Uma Arquitetura de Referência Baseada em Plugins para Sistemas de Informação Mobile

Enoque Joseneas*
Instituto Federal da Bahia
Rua Emídio dos Santos, S/N, Barbalho
Salvador-Ba, Brasil
enoquejoseneas@ifba.edu.br

Sandro Andrade†
Instituto Federal da Bahia
Rua Emídio dos Santos, S/N, Barbalho
Salvador-Ba, Brasil
sandroandrade@ifba.edu.br

RESUMO

O desenvolvimento de aplicativos móveis trouxe uma série de desafios para a ciência da computação. Com limitações de recursos como a bateria, armazenamento e memória, o desenvolvimento de software para dispositivos móveis impõe requisitos não-funcionais importantes a serem considerados no projeto de aplicativos. Com a popularização da Internet e a expansão das redes móveis, os aplicativos tornaram-se populares e projetá-los de forma fácil com componentes de alto nível, baixo acoplamento e bom desempenho não é uma tarefa trivial. Este trabalho apresenta uma arquitetura de referência para o desenvolvimento de aplicativos móveis orientado a plugins no contexto de sistemas de informação, proporcionando baixo acoplamento entre os componentes e escalabilidade de recursos através de plugins. A arquitetura dispõe de componentes de alto nível reusáveis para acesso a rede, persistência de dados, notificações do aplicativo e comunicação entre objetos através de eventos. A avaliação realizada neste trabalho, foi baseada em métricas extraídas a partir do código de duas versões de um aplicativo, sendo uma versão baseada nesta arquitetura.

Plavras-chave

Aplicativos Móveis; Arquitetura de Software; Qt; Sistemas de Informação.

1. INTRODUÇÃO

Os dispositivos móveis apresentam a cada dia novas oportunidades e desafios para as tecnologias da informação, tais como o acesso ubíquo, a portabilidade, a democratização do acesso à informação e novas oportunidades de negócio. Com a expansão da Internet e o grande volume de dados compartilhados nas redes sociais e aplicativos de troca de mensagens, surgiram novos paradigmas (e.g. *Big Data*, *Cloud Computing*, *NoSQL*, etc.), novas tecnologias como o *Push Notification* e também novas oportunidades de trabalho e profissões (e.g. O analista de dados, o desenvolvedor mobile, o *design UX*, etc.), além de pesquisas importantes na ciência da computação que abrange tanto hardware como software.

Os smartphones inovam a cada dia diversas áreas do conhecimento, tais como a engenharia elétrica, no projeto de baterias cada vez mais eficientes, o design, no projeto de interfaces cada vez mais intuitivas e influenciam diretamente na evolução da Internet e dos meios de comunicação como as redes móveis que expandem as áreas de cobertura para atender ao crescente número de aparelhos conectados. Os dispositivos móveis também permitem bons empreendimentos através dos aplicativos. Atualmente, o número de downloads cresce a cada dia na *App Store* e *Google Play*, demonstrando uma certa disponibilidade dos usuários de passarem cada vez mais tempo utilizando os aplicativos do que os próprios navegadores de Internet [30]. Através dos aplicativos, é possível monetizar e gerar receitas via marketing digital e desenvolver soluções para diversos segmentos, tais como o *e-commerce*, redes sociais e sistemas de informação.

O desenvolvimento de aplicativos apesar de contar com inúmeras ferramentas tais como as IDEs (Android Studio e Eclipse) e frameworks (*Ionic* e *PhoneGap*), ainda apresentam limitações, dentre elas, a falta de soluções arquiteturais de alto nível, ausência de componentes de *UI*[1] flexíveis e de alto nível. No Android por exemplo, para construir uma interface gráfica utiliza-se arquivos xml incorporados por objetos java. Outra limitação encontrada no desenvolvimento mobile, é a falta de suporte facilitado para comunicação *RESTful*, visto que os aplicativos móveis utilizam na maioria dos casos algum *web service*.

Este trabalho teve como objetivo o projeto, implementação e avaliação de uma arquitetura orientada a plugins e reutilizável para o desenvolvimento de sistemas de informação *mobile*. Dentre os benefícios desenvolvidos destaca-se uma arquitetura de plugins, que permite ao desenvolvedor implementar as funcionalidades do aplicativo com maior facilidade de extensão, manutenção e baixo acoplamento entre os componentes. O projeto desta arquitetura visa atender quatro requisitos funcionais, disponibilizando para cada um deles, componentes de alto nível para os plugins. Os requisitos são: acesso a rede para comunicação com serviços *RESTful*, persistência de dados local via *SQLITE*, notificações do sistema via *push* e local (partindo do próprio aplicativo) e um mecanismo de comunicação entre objetos através de eventos.

Para desenvolver este trabalho, foi utilizado o Qt, ele dispõe de APIs que facilitaram o desenvolvimento e o atendimento dos requisitos funcionais da arquitetura, além do QML, que é a linguagem implementada nos componentes reusáveis e deve ser utilizada na construção de objetos de *UI* pelos plugins.

Para avaliar a arquitetura proposta neste trabalho, foi desenvolvido duas versões de um aplicativo móvel, sendo uma versão baseada nesta arquitetura e a outra versão sem utilizá-la. Após finalizar o desenvolvimento das duas versões, foram extraídas algumas métricas dos dois modelos com o objetivo de destacar os benefícios de utilizar a arquitetura proposta neste trabalho. As métricas definidas para avaliação foram: número de linhas de código implementado e densidade de bugs encontrado em cada versão. Para complementar a avaliação, foi extraído uma lista de outros aspectos destacando detalhes observados do desenvolvimento de cada versão do aplicativo.

Este trabalho está organizado como segue. A Seção 2 apresenta

*Graduando em Análise e Desenvolvimento de Sistemas
†Prof. Doutor em Ciência da Computação

[1] *User Interface* ou interface do usuário

arquiteturas e tecnologias para sistemas mobile. Em seguida, na Seção 3, serão apresentados os trabalhos relacionados. A Seção 4 detalha o projeto da arquitetura. A Seção 5 apresenta a avaliação realizada e a Seção 6 descreve as conclusões obtidas seguido das limitações encontradas e os possíveis trabalhos futuros.

2. ARQUITETURAS E TECNOLOGIAS PARA SISTEMAS MOBILE

Esta Seção, apresenta as principais referências que contextualizam este trabalho. A Subseção 2.1 apresenta o Qt e o QML. A Subseção 2.2 descreve sobre Arquitetura de Software. A Subseção 2.3 apresenta Arquitetura de Referência. A Subseção 2.4 resume Sistemas de Informação. A Subseção 2.5 apresenta Arquiteturas de Aplicativos Móveis. A Subseção 2.6 detalha Projetos de Aplicativos Móveis. A Subseção 2.7 discute sobre Desenvolvimento Orientado a Componentes. Para finalizar, a Subseção 2.8 faz uma breve revisão sobre as tecnologias suportadas nesta arquitetura: *web services*, o estilo arquitetural *RESTful*, o *Push Notification* e o JSON.

2.1 O Qt e o QML

O Qt é um *toolkit cross-platform* para desenvolvimento de aplicações com interface gráfica. O Qt é muito mais que um SDK, ele é uma estratégia de tecnologia que permite ao desenvolvedor, de forma rápida e econômica, projetar, desenvolver, implementar e manter uma aplicação multiplataforma oferecendo uma experiência de usuário perfeita em todos os dispositivos [9]. No entanto, programas sem interface gráfica podem ser desenvolvidos, como ferramentas de linha de comando e consoles para servidores [2].

O Qt possui um amplo apoio à internacionalização e outros recursos, tais como o acesso a um banco de dados *SQL*, *parsing* de XML, *parsing* de JSON, gerenciamento de *threads* e suporte a rede [8]. O Qt dispõe ainda de uma linguagem declarativa e interpretada para construir interface gráfica, o QML. O QML é uma especificação de interface de usuário e linguagem de programação que permite a desenvolvedores e designers criar aplicativos de alta performance, fluidamente animados e visualmente atraentes. O QML oferece uma sintaxe JSON, altamente legível e declarativa, com suporte para expressões imperativas JavaScript combinadas com ligações de propriedades dinâmicas [1].

2.2 Arquitetura de Software

De acordo com a definição clássica proposta por *Shaw e Garlan* [21], arquitetura de software define o que é sistema em termos de componentes computacionais e os relacionamentos entre eles, os padrões que guiam suas composições e restrições. Arquitetura de software pode ser compreendida como uma especificação abstrata do funcionamento de um sistema e permite especificar, visualizar e documentar a estrutura e o funcionamento de um programa independente da linguagem de programação na qual ele será implementado [4].

Os softwares estão em constante evolução e sofrem mudanças periodicamente, que ocorrem por necessidade de corrigir *bugs* ou de adicionar novas funcionalidades. As mudanças ocorridas no processo de evolução de um software podem torná-lo instável e predisposto a defeitos, além de causar atraso na entrega e custos acima do estimado. Porém, um software que é projetado orientado a arquitetura, possibilita os seguintes benefícios:

- Melhor escalabilidade;
- Maior controle intelectual;
- Menor impacto causado pelas mudanças;
- Melhor atendimento aos requisitos não-funcionais;
- Maior agilidade na manutenção do código;
- Padronização de comunicação entre os componentes e;
- Suporte a reuso de componentes e maior controle dos mesmos.

O desenvolvimento de software envolve muitas partes (e.g., levantamento de requisitos, modelagem, implementação, testes, refatoração e etc.). O objetivo de um software é o que motiva a sua construção, e o que fomenta todas as partes que envolve o seu desenvolvimento é o problema que ele tenta solucionar no mundo real e parte do mérito de uma boa solução é devido ao uso de uma boa arquitetura.

Neste trabalho, arquitetura de software pode ser compreendida nas decisões de implementação, nas restrições impostas pelo uso dos recursos disponibilizados e dos componentes reusáveis, além dos estilos arquiteturais provenientes das APIs utilizadas, dentre elas, o *Event-Based*, mecanismo de comunicação baseado em eventos provido pelo Qt. Outro aspecto arquitetural deste trabalho é um estilo de desenvolvimento orientado a plugins. Os plugins devem representar os componentes específicos e as funcionalidades de cada projeto baseado nesta arquitetura, eles são independentes entre si e proporcionam baixo acoplamento entre as funcionalidades do sistema.

2.3 Arquitetura de Referência

Uma arquitetura de referência consiste em uma forma de apresentar um padrão genérico para um projeto [32]. Com base nessa arquitetura, o desenvolvedor projeta, desenvolve e configura uma aplicação prototipando-a por meio de componentes reutilizáveis [32]. Para compor uma arquitetura de referência é necessário apresentar os tipos dos elementos envolvidos, como eles interagem e o mapeamento das funcionalidades para estes elementos [16]. De maneira geral, uma arquitetura de referência deve abordar os requisitos para o desenvolvimento de soluções, guiado pelo modelo de referência e por um estilo arquitetural de forma a atender as necessidades do projeto [10].

A concepção de uma arquitetura de referência pode ser entendida neste trabalho como uma forma de disponibilizar um padrão genérico para o desenvolvimento de novos aplicativos no contexto de sistemas de informação, partindo de quatro requisitos funcionais que serão apresentados em uma seção mais adiante.

2.4 Sistemas de Informação

Um sistema de informação pode ser definido como um conjunto de componentes inter-relacionados trabalhando juntos para coletar, recuperar, processar, armazenar e distribuir informações com a finalidade de facilitar o planejamento, o controle, a coordenação, a análise e o processo decisório em organizações [20].

O escopo desta arquitetura está focado em sistemas de informação, porém, não está limitado somente a este tipo de software. Os requisitos de um aplicativo baseado nesta arquitetura, devem ser implementados através de plugins que podem se comunicar, persistir dados e conectar à Internet de forma facilitada, usando APIs de alto nível. No entanto, os requisitos funcionais atendidos por esta arquitetura são muito comuns em sistemas de informação e este projeto tem o objetivo de facilitar a construção de aplicativos para este segmento.

2.5 Arquiteturas de Aplicativos Móveis

Arquitetura para aplicações móveis abrange quatro camadas: Interação Humana-Computador, Aplicação Móvel, *Middleware* e *Enterprise Backend* [24]. Neste trabalho, a arquitetura foi concentrada apenas nas camadas de interação, aplicação e *middleware*.

2.5.1 Camada de Interação Humano-Computador

A camada de Interação Humano-Computador (mais conhecida como *IHC*, interface de usuário ou simplesmente *UI*) define os elementos de interação entre o usuário e os recursos do aplicativo. De forma abstrata, a camada de interface do usuário descreve o tipo de mídia suportada pelo aplicativo (por exemplo, texto, gráficos, imagens, vídeo ou som), os tipos de mecanismos de entrada (por exemplo, teclado alfa-numérico, ponteiros de caneta ou toques na tela) e os tipos de mecanismos de saída (por exemplo, uma notificação na bandeja do sistema, a tela, os alto-falantes ou algum tipo de *feedback* como vibrar o dispositivo) [24]. Um exemplo de um componente desta camada é o objeto *Image* do QML que corresponde ao carregamento e exibição de uma imagem na tela.

2.5.2 Camada de Aplicação

A camada de aplicação corresponde ao processamento de ações e eventos provenientes da camada de interação com o usuário, como por exemplo, escutando eventos de toque e realizando processamento em segundo plano. Esta camada, corresponde a componentes não visuais e interagem diretamente com a camada de *middleware*. Objetos da camada de aplicação podem por exemplo, gerenciar e controlar a criação de outros objetos. Um exemplo de objeto desta camada é o *Loader* do QML, ele cria objetos dinamicamente e emite um sinal quando o item recém criado estiver pronto.

2.5.3 Camada de Middleware

A camada de *middleware* intercala entre a camada de aplicação com a camada de *backend*. O objetivo dessa camada é fornecer de forma abstrata e genérica um meio de comunicação entre o modelo de dados da aplicação com a camada de *backend* [24]. Ela é também responsável por interagir com o meio de comunicação disponível no dispositivo abstraindo para a camada de aplicação qual foi a interface de hardware utilizada. Um exemplo de objeto que trabalha nessa camada é o *RequestHttp* disponibilizado nesta arquitetura, ele é resposável por realizar requisições HTTP ao *web service* de forma assíncrona, notificando o objeto da camada de aplicação quando a resposta for obtida.

2.5.4 Camada de Backend

A camada *backend* consiste de uma outra aplicação que responde pelas requisições do aplicativo através de uma rede via protocolo *HTTP*. Esta camada está associada ao *web service* ou serviço REST. O *web service* pode atender a diferentes requisições e dispositivos, além de abstrair para o cliente, a lógica de negócios referente ao armazenamento e processamento dos dados entregues como resposta das requisições. A implementação desta camada pode ser desenvolvida sobre uma outra arquitetura, além de implementar regras de negócio inerentes ao seu funcionamento e portanto, não será detalhada neste trabalho.

2.6 Projeto de Aplicativos Móveis

Um projeto é um esforço temporário empreendido para criar um produto, serviço ou resultado exclusivo. O termo temporário quer dizer que o projeto possui um ciclo de vida com início e final determinados [6]. O projeto termina quando seus objetivos forem alcançados ou quando existirem motivos para não continuá-lo [6].

Um aplicativo móvel ou aplicação móvel ou simplesmente *app*, é um sistema desenvolvido para ser instalado e executado em um dispositivo eletrônico portátil, como tablets e smartphones [7]. Um aplicativo móvel pode ser baixado diretamente no aparelho eletrônico, desde que o dispositivo possua conexão com a Internet. O mercado de dispositivos móveis é ramificado por diferentes fabricantes, o que inclui uma variação de plataformas de desenvolvimento, sistemas operacionais, versões do SO e configuração variada de hardware. Na construção de um aplicativo para dispositivo móvel, a implementação é um ponto muito importante, pois, além de representar a parte concreta dos requisitos funcionais do aplicativo também refletem diretamente nos requisitos não funcionais e consequentemente na qualidade do software e na satisfação do usuário.

O sucesso de aplicativos para dispositivos móveis vai além das medidas de desempenho, portabilidade e usabilidade tradicionais [19]. Os aplicativos devem estar em conformidade com a personalidade, preferências, objetivos, experiências e conhecimento de seus usuários [31]. Além disso, o contexto físico, social e virtual onde ocorrem as interações deve, sempre que possível, ser levado em consideração [22].

Torna-se evidente que são muitos requisitos a serem considerados em um projeto de um aplicativo móvel. O esforço dedicado para atender a todos os requisitos pode tornar o projeto enfadonho, além de exigir tempo e mão de obra. O processo de desenvolvimento pode ser otimizado através de ferramentas como *frameworks* ou uma arquitetura de software que disponha de componentes reutilizáveis e fácil extensibilidade através de plugins.

2.7 Desenvolvimento Orientado a Componentes

O desenvolvimento de software orientado a componentes é um paradigma da engenharia de software caracterizado pela composição de partes já existentes, ou desenvolvidas independentemente e que são integradas para atingir um objetivo final [14]. Construir novas soluções pela combinação de componentes desenvolvidos aumenta a qualidade e dá suporte ao rápido desenvolvimento, levando à diminuição do tempo de entrega do produto final ao mercado [14]. Os sistemas definidos através da composição de componentes permitem que sejam adicionadas, removidas e substituídas partes do sistema sem a necessidade de sua completa substituição. Com isso, o desenvolvimento baseado em componentes auxilia na manutenção do software, por permitir que o sistema seja atualizado através da integração de novos componentes ou atualização dos objetos já existentes [28].

O reuso de componentes é um recurso desta arquitetura, pois dispõe de onze componentes (visuais e não visuais) para auxiliar no desenvolvimento de novos aplicativos. Esses componentes são arquivos QML que suportam diferentes customizações através das propriedades disponibilizadas pelos objetos internos de cada componente. Os benefícios da componentização estão ligados a manutenibilidade, reúso, extensibilidade e escalabilidade [13].

2.8 Web Services, RESTful, Push Notification e JSON

Web Services constituem uma tecnologia emergente da Arquitetura Orientada a Serviços (SOA) [25]. Com a expansão da Internet e a necessidade de integração entre aplicações web, tornou-se necessário a centralização de informações para serem acessados por diferentes clientes. Para esse propósito, foi criada a tecnologia de *web services* [12].

RESTful é um estilo arquitetural para a construção de sistemas distribuídos [15]. O elemento fundamental da arquitetura *RESTful* é o *resource* ou recurso. Um recurso pode ser uma página web con-

tendo um documento estruturado, uma imagem ou até mesmo um vídeo. Para localizar os recursos envolvidos em uma interação entre os componentes da arquitetura *RESTful* é utilizado o chamado identificador de recurso ou *URI*. Com isso, um recurso pode ser representado através de diferentes formatos e o mais comum e utilizado é o *JSON*.

Push Notification é descrito por *Acer et al.* [11] como mensagens pequenas, usadas por aplicações de celular para informar aos usuários sobre novos eventos e atualizações. As notificações na maioria dos casos, estão associadas aos aplicativos instalados no dispositivo. O termo *push* indica que a mensagem parte do servidor para o dispositivo. Os principais provedores de notificações via *push* são o *Apple Push Notification Server* (APN) e o *Firebase* antigo *Google Cloud Messaging*.

JSON (*JavaScript Object Notation*) é um conjunto de chaves e valores, que podem ser interpretados por qualquer linguagem. Além de ser um formato de troca de dados largamente utilizado em serviços *RESTful*, é fácil de ser entendido e escrito pelos programadores. Estas propriedades fazem do JSON um objeto ideal para o intercâmbio de dados em aplicações web tal como o XML [17].

3. TRABALHOS RELACIONADOS

Nesta seção, serão apresentados os trabalhos relacionados com este projeto. Para cada trabalho relacionado, será descrito um resumo extraído do próprio artigo ou monografia e no final de cada subseção, será exibido uma tabela comparando as principais características deste projeto com o trabalho relacionado.

Arquitetura de Referência para o Desenvolvimento de Sistemas Colaborativos Móveis Baseados em Componentes [23].

A arquitetura de referência proposta, denominada CReAMA – *Component-Based Reference Architecture for Collaborative Mobile Applications*, teve como principal objetivo orientar o desenvolvimento de sistemas colaborativos móveis baseados em componentes para a plataforma Android. Sistemas desenvolvidos de acordo com essa arquitetura, devem dar suporte ao desenvolvimento de componentes e à criação de aplicações colaborativas por meio da composição desses componentes.

As aplicações e componentes são desenvolvidos para plataformas móveis, facilitando o uso de recursos inerentes a essas plataformas, tais como informações de sensores embarcados. Com base na arquitetura de referência, o desenvolvedor poderá ser guiado para criar componentes e compor novas aplicações seguindo os padrões estabelecidos. Por exemplo, será possível construir *toolkits* que forneçam componentes para um domínio específico. É importante ressaltar que a arquitetura foi definida considerando-se: aspectos da plataforma móvel, de sistemas colaborativos e da própria orientação a componentes. Com relação à plataforma móvel, optou-se por uma plataforma específica, visando-se a definição de uma arquitetura otimizada para as características da respectiva plataforma.

O trabalho proposto por Maison Melotti se relaciona com este trabalho pelo fato de terem objetivos semelhantes, que é propor uma arquitetura para facilitar o desenvolvimento de aplicativos móveis, permitindo o reuso facilitado de componentes. Apesar de estarem focados em domínios diferentes, os trabalhos se relacionam no atendimento de três requisitos funcionais: o cache de dados (persistência local); notificações do aplicativo (local e *push notification*) e acesso a rede.

MoCA: Arquitetura para o Desenvolvimento de Aplicações Sensíveis ao Contexto para Dispositivos Móveis [27].

MoCA (*Mobile Collaboration Architecture*) é uma arquitetura que oferece recursos para o desenvolvimento de aplicações distribuídas sensíveis ao contexto que envolvem usuários móveis. Esses recursos incluem um serviço para a coleta, armazenamento e distribuição de informações de contexto e um serviço de inferência de localização de dispositivos móveis. Além disso, a arquitetura provê APIs para o desenvolvimento de aplicações que interagem com estes serviços como consumidores de informações de contexto. Os serviços providos pela *MoCA* livram o programador da obrigação de implementar serviços específicos para a coleta e tratamento de contexto.

O conjunto de APIs oferecidas pela *MoCA* para desenvolvimento de aplicações compreende três grupos: as APIs de comunicação, que fornecem interfaces de comunicação síncrona e assíncrona (componentes de *UI* que utilizam eventos); as APIs principais que fornecem interfaces de comunicação com os serviços básicos da arquitetura; e as APIs opcionais que facilitam o desenvolvimento de aplicações baseadas na arquitetura cliente-servidor.

A relação deste trabalho com a arquitetura proposta em *MoCA* pode ser entendida pelo uso do estilo arquitetural *Event Based*, além provê APIs de alto nível para operações de rede, persistência de dados no dispositivo e notificações do aplicativo. No entando, *MoCa* foi construído para trabalhar com um servidor próprio, atendendo requisições específicas de seu domínio, enquanto que esta arquitetura propõe um modelo de comunicação cliente-servidor através de serviços *RESTful*.

Solução Multiplataforma para Smartphone Utilizando os Frameworks SenchaTouch e PhoneGap Integrado à web service Java [18].

O trabalho teve como objetivo principal, realizar análise e estudo sobre as tecnologias de desenvolvimento de aplicativos móveis multiplataforma, utilizando a junção dos frameworks *PhoneGap* e *Sencha Touch*. Os aplicativos desenvolvidos usando o *PhoneGap* são aplicações híbridas onde partes do aplicativo, principalmente a interface do usuário, a lógica da aplicação e a comunicação com um servidor, é baseado em HTML, Javascript e CSS. A outra parte, que se comunica com o sistema operacional do dispositivo é baseada no idioma nativo de cada plataforma, ou seja, *Java* no android e *Objective C* no iOS.

O estudo propôs uma modelagem facilitada de integração com outro sistema por meio de serviços web, através de uma aplicação *RESTFul* utilizando Java EE. Com a análise das ferramentas e tecnologias levantadas, pode-se concluir que o desenvolvimento de aplicativos utilizando os frameworks *PhoneGap* e *Sencha Touch* tem muitas vantagens. Uma delas é a facilidade de portar o aplicativo para qualquer plataforma móvel. O *PhoneGap* dispõe uma arquitetura *MVC* e diversos componentes para acesso a recursos do dispositivo, como câmera, acelerômetro e GPS através de objetos javascript. O *Sencha Touch* dispõe de objetos focado em UI, principalmente suporte a eventos de toque na tela.

O trabalho realizado por Jauri da Cruz Junior se relaciona com esta arquitetura como um estudo comparativo dos recursos provido pelo Qt com o *PhoneGap*. Identificou-se que o *PhoneGap* possui APIs para o *build* do aplicativo nas plataformas mobile e dispõe de uma API de alto nível em Javascript para o desenvolvedor utilizar os recursos do dispositivo independente da plataforma. Os recursos podem ser desde a câmera do aparelho até notificações do sistema. No entanto, o *PhoneGap* não possui componentes de interface prontos para serem adicionadas na aplicação, por isso no trabalho de Jauri foi utilizado outro framework para composição das telas.

Avaliação do Framework Xamarin.Forms Para Desenvolvimento de Aplicativos Móveis Multiplataforma [26].

Como consequência do aumento de dispositivos móveis, foram criados diferentes sistemas operacionais para gerenciar tais dispo-

sitivos. Na tentativa de atingir o maior número de usuários, os desenvolvedores de aplicativos móveis têm que compreender as características de cada plataforma, dentre elas a linguagem, bibliotecas, IDEs e APIs. Disponibilizar um aplicativo para mais de uma plataforma ou dispositivo móvel requer tempo, experiência e pode acabar custando mais do que o estimado. A plataforma *Xamarin* permite que desenvolvedores criem aplicativos multiplataforma nativos usando o mesmo código base, apenas implementando uma interface para cada sistema alvo. Como forma de reduzir ainda mais a necessidade de desenvolvimento específico de interface de usuário, um novo modelo de framework para desenvolvimento surgiu no mercado – o *Xamarin.Forms* – e assegura que é possível resolver esses obstáculos com apenas uma implementação. O propósito geral deste trabalho é explorar o framework multiplataforma *Xamarin.Forms*, a fim de verificar a eficiência do modelo de desenvolvimento onde a escrita do código é feita em uma única linguagem, e o aplicativo resultante pode ser executado em diferentes plataformas móveis, tais como o Android e o iOS. Além disso, avaliou-se como o *Xamarin.Forms* torna transparente os detalhes de implementação de cada plataforma, e também, se existem áreas onde o desenvolvedor deve conhecer mais a fundo a plataforma onde o aplicativo final vai ser implantado. Para tanto, este trabalho descreve o desenvolvimento de um aplicativo para a interpretação de laudos de análise de solo com um único código base, mas que executa nas principais plataformas móveis existentes no mercado.

O trabalho apresentado por Lisandro, relaciona-se com este artigo pelo fato de ambos terem desenvolvido APIs de utilizarem outros frameworks para o desenvolvimento do projeto. Enquanto que nesta arquitetura optou-se pelo Qt que utiliza a linguagem C++, o trabalho de Lisando utilizou o *Xamarin* que utiliza a linguagem C#. Alguns recursos Disponibilizados pelo Qt também estão disponíveis no *Xamarin*. No entanto, o trabalho de Lisandro não focou em entregar um aplicativo genérico que pudesse ser reaproveitado para outros aplicativos, mais fez uso dos mesmos recursos disponibilizados nesta arquitetura.

A tabela 1 apresenta um comparativo dos recursos disponibilizados na arquitetura descrita neste artigo com os trabalhos relacionados descritos anteriormente.

4. PROJETO DA ARQUITETURA

A arquitetura proposta neste trabalho utiliza os estilos arquiteturais *Client-Server* e *Event-Based*. *Client-Server* pelo fato de permitir ao aplicativo consumir algum serviço *RESTful* através de uma API de alto nível para acesso a rede, tonando o aplicativo um cliente, enquanto que o serviço *RESTful* representa o servidor [29]. Já o *Event-Based* é estilo de arquitetural que caracteriza a comunicação entre objetos através de eventos [29], que neste trabalho, é um recurso disposto através de componentes reusáveis. A arquitetura dispõe de objetos de baixo nível escritos em C++ que trabalham na camada de aplicação, recebendo dados de objetos da camada *UI* e interagem com objetos da camada de *middleware* que realizam a comunicação com o serviço *RESTful* configurado para o aplicativo, caracterizando a arquitetura como um modelo em camadas. No modelo em camadas, a conexão entre os componentes pode ser realizado tanto por eventos como por objetos compartilhados ou, através de leitura e escrita em um arquivo ou em uma base de dados. Porém, nesta arquitetura, foi utilizado somente eventos para comunicação entre os objetos. A escolha de eventos como principal conector entre os objetos foi feita principalmente por proporcionar comunicação assíncrona entre emissor e ouvinte e pelo fato de não acoplar os componentes, garantindo maior independência entre eles. Outro motivo da escolha de eventos, é o fato do Qt provê nativamente um mecanismo de comunicação através de eventos.

4.1 Tecnologias Utilizadas

As tecnologias utilizadas consiste de todos os elementos utilizados durante o desenvolvimento deste trabalho. O Qt e o *QtCreator* foram os elementos mais importantes, pois, forneceram os recursos e ferramentas para a construção das principais características da arquitetura. Dentre os recursos providos pelo Qt destaca-se os eventos, que permitem interligar objetos através de sinais e slots[2] ou *signal handles*, e as APIs providas em classes C++ que integram os recursos da arquitetura, tais como, persistência de dados (via *QSettings* e *QSqlDatabase*) e rede (via *QNetworkAccessManager*). O *QtCreator* é uma IDE que possui recursos integrados à um projeto Qt com destaque para facilidade de *build* do projeto, construção do executável do aplicativo e o *deploy* em um *smartphone*. O *QtCreator* também foi utilizado como editor de código fonte.

4.2 Processo de Projeto Arquitetural

Esta arquitetura foi desenvolvida sob uma metodologia ágil com destaque para uma programação extrema e teste contínuo. A arquitetura recebeu alterações durante 10 meses e a primeira etapa de desenvolvimento introduziu o suporte aos plugins. O primeiro desafio foi desacoplar os plugins do arquivo *qrc*[3] e permitir que a aplicação carregasse-os dinamicamente. Também nesta primeira etapa, foi implementado alguns recursos associados aos plugins, como controle de cache dos arquivos QML, ordenação e *parsing* das páginas (definido pelos plugins), além da criação de um componente genérico a ser estendido por todas as páginas do aplicativo. O controle de cache consiste em regenerar o cache dos arquivos QML após uma atualização para garantir o carregamento de mudanças em cada arquivo a cada *release*. O componente genérico foi definido como *BasePage.qml*, ele é um objeto da camada *UI* e foi criado para garantir o atendimento de alguns requisitos mínimos de aparência, estrutura e simplificar a criação de páginas. Na segunda etapa, foi implementado classes C++ para gerenciar as configurações da aplicação, uma classe utilitária com métodos a serem invocados pelos plugins para operações de baixo nível que ainda não são suportados pelo QML.

Na terceira etapa, foi definido os layouts visuais suportados pela arquitetura e dois modelos foram implementados: O layout em pilha, que faz uso do *container StackView* e o layout em linha, que faz uso do *container SwipeView*, ambos do *Quick Controls*[4]. Em etapas seguintes, foi desenvolvido componentes visuais reutilizáveis, além das APIs para acesso a rede, notificações do aplicativo e persistência de dados. Os *containers* de layout trabalham na camada *UI* e gerenciam a criação e remoção das páginas do aplicativo. É importante destacar que em ambos os layouts, somente uma página pode ser visualizada por vez. As imagens a seguir apresentam os layouts em pilha e em linha.

[2]funções javascript ou métodos de uma classe c++ invocados quando o sinal o qual estão conectados for emitido, recebendo em seus parâmetros os argumentos enviado pelo sinal.

[3]qrc – *Qt resource collection* é um arquivo xml que mapeia os arquivos que serão empacotados no aplicativo.

[4]módulo do Qt que provê um conjunto de componentes QML para construção de interfaces gráficas.

Recurso	Este trabalho	CReAMA	MoCA	Trabalho de Jauri com SenchaTouch e PhoneGap	Trabalho de Lisandro utilizando Xamarin.Forms
Provê suporte multiplataforma Android e iOS	Sim	Não	Não	Sim	Sim
Provê suporte a plataforma desktop	Sim	Não	Não	Não	Não
Provê suporte a recursos extensíveis através de plugins	Sim	Não	Não	Não	Não
Provê APIs de alto nível para recursos de rede (HTTP), banco de dados e UI	Sim	Sim	Sim	Sim	Sim
Provê suporte a reuso de componentes	Sim	Sim	Sim	Sim	Sim
Provê suporte a comunicação entre objetos através de eventos	Sim	Não	Sim	Sim	Sim

Tabela 1: Tabela comparativa entre este trabalho e os relacionados

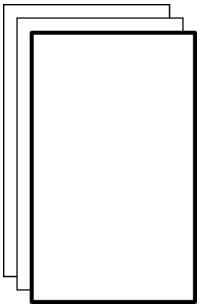

Figura 1: Estrutura de um layout em pilha

No layout em pilha, o componente *ToolBar.qml* será instanciado e posicionado no topo da janela do aplicativo e trabalhará em conjunto com o *StackView*. O *ToolBar* fará *bindings* com algumas propriedades da página ativa, como por exemplo, adicionando ou removendo botões com ações para a página atual.

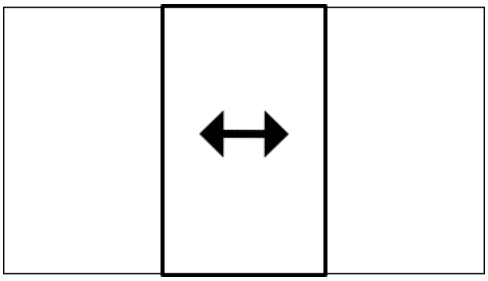

Figura 2: Estrutura de um layout em linha

No layout em linha, o componente *TabBar.qml* será instanciado e posicionado no rodapé da janela do aplicativo. O TabBar corresponde ao menu no layout em linha e trabalhará em conjunto com o *SwipeView*. A arquitetura suporta intercalar os dois layouts ao mesmo tempo e um objeto *Binding* do QML manterá o *SwipeView* visível somente quando não houver páginas na pilha do *StackView*. No entanto, o *SwipeView* será instanciado somente se a propriedade *usesSwipeView* for definida para *true* no arquivo de configuração e se tornará o *container* principal. Os objetos correspondentes aos layouts serão instanciados na inicialização e os plugins poderão adicionar ou remover páginas dinamicamente utilizando os objetos *swipeView* e *pageStack* quando for necessário navegar para uma determinada página a partir de outra, sem ser pelo menu.

4.3 Requisitos Funcionais Suportados

O projeto desta arquitetura visa atender quatro requisitos funcionais entendidos como básicos em todo sistema de informação. Para atender aos requisitos, foi implementado APIs usando classes C++ nativas do Qt. Apesar de o QML dispôr de funcionalidades que poderiam atender a estes requisitos, foi decidido implementar em C++ por questões de desempenho devido menos código interpretado, melhor gerenciamento de memória e para simplificar a implementação de código nos plugins. Os requisitos listados a seguir, foram disponibilizados na arquitetura através de APIs de alto nível que serão apresentadas em tópicos específicos posteriormente, estes quatro requisitos são:

1. Acesso a rede para comunicação com serviços *RESTful*;
2. Persistência de dados local via *SQLITE*;
3. Notificações do aplicativo via *push* e local;
4. Comunicação entre objetos facilitado.

4.4 Visão Estrutural

A figura a seguir, apresenta um diagrama de pacotes e arquivos destacando uma visão lógica dos principais elementos da arquitetura. Em seguida, será descrito a responsabilidade e o conteúdo de cada um deles.

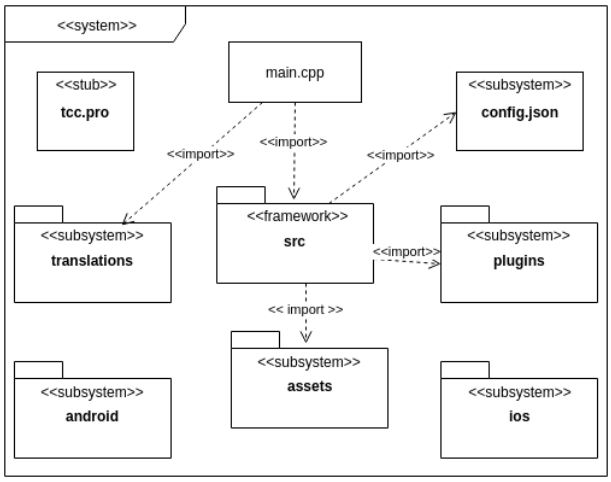

Figura 3: Pacotes principais da arquitetura

1 *main.cpp*: Arquivo inicialização da aplicação Qt. Esse arquivo é responsável por instanciar as classes do Qt que exibem a janela do aplicativo e gerenciam o *loop* da aplicação, além do interpretador QML e classes da camada de aplicação. O *main.cpp* também é responsável por carregar o arquivo de tradução de acordo com o idioma do dispositivo e registrar objetos no contexto da aplicação a serem utilizados pelos plugins.

2 *tcc.pro*: Arquivo de configuração de todo projeto Qt. Nele é definido os módulos do Qt a serem utilizados na aplicação, as classes C++ que serão compiladas e adicionadas no executável, os aquivos *qrc* que mapeiam os componentes QML, as imagens e arquivos genéricos a serem empacotados no aplicativo, além de módulos e arquivos de configuração para cada plataforma (linux, osx, android e iOS). É neste arquivo que fica definido onde os plugins serão instalados no dispositivo.

3 *config.json*: Arquivo de configuração do aplicativo que deve ser editado pelo desenvolvedor. Ele contém propriedades que indicarão alguns comportamentos iniciais, além do tipo de layout a ser utilizado no aplicativo. Outra propriedade presente nesse arquivo, é a opção de exibir ou não os termos de uso na primeira inicialização (carregará o arquivo *assets/eula.html* definido pelo desenvolvedor), se tem login ou não (caso sim, instanciará um objeto que gerencia o perfil do usuário e exibirá a página de login na inicialização) entre outras propriedades. Os detalhes deste arquivo serão apresentados em um tópico mais adiante.

4 *src*: Diretório de código fonte. É onde estão as classes C++, componentes QML utilizados internamente e dispostos para os plugins como reusáveis. Esse diretório está sub-dividido em outros seis diretórios que organizam as classes por tipo de API e são eles:

- *core*: contém classes do núcleo da aplicação dentre elas *Settings*, *PluginManager*, *Observer*, *Subject* e *Utils*;
- *database*: contém classes da API de persistência de dados;
- *extras*: contém classes de customização de estilo no android;
- *notification*: contém as classes que gerenciam as notificações baseadas na plataforma;
- *network*: contém as classes da API de rede (HTTP);
- *qml*: contém os arquivos QML sub-divididos em *private* e *public*, sendo os componentes em *private* os que são utilizados internamente pela aplicação e os que estão em *public* os reutilizáveis.

5 *plugins*: Diretório de plugins. Cada plugin deve estar em um sub-diretório com no mínimo um arquivo de configuração de nome *config.json* e os arquivos QML necessários para o seu funcionamento. Os detalhes dos requisitos para o carregamento de um plugin serão descrito em um tópico posterior.

6 *translations*: Diretório contendo os arquivos de tradução. Os arquivos de tradução devem ser gerados ou atualizados antes de cada *release* do aplicativo. Um arquivo de resources *translations.qrc* existe neste diretório e deve ser utilizado para mapear os arquivos de idioma suportados pelo aplicativo. Cada arquivo de tradução deve ser nomeado seguindo o padrão *language_COUNTRY* com extensão *ts*, por exemplo: *pt_BR.ts*. Ao iniciar a aplicação, o *main*, tentará identificar o idioma do dispositivo e o arquivo de tradução correspondente (se houver) será carregado para que os textos visíveis sejam traduzidos para o usuário. Para gerar as traduções, deve-se utilizar o comando *lupdate -recursive . -ts translations/pt_BR.ts* (na raíz do projeto) para criar ou atualizar o arquivo *ts* do idioma especificado.

7 *android*: Diretório contendo os arquivos de configuração do aplicativo para a plataforma android. Outros sub-diretórios guardam arquivos do *gradle* utilizados para o *build* do APK, ícones do lançador do aplicativo e classes java, além de uma versão da lib *openssl* compilada para o funcionamento de requisições HTTP.

8 *assets*: Diretório contendo imagens e arquivos de configuração do *qtquickcontrols2*, além de um arquivo html que pode ser usado para exibir os termos de uso do aplicativo quando necessário (se a propriedade *showEula* for definida para *true* no arquivo de configuração). Um arquivo de *resources assets.qrc* mapeia todos os arquivos contidos neste diretório.

9 *ios*: Diretório contendo os arquivos de configuração do aplicativo para a plataforma iOS. Pode conter os ícones e imagens requeridas pelo iOS, tais como as imagens de *splash-screen*, além do arquivo de configuração *Info.plist* que define nome, versão do aplicativo e os recursos do sistema requerido para o funcionamento da aplicação no iOS (permissão para acessar a câmera do dispositivo, enviar mensagens via *push* etc.).

4.5 Infraestrutura de Plugins

As funcionalidades de um aplicativo baseado nesta arquitetura devem ser implementadas através de plugins, atendendo aos requisitos do aplicativo a ser desenvolvido utilizando apenas QML. Os plugins são independentes entre sí e podem incluir arquivos QML, TXT, HTML e imagens em seu diretório. Qualquer componente de um plugin pode reutilizar os componentes públicos usando a diretiva *import "qrc:/publicComponents/"*. Ao todo, dez componentes foram disponibilizados e serão apresentados em uma seção posterior.

Os plugins estão desacoplados do núcleo da aplicação e serão conhecidos em tempo de execução. Ao adicionar um novo plugin no diretório *plugins*, ele será carregado no próximo *build*. Para que um plugin seja identificado pelo objeto gerenciador de plugins e carregado na aplicação, é necessário obedecer as seguintes restrições:

1ª estar em um sub-diretório dentro de *plugins*;

2ª conter um arquivo *config.json* neste sub-diretório;

3ª conter pelo menos um arquivo QML.

O arquivo *config.json* de um plugin deve ser um objeto json contendo as seguintes propriedades:

1. *listeners* (array): uma lista de strings que identifica os arquivos do plugin (componentes QML não visuais) que serão instanciados como observadores de eventos da aplicação. O preenchimento dessa propriedade é opcional e pode ser preenchida mesmo que a propriedade *pages* seja definida. Os itens especificados em *listeners* serão instanciados dinamicamente na inicialização do aplicativo e poderão ser destruídos quando não forem mais necessários, invocando o método *destroy* a partir do elemento raiz do objeto.

2. *pages* (array): uma lista de objetos que identifica as páginas do plugin que serão acessadas a partir dos menus do aplicativo e deve ser preenchido se a propriedade *listeners* estiver vazia.

Cada objeto em *pages* poderá conter as seguintes propriedades:

- *qml* (string): O nome do arquivo correspondente a página. Se essa propriedade não for definida, a página não será carregada;
- *title* (string): O título correspondente a página a ser exibido no menu. Esse valor também é requerido, se não for definido, a página não será adicionada ao menu;
- *awesomeIcon* (string) (opcional): O nome de um ícone do *Awesome Icons*[5] que será exibido no menu, em conjunto com o título. Se esse valor não for definido, um ícone padrão (*gear*) será utilizado;
- *roles* (array): Uma lista de strings contendo os nomes de perfil de usuário que poderão acessar a página. Essa lista será útil somente se for definido o tipo de perfil do usuário no objeto *userProfile.profile*. O objeto *userProfile* é *null* por padrão e será instanciado na inicialização do aplicativo se a propriedade *usesLogin* for definida para *true* no arquivo de configuração. Caso *roles* não for definido, será setado um array vazio. Porém, se *userProfile* for instanciado, ou seja, tiver login na aplicação e for definido uma string contendo o tipo de perfil do usuário e, essa string não tiver em *roles*, a página não será exibida para o usuário. Informações sobre o objeto *userProfile* será detalhado em um tópico posterior;
- *order* (int): Um valor numérico que define a ordem em que a página será exibida na lista de itens nos menus. O desenvolvedor deverá definir um valor acima de zero e quanto maior o valor, maior a prioridade da página na lista de itens dos menus;
- *isLoginPage* (bool): Um valor booleano que indica se a página representa a tela de login do aplicativo e deve ser definido pela página correspondente se a propriedade *usesLogin* for definido para *true* no arquivo de configuração. Se o aplicativo usa login, o *path* da página definida como login será persistido pois, será lido por funções internas do aplicativo na inicialização e quando o usuário fizer *logout*. Se mais de uma página definir *isLoginPage* para *true* dentre os plugins, será utilizada a última página identificada pelo gerenciador de plugins;
- *isHomePage* (bool): Um valor booleano que indica se a página corresponde a primeira página exibida para o usuário e deve ser utilizado pela página correspondente quando o layout em pilha estiver sendo utilizado. O *path* dessa página será persistido durante o carregamento dos plugins e será carregada por funções internas na inicialização quando não houver login ou, após o usuário efetuar o login e o json do usuário for passado em um evento específico que inicializa o perfil do usuário (esse recurso será detalhado posteriormente). Se mais de uma página definir *isHomePage* para *true*, será utilizada a última página identificada pelo gerenciador de plugins;
- *showInDrawer* (bool): Um valor booleano que indica se a página poderá ser exibida no menu de layout em pilha. Por padrão, o menu de layout em pilha será carregado quando *StackView* for utilizado. Porém, poderá ser instanciado no layout em linha se a propriedade *usesDrawer* for definida para *true* no arquivo de configuração. O layout em linha utiliza uma barra de botões como menu e através dessa propriedade, possibilita ao usuário acessar páginas diferentes a partir dos dois menus do aplicativo;
- *showInTabBar* (bool): Um valor booleano que indica se a página poderá ser exibida no menu de layout em linha. O objetivo dessa propriedade é permitir exibir páginas diferentes nos menus quando o *drawer menu* for instanciado.

O código a seguir, apresenta um exemplo de um plugin com uma lista de páginas, que podem ser visualizadas nas figuras 4 e 5.

```
{
    pages:[
        {
            qml:"Page1.qml",
            title:"Pagina 1",
            awesomeIcon:"commenting",
            order:3,
            roles:["student"],
            showInDrawer:true,
            showInTabBar:true
        },
        {
            qml:"Page2.qml",
            title:"Pagina 2",
            awesomeIcon:"calendar",
            order:4,
            roles:["student","teacher"],
            showInDrawer:true,
            showInTabBar:true
        },
        {
            qml:"Page3.qml",
            title:"Pagina 3",
            awesomeIcon:"code_fork",
            order:5,
            roles:["student","teacher"],
            showInDrawer:true,
            showInTabBar:true
        },
        {
            qml:"ProfileView.qml",
            title:"Meu perfil",
            roles:["student","teacher"],
            awesomeIcon:"user",
            order:1,
            showInDrawer:true,
            showInTabBar:true
        },
        {
            qml:"AboutPage.qml",
            title:"Sobre",
            awesomeIcon:"question_circle",
            roles:["student","teacher"],
            order:0,
            showInDrawer:true,
            showInTabBar:true
        }
    ]
}
```

As páginas serão instanciadas sob demanda quando o aplicativo estiver utilizando o layout em pilha, que ocorrerá quando o usuário

[5]https://fontawesome.com/icons

clicar em um item da lista. No layout em pilha, o menu é exibido pelo componente *Drawer.qml* que consiste de uma instância do objeto *Drawer* do *Quick Controls* e as páginas serão listadas verticalmente (exemplo na figura 4).

Quando o layout em linha estiver sendo utilizado, uma lista horizontal de botões será adicionado no rodapé da janela do aplicativo permitindo ao usuário alternar entre as páginas disponíveis. Porém, no layout em linha, todas as páginas serão instanciadas no início da aplicação e terá um botão associado a cada página. No layout em linha, o menu corresponde ao componente *TabBar.qml* que consiste de uma instância do objeto *TabBar* do *Quick Controls* com algumas modificações (exemplo na figura 5). As figuras a seguir, apresentam exemplos dos layouts em pilha e em linha.

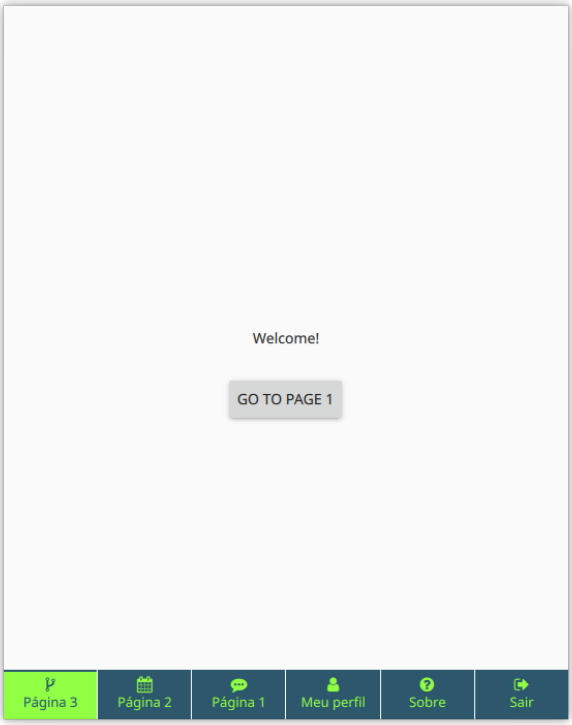

Figura 5: Lista de páginas exibidas no menu de layout em linha

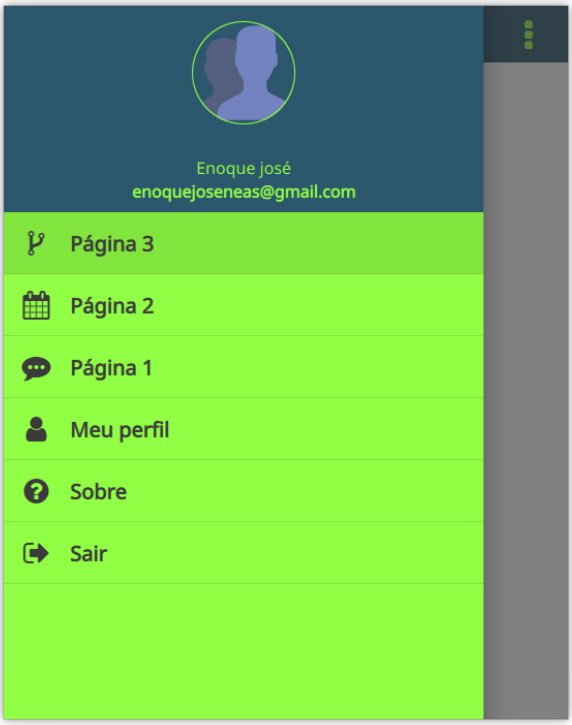

Figura 4: Lista de páginas exibidas no menu de layout em pilha

4.5.1 Gerenciamento de Plugins

A classe *PluginManager* é responsável por carregar os plugins, basicamente, iterando os arquivos dentro do diretório *plugins* e analisando as propriedades do arquivo de configuração de cada plugin. Em cada arquivo, o objeto verificará as definições de cada página, transformando cada item em um objeto e adicionado em uma lista. Após ler todos os plugins, o array de objetos será persistido nas configurações da aplicação para que na próxima inicialização não precise iterar novamente o diretório, lendo as definições dos plugins das configurações. Os plugins serão recarregados após uma atualização do aplicativo ou quando a aplicação for executada em modo *debug*.

A cada inicialização, será feito uma verificação da versão do aplicativo, que será persistida nas configurações na primeira execução e sobre-escrita a cada atualização. Se houver diferença entre versão em execução da versão salva anteriormente (se houver), os plugins serão recarregados. Além disso, esta classe também é responsável por deletar todos os arquivos de cache contido no diretório de cache da aplicação a cada *release*. Outra responsabilidade dessa classe, é a criação da tabela do plugin no banco de dados do aplicativo, se existir um arquivo *plugin_table.sql* no diretório do plugin. O banco de dados da aplicação será criado por outro objeto que controla as operações de persistência e será detalhado em outra seção. O diagrama a seguir, apresenta as operações e atributos da classe *PluginManager*.

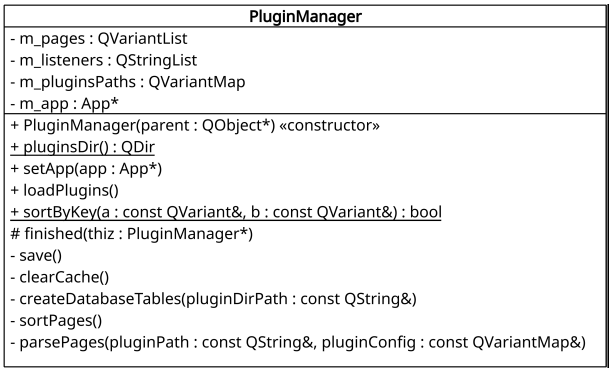

Figura 6: Diagrama da classe PluginManager

O carregamento dos plugins será feito na inicialização do aplicativo, antes de instanciar qualquer componente visual, na invocação do método *loadPlugins*.

4.6 Gerenciamento de Configurações

A arquitetura provê uma API simples e de alto nível para persistir dados utilizando o conceito *chave-valor* através de invocação síncrona. Nesse modelo de persistência, a chave identifica o dado a ser armazenado e o valor é o dado propriamente dito. Os tipos de dados são: strings, números, objetos ou arrays e os métodos disponíveis podem ler, persistir ou deletar. As aplicações simples baseadas nesta arquitetura que não precisam de *SQLITE*, poderão utilizar esse mecanismo para persistir dados no dispositivo. Essa API é provida através da classe *Settings* descrita a seguir.

4.6.1 A classe Settings

A classe *Settings* é um componente importante nesta arquitetura e suas principais responsabilidades consiste em carregar o arquivo de configuração *config.json* e gerenciar a persistência das configurações do aplicativo via *QSettings*[6]. *Settings* será instanciada na inicialização do aplicativo e registrada no contexto da aplicação identificada como um objeto global *"Settings"* para que os plugins possam invocar seus métodos públicos. O arquivo de configuração será convertido em um objeto javascript e registrado no contexto da aplicação como um objeto global em modo leitura, referenciado por *"Config"*. *Settings* adicionará o *path* de cada plugin em uma propriedade no objeto *Config* para que os plugins possam acessar os seus arquivos usando uma sintaxe mais legível (via *Config.plugins.plugin_directory_name*). Por exemplo, considerando que *Session* é um plugin, os arquivos em seu diretório poderão ser acessados da seguinte forma: *Config.plugins.session* + *"File.qml"*. O objetivo dessa propriedade é fornecer aos plugins uma forma simplificada de acessar os arquivos em seu diretório, visto que os plugins serão colocados em diretórios virtuais nas plataformas *mobile*: *"assets://"* no android e "assets_catalogs://"no ios. Quando executado em desktop, será utilizado o caminho absoluto de cada arquivo. *Settings* também dispõe ainda de seis atributos booleanos que indicam qual a plataforma o aplicativo está sendo executado. A estrutura da classe *Settings* será apresentada na figura abaixo.

[6]https://doc.qt.io/qt-5/qsettings.html.

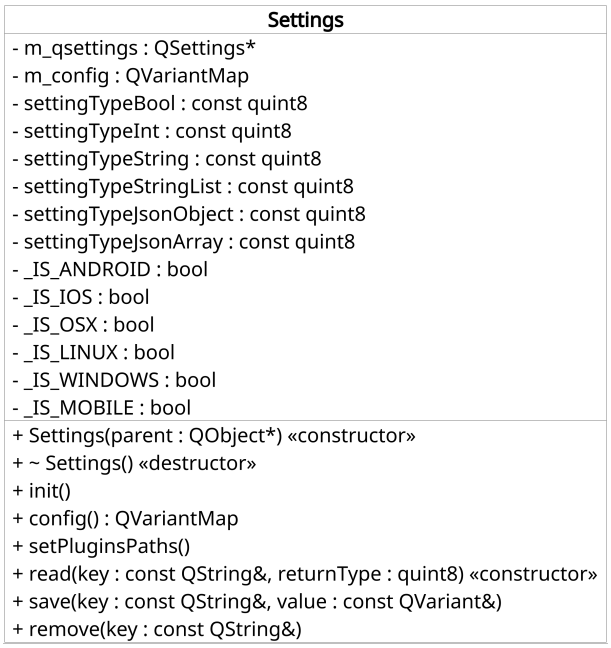

Figura 7: Diagrama da classe Settings

Settings é quem define o estilo de *widgets* utilizado pelos componentes QML (Material, Universal, etc.). A definição do estilo deve ser feito pelo desenvolvedor no arquivo de configuração na propriedade *applicationStyle* que será passado para um objeto *QQuickStyle* do Qt na inicialização. Os possíveis valores para esta propriedade serão detalhados na seção que descreve o arquivo *config.json*.

Os métodos da classe *Settings* serão descritos a seguir, seguido de um exemplo de como utilizá-los:

- *read*: Utilizado para ler um dado salvo utilizando uma string que identifique a informação a ser retornada. Por padrão, o tipo de retorno é string. Porém, se o dado não for string, o segundo parâmetro deve ser passado indicando o tipo específico a ser retornado. Os tipos podem ser especificados para evitar o uso de *cast* no QML. Os tipos possíveis são:
  - *TypeBool*: para retornar um valor booleano;
  - *TypeInt*: para retornar um inteiro;
  - *TypeStringList*: para retornar uma lista de strings;
    TypeJsonObject: para retornar um objeto;
  - *TypeJsonArray*: para retornar um array de objetos.
- *save*: Utilizado para persistir alguma informação. Esse método é *void* e possui dois parâmetros requeridos. O primeiro é uma string que identifica o dado a ser persistido, e o segundo, é o dado que será armazenado. O dado pode ser uma string, um valor numérico, um objeto ou array json.
- *remove*: Utilizado para apagar alguma informação das configurações. Esse método requer apenas um parâmetro, uma string que identifica o dado a ser deletado.

O código a seguir, apresenta um exemplo de persistência e leitura de dados usando o mecanismo *chave-valor* a partir do objeto *Settings*:

```
import "QtQuick" 2.8

Item "{
    Component.onCompleted:{
        var "foo" = "Settings.read("foo")
        if "(bar "!= "foo)
            Settings.save("foo",bar)
        Settings.remove("bar")
    }
}
...
Item "{
    ...
    key:"bar"
    count:Settings.read(key,Settings.TypeInt)
    ...
    Component.onCompleted:{
        Settings.save(key,++count)
    }
}
```

4.7 O Arquivo de Configuração

O arquivo *config.json* presente na raiz do projeto é um componente importante nesta arquitetura e contém as definições de configuração do aplicativo. No entanto, as informações desse componente não serão persistidas e o conteúdo (propriedades e valores) será passado para a aplicação como um objeto javascript global. As propriedades listadas a seguir, serão utilizadas por componentes internos e o desenvolvedor poderá adicionar outras propriedades quando for necessário compartilhar informações entre os plugins. Os plugins poderão ler as propriedades declaradas neste arquivo via *Config.property_name*. As propriedades definidas inicialmente e utilizadas pela aplicação serão descritas a seguir:

- *applicationName* (string): O nome do aplicativo. Este valor será passado para o Qt na inicialização da aplicação. O valor dessa propriedade será utilizado para definir o nome dos arquivos de configuração (*QSettings*) e do banco de dados *SQLITE* (quando houver);
- *organizationName* (string): O nome da organização do aplicativo. Este valor também será passado para o Qt na inicialização do aplicativo e será utilizado para nomear e identificar o diretório de configurações e cache do aplicativo;
- *organizationDomain* (string): O endereço de domínio da organização, por exemplo, *qt.project.org*. O valor dessa propriedade será utilizado pelo Qt internamente;
- *applicationStyle* (string): O nome do estilo a ser aplicado nos *widgets* (*Button*, *TabBar*, *ToolTip* etc.) do *Quick Controls*. Os possíveis valores são: *Material*, *Universal* ou *Default*;
- *forceEulaAgreement* (bool): Um valor booleano que indica se a aplicação deverá exigir do usuário confirmação de aceitação dos termos de uso para continuar usando o aplicativo. Esse valor só terá efeito se a propriedade *showEula* for definida para *true*;
- *hasLogin* (bool): Um valor booleano que indica se o aplicativo deverá carregar uma página de login na inicialização. Se for definido para *true*, a aplicação irá utilizar a página que definiu (dentre os plugins) *isLoginPage* para *true*;
- *showEula* (bool): Um valor booleano que indica se a aplicação exibirá para o usuário uma página contendo os termos de uso do aplicativo. Caso seja setado para *true*, o arquivo *assets/eula.html* será carregado e exibido na primeira execução do aplicativo. Após o usuário ler e aceitar os termos de uso, o usuário irá para a página de login ou a *home page*. No entanto, se essa propriedade for utilizada o desenvolvedor deverá escrever os termos de uso no arquivo *assets/eula.html*;
- *showTabButtonText* (bool): Um valor booleano que indica se o título da página será exibida nos botões do menu em linha (*TabButton* do *Quick Controls*). Essa propriedade terá efeito somente quando a aplicação tiver usando o layout em linha;
- *usesSwipeView* (bool): Um valor booleano que indica se o layout principal do aplicativo será em linha. Caso essa propriedade seja *true*, o aplicativo utilizará o container *SwipeView* do *Quick Controls*. No entanto, o container responsável pelo layout em pilha (*StackView*) ainda continuará disponível na aplicação, porém como container secundário. Um objeto fará o *Binding* entre ambos, ocultando o *SwipeView* quando alguma página for adicionada a pilha do *StackView*. No *SwipeView* o usuário poderá alternar entre as páginas deslizando horizontalmente;
- *usesDrawer* (bool): Um valor booleano que indica se o menu lateral usado no layout em pilha, será instanciado. Essa propriedade terá efeito apenas no layout em linha, ou seja, se *usesSwipeView* for *true*, pois no layout em pilha ele será instanciado mesmo que o valor seja *false*. O objetivo dessa propriedade é permitir que o desenvolvedor possa utilizar o menu lateral junto com o *SwipeView*, intercalando as páginas que serão visíveis em cada um dos menus através das propriedades *showInDrawer* e *showInTabBar* na configuração das páginas de cada plugin;
- *showDrawerImage* (bool): Um valor booleano que indica se a imagem do menu será carregada. Por padrão a imagem não será exibida. Se essa propriedade for definida para *true*, a imagem definida em *assets/drawer.jpg* será utilizada e o desenvolvedor poderá substituí-la por uma de sua preferência;
- *restService* (object): Um objeto contendo as definições do serviço *RESTful*, como url base e os parâmetros de autenticação básica (*Basic Authentication*) usuário e senha. Os valores das propriedades *userName* e *userPass* serão convertidos em um *hash base64* e passado no cabeçalho de cada requisição HTTP. As seguintes propriedades são requeridas:
  - *userName* (string): O nome do usuário do serviço *RESTful*;
  - *userPass* (string): A senha de usuário do serviço *RESTful*;
  - *baseUrl* (string): A url base do serviço *RESTful*. Essa propriedade será utilizada pelo objeto *RequestHttp* nos métodos de requisições *GET*, *POST* etc. A sugestão é que o desenvolvedor adicione apenas o *path* nos objetos que fazem requisições HTTP. Desta forma, se a url do *web service* for modificada, basta alterar apenas no arquivo de configuração. Internamente, o objeto que realiza requisições concatenará o valor dessa propriedade com o *path* passado no primeiro parâmetro dos métodos.
- *fontSize* (object): Um objeto com as definições de valores inteiros para os tamanhos de fonte a serem utilizadas em elementos textuais tais como o *Label*. Essa propriedade possui quatro atributos: *small*, *normal*, *large* e *extraLarge*;

- *theme* (object): Um objeto com as definições de cores utilizada nos elementos visuais, tais como Botões, *ToolBar*, *TabBar*, cor de fundo das páginas e cor de fundo do *drawer*;
- *events*: (array): Um array de strings com os eventos utilizados pelos plugins e pode ser adicionado pelo desenvolvedor. Essa propriedade será utilizada por objetos *observer* para identificar de qual evento estão sendo notificados, e o objetivo é padronizar os nomes dos eventos e reduzir a replicação de strings. Os objetos da aplicação podem disparar eventos através de uma instância da classe *Subject*, via *procedure call* com a seguinte notação: *Subject.notify(Config.events.foo, null)*. Os observadores do evento *"foo"* serão notificados recebendo o argumento *null*. O objeto *Settings* adicionará treze itens neste array que identificam os eventos disparados por objetos internos da aplicação. Esses valores podem ser utilizados pelos plugins para executarem ações específicas em dado momento. Os eventos adicionados pelo objeto *Settings* serão descritos a seguir:
  - *cameraImageSaved* (string): Utilizado para notificar observadores de que uma imagem foi capturada pela câmera do dispositivo e salva localmente. Para utilizar a câmera do dispositivo, a arquitetura dispõe de um componente QML *CameraCapture* que pode ser utilizado pelos plugins para permitir ao usuário capturar uma imagem. A url da imagem salva será passado no argumento do evento como uma string;
  - *cancelSearch* (string): Utilizado pelo *ToolBar* para indicar que o campo de busca não está ativo e notificar a página ativa que atualize o conteúdo para o usuário. O *ToolBar* possui um campo texto para pesquisa e ficará visível quando a página alterar o valor da propriedade *toolBarStatus* (string) para *"search"*. Essa funcionalidade permitirá que uma página filtre os resultados exibidos na sua *view*, recebendo em um atributo string *searchText* o valor digitado no campo. O usuário poderá cancelar a busca clicando em um botão voltar que ficará visível ao lado esquerdo do campo de busca e nesse momento, este evento será disparado;
  - *logoutApplication* (string): Utilizado pelo objeto *UserProfile* para atualizar o status da propriedade *isUserLoggedIn* para *false* e em seguida carregar na página de login. No entanto, esse evento deverá ser utilizado quando houver login na aplicação e deve ser disparado por algum plugin;
  - *newActionNotification* (string): Utilizado para notificar a aplicação quando o usuário clicar em uma notificação e o aplicativo ficar em *foreground*, válido para notificações via *push* e local. Esse evento será disparado pela atividade do Android (*CustomActivity*) e pelo objeto *QtAppDelegate* no iOS, repassando para a aplicação um objeto como argumento do evento. O argumento do evento (*eventData*) conterá os dados da mensagem em uma string sendo necessário fazer o *parsing* caso seja um objeto json;
  - *newPushNotification* (string): Utilizado sempre que uma notificação via *push* chegar no dispositivo e o mesmo estiver em execução, em *foreground* ou *background*. Ele será disparado a partir dos serviços de notificação que executam em outro processo e serão repassados para a aplicação através de uma conexão entre o objeto java *CustomAtivity* no android e o *QtAppDelegate* no iOS. O argumento do evento (*eventData*), conterá os dados da mensagem em uma string sendo necessário fazer o *parsing* caso seja um objeto json;
  - *newPushNotificationToken* (string): Utilizado quando o registro no *Firebase* for realizado com sucesso. Esse evento será disparado por objetos que executam em outro processo e serão passados para a aplicação através de uma conexão entre o objeto java *CustomAtivity* no android e o *QtAppDelegate* no iOS. O argumento do evento (*eventData*) conterá o token em uma string;
  - *openDrawer* (string): Utilizado intermante para abrir o menu do layout em pilha quando o usuário clicar no botão de menu, posicionado no canto esquerdo do *ToolBar*. O menu do layout em pilha fica oculto por padrão e o usuário poderá torná-lo visível arrastando da esquerda para a direita na janela do aplicativo;
  - *popCurrentPage* (string): Esse evento deverá ser disparado quando for necessário remover a página ativa da pilha do *StackView*. Será disparado internamente pelo *ToolBar* quando o usuário clicar no botão voltar (seta para a esquerda, exibido se a página definir a propriedade *toolBarState* para *"goBack"*) ou quando o botão *back-button* do android for pressionado. O *StackView* por exemplo, removerá a página da pilha quando este evento for emitido;
  - *appendOptionPage* (string): Esse evento pode ser utilizado para adicionar um novo item na lista de opções do menu. Esse evento poderá ser utilizado em ambos os layouts em pilha ou em linha e o argumento do evento deve ser um objeto contendo as propriedades de uma página utilizado no *config.json* de um plugin;
  - *requestUpdateUserProfile* (string): Esse evento deve ser emitido para atualizar as informações ou dados do usuário no objeto *UserProfile*. Por exemplo, uma página que permite editar os dados do usuário em um formulário, após o usuário atualizar alguma informação, a página poderá emitir esse evento passando como argumento um objeto contendo as informações do usuário no estilo *chave-valor*. O objeto *UserProfile* observará esse evento e salvará as informações localmente;
  - *initUserProfile* (string): Esse evento deve ser utilizado para inicializar o perfil do usuário após o login, recebendo um objeto json contendo as informações que serão persistidas localmente. O json deverá ser passado no argumento do evento, contendo no mínimo as propriedades *id* e *email*. Um exemplo de uso deste evento, pode ser feito pela página de login, após sucesso na autenticação. A página de login poderá emitir esse evento e passar como argumento o objeto retornado pelo serviço *REST*. O objeto *UserProfile* observará esse evento e quando o mesmo for emitido, salvará os dados do usuário na propriedade *profile*. Em seguida, a propriedade *isLoggedIn* será atualizada para *true* e a *home page* será carregada;
  - *setUserProfileData* (string): Esse evento também deve ser utilizado quando houver um perfil de usuário e permitirá adicionar ou atualizar uma informação no perfil do usuário. Logo, o argumento do evento deve conter a propriedade *key* indicando o nome da propriedade a ser adicionada e *value* contendo o respectivo valor. Por

exemplo, o seguinte objeto pode ser utilizado como argumento deste evento: *{"key": "username", "value": "enoque"}*;

– *userProfileChanged* (string): Esse evento será disparado pelo objeto *UserProfile* sempre que ocorrer alguma atualização nos dados do usuário. No argumento do evento será passado uma referência para o objeto *profile*.

4.8 Acesso a Rede (HTTP)

O acesso a rede é um requisito funcional provido nesta arquitetura e o objetivo é permitir a comunicação entre o aplicativo e um serviço *RESTful* de forma facilitada. Para disponibilizar o uso de rede pelos plugins, foi criado uma classe C++ com suporte a autenticação básica, *download* e *upload* de arquivos, além de requisições via *GET*, *POST* e *PUT*. O objetivo dessa classe é oferecer um componente rico em recursos, com bom desempenho e de alto nível para os plugins. Os plugins que utilizarem requisições HTTP, receberão a resposta de cada pedido através de objetos json dispensando o uso de *parsing*. Para simplificar o uso da API de rede, o desenvolvedor deverá adicionar na propriedade *restService* no arquivo de configuração, a url do serviço *RESTful* em *baseUrl* e o nome e a senha do usuário em *userName* e *userPass* respectivamente. O objetivo é que o desenvolvedor utilize apenas o *path* da url nos métodos de requisições. A subseção a seguir, descreve os detalhes da classe *RequestHttp* seguido de exemplos de como instanciá-la e como fazer requisições ao *web service* do aplicativo.

4.8.1 A classe RequestHttp

A classe *RequestHttp* será registrada como um tipo QML no contexto da aplicação e para instanciá-la basta adicionar a diretiva *import RequestHttp 1.0* e em seguida, declarar um objeto QML *RequestHttp*. Internamente, os objetos inicializarão os atributos *baseUrl*, nome de usuário e senha, mas, o desenvolvedor pode sobrescrevê-los quando for necessário. Essa classe utiliza *QNetworkAccessManager* para gerenciar as operações de rede abstraindo para o aplicativo a interface de rede utilizada no dispositivo e *QNetworkRequest* para encapsular os parâmetros da requisição. Os atributos e métodos disponíveis serão descritos a seguir, seguido de um exemplo de uso:

- *baseUrl* (string): atributo que define o endereço base do serviço *REST*. Não precisa ser definido, pois será feito automaticamente no construtor do objeto. Porém, poderá ser utilizado pelo desenvolvedor quando a url definida no arquivo de configuração for diferente da url de destino de uma requisição específica;

- *authorizationUser* (string): atributo que define o nome do usuário do serviço *REST*. Também não precisa ser definido, mas está disponível para que o desenvolvedor possa fazer requisições a uma url ou serviço com outro nome de usuário, além do que está definido no arquivo de configuração;

- *authorizationPass* (string): atributo que define a senha do usuário do serviço *REST*. Também não precisa ser definido, mas poderá ser modificado pelo desenvolvedor quando for preciso;

```
import "QtQuick" 2.8
import "RequestHttp" 1.0

RequestHttp "{
    id: requestHttp
    baseUrl: "https://static.api.foo.com"
    authorizationUser: "foo"
    authorizationPass: "bar.2018"
}
...
Item "{
    Component.onCompleted: {
        var "queryString "= "{
            arg1: "123",
            arg2: "321"
        }
        requestHttp.get("bar", queryString)
    }
}
```

- *get*: método que realiza requisições do tipo *GET* exigindo apenas a url de destino. Esse método possui dois parâmetros opcionais, o primeiro deles é um objeto do tipo *chave-valor* e se for passado, adicionará a url uma *query-string*. O segundo parâmetro opcional é também um objeto e pode ser passado quando for necessário adicionar dados no cabeçalho da requisição. O código a seguir, apresenta um exemplo de uso desse método:

```
import "QtQuick" 2.8
import "RequestHttp" 1.0

RequestHttp "{
    id: requestHttp
}
...
Item "{
    Component.onCompleted: {
        var "queryString "= "{
            someKey: "foo",
            paginate: listView.count
        }
        //... foo?someKey=foo&paginate=16
        requestHttp.get("foo", queryString)
    }
}
```

- *post*: método que realiza requisições do tipo *POST* exigindo a url de destino e o dado a ser postado em formato string. O terceiro parâmetro desse método é opcional e pode ser passando quando for necessário adicionar dados no cabeçalho da requisição. É importante destacar que o *Content-Type* será sempre *application/json* e se o método post exigir outros formatos tais como o *x-www-form-urlencoded*, o desenvolvedor deverá passar um objeto (*Content-Type: [Type]*) como argumento extra no cabeçalho da requisição através do terceiro parâmetro desse método. O código a seguir, apresenta um exemplo de uso desse método:

```
import "QtQuick" 2.8
import "RequestHttp" 1.0

RequestHttp "{
    id: requestHttp
}
...
Item "{
    Component.onCompleted: {
        var "postData "= "JSON.stringfy({
            someKey: "bar",
            email: textField.text
```

```
        })
        requestHttp.post("bar",postData)
    }
}
```

- *uploadFile*: realiza uma requisição do tipo *POST* ou *PUT* exigindo a url de destino e um array de strings contendo os endereços dos arquivos locais a serem enviados para o servidor. A requisição será do tipo *multipart-formdata* e do tipo *POST* (por default). O terceiro parâmetro (booleano, default *false*) pode ser passado quando for necessário utilizar o método *PUT*. Já o último parâmetro, poderá ser passado quando for necessário adicionar dados no cabeçalho da requisição. Após invocar esse método, o sinal *uploadFinished* será emitido para cada arquivo enviado. Durante o envio de cada arquivo, será emitido o sinal *uploadProgressChanged* contendo os bytes do arquivo (local) e o total de bytes já enviados para o servidor. O código a seguir, apresenta um exemplo de uso desse método:

```
import "QtQuick" 2.8
import "RequestHttp" 1.0

RequestHttp "{
    id:requestHttp
}
...
Item "{
    Component.onCompleted:{
        var "files "=" [
            "/data/app/myapp/files/img1.png",
            "/data/app/myapp/files/img2.png"
        ]
        requestHttp.upload("bar",files)
    }
}
```

- *downloadFile*: realiza uma requisição do tipo *GET* exigindo apenas uma lista de urls de arquivos a serem baixados para o dispositivo. Por padrão, os arquivos serão salvos no diretório público de *downloads* no dispositivo. No entanto, o segundo parâmetro *saveInAppDirectory* (booleano, default *false*) pode ser utilizado para alterar o diretório de destino dos arquivos para uma pasta interna do aplicativo. Após invocar esse método, para cada arquivo salvo, o sinal *downloadFileSaved* será emitido passando o *path* do arquivo salvo localmente. Durante o download de cada arquivo, o sinal *downloadProgressChanged* será emitido passando dois argumentos, o primeiro indica o total de bytes do arquivo que está sendo baixado e o segundo, um valor inteiro indicando os bytes que já foram baixados. O código a seguir, demonstra um exemplo de requisição para download de arquivos:

```
import "QtQuick" 2.8
import "RequestHttp" 1.0

RequestHttp "{
    id:requestHttp
}
...
Item "{
    Component.onCompleted:{
        var "urls "=" [
            "https://.../files/f1.png",
            "https://.../files/f2.png"
        ]
        requestHttp.downloadFile(urls)
    }
}
```

Os métodos descritos anteriormente são todos *void*, assíncronos e declarados como *Q_INVOKABLE* [7] para permitir o uso dos métodos em componentes QML. Para obter a resposta de uma requisição, é preciso criar uma conexão com o sinal *finished* usando o objeto *Connections*[8] do QML, especificando como *target* o objeto *requestHttp*. O sinal *finished* será emitido por todos os métodos da classe *RequestHttp* descritos anteriormente quando não houver erros no pedido e logo após o envio da resposta pelo servidor. O sinal *finished* passará dois argumentos: *statusCode* — um valor inteiro indicando o status da resposta (200, 400, 500, etc.) e *response* — o dado da resposta enviado pelo servidor. Se a resposta enviada pelo servidor for um json válido, *response* será convertido em objeto ou array javascript, caso contrário, uma string contendo o dado bruto.

O sinal *error* será emitido quando houver erros em um pedido e passará dois argumentos, *statusCode* (integer) indicando o código HTTP correspondente ao erro, e *message* (string) contendo a mensagem do erro.

A propriedade *status* (integer) declarada como *Q_PROPERTY*[9] indicará o *status* atual de uma requisição. Essa propriedade pode ser comparada com qualquer uma das seguintes meta-propriedades (útil para fazer *bindings* com outros objetos):

- *Error*: indica um erro no pedido, quando o servidor não responder ou o status da requisição for zero;
- *Finished*: indica que a requisição terminou e será setada em *status* antes da emissão do sinal *finished*;
- *Loading*: indica que a requisição está em andamento ou carregando;
- *Ready*: indica que a requisição está pronta, será setada em *status* no construtor e logo após a emissão do sinal *finished*.

O trecho de código a seguir, apresenta um exemplo de como criar um *Bind* com o status de uma requisição em uma propriedade de um objeto *Button*.

```
import "RequestHttp" 1.0

RequestHttp "{
    id:request
}
...
Button "{
    text:"Submit"
    enabled:request.status "==" request.Ready
    onClicked:request.get("foo",{"bar":1})
}
...
```

O diagrama a seguir, apresenta os atributos e métodos da classe *RequestHttp*:

[7] https://doc.qt.io/qt-5/qtqml-cppintegration-exposecppattributes.html
[8] http://doc.qt.io/archives/qt-4.8/qml-connections.html
[9] http://doc.qt.io/qt-5/properties.html

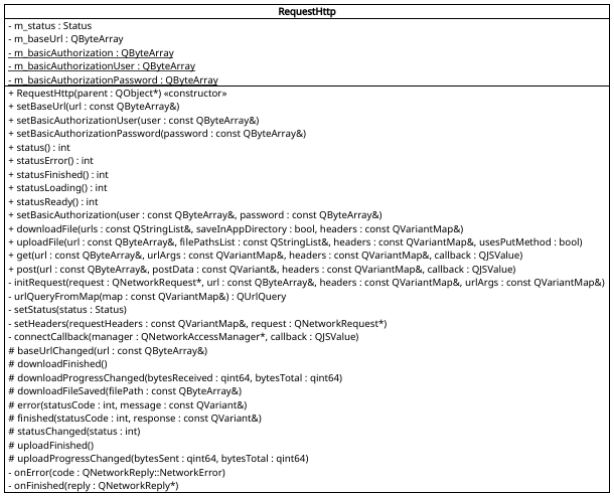

Figura 8: Diagrama da classe *RequestHttp*

4.9 Persistência de Dados

Persistência de dados é um requisito funcional implementado e disponível nesta arquitetura e visa fornecer aos plugins a possibilidade de persistir dados em um banco *SQLITE* e o funcionamento *offline* do aplicativo com maior robustez. Cada plugin pode criar uma ou mais tabelas no banco de dados e realizar as operações de inserção, atualização e busca de dados em suas tabelas, basta importar o componente *Database* com a diretiva *import Database 1.0* e declarar um objeto QML. Para criar as tabelas no banco de dados do aplicativo, o plugin deve fornecer um arquivo *plugin_table.sql* em seu diretório, contendo os comandos de criação, alteração ou remoção das tabelas. A cada *release*, esse arquivo será carregado e executado como uma *query* sql permitindo aos plugins atualizar as tabelas (adicionar, remover ou modificar as colunas) quando for necessário.

Para gerenciar as operações de banco de dados, foi criado uma classe C++ chamada *Database* que encapsula em métodos de alto nível as operações necessárias para criar o banco de dados, conectar e executar as operações sql. No entanto, essa classe não será utilizada pelos plugins diretamente, foi criado outra classe chamada *DatabaseComponent* que agrega uma instância de *Database* e delegará as operações para esse objeto. *DatabaseComponent* fornecerá alguns recursos para simplificar ainda mais a persistência de dados. As subseções a seguir apresentam detalhes de ambas as classes *Database* e *DatabaseComponent*.

4.9.1 A classe Database

A classe *Database* é responsável por criar o banco de dados *SQLITE* na primeira execução do aplicativo se houver necessidade, ou seja, se algum plugin dispor de um arquivo *plugin_table.sql* em seu diretório e será instanciada pelo objeto *PluginManager* para criar a tabela do plugin. Se nenhum plugin fornecer esse arquivo, o banco de dados *SQLITE* não será criado. A classe *Database* utiliza as classes *QSqlDatabase* para criação e conexão com o banco de dados e as classes *QSqlQuery* e *QSqlRecord* para as realizar *querys* e retornar os resultados das consultas. Os métodos principais desta classe serão descritos a seguir:

- *tableColumns*: esse método retorna uma lista de strings contendo os nomes das colunas da tabela especificada no único parâmetro do método;
- *createTable*: esse método é invocado pelo gerenciador de plugins quando houver um arquivo sql para criação de uma tabela no banco *SQLITE* do aplicativo. Ele requer um único parâmetro string com o path do arquivo contendo os comandos de criação da tabela do plugin;
- *select*: esse método pode ser utilizado para recuperar dados de uma tabela através de uma consulta *sql* e possui três parâmetros, o primeiro é nome da tabela onde será feito a consulta, o segundo é um objeto contendo os parâmetros da consulta (condição *where*) no estilo *nome-da-coluna.valor*, e o último parâmetro, um outro objeto contendo argumentos adicionais, tais como *limit*, *offset* e *order by*. Esse método retornará uma lista de objetos resultante da consulta, e cada objeto contido na lista é composto de *nome-da-coluna.valor*. Se a consulta não for efetuada com sucesso, será retornado uma lista vazia;
- *insert*: esse método pode ser utilizado para inserir dados em uma tabela e os parâmetros requeridos são: o nome da tabela onde será feito a inserção e um objeto no estilo *nome-da-coluna.valor* contendo os dados a serem persistidos. Se a inserção for efetuada com sucesso e a tabela possuir uma coluna auto-incrementada como chave-primária o valor incrementado será retornado. Caso contrário, não possuir a coluna auto-incrementada, retornará o valor inteiro 1 (um) ou, retornará zero se houver erros na inserção;
- *remove*: esse método pode ser utilizado para deletar um ou mais registros em uma tabela e três parâmetros são requeridos: o primeiro é o nome da tabela onde será feito a remoção, o segundo é um objeto contendo os argumentos ou filtros da *query* no estilo *nome-da-coluna.valor*. O terceito parâmetro é opcional e pode ser passado para customizar o operador de comparação para cada par *nome-da-coluna.valor* no segundo parâmetro. Por default, será utilizado o operador de igualdade (“=”);
- *update*: esse método pode ser utilizado para atualizar um ou mais registros em uma tabela específica. Os parâmetros desse método são basicamente os mesmos do método *remove* descrito no item anterior, com uma adição, o parâmetro *updateData*, um objeto contendo os dados a serem atualizados também no estilo *nome-da-coluna.valor*;
- *queryExec*: esse método pode ser utilizado para realizar uma consulta sql a partir de uma string passada como parâmetro. O valor booleano *true* será retornado se a *query* for efetuada com sucesso, caso contrário *false*. O resultado da consulta se houver, deverá ser obtido através do método *resultSet* descrito a seguir;
- *resultSet*: esse método pode ser utilizado para recuperar um conjunto de dados resultante da última consulta sql efetuada, por exemplo, após a chamada ao método *queryExec*. O tipo de retorno é uma lista de objetos no estilo *nome-da-coluna.valor*. O método *select* por exemplo, utiliza esse método como retorno da consulta efetuada internamente. Esse método possui um parâmetro opcional que deve ser utilizado quando a tabela a qual a última consulta efetuada for do tipo *meta_key.meta_value* para que a chave do objeto seja a chave na tabela e o valor, o dado contido em meta_value;

É importante destacar, que essa classe é um *singleton* e será instanciada na inicialização do aplicativo pelo objeto *PluginManager*. Se múltiplos plugins realizarem operações sql, eles utilizarão a mesma instância dessa classe. O diagrama a seguir apresenta os atributos e métodos da classe *Database*.

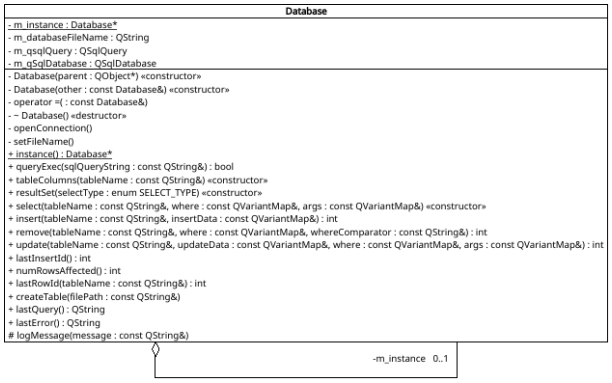

Figura 9: Diagrama da classe Database

4.9.2 *A classe DatabaseComponent*

A classe *DatabaseComponent* foi criada com o objetivo de fornecer um componente de alto nível que simplificasse para os plugins as operações em uma tabela específica. Essa classe será registrada no contexto da aplicação como um tipo QML com o nome de "*Database*"e permitirá aos plugins a utilizarem declarando um objeto QML. *DatabaseComponent* agrega uma instância da classe *Database* e delega as operações para esse objeto. No entanto, ela possui alguns atributos (*Q_PROPERTY*) que permitem aos plugins informar o nome da tabela no banco de dados, o nome da chave-primária do tipo string (quando a chave-primária não for numérica auto-incrementada), além de colunas que guardam objetos json. As colunas json evitam que os plugins tenham que fazer *parsing* nas views.

Outro atributo *totalItens* manterá atualizado o número de registros na tabela para fins de comparação com o número de itens disponível no serviço *REST* com os itens já baixados para o dispositivo, útil para paginação de dados na *view*. O diagrama a seguir, apresenta os atributos e métodos dessa classe:

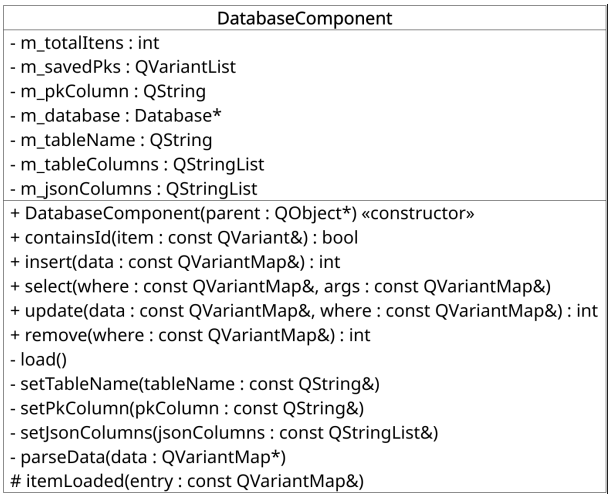

Figura 10: Diagrama da classe DatabaseComponent

É importante destacar que os métodos de busca, inserção, remoção e atualização foram simplificados para que os objetos informem apenas os parâmetros do método sem o nome da tabela. Outro recurso importante desse componente é que o método de busca é assíncrono e os resultados de uma busca, quando houver, serão retornados através do sinal *itemLoaded* que passará como argumento, um objeto no estilo *nome-da-coluna.valor*. *DatabaseComponent* possui ainda o método *containsId* que verifica se um item (pela chave primária) já está salvo localmente e pode ser útil para sincronizar com a *view* os itens já baixados do serviço *REST*.

O código a seguir, apresenta um exemplo de como um plugin poderá utilizar *DatabaseComponent* referenciada apenas como "Database"para persistir dados em uma tabela:

```
import "Database" 1.0
...
Database "{
    id: database
    jsonColumns: ["source"]
    tableName: "news"
    pkColumn: "title"
    onItemLoaded: listViewModel.append(entry)
}
...
Item "{
    Component.onCompleted: {
        var "args "=" {
            title: "GNU is not Unix!",
            message: "GNU is an operating ...",
            datetime: "2017-12-30T14:29:33"
            source: {
                url: "https://foo.com/api",
                code: "foo",
                value: "bar"
            }
        }
        database.insert(args)
    }
}
...
```

4.10 Notificações do Aplicativo

Notificações do aplicativo é um requisito funcional disponibilizado nesta arquitetura e o objetivo é permitir o envio de notificação ao usuário via *push* ou local (partindo do próprio aplicativo). As notificações via *push* é suportado apenas no Android e iOS e as notificações locais podem ser enviadas nos dispositivos móveis, linux desktop e MacOS. As notificações podem ser utilizadas para alertar o usuário de algum evento na aplicação e pode conter um título, uma descrição e nas plataformas mobile, pode vibrar o dispositivo e emitir um som. Outro recurso disponibilizado nas notificações é a possibilidade de adicionar algum dado extra em cada notificação. Esse dado deve ser um objeto no estilo *chave.valor*. As notificações locais serão enviadas pela própria aplicação via *procedure call* e as notificações via *push* através do *Firebase*. As subseções a seguir, descrevem detalhes de cada tipo de notificação.

4.10.1 *Notificações PUSH*

O suporte a notificações via *push* já está configurado na arquitetura e o que o desenvolvedor deve fazer para enviar mensagens via *push* é criar um projeto no *Firebase*, exportar o arquivo *google-services.json* e adicioná-lo no diretório *android* e o arquivo *google-services.xml* correspondente ao iOS no diretório *ios*, substituindo os arquivos existentes, ambos criados como exemplo. É importante destacar que o *package name* do aplicativo adicionado no *AndroidManifest.xml* e o *CFBundleIdentifier* no *Info.plist* do iOS, devem ser o mesmo utilizado ao criar o projeto no *Firebase*. Caso contrário, ocorrerá um erro ao construir o APK ou IPA, pois as bibliotecas correspondentes ao serviço de *push* (adicionadas ao projeto

durante o *build*) farão um *parsing* dos arquivos e abortará a compilação caso ocorra algum erro. No android, o *package name* deve ser adicionado manualmente no arquivo *build.gradle* presente no diretório *android* na propriedade *defaultConfig.applicationId*.

No android, o funcionamento de *push notification* requer duas classes java que serão instanciadas na inicialização do aplicativo e funcionarão como dois serviços. Essas classes já estão no projeto e o desenvolvedor não precisa modificá-las. O primeiro serviço registra o dispositivo no *Firebase* e retorna um token. O segundo serviço é um *listener* de notificações via *push* e ficará em execução mesmo que a aplicação seja fechada. Ao enviar uma notificação utilizando o console do *Firebase* ou através de algum *web service*, as notificações serão recebidas nesse serviço e enviadas para o *system tray* automaticamente. Se o aplicativo estiver em execução, a notificação será encaminhada para o aplicativo em um objeto json contendo todos os dados da mensagem, tais como, o título, a data e a hora de envio e o *payload* (um dado adicional não visível enviado na mensagem). Para adicionar o dado de *payload*, é necessário que o *web service* adicione um objeto json na requisição que é enviada ao *Firebase*[10].

No iOS, o arquivo *QtAppDelegate.mm* presente no diretório *ios* realizará o registro do dispositivo no *Firebase* e contém um método que receberá as notificações via *push* e re-encaminhará para a aplicação usando o mesmo evento utilizado no android. Em ambas as plataformas, o token será passado para a aplicação através do evento *newPushNotificationToken* e as mensagens de push, no evento *newPushNotification*. Se a aplicação estiver em execução e o usuário clicar em uma notificação colocando o aplicativo em foreground, o evento *newActionNotification* será disparado passando como argumento os dados de *payload* da mensagem.

A arquitetura dispõe do plugin de exemplo *Listeners* que contém o componente *PushNotificationRegister.qml*. Esse componente demonstra um exemplo de como obter o token de registro no *Firebase* e enviar para o serviço *REST do aplicativo*. O trecho de código a seguir, apresenta um exemplo de como acessar o token do *Firebase*.

```
import "Observer" 1.0

Observer "{
    id: ob
    objectName: "TokenObserver"
    event: Config.events.newPushNotificationToken
    onUpdated: {
        //enviando "o" token "para "o" web "service
        //eventData "=" o "argumento "do" evento
        //contendo "uma" string "com" o "token
        sendTokenToServer(eventData)
    }
    //adiciona "o" observador "ao" subject
    //para "o" evento "de" registro "do" token
    Component.onCompleted: Subject.attach(ob, event)
}
```

4.10.2 Notificações Locais

Para gerenciar as notificações locais, foi criado a classe *Notification* que abstrai a plataforma e dispõe de um método para enviar uma notificação para a área de notificações do sistema em execução. Essa classe será instanciada na inicialização do aplicativo e registrada no contexto da aplicação para que os plugins possam invocar o método *Notification.show*, passando o título e a mensagem da notificação e opcionalmente um objeto contendo o argumento a ser passado para a aplicação quando o usuário clicar na notificação. Quando o usuário clicar na notificação, o aplicativo ficará em *foreground* (caso não esteja) e o argumento da notificação será passado para o aplicativo através do evento *newActionNotification*.

O trecho de código a seguir, apresenta um exemplo de como exibir uma notificação local por algum objeto de um plugin, passando um título, uma mensagem e um objeto como argumento.

```
import "QtQuick" 2.8

Item "{
    Component.onCompleted: {
        var "title "=""Novo item carregado!"
        var "message "=""Clique para visualizar!"
        var "argument "="{
            key: "foo",
            value: "bar"
        }
        Notification.show(title, message, argument)
    }
}
```

4.11 Comunicação Entre os Objetos e Plugins

Esta seção descreverá o processo de comunicação entre objetos no aplicativo que está também disponível para os plugins. A comunicação entre os objetos facilitado é um requisito funcional desta arquitetura e consiste em disponibilizar uma API de alto nível para troca de mensagens entre objetos através de eventos e consiste na implementação do padrão de projeto *Observer*, com os eventos identificados por um nome. Essa API foi desenvolvida através das seguintes classes C++: *Subject* e a outra *Observer*. O nome do evento é uma string que identifica um evento específico e visa notificar apenas os objetos interessados, evitando o *broadcast* na aplicação e consequentemente aumentando o desempenho. Nessa API, é possível enviar um argumento para o objeto observador.

A classe *Subject* possui um atributo privado *attacheds*, uma instância da classe *QMap* do Qt que guardará um vetor de observadores para cada string que identifica determinado evento. Quando um objeto deseja observar um evento, ele deve primeiramente, informar o nome do evento que deseja ser notificado, e registrar-se no *Subject*. Quando determinado objeto deseja notificar observadores, ele deverá invocar o método *Subject.notify* passando no primeiro parâmetro uma string que identifica o evento, seguido de um objeto contendo o argumento a ser passado para os observadores.

A classe *Subject* será instanciada na inicialização e registrada no contexto da aplicação identificada como *"Subject"*. Os seguintes métodos são públicos e podem ser invocados pelos plugins:

- *attach* (void): Esse método poderá ser utilizado para adicionar observadores a uma lista de eventos na aplicação e dois parâmetros são requeridos: o primeiro é uma referência para o *observer* e o segundo, é uma lista de strings contendo os eventos que o observador deseja ser notificado;
- *detach* (void): Esse método poderá ser utilizado para remover um observador de uma lista de eventos na aplicação e dois parâmetros são requeridos: o primeiro é um ponteiro para o *observer* e o segundo é a lista de eventos da qual o observador será removido;
- *notify* (void): Esse método poderá ser utilizado por qualquer objeto da aplicação para notificar observadores de um determinado evento. Esse método requer dois parâmetros, o primeiro é uma string contendo o nome do evento, o segundo é

[10] https://firebase.google.com/docs/cloud-messaging/concept-options?hl=pt-br

um objeto variante contendo o dado a ser passado para o observador. Ao chamar esse método, o *Subject* irá iterar o vetor de observadores correspondente ao evento solicitado, chamando para cada observador o método público *update* que receberá como argumento, o nome do evento em que está sendo notificado e o argumento enviado pelo emissor.

O desenvolvedor poderá adicionar os eventos que deseja utilizar entre os plugins no arquivo de configuração, na propriedades *events* e utilizá-los via acesso ao objeto *Config*. Os eventos são basicamente strings no estilo e o acesso pode ser feito da seguinte forma: *Config.events.event_name*. Outro detalhe é que *Subject* criará uma *thread* para cada *observer* quando for notificá-los de algum evento. Isso significa que a chamada ao método *update* em cada *observer* será assíncrono para garantir o melhor desempenho. O código a seguir, apresenta um exemplo de como utilizar o *Observer* por objetos nos plugins.

```
import "Observer" 1.0
...
Observer "{
    id: observer
    objectName: "obs1"
    events: [Config.events.newSourceAdded]
    onUpdated: database.insert(eventData)
}
...
Item "{
    Component.onCompleted: {
        //pede "ao" Subject "para" adicionar
        //o "observador" a "lista" do "evento
        var "evt" = "Config.events.newSourceAdded
        Subject.attach(observer, evt)
    }
}
...
```

O código a seguir, apresenta um exemplo de como notificar observadores de um determinado evento:

```
import "QtQuick" 2.8
...
Item "{
    id: rootItem
    Component.onCompleted: {
        var "event" = "Config.events.newSourceAdded
        var "args" = "{
            key: "foo",
            value: "bar"
        }
        Subject.notify(event, args, rootItem)
    }
}
```

4.12 O Perfil de Usuário

A arquitetura dispõe o componente *UserProfile.qml* para gerenciar os dados do usuário do aplicativo, ele será instanciado na inicialização da aplicação e setado no objeto *userProfile* se houver login na aplicação, ou seja, se o desenvolvedor definir nas configurações a propriedade *usesLogin* para *true*. Esse objeto (*userProfile*) estará disponível para os plugins como um objeto global, e a propriedade *profile* poderá ser utilizada em *bindings* com outros objetos. No entanto, atualizações no perfil do usuário não devem ser feitas diretamente na propriedade *profile* e sim, através de eventos.

O componente *UserProfile.qml* observará três eventos na aplicação pelo qual os plugins devem utilizá-los para setar ou editar as informações do usuário. O primeiro evento *initUserProfile*, poderá ser utilizado para inicializar o perfil do usuário após o login e o argumento do evento deve conter um objeto enviado pelo serviço *REST* contendo no mínimo os campos *id* e *email*. O segundo evento *setUserProfileData*, deve ser utilizado para atualizar ou adicionar alguma informação ao perfil do usuário. Nesse evento, o argumento deve conter as propriedades *key* com o nome do campo a ser atualizado (ou adicionado) e *value* a informação para o campo correspondente. O terceiro evento *logoutApplication*, deve ser disparado pela página de *logout* para avisar que a seção foi encerrada e o usuário será direcionado para a página de login. No *logout*, o argumento do evento pode ser *false* ou simplesmente *null*. O componente *UserProfile* dispõe das seguintes propriedades:

- *profile* (object): Um objeto javascript que guardará os dados do perfil do usuário, no estilo *campo.valor*. Os plugins podem fazer *binding* com esse objeto em elementos visuais tais como, exibir a imagem de perfil através do campo *image_url* (via *profile.image_url*). Esse objeto será persistido a cada alteração;

- *profileName* (string): Uma string contendo o nome do perfil do usuário, por exemplo, *administrator*, *editor*, *student* etc. Essa propriedade será setada internamente quando o *profile* for definido ou alterado. No entanto, o nome do perfil do usuário deve ser passado pelo serviço *REST* na resposta do login. É importante destacar que essa propriedade será definida somente se o serviço *REST* adicionar no json (do perfil do usuário, retornado no login) a propriedade *permission* como um objeto contendo *name* ou *description*. A exibição das páginas para o usuário será baseado nessa propriedade através de *bindings* nos menus do aplicativo. Se a propriedade *roles* na configuração das páginas dos plugins for definido e *profileName* for uma string vazia o usuário não visualizará as páginas no menu do aplicativo;

- *isLoggedIn* (bool): Essa propriedade será definida para *true* quando *profile* for modificado contendo alguma informação válida, ou seja, *id* e *email* (no mínimo). *isLoggedIn* será *false* quando não houver informações em *profile*. Essa propriedade será modificada internamente após os eventos *initUserProfile* e *logoutApplication*. O valor de *isLoggedIn* será persistido sempre que for modificado.

O objeto *userProfile* invocará uma função interna que carregará a página inicial (*home page*) após *profile* ser modificado, ou seja, após o evento *initUserProfile*. A página de login também será carregada após o evento *logoutApplication*. A arquitetura disponibiliza o plugin de exemplo *Login* contendo alguns arquivos que demonstram a utilização do *login*, *logout*, exibição do perfil e como solicitar alterações nos dados do usuário através do evento *setUserProfileData* (o arquivo ProfileEdit.qml).

É importante destacar, que após o sucesso do primeiro login, não será necessário logar novamente até que o usuário faça logout ou limpe o cache do aplicativo nas configurações do dispositivo. Além disso, a imagem presente em *assets/default_user_profile.svg* será setado no perfil do usuário após o login, na propriedade *image_url* e o desenvolvedor poderá sobrescrever esse arquivo por uma imagem de sua preferência.

O código a seguir, apresenta um exemplo de como utilizar o perfil do usuário através do objeto *userProfile*, exibindo o informações como email e imagem do usuário.

```
import "QtQuick" 2.8
...
Column "{
    spacing : "10

    // userProfile "eh "um "objeto "global ,
    // criado "e "declarado "no "main "window !
    property "var "profileData : userProfile . profile

    // exibindo "a "imagem "de "perfil
    Image "{
        id : userImg
        asynchronous : true ; cache : true
        source : profileData . image_url
    }

    // exibindo "o "email "do "usuario
    Label "{
        id : userEmail
        text : profileData . email
    }
}
```

4.13 O Componente BasePage

O componente *BasePage* é um arquivo genérico que fornece algumas propriedades para as páginas dos plugins, além de fazer *bindings* com o *ToolBar*, o *TabBar*. *BasePage* deverá ser utilizada pelas páginas de plugins sempre que possível. Esse componente irá instanciar um *ListView* contendo um *ListModel* e um *RequestHttp* para facilitar o trabalho do programador e reduzir a escrita de código pelos plugins. *BasePage* pode ser visto como uma classe abstrata que possui alguns objetos agregados e métodos públicos prontos para uso pelos componentes que estenderem *BasePage*. As propriedades a seguir, compõem o *BasePage* e devem ser utilizadas para customizar o layout da página.

- *toolBarButtons* (array): uma lista de objetos contendo as propriedades de um botão *ToolBarButton* a serem adicionados no *ToolBar* dinamicamente, quando a página for carregada ou quando ficar visível para o usuário. Cada objeto deve conter as seguintes propriedades: *iconName*, uma string contendo o nome de um ícone do *Awesome Icons* e *callback* uma função javascript que será invocada quando o botão for pressionado pelo usuário;

- *toolBarState* (string): essa propriedade poderá ser utilizada para definir o estado do *ToolBar* e três valores estão disponíveis. O primeiro deles é *normal* que é o valor *default*. O segundo valor é *goBack* e quando for utilizado, permitirá ao usuário sair da página atual e retornar para a página anterior clicando em uma seta para a esquerda, essa seta será adicionada pelo *ToolBar*. O último valor é *search* que exibirá um campo de busca no *ToolBar* permitindo ao usuário digitar um texto para pesquisar algo na *view* da página ativa. No modo *seach*, a propriedade *searchText* também de *BasePage* receberá uma cópia do texto digitado pelo usuário e o plugin deverá criar uma conexão com *searchText* para realizar a busca do conteúdo solicitado pelo usuário;

- *absPath* (string): essa propriedade deverá ser definida para que o menu declare um *bind* entre o item correspondente (na lista de itens do menu, tornado-o selecionado) com a página atualmente vista pelo usuário. Para setar essa propriedade, a página poderá utilizar o *path* do plugin mais o nome do arquivo, por exemplo: *Config.plugins.plugin_name* + o nome do arquivo QML correspondente a página. Outro exemplo, considerando um plugin chamado *LoadMessages* e a página *View.qml*, essa propriedade pode ser definida da seguinte forma: *absPath: Config.plugins.loadmessages* + *"View.qml"*;

- *showToolBar* (bool): essa propriedade deverá ser utilizada se a página deseja ocultar o *ToolBar*;

- *showTabBar* (bool): essa propriedade deverá ser utilizada quando a página precisar ocultar o menu do layout em linha, que é uma instancia do *TabBar* do *Quick Controls*;

- *hasNetworkRequest* (bool): essa propriedade é *true* por *default* e se mantida com o valor padrão, instanciará um objeto *RequestHttp* quando a página for carregada. Outra propriedade de *BasePage* associada a esta, é *requestHttp* que receberá uma referência para o objeto e poderá ser utilizada pela página para fazer requisições HTTP. No entanto, se a página não realizará requisições HTTP, deverá setar *false* nessa propriedade;

- *hasListView* (bool): essa propriedade é *true* por *default* e se mantida com o valor padrão, instanciará um *ListView* do *Quick Controls* e passará a referência para a propriedade *listView*. O *ListView* já terá um *ListModel* do QML que será atribuído a propriedade *listViewModel* também de *BasePage* e poderá ser utilizada pela página para fazer *append* ou remover itens da *view*;

- *isPageBusy* (bool): essa propriedade é *false* por *default* e fará um *bind* com o status de cada requisição HTTP feita pela página quando o objeto *RequestHttp* for instanciado. Logo, o *bind* será criado somente se a página manter *hasNetworkRequest* como *true*;

- *isActivePage* (bool): essa propriedade fará um *bind* com o *window.currentPage* que é uma referência para a página atualmente vista pelo usuário;

- *listViewDelegate* (Component): essa propriedade é *null* por *default* e deve ser definida pela página quando utilizar *ListView* atribuindo ao *delegate* correspondente. Se *ListView* for instanciado, essa propriedade será atribuída ao *delegate* do *ListView*;

- *pageBackgroundColor* (Color): essa propriedade pode ser utilizada para definir uma cor de fundo personalizada para a página, pode ser tanto hexadecimal como RGBA. A cor padrão utilizada será o valor definido no arquivo de configuração na propriedade *theme.pageBackgroundColor*.

É importante destacar que *BasePage extends* o componente *Page* do *Quick Controls* e as propriedades definidas em *Page* serão herdadas, como por exemplo, *title*, que deve ser definido para exibir ao usuário o título da página que ele está visualizando. Para utilizar *BasePage*, basta adicionar a diretiva *import "qrc:/publicComponents"* e declarar um objeto *BasePage* como no exemplo a seguir:

```
import "QtQuick . Controls "2.0
import ""qrc :/ publicComponents /""as "Components

Components . BasePage "{
    id : page

    // ignora "o "uso "do "ListView
    hasListView : false
```

```
    //define o path absoluto
    absPath: Config.plugins.pages+"Page1.qml"

    //exibe um titulo no ToolBar
    title: qsTr("Page 1")

    //permite sair da pagina atual
    //usando o botao voltar no ToolBar
    toolBarState: "goBack"

    //trata as respostas de um pedido http
    onRequestFinished:{
        console.log("status:",statusCode)
        console.log("response:",response)
    }
...
}
```

4.14 Componentes Reutilizáveis

A arquitetura dispõe de dez componentes reusáveis para os plugins, alguns são visuais e outros não. Para utilizá-los basta adicionar a diretiva *import "qrc:/publicComponentes/"* e declarar um objeto QML. Os componentes disponíveis serão descritos a seguir:

- *AwesomeIcon.qml*: esse componente consiste utiliza um arquivo de fonte OTF contendo 720 ícones da biblioteca *Awesome Icons* e poderá ser utilizado para exibir um ícone clicável nos fragmentos de uma página. As propriedades *name*, *size* e *color* podem ser utilizadas para definir o ícone, o tamanho e a cor do ícone;

- *BasePage.qml*: esse componente é o elemento descrito na seção anterior e pode ser utilizado pelos plugins nas páginas no aplicativo;

- *CameraCapture.qml*: esse componente pode ser utilizado para abrir a câmera do dispositivo móvel ou a *webcam* em laptops. Ao adicionar esse componente no *StackVIew*, ele inicializará a câmera disponível no dispositivo e permitirá ao usuário capturar uma imagem clicando em qualquer ponto da tela ou, utilizando o botão *photo* no rodapé da janela. Outros dois botões estarão disponíveis nos cantos da tela. O botão do lado esquerdo permitirá ao usuário alternar entre as câmeras frontal ou de fundo (se disponível), e o do lado direito, abre a janela para seleção de arquivos no dispositivo. Se o usuário capturar uma imagem com a câmera, a imagem capturada será salva em um diretório público do dispositivo e o evento *cameraImageSaved* será disparado passando como argumento, uma string com o *path* da imagem;

- *CustomListView.qml*: esse componente *extends* o *ListView* do *Quick Controls* e já dispõe de um *ListModel* que será instanciado e definido como *model* do *ListView*. Também será adicionado efeitos de transição, além de um objeto *ScrollIndicator* que exibirá uma barra de rolagem vertical dinamicamente;

- *Datepicker.qml*: esse componente pode ser utilizado para exibir um caledário contendo opção de seleção de dia, mês e ano. Após o usuário selecionar uma data, o sinal *dateSelected(int day, int month, int year)* será emitido pelo objeto;

- *ListItem.qml*: esse componente é um *ItemDelegate* do *Quick Controls* e permite adicionar até quatro elementos visuais, além de uma borda no rodapé que será utilizada como separador em uma lista de elementos. A utilização desse componente pode ser feita tanto como *delegate* de um *ListView* como em uma página dentro de um *ColumnLayout*. Os elementos visuais são: um ícone do *Awesome Icon* através da propriedade *primaryIconName* ou uma imagem setando o *source* na propriedade *primaryImageSource* e será posicionado no lado esquerdo e centralizado verticalmente.

  Ao lado direito do ícone (lateral esquerdo, se for definido), pode ser adicionado um texto e uma descrição através das propriedades *primaryLabelText* e *secondaryLabelText* respectivamente, um abaixo do outro, sendo a descrição com o *font-size* e opacidade reduzidos. O último elemento visual, é o mesmo do primeiro só que, posicionado no lado direito e pode ser um ícone do *Awesome Icon* ou uma imagem. A imagem em ambos os lados pode ser um endereço remoto ou a partir do *qrc*;

- *PasswordField.qml*: esse componente *extends* o *TextField* do *Quick Controls* e pode ser utilizado para exibir um campo de senha para o usuário, contendo um ícone do *Awesome* ao lado direito centralizado verticamente. O ícone permitirá exibir a senha digitada no campo quando for clicado, alternando o valor da propriedade *echoMode* entre *Password* e *Normal*;

- *PhotoSelection.qml*: esse componente é um *Popup* do *Quick Controls* e pode ser utilizado para exibir uma lista vertical de itens clicáveis para o usuário. Três opções estão disponíveis: A primeira opção, ao ser clicada, abre a câmera do dispositivo, ou seja, irá instanciar *CameraCapture.qml*. A segunda opção abrirá a galeria de imagens do dispositivo para seleção de um único arquivo. A terceira opção, ao ser clicada, nada será feito internamente, apenas emitirá o sinal *removeCurrentPhoto* para que algum plugin possa executar essa ação de remover a imagem de perfil do usuário;

- *RoundedImage.qml*: esse componente exibe uma imagem circular e dispõe as propriedades *imgSource* para o *path* da imagem (local ou remota) e *borderColor* para definir uma cor para a borda (o padrão é transparente). A imagem será adicionada em um retângulo com um *MouseArea*, tornando possível capturar eventos de *click* ou pressionamento na imagem;

- *TimePicker.qml*: esse componente exibirá um relógio baseado em *hora:minuto:segundos*. Ele *extends Popup* do *Quick Controls* e para abrir o diálogo, é preciso chamar o método *open()* a partir do objeto declarado. Quando o usuário selecionar a hora e clicar no botão *OK*, o sinal *timeSelected(var time)* será emitido.

4.15 Fluxo de Execução

A figura exibida no final desta seção, demonstra o *workflow* de um aplicativo baseado nesta arquitetura. As aplicações Qt possuem nas plataformas mobile, objetos *Java* e *ObjectiveC* no android e iOS respectivamente, que inicializam o aplicativo em cada plataforma e em seguida executam a aplicação tendo como ponto de entrada o *main.cpp*.

Os elementos visuais serão carregados no *main.qml* que será instanciado no último passo antes de iniciar o *loop* da aplicação. O *main.qml* corresponde a *ApplicationWindow* do *Quick Controls* e poderá ser acessado por qualquer objeto através do id *window*. Os *listeners* serão carregados quando *window* estiver pronto e poderão acessar qualquer uma das propriedades declaradas no escopo do *window*, tais como *userProfile*. O *window* contém dois *widgets* que

podem ser utilizados para exibir avisos ao usuário. Os *widgets* são *Snackbar*[11] e *Toast* e foram inspirados nos respectivos componentes do *Material Design*[12]. Os *widgets* dispõe da função *show* que exige uma string como parâmetro, essa string será o texto exibido ao usuário.

A figura a seguir, apresenta de uma forma resumida o fluxo de inicialização e a sequência de objetos instanciados em um aplicativo baseado nesta arquitetura.

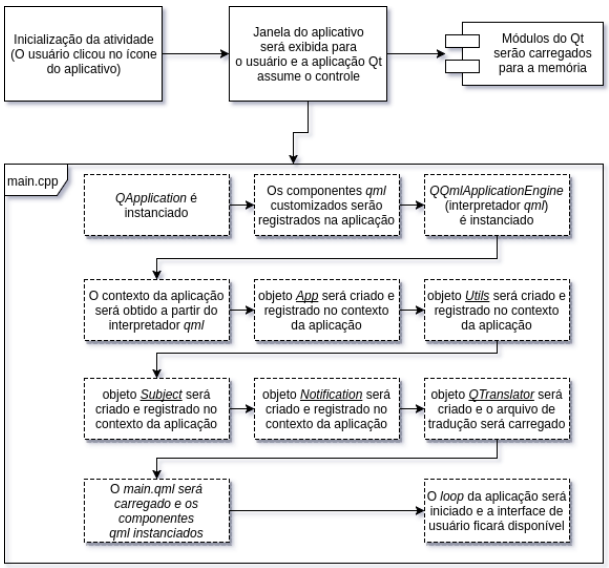

Figura 11: *workflow* da execução de um aplicativo Qt

4.16 Processo de Instanciação da Arquitetura

Para utilizar a arquitetura desenvolvida, é necessário seguir uma determinada ordem de atividades que serão descritas abaixo. O código fonte da arquitetura encontra-se em uma página do *github*, no link [3] e é necessário utilizar o *git* para obter uma cópia e iniciar a criação de um aplicativo. É preciso configurar o ambiente de desenvolvimento Qt para android ou iOS, além de baixar o android *SDK* e o *NDK*. No primeiro *build* do aplicativo (na versão para android), o *Gradle* baixará algumas bibliotecas para a máquina do desenvolvedor requeridas para o funcionamento do *push notification*. Para criar um aplicativo usando esta arquitetura basta seguir o passo-a-passo abaixo:

1. Acessar a página do projeto no *github* e criar um *fork* para a conta do desenvolvedor.

2. Em seguida, basta clonar o projeto para a computador que será utilizado. É possível também, clonar a partir da página do projeto e alterar a url de origem.

3. Após finalizar o processo de clone via git, renomear a pasta clonada e o arquivo *tcc.pro* para o nome do aplicativo a ser desenvolvido.

4. Importar o projeto (arquivo .pro) no *QtCreator* e configurar as plataformas suportadas pelo aplicativo.

5. Abrir o arquivo *config.json* presente na raíz do projeto (manualmente usando algum editor) ou pelo *QtCreator* na aba Resources *config.qrc/config.json* e configurar as propriedades do aplicativo como foi descrito na seção 4.10. Os valores pré-definidos foram setados apenas como exemplo.

6. Abrir o arquivo *AndroidManifest.xml* no *QtCreator* e renomear o *package name*, de *org.qtproject.example* para o nome do pacote do aplicativo a ser desenvolvido.

7. Para utilizar *push notification*, é necessário criar um projeto do *Firebase* e adicionar o suporte a *push notification*[13]. Em seguida, configurar as opções do projeto do *Firebase* e no final do *wizard* de configuração é necessário exportar o arquivo *google-services.json* e salvar na pasta *android* e IOS substituindo os arquivos existentes, também criados apenas como exemplo.

8. No android, é necessário mais um passo para habilitar o *push notification*. É preciso editar o arquivo *android/build.gradle* e renomear o valor da propriedade *defaultConfig.applicationId* (linha 71) para o *package name* do aplicativo.

9. Para customizar as cores do aplicativo no android (*Action Bar*, *Status Bar* e na janela do aplicativo), basta editar o arquivo *android/res/values/colors.xml*. A cor definida em *colorPrimary* será utilizada como cor de fundo no ícone de notificações via *push* ou local.

10. Em seguida, adicionar os ícones do aplicativo no diretório *android/res/drawable-(*)dpi* e *ios/icons*. O projeto vem com ícones nos tamanhos ideais (para android). No android, é possível utilizar alguma ferramenta online[14] para gerar os ícones. No iOS, é recomendável utilizar a interface do *XCode* para importar os ícones, eles serão mapeados no arquivo xml *Info.plist* que é equivalente ao *AndroidManifest.xml*.

11. Se algum plugin precisar exibir a logo ou o ícone do aplicativo, deve ser adicionado uma cópia de um ícone na pasta *assets* que está na raiz do projeto. Os ícones da pasta *android* ou *ios* não serão acessíveis por objetos da aplicação e o ícone nesta pasta será utilizado em modo desktop. Neste mesmo diretório, está a imagem *drawer.jpg* utilizada no menu do layout em pilha, ela será carregada se a propriedade *showDrawerImage* for *true* no arquivo de configuração e pode ser substituída pelo programador. A imagem *default_user_image.svg* será utilizada no perfil do usuário e também pode ser substituída por outra que combine com o design do aplicativo.

12. O último passo é implementar as *features* do aplicativo através de plugins. É possível iniciar o desenvolvimento de um plugin a partir dos exemplos fornecidos. Os plugins não serão mapeados em arquivo qrc sendo necessário criá-los manualmente, mantendo-os em uma pasta com o nome do plugin seguido de um arquivo *config.json*, além de aquivos QML e imagens. Tudo que tiver na pasta do plugin será empacotado no APK ou IPA. Não é recomendável utilizar o *wizard* de adição de arquivos do *QtCreator*, pois ele adicionará os arquivos em *public.qrc* ou *private.qrc*, que contém os componentes internos e reutilizáveis criando um acoplamento entre plugins e núcleo, violando um requisito não funcional desta arquitetura.

[11] https://material.io/guidelines/components/snackbars-toasts.html
[12] https://material.io/guidelines
[13] https://console.firebase.google.com/project/novo-projeto-do-firebase/notification
[14] https://jgilfelt.github.io/AndroidAssetStudio/icons-launcher.html

5. AVALIAÇÃO EXPERIMENTAL

Esta seção apresentará o estudo conduzido para avaliar a arquitetura proposta neste trabalho. A avaliação consistiu no desenvolvimento de duas versões de um aplicativo móvel e na coleta de métricas relacionadas ao desenvolvimento de cada versão. As métricas foram utilizadas para validar a eficácia e os benefícios da arquitetura proposta neste trabalho no desenvolvimento de aplicativos móveis. As subseções a seguir, apresentarão os detalhes do estudo conduzido na avaliação.

5.1 Planejamento

O objeto de estudo utilizado na avaliação foi o aplicativo *Emile*. O *Emile* consiste de um sistema para facilitar a comunicação acadêmica, permitindo aos professores enviar mensagens aos alunos de suas turmas de eventos diversos que possam ocorrer durante o semestre letivo. O Emile é um sistema desenvolvido pelo GSORT – um grupo de pesquisa do Instituto Federal da Bahia e mais detalhes sobre ele pode ser encontrado na página do projeto [5].

O processo da avaliação consistiu na implementação de três *features* do aplicativo em duas versões separadas, sendo uma com o framework proposto neste trabalho e outra versão escrita sem utilizar o framework. A versão sem utilizar o framework também foi escrita em Qt/QML e todos os elementos visuais utilizados na versão com o framework foram implementados na versão sem o framework para que o estilo do aplicativo fosse padronizado em ambas as versões. As features escolhidas estão descritas a seguir:

- Login do usuário. Esse recurso inclui o logout para permitir que o usuário possa encerrar uma seção.
- Gerenciamento de mensagens. Esse recurso inclui o envio de mensagens usando um perfil de professor e a visualização das mensagens enviadas pelo professor, além de visualização das mensagens recebidas por um aluno, ou seja, o aluno também poderá logar no aplicativo e visualizar as mensagens enviadas para a turma a qual ele está matriculado.
- Gerenciamento de perfil do usuário. Esse recurso inclui a visualização e a edição dos dados do usuário. No entanto, apenas a edição dos campos email e senha foram suportados.

O objetivo do estudo era extrair métricas que pudessem destacar as vantagens e os benefícios de utilizar a arquitetura desenvolvida neste trabalho. As métricas escolhidas na avaliação serão descritas a seguir:

- Número de linhas de código implementado em cada versão.
- Densidade de bugs encontrados em cada versão.
- Número de alterações realizadas na versão com o framework, necessárias para poder utilizá-lo.

5.2 Execução e Coleta de Dados

A avaliação foi realizada no decorrer de quinze dias e primeiramente foi implementado a versão sem o framework. Uma sequência de passos deu início a configuração do aplicativo e as *features* nesta versão, foram implementadas através de três plugins, sendo um para cada uma das *features* escolhidas para avaliação. É importante destacar, que foi utilizado um serviço REST como servidor do aplicativo. O serviço é necessário para login e obtenção dos dados do usuário, além de listagem das turmas para envio de mensagem pelo professor. Este *web service* foi utilizado na avaliação pelas duas versões *Emile1* e *Emile2* e foi implementado em um serviço de host gratuito chamado *pythonanywhere* e pode ser acessado através do endereço *enoque.pythonanywhere.com*. O *web service* não faz parte da avaliação e constitui toda lógica de negócio inerente ao funcionamento do Emile. A sequência de passos realizado na versão com o framework será descrita abaixo, ela descreve as alterações realizadas para utilizar a arquitetura descrita neste artigo:

1. Foi realizado o clone do projeto da arquitetura para o computador utilizado no desenvolvimento, através do link contido no github, na página do projeto.
2. Em seguida, a pasta do projeto foi renomeada de *tcc* para *Emile1*.
3. O arquivo de projeto Qt *tcc.pro* foi renomeado para *Emile1.pro*.
4. Realizado edições no arquivo de configuração do framework (*config.json*), para atualizar as propriedades *applicationName*, *organizationName* e *organizationDomain* para os dados referentes ao aplicativo.
5. Adicionado os dados nas propriedades *baseUrl*, *userName* e *userPass* correspondente a API REST do aplicativo, também no arquivo de configuração do framework.
6. Foi deletado os plugins *About*, *AppendPage* e *Pages*, além do arquivo *Listeners/Listener1.qml*, ambos disponibilizados como exemplo no framework.
7. Alterado o arquivo *Listeners/config.json*, onde foi removido uma linha do arquivo.
8. Adicionado a pasta *Messages* (representando o plugin *Messages*) para implementação da *feature* de gerenciamento de mensagens (exibição e envio).
9. Adicionado o arquivo *config.json* na pasta do plugin *Messages*.
10. Adicionado os arquivos: *CourseSectionSelectPage.qml*, *DestinationGroupSelectPage.qml*, *SendMessagesPage.qml*, *ViewMessagesPage.qml*, *plugin_functions.js*, *plugin_table.sql* e *ViewMessagesPageDelegate.qml* no plugin *Messages* contendo a lógica de negócio para seleção de destinatário, escrita da mensagem e listagem de mensagens disponíveis para o usuário. Este plugin utilizará persitência de dados local para permitir leitura das mensagens *offline*.
11. Alterado o arquivo *Messages/config.json* e adicionado as páginas visíveis do plugin, além da permissão do usuário (propriedade *roles*) requerida para acessar cada página.
12. Alterado a propriedade *order* do plugin *UserProfile* para o valor inteiro *5* para que a exibição da opção *Ver perfil* permanecesse abaixo das opções de envio e visualização de mensagens.
13. Adicionado os arquivos *LoadUserCourseSections.qml* e *LoadUserProgram.qml* no plugin *Login* para carregarem dados adicionais do usuário após o login.
14. Removido as strings presente na propriedade *roles* do plugin *UserProfile* e mantido somente os valores *teacher* e *student*.
15. Removido as strings presente na propriedade *roles* do plugin *Login* e mantido somente os valores *teacher* e *student*.

16. Adicionado a logo do aplicativo na pasta *assets* (que está na raiz do projeto), substituindo o ícone fornecido como exemplo.

17. Editado o arquivo *assets/qtquickcontrols2.conf* onde foi alterado as cores do aplicativo, nas propriedades *Accent*, *Primary*, *Foreground* e *Background*.

O framework dispõe de alguns plugins de exemplo que demonstram como utilizar alguns dos recursos disponibilizados na arquitetura, dentre eles, um plugin para login e outro para o gerenciamento do perfil do usuário e ambos foram reaproveitados na versão com o framework. Os plugins reaproveitados sofreram poucas alterações, tais como, a url de destino e os parâmetros da requisição de login e de atualização do perfil do usuário, onde foi adicionado os argumentos que são requeridos pelo *web service* do Emile. Ao todo, seis alterações foram feitas nestes dois plugins. Na versão sem o framework, as implementações foram feitas através de arquivos separados e acoplados na aplicação, ou seja, não foi criado plugins. No entanto, nessa versão, foi utilizado os mesmos componentes do *Quick Controls* que são utilizados no framework, tais como *Page*, *ListView*, *ListModel*, etc. O código fonte de ambas as versões (Emile1 e Emile2), está na página do projeto criado para a avaliação e pode ser adquirida utilizando o git, através da url *https://github.com/joseneas/avaliacao-tcc.git*.

Os testes de execução do aplicativo foram feitos seguindo uma sequência de passos com o objetivo de testar as três *features* implementadas para avaliação. Os testes foram feitos em um dispositivo Android, modelo Samsung J3 contendo a versão 5.1 de nome *Lollipop*. O mesmo teste foi realizado nas versões *Emile1* e *Emile2*. A lista a seguir, descreve a sequência de passos realizado:

1. O aplicativo foi aberto via click no ícone do aplicativo, na banjeija de aplicativos do sistema;

2. Com o aplicativo pronto para uso, foi realizado o login usando um perfil de professor;

3. Após o login, adicionado click no ícone de menu para visualização das páginas disponíveis;

4. Com o menu lateral aberto, foi solicitado acesso a página *Enviar mensagem* via clique na tela;

5. A página de seleção de destinatário foi aberta, e clicou-se na opção “Todas as turmas”;

6. A página de envio de mensagem foi aberta e uma mesagem foi enviada para todas as turmas;

7. Após o envio da mensagem, cliou-se na opção “Ver mensagem”no menu;

8. A página de listagem de mensagens foi aberta e a mensagem recém enviada estava disponível para leitura, indicando o destinatário e o texto submetido;

9. Adicionado click no ícone de menu para visualização das páginas disponíveis;

10. Selecionado a opção “Ver perfil”;

11. Adicionado a opção de edição do perfil e em seguida a página foi aberta;

12. Alterado o email do usuário e em seguida foi solicitado envio para o serviço REST;

13. Via console da IDE, observou-se a resposta com status *200-OK* do servidor, indicando que o email foi atualizado com sucesso.

A seção a seguir, descreverá os resultados obtidos na avaliação.

5.3 Resultados e Discussão

Após finalizar o desenvolvimento das versões *Emile1* e *Emile2*, observou-se que a versão sem o framework resultou em um arquivo final (APK) um pouco menor do que a versão com o framework, além do *build* do projeto ocorrer em menor tempo.

Em relação ao número ao linhas, na versão sem o framework foi considerado apenas os plugins, ou seja, foi efetuado a contagem somente dos arquivos presentes no diretório *plugins*, pois a implementação das features se deu somente através de plugins. A versão sem o framework foi efetuado a contagem de todos os arquivos. Para utilizar a contagem, foi utilizado o comando *shell*: *wc -l 'find <folder_path> -type f'*. Os resultados obtidos foram:

- Emile1 (versão com o framework): **1116** total de linhas.
- Emile2 (versão sem o framework): **5206** total de linhas.

Em relação ao bugs identificados em cada versão, destaca-se maior complexidade dos bugs na versão com o framework, pois era difícil de depurá-los, pois foram erros internos que exige do usuário do framework conhecimento do código fonte. Na versão sem o framework, foi indentificado os seguintes bugs:

1. Quando o dispositivo não estava conectado a Internet, as requisições HTTP ficavam intermináveis, o elemento visual *BusyIndicator* ficava visível por tempo indeterminado.

2. Erros de requisição HTTP não capturados quando ocorre exceção no servidor e o mesmo não responde um json válido, retornando um HTML.

3. O click no botão “voltar”do android, não retorna para a página anterior ou minimiza o aplicativo quando apenas uma página foi acessada pelo usuário. O que ocorre é o fechamento inesperado do aplicativo. Na versão com o framework esse recurso já está implementado internamente.

4. Não foi possível permitir que o usuário altere a imagem de perfil, selecionando uma imagem da galeria de arquivos do sistema, pois o QML não dispõe de um componente para acesso a galeria de imagens do dispositivo no android, sendo necessário implementar uma API em java ou C++ com JNI. Na versão com o framework esse recurso também já está implementado e pode ser utilizado via *procedure call*.

A lista a seguir, descreve os bugs encontrados na versão com o framework:

1. Após finalizar a implementação, em alguns momentos o aplicativo finalizou inesperadamente com a seguinte mensagem de erro: *QThread: Destroyed while thread is still running*. Um erro difícil de ser depurado, pois não há detalhes de onde e porquê ocorreu.

2. Após um determinado tempo utilizando o aplicativo, após logar, visualizar as mensagens e navegar em algumas páginas, algumas requisições não são iniciadas ou finalizadas. Por exemplo, após fazer o logout e tentar logar novamente o aplicativo não responde.

3. A precisão de cliks no *ToolBar* para abrir o *Drawer* Menu não é muito boa, em algums momentos é preciso clicak até três vezes para abrir o menu.

4. A nível de implementação, é preciso conhecer muito bem os detalhes do framework para utilizar bem os recursos, pois ao implementar a persistência de mensagens foi preciso olhar o código fonte da classe *DatabaseComponent* para saber o nome do parâmetro do sinal *itemLoaded* emitido após realizar uma consulta. Esse parâmetro contém os dados extraídos da consulta e era preciso acessá-lo para adicionar os dados da *query* na *view*. Isso é devido o fato do método *select* ser assícrono, emitindo o resultado através de um evento.

5. O build do projeto é muito mais lento do que a versão sem o framework e o tamanho do arquivo final (apk) é em torno de 20% maior.

6. CONCLUSÃO

Este artigo apresentou o projeto, implementação e documentação de uma arquitetura de software baseado em plugins para o desenvolvimento de sistemas de informação mobile. Destacou-se os recursos de baixo acoplamento entre objetos e facilidade de implementar features para comunicação com serviços *RESTful*. A arquitetura destaca-se rica em recursos para operações básicas como acesso a rede, persistência de dados e notificações ao usuário tanto via push como local, através de APIs de alto nível. É possível ainda desenvolver objetos de UI usando componentes de alto nível através do QML.

Também foi realizado uma avaliação da arquitetura através do desenvolvimento de um aplicativo em duas versões, uma com e outra sem a arquitetura com o objetivo de analisar as vantagens e os benefícios de utilizar o framework proposto neste trabalho. Através da avaliação, concluiu-se que a arquitetura carece de depuração facilitada e melhorias na API de acesso a rede, além de uma documentação para cada uma das features disponibilizadas.

A característica de plugins da arquitetura desacopla as features do aplicativo do núcleo da aplicação, o que permite ao desenvolvedor adicionar, modificar ou remover recursos com maior facilidade, já que a comunicação entre objetos e recursos devem ser realizadas através de eventos. Através dos plugins, o desenvolvedor não precisa modificar o núcleo do aplicativo quando precisar alterar ou adicionar alguma funcionalidade, além de obter atualizações do framework sem perder as alterações feitas. Outra característica, é que o desenvolvedor irá escrever muito menos código, pois o framework já implementa muitas funcionalidades que os plugins podem utilizar através de APIs de alto nível, evitando ter que escrever código java, C++ ou Objective C. A arquitetura dispõe ainda de dois estilos para o aplicativo, usando *containers* como *StackView* e *SwipeView* e permite ao desenvolvedor escolher qual usar editando apenas uma propriedade em um arquivo de configuração.

Os possíveis trabalhos futuros podem disponibilizar uma documentação das APIS da arquitetura, além de melhorar as features disponibilizadas. Os bugs identificados na avaliação também podem ser corrigidos e outros recursos podem ser adicionados no futuro a fim de incrementar funcionalidades através de APIs de alto nível a serem utilizadas pelos plugins. Dentre os trabalhos futuros, o suporte a plugins C++ é o maior dentre os desafios, pois permitiria aos plugins implementar features mais complexas através de implementações de mais baixo nível.

6.1 Limitações Deste Trabalho

A lista a seguir apresenta as limitações identificadas nesta arquitetura.

1. Não há suporte a plugins escritos em C++: esse recurso seria importante, pois permitiria aos plugins delegar a lógica de negócio e operações de baixo nível a objetos C++, em vez de componentes QML como é atualmente. O problema é que as classes C++ devem ser conhecidas em tempo de compilação, pois em um projeto Qt as classes C++ devem estar mapeadas no arquivo *.pro* gerando um acoplamento do núcleo da aplicação com os plugins;

2. Instalação de plugins é feita somente durante o *build:*: a arquitetura não suporta a instalação de plugins dinamicamente. Esse recurso permitiria estender as *features* do aplicativo sem precisar de um *rebuild* e novo *deploy*;

3. Há suporte somente a um tipo de autenticação na API de rede: A arquitetura suporta apenas *Basic Authentication* nas requisições HTTP e carece de outros tipos de autenticação suportados por *web services* RESTful, tais como *OAuth*, *BEARER*, *DIGEST Auth* entre outros;

4. Atualização da arquitetura: Para criar um aplicativo utilizando este projeto requer alteração em diversos arquivos e isso implica em obter as atualizações futuras da arquitetura, pois os arquivos modificados serão sobrescritos e o desenvolvedor terá que atualizá-los manualmente. Seria interessante que todas as alterações necessárias fossem feitas em arquivos separados e durante o *build* fosse feito o *merge* entre os arquivos do núcleo e os arquivos modificados pelo desenvolvedor. Em projetos android usando a versão mais recente do gradle, é possível fazer *merge* do arquivo *AndroidManifest.xml* e o próprio arquivo *build.gradle* que contém as APIs do *Firebase* e outras propriedades de instalação e deploy no android.

6.2 Trabalhos Futuros

Os itens abaixo, apresenta os possíveis incrementos futuros na arquitetura para facilitar ainda mais o desenvolvimento de plugins e melhorar a qualidade de um aplicativo baseado neste projeto.

1. Adicionar os componentes reusáveis em um módulo estilo *qmldir* para que o *import* não seja feito através de uma string via *qrc* e sim, via módulo similar ao *import* do *QtQuick*;

2. Adicionar o suporte a outros métodos na API de requisições HTTP tais como *PUT*, *OPTIONS*, *DELETE* e *HEAD*;

3. Atualizar a implementação de *push notification* para a API em C++ do *Firebase* que já está em versão estável. Isso irá simplificar a API de notificações via *push* evitando ligações com objetos java via *JNI*;

4. Melhorar o suporte ao iOS atualizando as bibliotecas do *Firebase* para a versão mais recente, pois o *xCode* exibe *warnings* de que a versão da biblioteca utilizada possui APIs *deprecated*, além de melhorar o suporte aos componentes visuais em telas retina declarando *font-size* e dimensões baseadas no *DPI* do dispositivo;

5. Permitir que durante o desenvolvimento em modo Desktop, as alterações nos arquivos dos plugins sejam identificadas e carregadas a cada execução da aplicação, não exigindo o *rebuild* do projeto, similar ao que acontece com os arquivos mapeados no *qrc*. Atualmente, os plugins são copiados para o diretório do executável somente durante o *build*;

6. Melhorar o suporte a depuração, com mensagens de erros e exceções mais detalhados nas APIs disponibilizadas (notificação, rede e persistência de dados).

7. Desenvolver uma documentação com exemplos de código para cada *feature* da arquitetura, com o objetivo de facilitar o desenvolvimento de plugins.

7. REFERÊNCIAS

[1] About qml applications. Acessado em: 06/01/2018. Disponível em: http://doc.qt.io/qt-5/qmlapplications.html.

[2] About qt. Acessado em: 06/01/2018. Disponível em: https://en.wikipedia.org/wiki/Qt_(software).

[3] Código fonte da arquitetura. Disponível em: https://github.com/joseneas/tcc.

[4] Design da arquitetura de componentes. Acessado em: 15/12/2017. Disponível em: https://www.dimap.ufrn.br/“jair/ES/c7.html.

[5] Emile mobile | gsort. Acessado em 10/03/2018. Disponível em: http:/http://emile.ifba.edu.br/.

[6] Gerenciando projetos com pmbok. Acessado em 14/08/2017. Disponível em: http://www.governancadeti.com/2011/03/gerenciando-projetos-com-pmbok/.

[7] O que é um aplicativo móvel? Acessado em: 11/07/2017. Disponível em: http://blog.stone.com.br/aplicativo-movel.

[8] Qt toolkit. Acessado em: 06/01/2018. Disponível em: http://www.linuxjournal.com/article/201.

[9] Qt – cross-platform software development for embedded and desktop. Acessado em: 06/01/2018. Disponível em: https://www.qt.io.

[10] Reference model for service oriented architecture. oct 2006. Acessado em: 10/01/2018. Disponível em: https://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html.

[11] U.”Acer, A.“Mashhadi, C.”Forlivesi, and F.“Kawsar. Energy efficient scheduling for mobile push notifications. In *Proceedings of the 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services on 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 100–109. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2015.

[12] E.”Christensen, F.“C. IBM”Research, G.“M. Microsoft, and S.”W. IBM“Research. Web services description language (wsdl) 1.1. Acessado em: 24/07/2017. Disponível em: https://www.w3.org/TR/wsdl.

[13] D.”F. D’Souza and A.“C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[14] R.”H.“B. Feijó. Uma arquitetura de software baseada em componentes para visualização de informações industriais. *Programa de Pós-Graduação em Engenharia Elétrica do Centro de Tecnologia da UFRN*, 2007.

[15] R.”T. Fielding. Architectural styles and the design of network-based software architectures. Acessado em: 26/07/2017. Disponível em: http://dl.acm.org/citation.cfm?id=932295.

[16] C.“Hofmeister, R.”Nord, and D.“Soni. *Applied Software Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, nov 2000.

[17] L.”Y.“M. Jun, Y.;”Zhishu. Json based decentralized sso security architecture in e-commerce. *In: Proceedings of the 2008 International Symposium on Electronic Commerce and Security - ISECS ’08, Washington, DC, USA: IEEE Computer Society*, page 471–475, 2008.

[18] J.“D.”C. Júnior. Solução multiplataforma para smartphone utilizando os frameworks sencha touch e phonegap integrado À tecnologia web service java, 2014.

[19] A.“H. Kronbauer, C.”A.“S. Santos, and V.”Vieira. Um estudo experimental de avaliação da experiência dos usuários de aplicativos móveis a partir da captura automática dos dados contextuais e de interação. In *Proceedings of the 11th Brazilian Symposium on Human Factors in Computing Systems*, IHC ’12, pages 305–314, Porto Alegre-RS, Brazil, 2012. Brazilian Computer Society.

[20] K.“C. L. J.”P. Laudon. *Sistemas de Informação*, volume Unico. LTC, Rio de Janeiro, Brasil, fourth edition, 1999.

[21] S.“M. and G.”D. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[22] J.“McCarthy and P.”Wright. Technology as experience. *interactions*, 11(5):42–43, sep 2004.

[23] M.“MELOTTI. Creama: Uma arquitetura de referência para o desenvolvimento de sistemas colaborativos móveis baseados em componentes., 2014.

[24] C.”Pabllo, R.“Soto, and J.”Campos. Mobile medication administration system: Application and architecture. In *Proceedings of the 2008 Euro American Conference on Telematics and Information Systems*, EATIS ’08, pages 41:1–41:4, New York, NY, USA, 2008. ACM.

[25] C.“Perepletchikov, M.;”Ryan. A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software. *IEEE Transactions on Software Engineering*, page 449–465, 2011.

[26] L.“Procedi. Avaliação do framework xamarin.forms para desenvolvimento de aplicativos móveis multiplataforma, criando uma aplicação real., 2016.

[27] V.”J. d.“S. Rodrigues. Moca: Uma arquitetura para o desenvolvimento de aplicações sensíveis ao contexto para dispositivos móveis, 2004.

[28] C.”Szyperski, J.“Bosch, and W.”Weck. Component-oriented programming. *Object-Oriented Technology ECOOP’99 Workshop Reader Lecture Notes in Computer Science*, page 184–192, 1999.

[29] R.“N. Taylor, N.”Medvidovic, and E.“M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[30] B.”G. e. L.“G. Valéria”Feijó. Heurística para avaliação de usabilidade em interfaces de aplicativos smartphones: utilidade, produtividade e imersão. *Design e Tecnologia*, 3(06):33–42, 2013. Acessado em: 22/07/2017. Disponível em: https://www.ufrgs.br/det/index.php/det/article/view/141.

[31] A.“P. O.”S. Vermeeren, E.“L.-C. Law, V.”Roto, M.“Obrist, J.”Hoonhout, and K.“Väänänen-Vainio-Mattila. User experience evaluation methods: Current state and development needs. In *Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries*, NordiCHI ’10, pages 521–530, New York, NY, USA, 2010. ACM.

[32] S.”P. Zambiasi. Uma arquitetura de referência para softwares assistentes pessoais baseada na arquitetura orientada a serviços. *Tese (doutorado) - UFSC, Centro Tecnológico. Programa de Pós-Graduação em Engenharia de Automação*

e Sistemas, (77):331, 2012.