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 popular 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, além de componentes reusáveis de alto nível para recursos básicos como acesso a rede, persistência de dados, notificações do aplicativo e comunicação entre objetos através de eventos.

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 além de 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 [27]. 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 contrução de objetos de *UI* pelos plugins.

Este trabalho está organizado como segue. A seção 2 apresenta o referencial bibliográfico. Em seguida, na seção 3, será apresentado 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. REFERENCIAL BIBLIOGRÁFICO

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 Aplicati-

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

[1] *User Interface* ou interface do usuário

vos 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 [8]. 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 [7]. 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* [20], 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 [29]. Com base nessa arquitetura, o desenvolvedor projeta, desenvolve e configura uma aplicação prototipando-a por meio de componentes reutilizáveis [29]. 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 [15]. 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 [9].

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 [19].

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* [23]. Neste trabalho, a arquitetura foi concentrada apenas nas camadas de interação, aplicação e *middleware*.

2.5.1 Camada de Interação Humana-Computador

A camada de Interação Humana-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) [23]. Um exemplo de um com-

ponente 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* [23]. 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 [5]. O projeto termina quando seus objetivos forem alcançados ou quando existirem motivos para não continuá-lo [5].

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 [6]. 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 [18]. Os aplicativos devem estar em conformidade com a personalidade, preferências, objetivos, experiências e conhecimento de seus usuários [28]. Além disso, o contexto físico, social e virtual onde ocorrem as interações deve, sempre que possível, ser levado em consideração [21].

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 [13]. 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 [13]. 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 [26].

O reuso de componentes é um recurso desta arquitetura, pois dispõe de dezesseis componentes (visuais e não visuais) para auxiliar no desenvolvimento de novos aplicativos. Esses componentes são arquivos QML que podem atender a diferentes customizações através das propriedades disponibilizadas pelos objetos internos de cada componente. Os benefícios da componentização estão ligados a manutenibilidade, reúso, extensibilidade e escalabilidade [12].

2.8 Web Services, RESTful, Push Notification e JSON

Web Services constituem uma tecnologia emergente da Arquitetura Orientada a Serviços (SOA) [24]. 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* [11].

RESTFul é um estilo arquitetural para a construção de sistemas distribuídos [14]. O elemento fundamental da arquitetura *RESTful* é o *resource* ou recurso. Um recurso pode ser uma página web contendo 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.* [10] 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 [16].

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 [22]

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. A tabela a seguir, apresenta as principais características deste trabalho com o trabalho de Melotti.

Recurso	Este trabalho	CReAMA
Provê suporte multiplataforma mobile Android e iOS	Sim	Não
Provê suporte a plataforma desktop	Sim	Não
Provê recursos extensíveis através de plugins	Sim	Não
Provê APIs de alto nível para recursos de rede (HTTP), banco de dados e UI	Sim	Sim
Provê suporte a reuso de componentes	Sim	Sim
Provê suporte a comunicação por eventos	Sim	Não

Tabela 1: Tabela comparativa desta arquitetura com *CReAMA*.

MoCA: Arquitetura para o Desenvolvimento de Aplicações Sensíveis ao Contexto para Dispositivos Móveis [25]

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*. A tabela a seguir, apresenta as principais características deste trabalho com a arquitetura *MoCA*.

Recurso	Este trabalho	MoCA
Provê suporte multiplataforma mobile Android e iOS	Sim	Não
Provê suporte a plataforma desktop	Sim	Não
Provê recursos extensíveis através de plugins	Sim	Não
Provê APIs de alto nível para recursos de rede (HTTP), banco de dados e UI	Sim	Sim
Provê suporte a reuso de componentes	Sim	Sim
Provê suporte a comunicação por eventos	Sim	Sim

Tabela 2: Tabela comparativa desta arquitetura com *MoCA*.

Solução Multiplataforma para Smartphone Utilizando os Frameworks SenchaTouch e PhoneGap Integrado à web service Java [17]

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. A tabela a seguir, apresenta as principais características deste trabalho com o trabalho de Jauri.

Recurso	Este trabalho	Trabalho de Jauri
Provê suporte multiplataforma mobile Android e iOS	Sim	Sim
Provê suporte a plataforma desktop	Sim	Não
Provê recursos extensíveis através de plugins	Sim	Não
Provê APIs de alto nível para recursos de rede (HTTP), banco de dados e UI	Sim	Sim
Provê suporte a reuso de componentes	Sim	Sim
Provê suporte a comunicação por eventos	Sim	Sim

Tabela 3: Tabela comparativa desta arquitetura com a Solução Multiplataforma *PhoneGap*/*SenchaTouch* proposto por Jauri Junior.

4. PROJETO DA ARQUITETURA

A arquitetura proposta neste trabalho utiliza os estilos arquiteturais *Client-Server* e *Event-Based*. *Client-Server* pelo fato de um aplicativo consumir algum serviço *RESTFul* sendo neste caso um cliente, enquanto que o serviço *RESTFul* torna-se o servidor. Já o *Event-Based* é um modelo de comunicação baseado em eventos, disposto neste trabalho como um recurso que reduz o nível de acoplamento entre objetos e permite ainda, a comunicação entre partes distintas da aplicação, como por exemplos, os plugins. 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 necessários para 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 Etapas de desenvolvimento

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 extendido 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 é suportado pelo QML, além de classes para recursos extra providos pela arquitetura, como por exemplo, uma classe que exibe uma janela de diálogo para seleção de arquivos no dispositivo.

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 *QuickControls*[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.

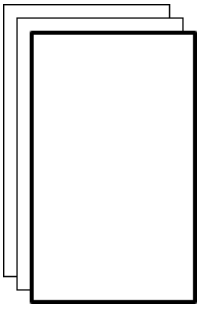

Figura 1: Estrutura de um layout em pilha

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.

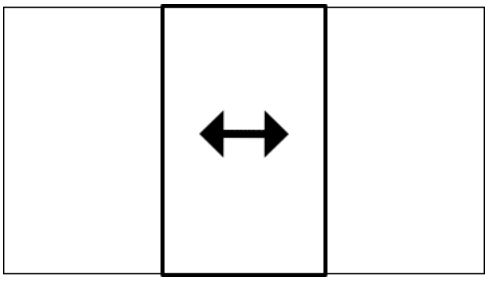

Figura 2: Estrutura de um layout em linha

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 identificadores *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 ententidos como básicos em todo sistema de informação, seja ele mobile ou não. 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 controle de consumo 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 a serem utilizadas pelos plugins. As APIs serão apresentadas em tópicos específicos posteriormente e os 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 geral da arquitetura

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.

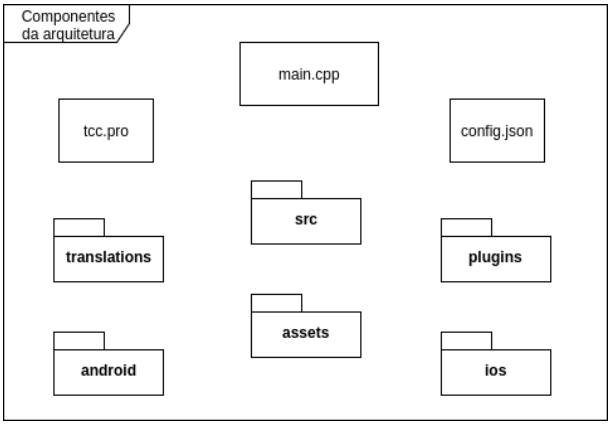

Figura 3: Pacotes principais da arquitetura

1 *main.cpp*: Arquivo inicial da aplicação. Esse arquivo é responsável por instanciar as classes do Qt que exibem a janela do aplicativo e o interpretador de código QML, além de classes da camada de aplicação, configuração e utilitários. 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 linkadas 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, pois contém propriedades que indicarão alguns comportamentos iniciais e escolha de tipos de objetos a serem instanciados, tais como, o tipo de layout a ser utilizado, se os termos de uso deve ser exibido 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 está 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 *App*, *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 obrigatóriamente 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 *.pro* (na raíz do projeto) para criar ou atualizar o arquivo *ts* principal.

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 do aplicativo e imagens requeridas pela plataforma, tais como as imagens de *splash-screen*, além do arquivo de configuração *Info.plist* que define nome e versão do aplicativo e os recursos do sistema requerido para o funcionamento da aplicação no iOS.

4.5 Arquitetura de plugins

As funcionalidades de um aplicativo baseado nesta arquitetura devem ser implementadas através de plugins, atendendo aos requisitos levantados para o 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:/publicComponentes/"*, ao todo quinze componentes foram disponibilizados.

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* dentro deste sub-diretório;

3ª conter pelo menos um arquivo QML visual ou *listener*.

Os *listeners* são componentes não visuais que observam eventos da aplicação. O arquivo *config.json* de um plugin deve ser um objeto contendo as seguintes propriedades:

1. *name* (string): o nome do plugin. Deve ser preenchido para que o plugin seja identificado pelo gerenciador de plugins;
2. *description* (string): um texto que descreve o plugin. Deve ser definido, caso contrário o plugin não será carregado;
3. *listeners* (array): uma lista de strings que indentifica os arquivos do plugin (componentes QML não visuais) que serão instanciados como observadores de eventos. O preenchimento dessa propriedade é opcional e pode ser preenchida mesmo que a propriedade *pages* seja preenchida;
4. *pages* (array): uma lista de objetos que indentifica 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* 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*. O objeto *userProfile* é *null* por default 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 e definido uma string com o perfil do usuário na propriedade *profile* e essa string não tiver em *roles*, a página não será exibida. 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 na lista de itens;

- *isLoginPage* (bool): Um valor boleano 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 for definida como *loginPage* entre os plugins, será utilizada a última página identificada pelo gerenciador de plugins;

- *isHomePage* (bool): Um valor boleano 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 (ou após) o login. Se mais de uma página for definida como *homePage*, será utilizada a última página identificada pelo gerenciador de plugins;

- *showInDrawer* (bool): Um valor boleano que indica se a página poderá ser exibida no menu de layout em pilha (*drawer menu*). Por padrão, o menu de layout em pilha será carregado. 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 com isso, é possível permitir acesso a páginas diferentes a partir dos dois menus;

- *showInTabBar* (bool): Um valor boleano 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* menu estiver sendo utilizado.

As páginas serão instanciadas sob demanda quando o aplicativo tiver utilizando o layout em pilha, neste caso, quando o usuário clicar em um item da lista no menu a página correspondente será instanciada e ficará visível para o usuário. No layout em pilha, o menu é exibido pelo componente *Drawer.qml* que consiste de uma instância do objeto *Drawer* do *QuickControls* e as páginas serão listadas verticalmente. Quando o layout em linha estiver sendo utilizado, uma lista horizontal de botões será acidionado 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* também do *QuickControls* com algumas modificações. As figuras a seguir apresentam um exemplo dos menus utilizado nos layouts em pilha e em linha.

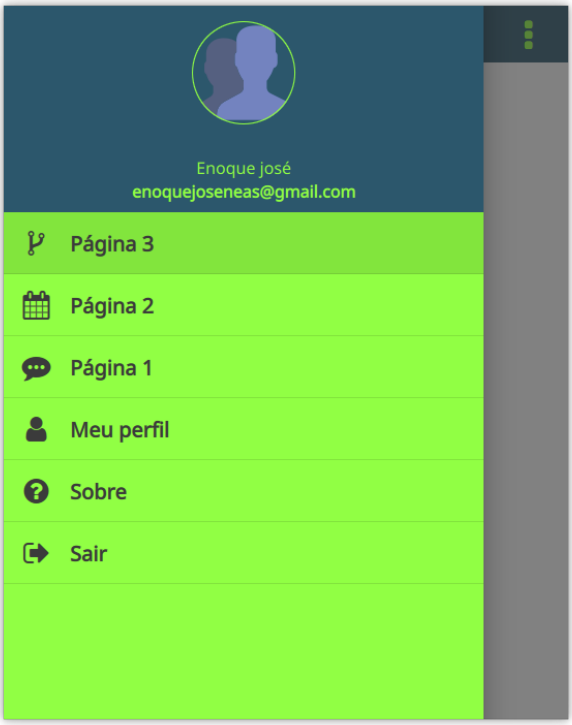

Figura 4: Lista de páginas exibidas no menu de layout em pilha

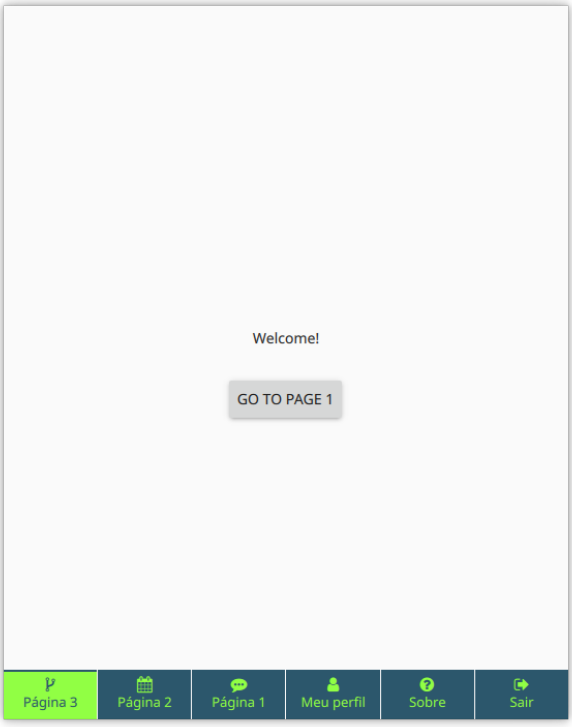

Figura 5: Lista de páginas exibidas no menu de layout em linha

A exibição do título da página no botão pode ser ocultada setando a opção *showTabButtonText* para *false* no arquivo de confi-

guração.

4.6 Gerenciamento de plugins

A classe *PluginManager* é responsável por gerenciar os plugins, basicamente, iterando os arquivos dentro do diretório *plugins* e analizando as propriedades do arquivo de configuração de cada plugin. Em cada arquivo, será verificado as propriedades para cada página, adicionando-a como objeto em um array. 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 do aplicativo e atualizada a cada *release*. Se houver diferença entre versão em execução da versão salva na execução anterior, 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 é criado e gerenciado 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*.

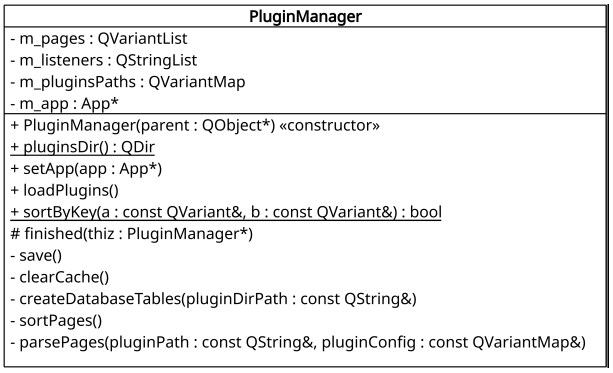

Figura 6: Diagrama da classe PluginManager

O carregamento dos plugins será feito antes de instanciar qualquer componente visual, na invocação do método *loadPlugins* feito pelo objeto *App*. A instância de *PluginManager* será destruída após o carregamento dos plugins para garantir baixo consumo de memória pelo aplicativo.

4.7 A classe App

A classe *App* é um componente importante nesta arquitetura, suas principais responsabilidades consiste de instanciar *PluginManager*, carregar o arquivo *config.json* que contém parâmetros da aplicação e gerenciar a persistência das configurações do aplicativo via *QSettings*[5]. *App* também é responsável por notificar os objetos através do sinal *eventNotify*. A classe *App* será instanciada na inicialização do aplicativo e registrada no contexto da aplicação identificada pela string "App"para que os plugins possam invocar seus métodos públicos. Dentre os métodos possíveis, está o *readSettings* que retorna um tipo genérico de dado requerendo apenas um parâmetro que identifique o valor a ser retornado. Outra tarefa que corresponde a esta classe, é a criação de uma conexão com a atividade do aplicativo no android e no iOS. A aplicação poderá receber parâmetros de eventos como *push notification* ou token de registro no *Firebase*, quando o serviço de *push* for utilizado. O registro do aplicativo no *Firebase* e o recebimento de notificações via *push* é realizado por objetos nativos de cada plataforma em um processo separado da aplicação. Em tempo de execução, as atividade passarão o token e os argumentos recebidos do *push* (título, mensagem, data, etc.) para o aplicativo através de uma chamada ao método estático *fireEventNotify* da classe *App* que irá notificará a aplicação através do sinal *eventNotify*. A figura abaixo apresenta o diagrama da classe *App*.

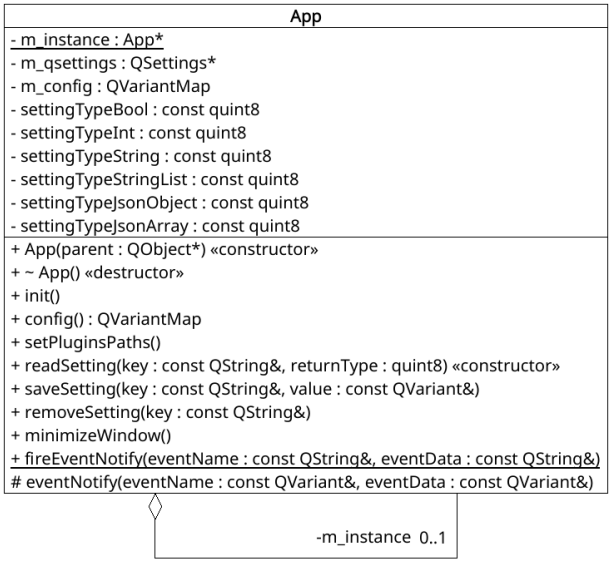

Figura 7: Diagrama da classe App

App é quem define o estilo de *widgets* utilizado pelos componentes QML (Material, Universal, etc.). A escolha do estilo deve ser feito pelo desenvolvedor no arquivo de configuração na propriedade *applicationStyle* e será passado diretamente para o objeto *QQuickStyle* na inicialização. Os possíveis valores para esta propriedade serão detalhados na seção que descreve o arquivo *config.json*. A instância da classe *App* adicionará no objeto *Config* uma propriedade contendo o *path* de cada plugin para facilitar o acesso aos arquivos no diretório de cada plugin. 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 "Documents/assets_catalogs://"no ios.

4.8 Gerenciamento de configurações

A arquitetura provê uma API simples e de alto nível para persistir dados utilizando o conceito *chave-valor* através do objeto *App*. Nesse modelo de persistência, a chave identifica o dado a ser armazenado e o valor é o dado propriamente dito. Os tipos de dados válidos são: strings, números, objetos ou arrays e os métodos disponíveis podem ler, persistir ou deletar. Os métodos serão descritos a seguir:

- *readSetting*: Utilizado para ler um dado salvo utilizando uma string que identifique a informação a ser retornada. Por padrão, o tipo de retorno é string e se o dado não for string o

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

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:

  - *SettingTypeBool*: para retornar um valor boleano;
  - *SettingTypeInt*: para retornar um inteiro;
  - *SettingTypeStringList*: para retornar uma lista de strings; *SettingTypeJsonObject*: para retornar um objeto;
  - *SettingTypeJsonArray*: para retornar um array de objetos.

- *saveSetting*: 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 propriamente dito. O dado pode ser uma string, um valor numérico, um objeto ou array.

- *removeSetting*: Utilizado para apagar alguma informação persistida. 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*:

```
import“QtQuick“2.8

Item“{
    Component.onCompleted:“{
        var“foo“=“App.readSetting("foo")
        if“(bar“!=“foo)
            App.saveSetting("foo",“bar)
        App.removeSetting("bar")
    }
}
```

Algoritmo“1: Exemplo de persistência através do objeto *App*

4.9 Gerenciamento de eventos

A classe *App* possui o sinal *eventNotify* e pode ser utilizado pelos plugins e objetos como conector. Este sinal possui dois parâmetros: o primeiro é uma string que identifica o evento e o segundo, um objeto contendo o dado ou argumento a ser passado para os objetos ouvintes do evento. Como a instância da classe *App* será adicionada ao contexto da aplicação como um objeto global, objetos podem disparar eventos fazendo uma chamada ao sinal da seguinte forma: *App.eventNotify(nome-do-evento, argumento)*. Observadores devem criar uma conexão com este sinal utilizando o objeto *Connections* do QML, especificando como *target* o objeto *App*. O código a seguir, apresenta um exemplo de uma conexão com este sinal:

```
import“QtQuick“2.8
...
Connections“{
    target:“App
    onEventNotify:“{
        if“(eventName“==“Config.events.foo)
            foo()
    }
}
```

Algoritmo“2: Exemplo de conexão com o sinal *eventNotify*

4.10 O Arquivo de Configuração

O arquivo *config.json* é um componente importante desta arquitetura, ele é utilizado como arquivo de configuração geral do aplicativo, porém, suas informações não são persistidas. O conteúdo deste arquivo será repassado para a aplicação como um objeto javascript e o usuário poderá adicionar propriedades a serem lidas pelos plugins. No entanto, as propriedades fornecidas no modelo desta arquitetura não devem ser removidas, pois, componentes internos fazem uso das propriedades definidas neste arquivo. Os plugins poderão ler essa propriedade acessando *Config.property_name*. As principais propriedades e seus possíveis valores podem ser entendidas a seguir:

- *appName* (string): O nome do aplicativo. Este valor será setado em *QApplication.setApplicationName* na inicialização do aplicativo. Essa informação será útil para Qt identificar as configurações do aplicativo;

- *appDescription* (string): Uma descrição sobre o aplicativo. Essa propriedade é opcional;

- *organizationName* (string): O nome da organização. Este valor será setado em *QApplication.setOrganizationName* na inicialização do aplicativo e é será utilizado internamente pelo Qt para definir o nome do diretório de configurações da aplicação. Basicamente, será na *home* do usuário (em modo desktop) e no diretório de cache de aplicativos nas plataformas móveis;

- *organizationDomain* (string): O endereço do domínio da organização, por exemplo, *qt.project.org*. Será utilizado pelo Qt internamente;

- *applicationStyle* (string): O nome do estilo a ser aplicado nos *widgets* (*Button*, *TabBar*, *ToolTip* e etc.). Os possíveis valores são: *Material*, *Universal* ou *Default*. O valor dessa propriedade será passada pelo objeto *App* para *QQuickStyle*;

- *forceEulaAgreement* (bool): Um valor boleano que inidica 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 boleano que indica se o aplicativo deverá carregar uma página de login na inicialização. Se for definido para *true*, a aplicação irá utilizar a página (dentre os plugins) que definiu *isLoginPage* para *true*, identificada pelo gerenciador de plugins durante o carregamento dos plugins;

- *showEula* (bool): Um valor boleano que inidica 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* se tiver sido definida;

- *showTabButtonText* (bool): Um valor boleano que indica se o título da página será exibida nos botões do menu em linha (*TabButton* do *QuickControls*). Essa propriedade só terá efeito quando a aplicação tiver usando o layout em linha;

- *usesSwipeView* (bool): Um valor boleano que indica se o layout principal do aplicativo será em linha, que consiste na utilização de um container do *QuickControls SwipeView*. 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 os containers, 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 boleano que indica se o menu lateral usado no layout em pilha, será instanciado. Esse componente corresponde ao *Drawer* do *QuickControls*. No entando, esse *flag* terá efeito apenas no layout em linha, ou seja, se *usesSwipeView* for *true*, pois no layout em pilha ele será instanciado mesmo que esse *flag* seja *false*. O objetivo dessa propriedade é permitir que o programador possa utilizar o menu lateral quando o layout for em linha, intercalando as páginas que serão visíveis em cada um dos menus através das propriedades *showInDrawer* e *showInTabBar* na configuração das páginas nos plugins;

- *showDrawerImage* (bool): Um valor boleano que indica se a imagem do *Drawer* menu será carregada. Por padrão a imagem não será exibida.

- *restService* (object): Um objeto contendo as definições do serviço *REST*, como url base e os parâmetros de autenticação básica, usuário e senha. O valor informado em *userName* e *userPass* serão passados em cada requisição HTTP como token baseado em um *hash base64*. As seguintes propriedades são requeridas:

  - *userName* (string): O nome do usuário do serviço *REST*;
  - *userPass* (string): A senha de usuário do serviço *REST*;
  - *baseUrl* (string): A url base do serviço REST. Essa propriedade será utilizada pelo objeto *RequestHttp* nos métodos de requisições *GET*, *POST* e etc. A sugestão é que nas páginas que fazem requisições HTTP adicione apenas o *path*, a fim de reduzir e simplificar o código. Com isso, uma alteração futura da url do serviço REST seria feita apenas no arquivo de configuração. Internamente, o objeto que realiza a requisição concatenará essa propriedade com o *path* passado no primeiro parâmetro do método;
  - *baseImagesUrl* (string): A url base dos arquivos de imagens, caso o serviço *REST* utilize uma url diferente ou um sub-domínio para os *resources*.

- *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 4 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;

- *events*: (object): Um objeto que mapeia os eventos, baseados em um par de strings *nome-do-evento.valor*. Essa propriedade será utilizada por objetos e componentes internos para identificar de qual evento estão sendo notificados, a fim de padronizar os nomes dos eventos e reduzir a replicação de strings na aplicação. Essa propriedade poderá ser utilizado tanto em conexões com o sinal *eventNotify* do objeto App como pela API do *Observer* disponibilizado pela arquitetura (será detalhado em um seção mais adiante). O objeto *App* adicionará treze eventos a essa propriedade na inicialização do aplicativo, alguns deles serão utilizados somente por objetos internos, outros podem ser utilizados pelos plugins para executarem ações específicas em dado momento. Esses eventos serão descritos a seguir:

  - *cameraImageSaved* (string): Utilizado para notificar observadores de que uma imagem foi capturada pela câmera do dispositivo e salva localmente. A url da imagem salva será passado no argumento do evento;
  - *cancelSearch* (string): Utilizado pelo *ToolBar* para indicar que o campo de busca não está ativo, para que a página corrente 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* 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;
  - *newActionNotification* (string): utilizado para notificar a aplicação quando o usuário clicar em uma notificação e o aplicativo ficar em *foreground*, vale para notificações via *push* e local. Esse evento será disparado pela atividade do Android (*CustomActivity*) e pelo objeto *QtAppDelegate* no iOS e repassado para a aplicação através do objeto *App*. 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 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 quando houver um perfil de usuário no aplicativo e permitirá 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*.
- *initUserProfile* (string): Esse evento deve ser utilizado quando houver um perfil de usuário e um objeto contendo os dados do usuário 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, passando como argumento o objeto retornado pelo serviço *REST*. O objeto *UserProfile* observa esse evento e quando o mesmo for emitido, ele irá salvar os dados do usuário no objeto *profile* e em seguida, atualizar a propriedade *isLoggedIn* para *true* e carregará a *home page*;
- *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*.

4.11 Acesso a Rede (HTTP)

O primeiro requisito funcional provido na arquitetura é o acesso a rede para comunicação com serviços *RESTFul*. Para permitir o uso de rede pelos plugins, foi criado uma classe C++ que realiza requisições HTTP com suporte a autenticação básica, *download* e *upload* de arquivos. O objetivo dessa classe é oferecer um componente rico em recursos, com bom desempenho e fornecer um componente de alto nível para os plugins. Essa classe utiliza componentes do Qt e entregará a resposta de cada requisição em objetos json dispensando o uso de *cast* pelos objetos. Para entregar uma API ainda mais fácil de utilizar, foi criado um componente QML com o mesmo nome da classe e disponibilizado como reusável. Para simplificar o acesso a rede, o desenvolvedor deverá adicionar na propriedade *restService* no arquivo de configuração, a url do serviço *REST* em *baseUrl*, o nome e a senha do usuário da API em *userName* e *userPass* respectivamente e utilizar apenas o *path* da url nos métodos de requisições. As subseções a seguir, descrevem os detalhes da classe *RequestHttp* e do componente correspondente.

4.11.1 A classe RequestHttp

A classe *RequestHttp* utiliza a classe *QNetworkAccessManager* para gerenciar as operações de rede abstraindo para o aplicativo a interface de rede utilizada no dispositivo. Os métodos disponíveis inicialmente são *GET* e *POST*, além de *download* de arquivos via *GET* e *upload* de arquivos via *POST* ou *PUT*. A classe *QNetworkRequest* do Qt será utilizada para encapsular os parâmetros da requisição. Os métodos principais dessa classe serão descritos a seguir:

- *get*: realiza uma requisição 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 "" "qrc:/publicComponents" "" as "Components"

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

- *post*: realiza uma requisição 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. O código a seguir, apresenta um exemplo de uso do método post:

```
import "QtQuick" 2.8
import "" "qrc:/publicComponents" "" as "Components"

Components.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 (boleano, 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. Esse método emitirá o sinal *uploadFinished* para cada arquivo enviado. Durante o upload 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 "" "qrc:/publicComponents" "as" Components

Components.RequestHttp "{
    id: "requestHttp
}
...
Item "{
    Component.onCompleted: "{
        var "files "=" [
            "/data/app/myapp/files/f1.png",
            "/data/app/myapp/files/f2.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* (boleano, default *false*) pode ser utilizado para alterar o diretório de destino dos arquivos para uma pasta interna do aplicativo. Para cada arquivo salvo, o sinal *downloadedFileSaved* será emitido passando o *path* do arquivo salvo localmente. Outro 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 já baixados. O código a seguir, demonstra um exemplo de requisição para download de arquivos:

```
import "QtQuick" 2.8
import "" "qrc:/publicComponents" "as" Components

Components.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 são declarados como *Q_INVOKABLE* [6] para permitir o uso dos métodos em componentes QML, como é o caso do componente *RequestHttp*. Para obter a resposta de uma requisição, é preciso criar uma conexão com o sinal *finished*. Esse sinal será emitido por todos os métodos descritos anteriomente se 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* contendo o dado enviado pelo servidor. Se a resposta enviada pelo servidor for um json válido, *response* conterá um objeto ou array, 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* 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*: inidica um erro no pedido, quando o servidor não responder ou o status da requisição for zero;
- *Finished*: indica que a requisição terminou e é 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 do objeto *RequestHttp* e logo após a emissão do sinal *finished*.

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

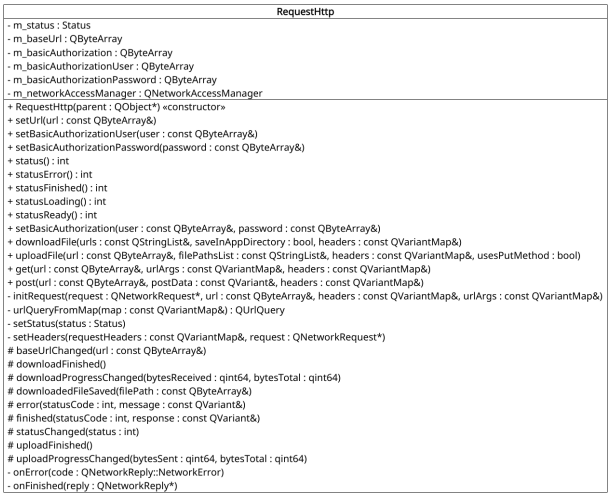

Figura 8: Diagrama da classe *RequestHttp*

4.11.2 O Componente RequestHttp

A classe *RequestHttp* será registrada como um tipo QML no contexto da aplicação e para ser instanciada basta adicionar a diretiva *import RequestHttp 1.0* e em seguida, declarar um objeto QML.

[6] https://doc.qt.io/qt-5/qtqml-cppintegration-exposecppattributes.html

No entando, para simplificar ainda mais o uso de rede, os plugins podem utilizar o componente *RequestHttp* que instancia a classe *RequestHttp* e inicializa os atributos *baseUrl* e os parâmetros de autenticação, além de exibir mensagens de erro personalizada para o usuário. Os exemplos de código apresentados na subseção anterior, utilizam o componente *RequestHttp* importando-o dos arquivos reusáveis. No entanto, objetos *listeners* que por algum motivo não utilizem o serviço *REST*, podem instanciar a classe *RequestHttp* diretamente. O código a seguir, apresenta um exemplo de uso da classe *RequestHttp*:

```
import“RequestHttp“1.0

RequestHttp“{
    baseUrl:“Config.restService.baseUrl
    authorizationUser:“Config.restService.userName
    authorizationPass:“Config.restService.userPass
    onError:“{
     var“message“=“"Erro␣ao␣conectar␣no␣servidor"
        if“("android"“==“Qt.platform.os)
            snackbar.show(message)
        else
            functions.alert("Error!",“message)
    }
}
```

Algoritmo“3: Código do componente *RequestHttp.qml*

4.12 Persistência de Dados

Persistência de dados é o segundo requisito funcional disponibilizado nesta arquitetura e visa fornecer aos plugins a possibilidade de persistir dados em um banco *SQLITE* e o funcionamento *offline* do aplicativo. 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 utilizá-lo como componente QML. Para criar as tabelas no banco de dados do aplicativo, o plugin deve fornecer um arquivo *plugin_table.sql* 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 delega as operações para esse objeto e fornece alguns recursos para simplificar ainda mais a persistência de dados. As subseções a seguir apresentam detalhes de ambas as classes *Database* e *DatabaseComponent*.

4.12.1 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. 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 cosultas. Os métodos principais desta classe serão descritos a seguir:

- *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 boleano *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 implementa o *singleton*, e será instanciada na inicialização da aplicação 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*.

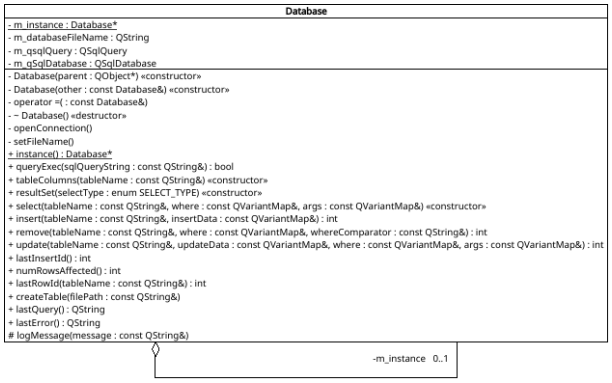

Figura 9: Diagrama da classe Database

4.12.2 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 utilizarem como componentes QML. *DatabaseComponent* agrega uma instância da classe *Database* e delega as operações para esse objeto. No entanto, ela possui alguns atributos declarados como *Q_PROPERTY* que permitem aos plugins informar o nome da tabela no banco de dados, uma coluna 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 a realização de *parsing* nas views ou *delegates* para objetos agregados, sendo retornados já convertidos em objetos ou arrays.

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

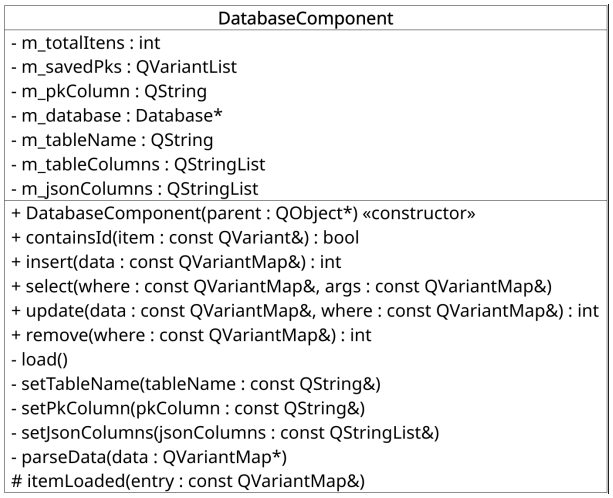

Figura 10: Diagrama da classe DatabaseComponent

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* 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)
    }
}
...
```

Algoritmo"4: Exemplo de como utilizar *DatabaseComponent*

4.13 Notificações do Aplicativo

O terceiro requisito funcional disponibilizado nesta arquitetura é o envio de notificação ao usuário do aplicativo e consiste de notificações via *push* e local. As notificações via *push* é suportado apenas no Android e iOS e as notificações locais é suportado nos dispositivos móveis além de 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 *chave.valor* e o valor pode ser tanto uma string como um valor numérico ou, outro objeto ou array. As notificações locais serão enviadas pela própria aplicação via chamada de método e as notificações via *push* através do *Firebase*. As subseções a seguir, descrevem detalhes de cada tipo de notificação.

4.13.1 Notificações via 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 con-

trário, 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 na arquitetura e o desenvolvedor não precisa modificá-las. O primeiro serviço registra o dispositivo no *Firebase* e retorna um token. Quando isso ocorrer, o token será passado para a aplicação através do sinal *eventNotify* do objeto *App*. 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: o título, a data e a hora de envio e o dado de *payload* (um dado adicional não visível enviado na mensagem).

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 sinal 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 do evento os dados de *payload* da mensagem.

A arquitetura dispõe do plugin de exemplo *Listeners* que contém o componente *PushNotificationRegister.qml*. Esse componente demostra um exemplo de como obter o token de registro no *Firebase* e enviar para o serviço *REST*, destruindo o componente (para otimizar consumo de memória) quando o token for recebido com sucesso pelo servidor.

```
import“QtQuick“2.8

Connections“{
    target:“App
    onEventNotify:“{
        var“e
        e“=“Config.events.newPushNotificationToken
        if“(e“==“eventName)“{
            //“enviando“o“token“para“o“web“service
            //“eventData“eh“o“argumento“do“evento
            //“contendo“uma“string“com“o“token
            sendTokenToServer(eventData)
        }
    }
}
```

Algoritmo“5: Exemplo de como acessar o token do *Firebase*

4.13.2 Notificações local

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 objeto *App* que notificará a aplicação via *eventNotify* contendo em *eventName* a string *newActionNotification* e em *eventData*, o argumento passado no método *show*. O 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)
    }
}
```

Algoritmo“6: Exemplo de como exibir uma notificação local

4.14 Comunicação entre os objetos e plugins

O quarto requisito funcional desta arquitetura é a disponibilização de um mecanismo de comunicação entre objetos e plugins facilitado. A seção 4.9 descreveu o uso do sinal *eventNotify* do objeto *App*, utilizado por componentes internos e pelos plugins disponibilizados como exemplo na arquitetura. Porém, esse não é o único mecanismo de comunicação disponibilizado, a arquitetura dispõe de uma implementação em C++ do padrão de projeto *Observer* baseado em categorias de eventos. A categoria, neste caso, é uma string que identifica um evento específico e visa notificar apenas os objetos interessados, evitando o *broadcast* na aplicação em que todos os observadores da serão notificados. A implementação do *Observer* permite ainda enviar um argumento para o objeto observador.

A implementação do Observer foi feito por duas classes C++, uma chamada *Subject* e outra *Observer*. Na inicialização da aplicação, a classe *Subject* será instanciada e o objeto registrado no contexto da aplicação. A classe *Observer* será registrada na aplicação como um tipo QML para que os plugins possam utilizá-lo como componente. 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 um determinado evento. Quando um objeto deseja observar um evento, ele deve informar o nome do evento e passar uma referência para o observer que será informado quando o evento for disparado. Quando um determinado objeto deseja notificar observadores, ele invocará o método *Subject.notify* passando no primeiro parâmetro uma string que identifica o evento, seguido de um objeto como argumento a ser passado para os observadores e por último, uma referência para sí mesmo que identificá como emissor do evento.

O objeto *Subject* (registrado no contexto da aplicação) disponibiliza os seguintes métodos:

- *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 um ponteiro 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 observado 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 três parâmetros, o primeiro é uma string contendo o nome do evento, o segundo é um objeto variante contendo o dado a ser passado para o observador e por último um ponteiro *QObject* para o emissor do evento. Ao chamar esse método, o *Subject* irá iterar o vetor de observadores correspondente ao evento enviado, chamando para cada observador o método público *update* que receberá como argumento, o nome do evento em que está sendo notificado, o argumento enviado e uma referência para o objeto emissor.

Os eventos podem ser adicionados no arquivo de configuração na propriedades *events* e utilizá-los na aplicação através do objeto *Config*. Os eventos devem formar um par de strings no estilo *nome-do-evento.valor*, por exemplo: *Config.events.foo*. 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
    events: "[Config.events.newSourceAdded]
    onUpdated: "database.insert(eventData)
}
...
Item "{
    Component.onCompleted: "{
        //"pede"ao"Subject"para"adicionar
        //"o"observador"a"lista"do"evento
        var"event"="Config.events.newSourceAdded
        Subject.attach(observer,"event)
    }
}
...
```

Algoritmo"7: Exemplo de como utilizar o *Observer*

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)
    }
}
```

Algoritmo"8: Exemplo de como notificar observadores

4.15 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 entando, atualizações no perfil do usuário não devem ser feitas diretamente no objeto 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* e 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 objeto (do perfil do usuário, retornado no login) a propriedade *user_role.name*, que é o valor atribuído a essa propriedade. A exibição das páginas para o usuário será baseado nessa propriedade através de *bindings*. 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*. *isLoggedIn* será persistida sempre que for modificada.

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 *Session* e 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 na próxima inicialização, pois, as informações do usuário serão persistidas e na próxima inicialização o carregamento da primeira página será feita de acordo com o valor definido *isLoggedIn*. Se o login for efetuado com sucesso na execução anterior, essa propriedade será *true*, pois ela inicializará com o último valor definido na execução anterior.

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

```
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
    }
}
```

Algoritmo"9: Exemplo de como exibir informações do usuário

4.16 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 atributos e métodos agregados. As propriedades a seguir, compõem o *BasePage* e devem ser utilizadas para customizar a estrutura das páginas.

- *toolBarButtons* (array): uma lista de objetos contendo as propriedades de um botão *ToolBarButton* a serem adicionados no *ToolBar* dinamicamente quando a página for carregada. 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* (página corrente). No modo *seach*, a propriedade *searchText* também de *BasePage* receberá uma cópia do texto digitado pelo usuário;

- *absPath* (string): essa propriedade deverá ser definida pela página 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 nome do plugin a partir do objeto *Config.plugins* + o nome do arquivo QML correspondente. Por 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 menu do layout em pilha, que é uma instancia do *ToolBar* do *QuickControls*;

- *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 *QuickControls*;

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

- *hasListView* (bool): essa propriedade é *true* por *default* e se mantida com o valor padrão, instanciará um *ListView* do *QuickControls* 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. Quando o *ListView* for instanciado, essa propriedade será atribuída a *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 *QuickControls* e as propriedades definidas em *Page* serão herdadas, como por exemplo, *title*, que deve ser definido pela página 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)
    }
...
}
```

Algoritmo“10: Exemplo de uso do *BasePage*

4.17 Componentes Reusáveis

A arquitetura dispõe de quinze componentes reusáveis para os plugins, há componentes visuais e não visuais e para utilizá-los basta adicionar a diretiva *import "qrc:/publicComponentes/"* e declarar como componente QML. Os componentes disponíveis serão descritos a seguir:

- *ActionMessage.qml*: esse componente exibe um botão contendo um texto e um ícone do *Awesome Icons* no centro da tela. O texto pode ser definido na propriedade *messageText* e o ícone na propriedade *iconName*. O ícone e o texto estão disposts em um *ColumnLayout* sendo o ícone acima do texto. Dois sinais serão emitidos: *clicked()* e *pressAndHold()* quando clicado e pressionado respectivamente pelo usuário;

- *AwesomeIcon.qml*: esse componente consiste de um *RoundButton* do *QuickControls* com a propriedade *flat* setado para *true* para que o background seja transparente. Ele utiliza um arquivo de fonte OTF contendo setessentos e vinte 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;

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

- *Brand.qml*: esse componente consiste de um retângulo e uma imagem centralizada. A imagem utilizada é por padrão o ícone definido em *assets/icon.png*. O plugin *About* fornecido como exemplo, utiliza esse componente no topo da tela para exibir a logo do 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 contendo como argumento do evento, uma string com o *path* da imagem;

- *CustomButton.qml*: esse componente consiste de um botão com as bordas arredondadas e com a paleta de cores definida para: *colorPrimary* como cor de fundo, *colorControlHighlight* como cor da borda e *colorAccent* como cor do texto. Essas cores devem ser definidas pelo programador no arquivo de configuração na propriedade *theme*;

- *CustomListView.qml*: esse componente *extends* o *ListView* do *QuickControls* e já dispõe de um *ListModel* que será instanciado e definido como model do *ListView*. Também será adicionado efeitos de transição (quando um item for adicionado ou removido), 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;

- *FloatingButton.qml*: esse componente exibe um botão circular flutuante no lado direito da tela, no rodapé da janela do aplicativo. É possível definir um ícone do *Awesome* através da propriedade *iconName* e possui os mesmos sinais de um *Button* do *QuickControls*, tais como o *clicked* e *pressAndHold*;

- *ListItem.qml*: esse componente é um *ItemDelegate* do *QuickControls* 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 *QuickControls* 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* ou *Normal*;

- *PhotoSelection.qml*: esse componente é um *Popup* do *QuickControls* e pode ser utilizado para exibir uma lista vertical de opções 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;

- *RequestHttp.qml*: esse componente já foi apresentado na seção que descreve a API de rede e deve ser utilizado quando for necessário realizar requisições HTTP. É importante lembrar que *BasePage* já possui uma instância desse componente permitindo as páginas iniciar requisições ao serviço *REST*. No entanto, esse componente pode ser utilizado em objetos *listeners*;

- *RoundedImage.qml*: esse componente exibe uma imagem arredondada 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 *QuickControls* 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.

4.18 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 *QuickControls* 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* e *Toast* e foram inspirados nos respectivos componentes do *Material Design*. 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.

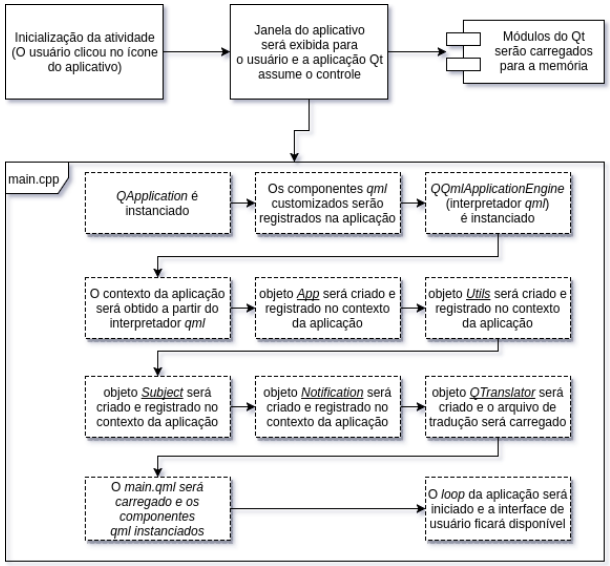

Figura 11: *workflow* da execução de um aplicativo Qt

4.19 Métodos para utilização 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 faça:

1. Acesse a página do projeto no *github* e crie um *fork* para a sua conta.

2. Clone o projeto *forkado* para a sua máquina.

3. Renomeie a pasta clonada e o arquivo *tcc.pro* para o nome do seu projeto (pode ser o nome do seu aplicativo).

4. Importe o projeto (arquivo .pro) no *QtCreator* e siga os passos seguintes para configurar as plataformas suportadas pelo aplicativo.

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

6. Abra o arquivo *AndroidManifest.xml* no *QtCreator* e renomeie o *package name*, de *org.qtproject.example* para o nome do pacote do aplicativo.

7. Para utilizar *push notification*, é necessário criar um projeto do *Firebase* e adicionar o suporte a *push notification*[7]. Configure as opções e no final do *wizard* de configuração exporte o arquivo *google-services.json* e salve na pasta *android* e IOS substituindo os arquivos existentes.

[7] https://console.firebase.google.com/project/novo-projeto-do-firebase/notification

8. No android, é necessário mais um passo para habilitar o *push notification*. Edite o arquivo *android/build.gradle* e renomeie 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. Adicione os ícones do aplicativo em *android/res/drawable-(*)dpi* e *ios/icons*. O projeto vem com ícones nos tamanhos ideais (para android). É possível utilizar alguma ferramenta[8] online para gerar os ícones.

11. Se algum plugin precisar exibir a logo ou o ícone do aplicativo, deve ser adicionado uma cópia de um ícone na pasta *images* 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 (*Drawer.qml*), ela será carregada se a propriedade *showDrawerImage* for *true* no arquivo de configuração e pode ser substituída por outra 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. Escreva os plugins. As funcionalidades do aplicativo deverão ser implementadas 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 *qml.qrc*, que contém os componentes internos e reutilizáveis criando um acoplamento entre plugins e núcleo, violando um requisito não funcional desta arquitetura.

5. AVALIAÇÃO EXPERIMENTAL

Aqui será apresentado a estudo de avaliação conduzido para valiar a solução desenvolvida.

6. CONCLUSÃO

Aqui será apresentado as conclusões obtidos em todo o trabalho, sejam elas positivas ou negativas.

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 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. 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 esta arquitetura 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. 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. Utilizar o *Observer* como único mecanismo de comunicação entre objetos, descartando o uso do sinal *eventNotify* da classe *App*. O *Observer* já está disponível na arquitetura mais não está sendo utilizado entre os componentes internos. As classes *Subject* e *Observer* em *src/core* foram implementadas para oferecer um mecanismo de comunicação por eventos reduzindo o *overhead* na aplicação notificando somente os objetos interessados no evento.

[8] https://jgilfelt.github.io/AndroidAssetStudio/icons-launcher.html

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

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

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

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

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

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

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.