de apresentação é realizada em uma *WebView* (um processo similar ao executado por navegadores para a exibição de páginas *WEB*) e não com componentes nativos da plataforma. Na mesma direção, os aplicativos não são considerados *WebApps*, como um site tradicional, pois além do modo de distribuição diferenciado, possuem acesso as APIs do dispositivo, normalmente bloqueadas para aplicações não-nativas. Essa diferenciação é importante pois possui

[1]MVC - ***M**odel **V**iew **C**ontroller*
[2]MVVM - ***M**odel **V**iew **V**iew **M**odel*

Tabela 1 – Exemplo de recursos disponíveis para algumas plataformas no *Apache Cordova*

	Android	Blackberry 10	iOS	Windows (8.1, 10)
Acelerômetro	SIM	SIM	SIM	SIM
Estado da Bateria	SIM	SIM	NÃO	SIM
Câmera	SIM	SIM	SIM	SIM
Captura de Mídia	SIM	SIM	SIM	SIM
Compasso	SIM	SIM	SIM	SIM
Conexão de Rede	SIM	SIM	SIM	SIM
Acesso aos Contatos	SIM	SIM	SIM	PARCIAL
Versão do Dispositivo	SIM	SIM	SIM	SIM
Geolocalização	SIM	SIM	SIM	SIM
Globalização	SIM	SIM	SIM	SIM
Browser InApp	SIM	SIM	SIM	PARCIAL
Notificações	SIM	SIM	SIM	SIM
Vibração	SIM	SIM	SIM	PARCIAL

um impacto significante na performance e fluidez de alguns aplicativos desenvolvidos nesse formato híbrido, tendo em vista que ao utilizar ferramentas web para renderizar a interface, lentidões e perdas de performance podem ser percebidas em comparação com aplicativos totalmente nativos (DIWAKAR, 2012).

De acordo com (LYNCH, 2014), um dos criadores do *Ionic Framework*, um ponto importante e muito contestado ao utilizar a abordagem de aplicações híbridas com o *Apache Cordova* é a disponibilidade de *plugins* para acessar os recursos nativos. Ao longo da evolução da ferramenta, muitos *plugins* foram criados visando oferecer um acesso transparente as funcionalidades em todas as plataformas, isso faz com que o *Apache Cordova*, ainda de acordo com o autor, seja capaz de realizar a interface com praticamente todo tipo de recurso que o desenvolver necessita na aplicação móvel. A tabela abaixo ilustra bem a grande disponibilidade e variedade de *plugins* e o suporte a cada um nas principais plataformas utilizadas com a ferramenta.

Considerando os diversos benefícios oriundos da utilização do *Apache Cordova*, diversas ferramentas foram construídas o utilizando como base, oferecendo uma camada adicional de abstração e produtividade. *Ionic Framweork* e *Telerik Platform* são alguns exemplos de ferramentas já difundidas no mercado. No decorrer desse artigo, o *Apache Cordova* será utilizado indiretamente para facilitar o acesso aos recursos nativos por meio do *Ionic Framework*.

4.3 Ionic Framework

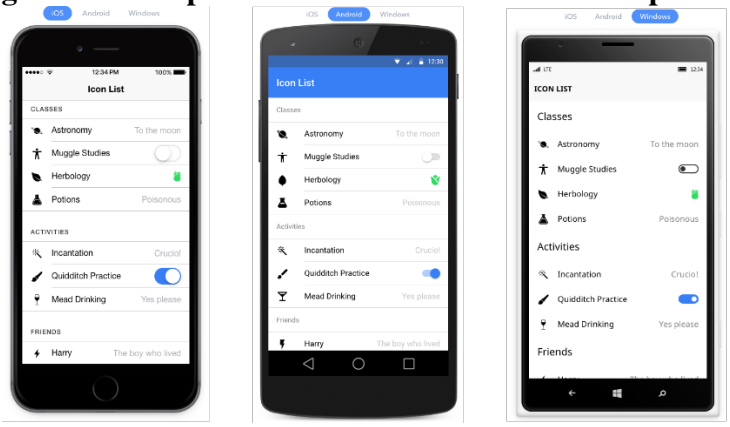
Ionic é um *framework JavaScript* de código aberto. Criado no final de 2012, é hoje a tecnologia multiplataforma mais popular para aplicativos móveis, tendo sido utilizado na criação de 1.3 milhões de aplicativos apenas em 2015 (LYNCH, 2016). De acordo com (BRADLEY, 2013), um dos criadores da ferramenta, o foco do *Ionic Framework* é aprimorar a interface do usuário ao oferecer uma série de bibliotecas com elementos HTML, CSS e *JavaScript* pré-

compilados e otimizados para dispositivos móveis. A ferramenta utiliza todas as vantagens do *Apache Cordova* e tenta suprir a necessidade dos aplicativos possuírem elementos de interface similares aos que os usuários já estão acostumados em cada plataforma. Ainda nas palavras do autor, o *framework* possui outros objetivos como promover padrões de design recomendados e documentar melhores práticas.

Os projetos criados com o *framework* utilizam, por padrão, o *AngularJS* visando facilitar o desenvolvimento de aplicações mais robustas. Diferentemente do primeiro, que oferece uma extensa gama de funcionalidades para diferentes necessidades, o *Ionic Framework* foca em oferecer componentes de interface e interação com o usuário para que a experiência de uso respeite o alto padrão de qualidade que os usuários esperam de aplicativos nativos atualmente (BRADLEY, 2013). Algumas de suas funções são a renderização otimizada de listas, criação de formulários com componentes nativos, customização e padronização dos elementos para a plataforma em que o aplicativo esteja executando, transição entre páginas e identificação de recursos por meio de API's simplificadas.

Um exemplo de como alguns componentes oferecidos pelo *framework* se adaptam para plataformas diferentes é ilustrado na Figura 1.

Figura 1 – Comparativo de telas em diferentes dispositivos



Fonte: (PRESSMAN, 2011)

4.4 Electron

Electron é um *framework JavaScript* de código aberto que possibilita o empacotamente de *websites* em aplicações nativas ao utilizar o motor de renderização *Chromium* em conjunto com outras tecnologias. Apesar de ser uma tecnologia relativamente nova, vem se concretizando no mercado e recebeu no ano de 2015, 1.2 milhões de downloads (GITHUB, 2016). O *framework* possui opções para gerar aplicativos nativos para os sistemas Mac, Windows e Linux de forma transparente para o usuário final e sem que o desenvolvedor necessite implementar por si só atualizações automáticas, menus nativos, geração de instaladores e outras funcionalidades que, normalmente, são extenuantes em um ciclo de desenvolvimento *desktop* tradicional. (GITHUB, 2016)

4.5 WebSockets

De acordo com a especificação da (GROUP, 2016) a interface de *WebSockets* foi introduzida para permitir que aplicações *WEB* mantenham comunicações bidirecionais e persistentes com processos sendo executados em servidores remotos.

Na arquitetura cliente-servidor tradicional, no qual a *WEB* foi construída, as conexões utilizam um padrão HTTP multiplexado, onde é de responsabilidade do cliente iniciar a conexão e requisitar dados, enquanto que a responsabilidade do servidor é atender essas requisições e retornar os dados solicitados. Esse paradigma foi, durante muitos anos, a única forma no qual as aplicações web conseguiam transmitir dados entre a aplicação cliente e o servidor. Com a especificação da interface de *WebSockets*, é possível que conexões persistentes sejam criadas conectando o cliente e servidor, possibilitando que ambas as partes iniciem a transmissão de dados a qualquer momento.

De acordo com a RFC 6455 (IEEE, 1990) a comunicação de *WebSockets* se inicia com um *handshake* e da mesma forma que o paradigma anterior, o cliente envia uma requisição HTTP com a única adição de um cabeçalho adicional do tipo *Upgrade*. Este cabeçalho informa ao servidor que o cliente deseja estabelecer uma conexão do tipo *WebSocket* e caso a requisição seja aceita e o servidor ofereça suporte a esse tipo de protocolo, a conexão bidirecional é estabelecida e ambos os lados podem iniciar a transmissão de dados , que são transmitidos por meio de mensagens baseadas em *frames*.

5 IMPLEMENTAÇÃO

Nessa seção será detalhada a arquitetura e as ferramentas utilizadas na implementação prática do protótipo.

5.1 *Arquitetura*

A arquitetura foi divida em três camadas principais, a fim de desacoplar e separar o conceito e responsabilidade de cada uma. São elas:

1. **Componentes:** Definição e Implementação dos componentes
2. **Aplicação:** Integração dos componentes
3. ***Deploy*:** Empacotamento do projeto para distribuição

Essas camadas funcionam de forma unidirecional, sempre passando o resultado da anterior para a próxima. Seguindo esse conceito, os componentes definidos e implementados

na camada de Componente são eventualmente integrados na camada de Aplicação e por fim empacotados para distribuição em plataformas específicas por meio da camada de *Deploy*.

Com essa abordagem o projeto pôde ser desenvolvido de forma modular, pois cada camada não precisa se preocupar com a responsabilidade das demais e por serem implementadas de forma separada cada uma pode ser substituída por uma implementação diferente sem que o fluxo sofra consequências negativas.

5.1.1 Camada de Componentes

O papel desta camada é a definição e implementação de componentes reutilizáveis. Os componentes definidos nessa etapa são encapsulados e planejados de forma a serem atômicos, modulares e independentes dos demais. Ao utilizar estas definições as implementações podem ser compartilhadas de forma mais simples e provêm maior facilidade na sua reutilização.

Dois grupos de componentes principais: apresentação e composição, foram definidos visando uma melhor separação de conceitos seguindo a abordagem sugerida por(ABRAMOV, 2015). Os componentes de apresentação são responsáveis por como os dados são apresentados e possibilitam um maior nível de reutilização por não salvarem nenhum estado próprio. Já os componentes de composição se preocupam em definir e implementar como as ações são tratadas e os dados se comportam internamente.

Durante a implementação os componentes foram codificados utilizando *TypeScript*, um *superset* da linguagem *JavaScript* que oferece suporte a interfaces, tipagem explícita, *generics* e outras funcionalidades não suportadas pela versão atual do *JavaScript* (ES6).

Ao longo desse processo, foi possível perceber os benefícios das tecnologias escolhidas. O *TypeScript* permitiu o uso de *decorators*, o que fez com que a codificação ficasse mais simples com a utilização de metadados para configuração dos componentes. A tipagem explícita também se provou útil ao possibilitar inspeções do código sem a necessidade de execução do interpretador de *JavaScript* para identificação de erros.

A separação do código em módulos permitiu que a reutilização se tornasse mais clara pois cada componente pôde ser separado e codificado de forma independente, como a arquitetura sugere. Tal prática se provou útil quando a aplicação começou a crescer e novos componentes foram adicionados.

A utilização de conceitos de programação funcional também foi essencial para a codificação eficiente desta camada. A minimização de efeitos colaterais com a ampla utilização de funções puras, que utilizam o conceito de imutabilidade de parâmetros e possuem comportamento previsíveis, permitiu que os componentes implementados possuíssem escopos mapeados, fazendo com que sua reutilização se tornasse mais eficaz ao encapsular seu funcionamento interno.

5.1.2 Camada de Aplicação

Esta camada é responsável por integrar os componentes implementados na anterior de tal maneira que o produto final possua uma árvore de componentes interconectados que possam representar a aplicação final. Essa proposta é a recomendação oficial da biblioteca utilizada, *AngularJS* e também é embasada no paradigma de desenvolvimento baseado em componentes (PRESSMAN, 2011).

Além da integração dos componentes de forma estrutural, esta camada também é responsável pelo tratamento de mensagens externas e o encaminhamento de forma normalizada desses dados para outras partes da aplicação por meio do controle de estados, que é realizado a fim de gerar comportamentos previsíveis e que possibilitem traçar uma linha do tempo das mudanças realizadas.

De uma forma geral, esta camada foca na composição dos componentes e a comunicação entre eles, por meio do controle de estados.Com essa abordagem os componentes podem ser implementados na camada anterior sem a necessidade de possuírem uma definição do contexto da aplicação final. Isso permite que uma extensa gama de componentes possa existir e que o desenvolvimento da aplicação em si seja apenas a combinação destes de uma forma funcional, o que, como já foi dito, é um dos princípios do DBC.

Durante a codificação alguns pontos foram repensados pois por utilizar *WebSockets* para comunicação externa, a aplicação necessita ser capaz de tratar diversos eventos assíncronos. Isso fez com que uma estrutura de estados utilizando o padrão de projeto *Publish/Subscribe* fosse implementada, porém foi detectado que muitas vezes os estados de variáveis independentes em partes distintas da aplicação não estavam sendo mantidos em sincronia. Tal fator, em adição à necessidade constante do desenvolvedor final sempre ser responsável por adicionar tratadores de eventos a fim de reagir a tais mudanças, fez com que esse padrão de projeto fosse substituído por uma implementação baseada nos conceitos sugeridos por (ABRAMOV, 2015), denominada *Redux*. Tal abordagem utiliza o conceito de ações explícitas, em que todas as mudanças de estados devem possuir uma ação correspondente, que por sua vez, deve ser emitida a fim de notificar as partes interessadas do evento ocorrido. Esta comunicação utiliza uma abordagem *top-down* de forma que todas as ações enviadas criam um novo estado, que é repassado para os componentes para que estes possam reagir da forma como foram definidos (ABRAMOV, 2016).

De forma similar a Camada de Componentes, a codificação foi realizada utilizando o *TypeScript*, que facilitou a integração dos componentes por ser a mesma tecnologia em que foram implementados. A composição dos componentes foi realizada utilizando uma aplicação iniciada com o *template* do *Ionic Framework* em conjunto com o *AngularJS*. O controle de estados, como já dito, foi implementado seguindo o padrão *Redux* e a parte visual se deu por conta da composição dos elementos CSS do *Ionic Framework* em conjunto com os componentes codificados na camada anterior.

5.1.3 Camada de Deploy

A camada final é a responsável por empacotar a solução implementada e incluir as funcionalidades necessárias para que a execução seja suportada em todas as plataformas necessárias.

Apesar de não ser opinativa na forma como o empacotamento deve ocorrer, a implementação que se provou mais viável foi a utilização de *scripts* pré-determinados com o intuito de possibilitar customizações específicas caso necessário. Os *scripts* devem ser separados por plataforma e implementados utilizando padrões da filosofia Unix (MCILROY, 1978). Apesar da separação, possuem o mesmo fluxo básico: A aplicação gerada na camada anterior é recebida e cada rotina deve tratar os arquivos de forma independente, gerando no final o artefato a ser executado na plataforma adequada.

Durante a codificação, foi possível identificar diversos pontos que se provaram inerentes para domínios de plataformas específicas. Em razão deste fato, durante a evolução do projeto esta camada foi a que mais sofreu alterações durante a codificação, pois diversas vezes foi necessário alterar os *scripts* a fim de padronizar o processo de empacotamento para cada plataforma, chegando até a quase se tornar um ponto que poderia refutar a arquitetura por falta de ferramentas que permitissem sua implementação prática. Isso fez com que uma pesquisa mais aprofundada em ferramentas de automação fosse realizada e após uma refatoração com base nas lições aprendidas foi possível implementar *scripts* expansíveis que possibilitaram o reuso de soluções e a padronização de tratamentos para casos específicos, fazendo com que a camada funcionasse de forma eficiente.

Diferentes implementações foram codificadas tendo em vista que algumas plataformas não aceitam a combinação de HTML + CSS + *JavaScript* para interpretação de forma nativa. Tal limitação foi identificada ao decorrer da implementação quando ao gerar o projeto final da aplicação para empacotamento, era necessário adicionar arquivos extras para que a plataforma conseguisse identificar a aplicação como válida. No caso do *Ionic Framework*, as plataformas suportadas na versão 2 não apresentaram problema, porém algumas vezes, aplicações executando na plataforma *Android* 4.4 ou inferior necessitaram ser embarcadas com um navegador próprio, a fim de viabilizar a identificação de diversos recursos nativos, aprimorar a performance e possibilitar um funcionamento transparente para o usuário final.

De forma similar, quando a aplicação foi inicialmente preparada para distribuição em computadores de mesa utilizando a ferramenta *Electron*, foi identificado a necessidade de adicionar arquivos e configurações extras a fim de permitir a renderização correta. O que foi identificado é que a ferramenta utiliza, fundamentalmente, duas *threads*, sendo uma responsável pelo código da aplicação e outra pela renderização do HTML. Esta separação, apesar de útil e performática, necessita um arquivo contendo codificação extra e configurações específicas para identificar momentos em que a aplicação necessita ser renderizada novamente, como por exemplo, ao receber eventos assíncronos via *WebSockets* quando fora de foco. Tal peculiaridade fez com que um *script* mais elaborado fosse desenvolvido a fim de permitir que as aplicações distribuídas por esse meio funcionassem da mesma forma que em aplicações sem uma separação

de *threads* similar.

5.2 Protótipo

Nessa seção contêm um detalhamento das principais funcionalidades e componentes do protótipo codificado.

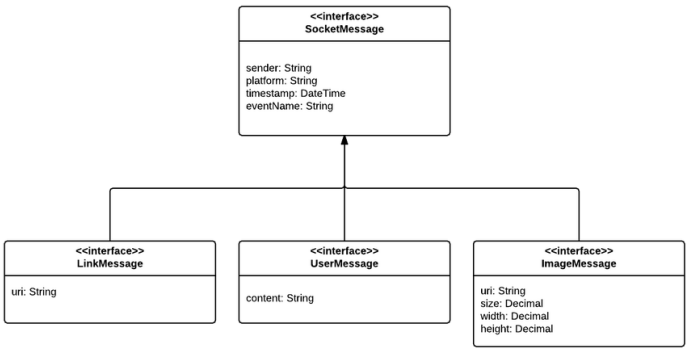
5.2.1 Principais funcionalidades

A aplicação construída durante a implementação desse artigo, oferece como principal funcionalidade o compartilhamento e exibição em tempo real de *links* entre os dispositivos conectados. Os *links* compartilhados irão apresentar uma breve descrição e uma imagem de exibição escolhida por inferência ao ser realizada uma breve análise dos metadados da página resultante. Cada *link* também irá possuir um indicador do tipo de plataforma de origem e qual usuário foi responsável por seu compartilhamento.

5.2.2 Estrutura de Comunicação

A comunicação entre os dispositivos conectados se dará por meio de eventos emitidos através de uma comunicação bidirecional com o *back-end* por meio de *WebSockets*. A figura 2 exemplifica as interfaces utilizadas para identificar as mensagens transmitidas, sendo que todas, quando recebidas, serão tratadas e despachadas para os componentes responsáveis utilizando o controle de estado da aplicação.

Figura 2 – Diagrama com as interfaces das mensagens transmitidas via *WebSockets* na aplicação



Fonte: (PRESSMAN, 2011)

5.2.3 Principais Componentes

Esta seção tem por objetivo explicar de forma mais detalhada como se exerceu a codificação dos componentes desenvolvidos para o protótipo.

De acordo com os conceitos definidos pela Camada de Componentes, ocorreu a separação de todas as funcionalidades da aplicação em componentes reutilizáveis com o objetivo de realizar a composição e integração da árvore de componentes que representa a aplicação final.

Durante a codificação, estes componentes foram separados em pastas distintas que contêm, no mínimo, um arquivo **.html** para definição da estrutura da página utilizando *tags* HTML e componentes próprios com a linguagem de *template* do *AngularJS*, um arquivo **.scss** que é utilizado pelo pré-processados de CSS, *Sass*, para a definição da folha de estilo da página e um arquivo **.ts** onde a classe que implementa a página é definida utilizando o *TypeScript*. Essa separação por pastas permite uma abordagem modular e faz com que todo o código e folhas de estilo sejam tratados como privados, ou seja, não afetam as outras partes da aplicação, o que normalmente não acontece em aplicações *WEB* tradicionais.

Com a utilização do *Ionic Framework*, foram desenvolvidos alguns componentes que representam as páginas da aplicação, que podem ser identificados no código por meio do *decorator* ***@Page*** e tem por objetivo definir a composição de uma página da aplicação. Alguns exemplos: *HomePage, LoginPage e CreatorModalPage*. A Figura 3 representa a codificação necessária para a página responsável pelo *login* dos usuários no sistema. Como pode ser observado, a classe implementada não possui nenhuma lógica interna, sendo responsável apenas por importar os módulos necessários e inicializar o formulário que será, eventualmente, responsável pela estratégia de *login*.

Figura 3 – Exemplo de código da página de Login

```
import {Page, NavController, Storage, SqlStorage} from 'ionic-angular';
import {FormBuilder, Validators, Control, ControlGroup} from "angular2/common";
import {SocketIO} from "../../providers/socket-io/socket-io";
import {HomePage} from "../home/home";
import {TabsPage} from "../tabs/tabs";
import {Store} from '@ngrx/store';
import {IApplicationState, JOIN_ROOM} from '../../providers/reducers/reducers';

@Page({
  templateUrl: 'build/pages/login/login.html',
})
export class LoginPage {
  private loginForm: ControlGroup;
  private username: string;
  private room: string;

  private storage = new Storage(SqlStorage);

  constructor(public nav: NavController,
              private form: FormBuilder,
              private socket: SocketIO,
              private store: Store<IApplicationState>
  ) {

    this.fetchValuesFromLocalStorage();

    this.loginForm = this.form.group({
      username: ['', Validators.required],
      room: ['', Validators.required]
    });
  }
}
```

Fonte: (PRESSMAN, 2011)

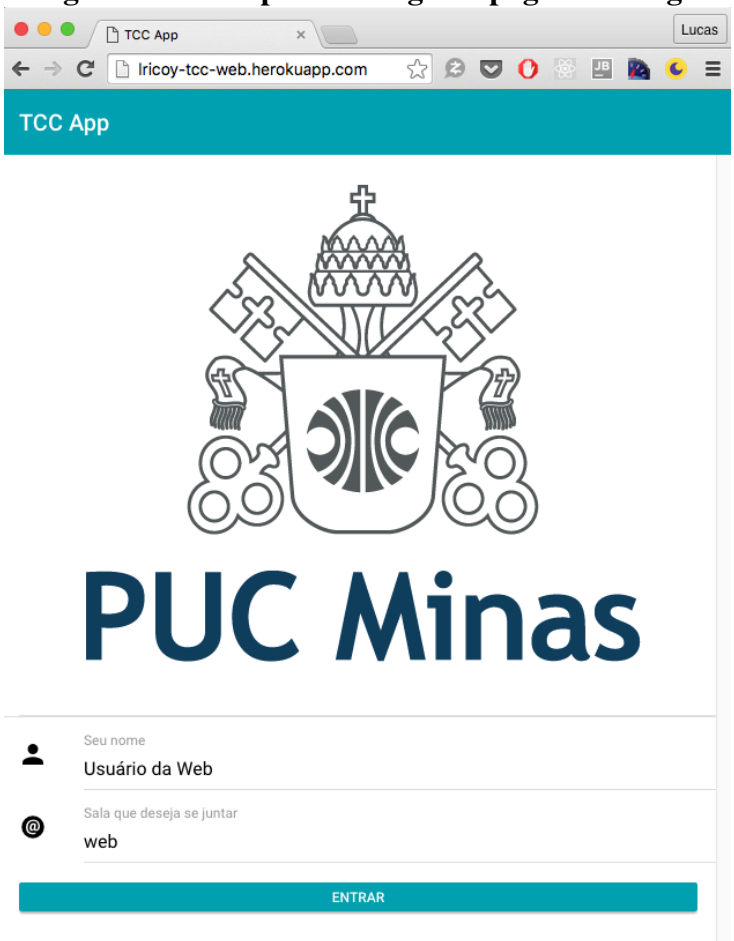
Em um nível abaixo dos componentes responsáveis pela definição das páginas da aplicação, foram implementados diversos componentes para atender os requisitos necessários da aplicação. Alguns exemplos são: *UrlPreview, UrlContentPreview, UrlImagePreview, UrlMessageForm, UrlList, WelcomeMessageCard, NewMessageButton*. Dentre estes componentes, é possível identificar alguns que, diferente dos citados anteriormente, possuem uma lógica interna, que seguindo os conceitos sugeridos por (ABRAMOV, 2015), são os chamados componentes de composição e apesar de possuírem uma importância singular para o projeto, estão presente em menor quantidade pois foi possível perceber durante a implementação do projeto, que os componentes menores eram reutilizados em maior quantidade e que a composição de componentes se provou mais útil do que a extensão dos mesmos. Com isso, a maioria dos componentes implementados, se encaixam na categoria de apresentação, de forma que apenas uma minoria possui alguma lógica interna.

A figura **??** representa uma árvore com os principais componentes implementados na aplicação protótipo.

5.2.4 Telas do Protótipo

A figura 4 representa o comportamento da aplicação final em diferentes plataformas.

Figura 4 – Exemplo de código da página de Login



Fonte: (PRESSMAN, 2011)

6 CONCLUSÃO

Com o desenvolvimento bem sucedido do protótipo utilizando a arquitetura sugerida, é possível concluir que o desenvolvimento de aplicações híbridas multiplataformas utilizando o mesmo código compartilhado é viável e possui capacidade para ser utilizada em diversas aplicações.

Com a utilização das ferramentas existentes nas etapas aqui registradas foi possível desenvolver uma solução expansível que possuíse uma certa gama de funcionalidades úteis e não totalmente simplórias de uma forma satisfatória. O nível de customização e qualidade final de renderização e reutilização também foram aceitáveis, indicando que a arquitetura utilizada possa servir de base para outras aplicações em variados contextos.

De uma forma geral, a arquitetura, apesar de expansível e modular, só foi testada com uma certa gama de ferramentas específicas em cada camada, ficando a substituição de cada ferramenta um trabalho futuro a fim de aprimorar o desacoplamento e diminuir as dependências da arquitetura em ferramentas específicas, focando mais em conceitos de responsabilidades do que implementações específicas.

Referências

ABRAMOV, Dan. **Presentational and Container Components**. 2015. Disponível em: <https://goo.gl/N9FMUV>. Acesso em: 25 de mar. 2016.

ABRAMOV, Dan. **Dcoumentação Redux**. 2016. Disponível em: <http://redux.js.org/>. Acesso em: 25 de mar. 2016.

ADOBE. **Adobe Announces Agreement to Acquire Nitobi, Creator of PhoneGap**. 2011. Disponível em: <http://www.adobe.com/aboutadobe/pressroom/pressreleases/201110/AdobeAcquiresNitob
Acesso em: 15 de mar. 2016.

BRADLEY, Adam. **Where does the Ionic Framework fit in?** 2013. Disponível em: <http://blog.ionic.io/where-does-the-ionic-framework-fit-in/>. Acesso em: 03 de mar. 2016.

DIWAKAR, Sapan. **Titanium vs Phonegap vs Native application development**. 2012. Disponível em: <http://www.sapandiwakar.in/api-research-study-iphone-and-android-applications/>. Acesso em: 03 de mar. 2016.

GITHUB. **Electron Framework Website**. 2016. Disponível em: <http://electron.atom.io/>. Acesso em: 2 de jun. 2016.

GOOGLE. **AngularJS Website**. 2016. Disponível em: <http://angular.io/> Acesso em: 2 de jun. 2016.

GROUP, WHATWG Web Hypertext Application Technology Working. **HTML Living Standard**. 2016. Disponível em: <https://html.spec.whatwg.org/multipage/comms.html>. Acesso em: 10 de mai. 2016.

IEEE. **IEEE Standard Glossary of Software Engineering Terminology**. [S.l.], 1990.

LYNCH, Max. **The Last Word on Cordova and PhoneGap**. 2014. Disponível em: <http://blog.ionic.io/what-is-cordova-phonegap/>. Acesso em: 03 de mar. 2016.

LYNCH, Max. **How 2015 Went for Ionic**. 2016. Disponível em: <http://blog.ionic.io/how-2015-went-for-ionic/>. Acesso em: 03 de mar. 2016.

MCILROY, Doug. **Basics of the Unix Philosophy**. 1978. Disponível em: <http://www.faqs.org/docs/artu/ch01s06.html>. Acesso em: 25 de mar. 2016.

PRESSMAN, Roger S. **Engenharia de Software: uma abordagem profissional**. 7. ed. [S.l.: s.n.], 2011.

SOMMERVILLE, Ian. **Engenharia de Software**. 9. ed. [S.l.: s.n.], 2011.

VALE, Tassio et al. Twenty-eight years of component-based software engineering. **The Journal of Systems and Software**, v. 111, p. 128–148, 2016.