

mobile

m a g a z i n e

Edição 53. Ano 5



Android

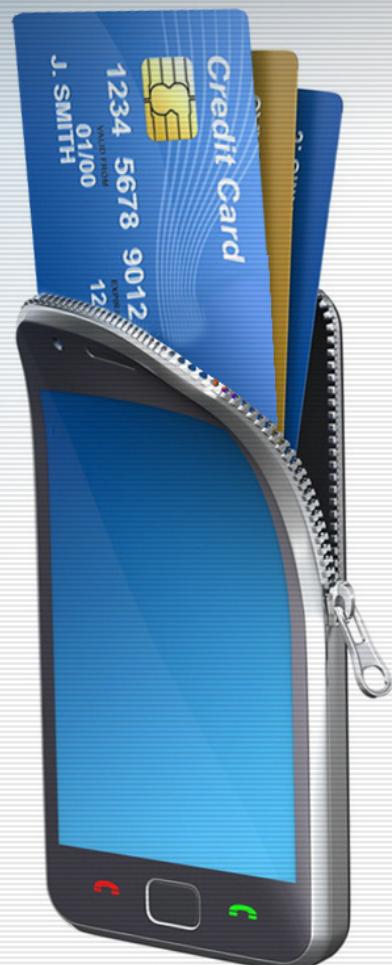
Conheça o Flurry para análise de eventos e crash report

Android

Aprenda a manipular dados e adapters na prática

PLANEJANDO SUA CARTEIRA MOBILE

Entenda como criar uma carteira eletrônica para dispositivos móveis



Do Instituto Infnet para as melhores empresas do mercado.

PÓS-GRADUAÇÃO

MIT em Engenharia de Software com Java
MIT em Engenharia de Software com .NET

GRADUAÇÃO

Análise e Desenvolvimento de Sistemas
Engenharia de Computação

FORMAÇÕES

Desenvolvedor Java
Desenvolvedor Java: Sistemas Distribuídos
Desenvolvedor Web .NET
MCITP Server Administrator
SQL Server

Aulas 100% práticas, corpo docente atuante no mercado, a mais atualizada biblioteca de TI do Rio, laboratórios com tecnologia de ponta e sala de estudos e exames.

Faça Infnet e seja um profissional valorizado.

“ Entrei na Graduação de ADS em março de 2009 (...). Hoje já sou Desenvolvedor Java Jr. de uma das maiores multinacionais de TI do mundo. ”

Rafael Santos
Aluno da Graduação de ADS - 2010.1

Acesse nosso site e conheça todos os nossos programas: www.infnet.edu.br/esti

ESCOLA SUPERIOR DA TECNOLOGIA DA INFORMAÇÃO
www.infnet.edu.br | 21 2122-8800



EXPEDIENTE

Editor

Rodrigo Oliveira Spínola (rodrigo.devmedia@gmail.com)

Subeditor

Eduardo Oliveira Spínola

Consultora Técnica

Daniella Costa (daniella@devmedia.com.br)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse www.devmedia.com.br/central, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique à vontade para entrar em contato com os editores e dar a sua sugestão!

Caso tenha interesse em publicar um artigo na revista ou no site Mobile Magazine, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Rodrigo Oliveira Spínola - Editor da Revista

rodrigo.devmedia@gmail.com



RODRIGO OLIVEIRA SPÍNOLA

Editor Chefe da SQL Magazine, Mobile e Engenharia de Software Magazine. Professor da Faculdade Ruy Barbosa, uma instituição parte do Grupo DeVry. Doutor e Mestre em Engenharia de Software pela COPPE/UFRJ.

Sumário

Conteúdo sobre Boas Práticas

04 – Planejando sua carteira eletrônica mobile

[Renato Galdino Machado]

Conteúdo sobre Boas Práticas

10 – Analisando o comportamento de aplicações Android

[Augusto Marinho]

Conteúdo sobre Boas Práticas

18 – Desenvolvendo um programa de cadastro usando SQLite e Adapters

[Robison Cris Brito e Ricardo Ogliari]

**Dê seu feedback sobre esta edição!**

A Mobile Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link: devmedia.com.br/webmobile/feedback

Planejando sua carteira eletrônica mobile

Entenda como criar uma carteira eletrônica para dispositivos móveis

Imagine a situação de estar diante de um caixa para pagar sua compra ou consumo, o vendedor informa o valor total, você tira seu *smartphone* do bolso, e com apenas um clique na tela, ou apenas aproximando-o do caixa, o pagamento é realizado e o comprovante aparece imediatamente na tela de seu *smartphone*. Ou ainda, a situação de poder transferir certo valor, pequeno ou grande, a um colega diretamente pelo *smartphone* e sem ter de tirar dinheiro de papel de sua carteira, mesmo que este colega não tenha conta em banco, ou tenha conta em um banco diferente de onde você possui conta corrente. Estas situações são possíveis com a Carteira Eletrônica.

O conceito inicial de Carteira Eletrônica é algo simples: efetuar compras e pagamentos através de um *smartphone*, em substituição aos cartões plásticos de uma carteira física. Desta forma, não seria mais necessário carregar diversos cartões de crédito, talões de cheque, dinheiro e comprovantes, dentro de uma grossa e pesada carteira em seu bolso. Com *smartphones* cada vez mais carregados de possibilidades tecnológicas, tais como câmeras que podem ler códigos de barras, bluetooth, NFC, criptografia avançada e conexão constante com a Internet, podemos implementar um *software* ou plataforma de carteira eletrônica com riqueza de funcionalidades e facilidades.

Diferente de uma carteira convencional, com cartões bancários, cartões de crédito e folhas de cheque, a Carteira Eletrônica pode e deve agregar valor à carteira convencional, tais como as possibilidades de consultar saldos e extratos, efetuar transferências on-line em tempo real, pagar contas, organizar ou planejar pagamentos, incrementar a segurança das operações financeiras, impedir clonagens ou roubos, dentre inúmeras outras facilidades de valor agregado que possam ser imaginadas.

Estas inúmeras e infinitas possibilidades de agregar valor à carteira física tradicional podem ser tentadoras

Resumo DevMan

Porque esse artigo é útil:

Neste artigo serão discutidos os conceitos e técnicas para implementação de uma carteira eletrônica, com uma breve apresentação dos conceitos básicos, apresentação das necessidades e possibilidades do produto, apresentações e sugestões técnicas para implementação, e finalizando com interessantes sugestões de aplicações práticas.

A implementação de uma carteira eletrônica é útil para desenvolvedores e empresas que desejam criar novos meios de pagamentos com algo valor agregado aos seus usuários e clientes.

e fantásticas, porém não devemos nos esquecer de sua premissa inicial, que é a de realizar compras e pagamentos. Um usuário de Carteira Eletrônica certamente ficará frustrado caso tenha a percepção de que é mais trabalhoso e demorado efetuar compras e pagamentos através de seu *smartphone*, do que simplesmente retirar sua “carteira de couro” do bolso da calça, retirar um cartão plástico ou notas de papel, e efetuar rapidamente a operação. Sim, operar a Carteira Eletrônica deve ser algo rápido, fácil e descomplicado. Quanto tempo, ou quantos segundos, uma pessoa comum leva para retirar um cartão de crédito da carteira e efetuar um pagamento? Quanto tempo, ou quantos segundos, esta mesma operação consome em sua Carteira Eletrônica? Novamente, não devemos nos esquecer da premissa inicial, a Carteira Eletrônica, antes de qualquer outra “coisa”, continua a ser uma carteira.

Ainda na premissa inicial, de que a Carteira Eletrônica é uma carteira, ela também deve ser segura. Sua “carteira de couro” está segura e protegida enquanto está em sua posse, a não ser que seja perdida, roubada, ou manipulada por outros sem seu conhecimento. Isto também é válido para um *smartphone*, que pode ser perdido, roubado, ou “emprestado”. A Carteira Eletrônica, nestas situações, possui uma vantagem, e uma desvantagem. A vantagem é poder cancelar ou bloquear toda a carteira eletrônica, remotamente, de uma só vez e em uma única operação, talvez pela Internet, sem ter de efetuar diversos e longos telefonemas para

comunicar perda ou roubo, como ocorre com as carteiras físicas tradicionais – note que acabamos de descobrir mais um valor agregado para a Carteira Eletrônica.

A desvantagem, por sua vez, é que seu *smartphone* continua a ser um telefone celular, geralmente deixado na mesa ou à vista de estranhos, não costuma ficar trancado em gavetas, e por vezes até emprestado, coisas que normalmente não fazemos com nossas carteiras físicas. Além do pequeno risco de ser invadido por *hackers* ou vírus computacionais. Talvez não venhamos a perceber em nosso cotidiano, mas nós protegemos nossas carteiras físicas com segurança e recursos físicos e, portanto, devemos proteger as carteiras eletrônicas com segurança digital e recursos digitais.

Outro fator cotidiano ligado às carteiras físicas: a organização. Nós costumamos organizar os itens dentro de nossas carteiras de maneira a facilitar nossa vida. Colocamos os cartões mais utilizados na frente dos demais, alguns deixam um talão de cheques inteiro e aberto, outros preferem dobrar e esconder umas poucas folhas. Alguns separam documentos em um lado e o dinheiro do outro, e algumas pessoas guardam os comprovantes de pagamentos na carteira, enquanto outras preferem jogá-los no lixo.

Nenhuma carteira é igual à outra, ao menos internamente. A ideia de organização dos diferentes objetos e informações dentro de uma carteira física não está ligada necessariamente ao perfeccionismo nem a algum tipo de “mania”, mas simplesmente objetivamos a facilidade e rapidez em acessarmos o que queremos, imediatamente, permanecendo o menor tempo possível com a carteira aberta e sem ter de vasculhar item por item até encontrarmos o que desejamos. Então, uma Carteira Eletrônica não deveria ser diferente, não exatamente pela necessidade de permitir uma organização das posições dos itens, mas por permitir encontrar logo “de cara” o que desejamos.

Percebemos sem muita dúvida, que a Carteira Eletrônica deve obedecer algumas regras de utilização de uma carteira convencional: rápida, fácil, e segura. Comparando-a ainda com a carteira convencional, a carteira convencional é também universal, ou seja, podemos efetuar pagamentos e recebimentos para qualquer pessoa, mesmo que esta pessoa não possua conta em banco, ou possua conta em um banco que nem conhecemos. Sim, afinal, também carregamos notas de dinheiro em nossas carteiras, que podem ser utilizadas para qualquer pagamento a qualquer pessoa, e podemos até mesmo pagar e receber em moedas estrangeiras. Independente das soluções dadas por desenvolvedores, engenheiros ou gênios de produtos, que podem ser muitas, variadas e criativas, devemos observar além da solução tecnológica, e adentrar nos cenários do “Produto Carteira Eletrônica”. Afinal, as soluções nascem diante de um desafio, e não o contrário.

Desafios do produto Carteira Eletrônica

Através de nossas carteiras convencionais, efetuamos pagamentos através de cartões de crédito, cartões de débito, dinheiro,

e cheques. Dentre estes, observemos o mundo dos cartões de crédito e débito.

Os cartões de crédito e débito, ao menos os mais conhecidos e utilizados, funcionam da mesma forma no mundo inteiro, com as mesmas regras e os mesmos tipos de participantes, de forma semelhante ao disposto na **Figura 1**. Em uma ponta, está o consumidor final, chamado de “portador”, é quem carrega consigo um cartão de crédito ou débito em sua carteira. O portador, para obter este cartão, possui relacionamento com um banco ou emissor de cartões, chamado de “emissor”, quem emite e gerencia esta base de cartões, bem como cobra ou credita o portador em suas operações com os cartões. Estes cartões, via de regra, são emitidos dentro de uma rede de aceitação, chamadas de “bandeiras”, tais como: Mastercard, Visa, American Express, Diners Club, JCB, EuroPay, UnionPay, Good Card, Premium Card, dentre inúmeras outras “bandeiras”. Estas bandeiras estabelecem uma rede de aceitação regional ou global, através dos chamados “adquirentes”, e são os adquirentes os responsáveis pelo relacionamento com os lojistas e estabelecimentos comerciais, provendo a eles a tecnologia necessária para a aceitação dos cartões de crédito, realizando também o gerenciamento financeiro e operacional dos pagamentos efetuados.

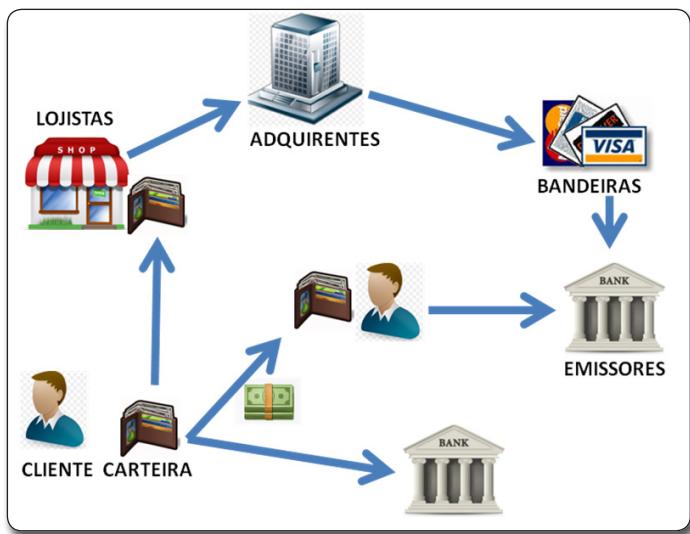


Figura 1. Fluxo global dos Meios de Pagamentos

Aqui começam os desafios do Produto Carteira Eletrônica. Os portadores dos cartões, ou seja, os usuários de Carteira Eletrônica são sempre “Pessoas Físicas” – mesmo os cartões emitidos para empresas Pessoa Jurídica devem possuir uma Pessoa Física como responsável pelo relacionamento com o emissor do cartão. O relacionamento com Pessoas Físicas, portadores, ocorre sempre com o emissor. Apenas o emissor possui relacionamento com o portador. As bandeiras não desejam efetuar nem manter relacionamento comercial com Pessoas Físicas, afinal este tipo de relacionamento é problemático e requer uma enorme estrutura de atendimento. Além do fato de as bandeiras não poderem atuar com relacionamento

Planejando sua carteira eletrônica mobile

comercial junto às Pessoas Físicas na grande maioria dos países onde estão presentes, incluindo aí o Brasil - pois não possuem a utilidade social de banco nem de financeira.

Já os adquirentes, pelas mesmas razões não podem nem desejam manter relacionamento comercial com Pessoas Físicas, portadores. Os adquirentes mantêm relacionamento comercial com lojistas e estabelecimentos comerciais, provendo a tecnologia necessária e o gerenciamento operacional dos pagamentos. E, na ponta final, os lojistas. Os lojistas devem obrigatoriamente manter um relacionamento tecnológico e operacional com um adquirente, caso queiram receber pagamentos eletrônicos, cartões de crédito e débito.

Resumindo a situação, nem as bandeiras nem os adquirentes desejam manter relacionamento comercial com Pessoas Físicas, e os lojistas estão “amarrados” à tecnologia provida pelos adquirentes, caso desejem receber pagamentos eletrônicos. Sobram os emissores, que podem e desejam manter relacionamento com os portadores. Porém, um emissor que criar uma carteira eletrônica certamente será obrigado a aceitar, e prover atendimento, aos cartões de emissores concorrentes – afinal não faria muito sentido criar uma carteira eletrônica que aceite apenas os cartões de um único banco. Já os adquirentes, para implementar a tecnologia de carteira eletrônica em suas lojas e estabelecimentos comerciais, teriam de obter autorização das bandeiras para operar uma nova e exclusiva tecnologia de pagamentos, ou fechar acordos comerciais com todos os bancos emissores de sua área de atuação – situações virtualmente e contratualmente impossíveis de ocorrer.

Por fim, especificamente no cenário brasileiro, há um impedimento Legal para custódia, ou seja, apenas os bancos podem fazer custódia de dinheiro e receber depósitos, e apenas bancos e financeiras podem conceder crédito, todos devidamente registrados junto ao BACEN – Banco Central do Brasil.

Resumindo mais ainda, e finalizando, ninguém consegue, sozinho, emplacar um produto universal de Carteira Eletrônica, seja por falta de interesse, seja por impedimentos legais ou contratuais. Mas, a despeito destas dificuldades, as plataformas de Carteira Eletrônica não deixaram de surgir nem deixaram de operar, mesmo no Brasil. Lembrando que, são as soluções que nascem diante dos desafios, não o contrário.

Diante das soluções de Carteira Eletrônica que continuam a surgir, superando todas as dificuldades, forma-se um cenário interessante que pode ser notado atualmente: a diversidade de carteiras eletrônicas. São plataformas que conseguiram estabelecer variados tipos de relacionamentos comerciais e tecnológicos, entre um ou mais participantes da cadeia de pagamentos. Mas como nenhuma Carteira Eletrônica conseguiu tornar-se universal, podemos ver uma diversidade de plataformas, de variados portes e tamanhos, cada qual alcançando seu grau de sucesso.

Boa parte da “engenharia de produto” e planejamentos, ao criar uma plataforma de Carteira Eletrônica, consiste em entender que a sua Carteira Eletrônica, muito provavelmente não será a primeira, nem a única, nem a última, a ser instalada nos smartphones de seus usuários e clientes. Pode não ser a única, mas, não poderia ser a melhor?

Criando sua própria Carteira Eletrônica

Vimos até agora que uma Carteira Eletrônica segue as premissas básicas de uma carteira física ou convencional, que deve ser segura, fácil, e rápida de operar. Também sabemos que uma Carteira Eletrônica pode e deve oferecer funções de valor agregado, e também que, entendidos os desafios e limitações do cenário de pagamentos, deve incluir soluções e diferenciais para que seja a mais universal possível.

Estas necessidades, entretanto, devem seguir uma hierarquia de prioridades, afinal, de nada adianta criar um *software* rápido e completo, mas inseguro; ou ainda uma solução inovadora mas extremamente complicada; ou algo fácil e rápido de operar mas que causa muitas panes e travamentos. Podemos então analisar formas de implementação destes itens e sua hierarquia, conforme disposto na **Figura 2**.

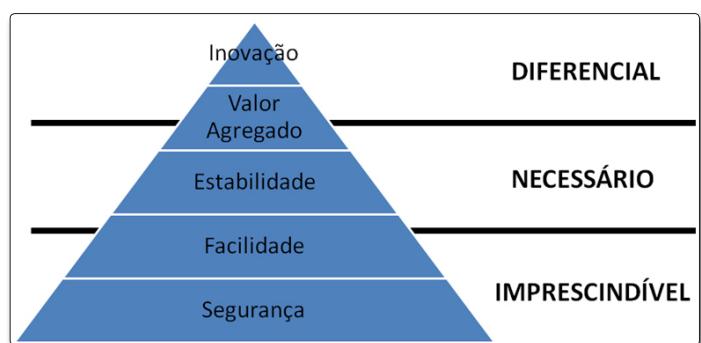


Figura 2. Hierarquia de Implementação de uma Carteira Eletrônica

Requisitos de Segurança:

- **Armazene no smartphone, o mínimo possível de informações confidenciais ou que possam identificar o usuário.** Prefira armazenar no máximo um identificador do usuário ou *login*, e um indicador de sessão. Armazene de forma criptografada, preferencialmente com chave dinâmica ou mesmo *hashes*;
- **Utilize dois ou mais fatores de autenticação.** Pode-se, por exemplo, utilizar uma senha para acesso ao aplicativo no smartphone, e um “PIN” ou senha curta para autenticar cada transação. Mesmo que o smartphone seja perdido, roubado, ou acessado indevidamente, com o aplicativo de Carteira Eletrônica aberto, a requisição de um PIN de quatro ou seis dígitos a cada transação financeira certamente impedirá a retirada indevida dos saldos na carteira;
- **Implemente uma forma de cancelamento e bloqueio remoto da carteira.** Preferencialmente, via Internet. Da mesma forma que na carteira física convencional, o roubo de um smartphone com Carteira Eletrônica requer que o uso de recursos financeiros seja bloqueado imediatamente pelo portador;
- **Esteja atento e siga as normas de PCI-DSS.** O PCI-DSS (acrônimo para *Payment Card Industry – Data Security Standards*, traduzido para Padrões de Segurança de Dados na Indústria de Cartões de Pagamentos) define uma série de regras de segurança que são universalmente seguidas à risca por todos os

participantes da cadeia de meios de pagamentos. Algumas destas regras consistem em não armazenar localmente dados sensíveis, a utilização de criptografia avançada no tráfego de informações, ou não permitir a exibição clara de números de cartões, dentre outras regras de segurança.

Requisitos de Facilidade:

- **Evite navegações complexas.** Da mesma forma que você encontra rapidamente os itens mais acessados em sua carteira convencional, não faça seu usuário ou cliente ter de navegar por menus e opções até encontrar o que precisa. Dentro do possível, evite a digitação pelo teclado, e use o quanto possível os toques em tela;
- **Simplifique termos e fluxos.** Você deseja que sua Carteira Eletrônica seja a mais universal possível, e certamente não deseja receber centenas de ligações de clientes com dúvidas sobre utilização. Não use termos complexos ou em idioma diferente do selecionado pelo usuário, evite termos técnicos, e dentro do possível, simplifique toda e qualquer operação a no máximo três toques na tela;
- **Uma senha pra entrar, outra para transacionar.** Recomenda-se a digitação de uma senha para acesso ao aplicativo de Carteira Eletrônica, mas após o acesso e durante sua utilização e navegação, não solicite esta senha novamente. Apenas no momento de efetuar uma transação financeira, solicite a segunda senha, preferencialmente uma senha numérica entre quatro e oito dígitos;
- **Design não é "frescura".** Lembre-se que sua Carteira Eletrônica está rodando em *smartphones* que podem possuir telas de diferentes tamanhos e rotações, diferentes resoluções e diferentes contrastes. Evite exibir listagens com letras muito pequenas, ou telas poluídas com excesso de informações. Seu time de *design* e de programação *front-end* devem estar atentos às limitações técnicas de exibição e navegação em *smartphones*.

Requisitos de Estabilidade:

- **Seu cliente sempre está com pressa, e segurando uma fila.** Nada frustra mais um consumidor do que não conseguir pagar, bem no momento de pagar. Procure meios técnicos de impedir a ausência ou demora de resposta na sua aplicação, não permita, jamais, que a aplicação trave ou congele, por qualquer motivo. Teste a conexão de Internet antes de iniciar qualquer operação online, e avise o usuário de que não há uma conexão. Não permita que o usuário da aplicação fique mais de sete segundos sem uma resposta satisfatória. Abuse das barras de progresso. Caso sua aplicação faça uso de algum *hardware* específico ou dispositivo externo, certifique-se de que estes dispositivos irão funcionar em qualquer condição de carga de bateria, conectividade, ou processamento. E nunca deixe de explicitar uma lista clara dos aparelhos compatíveis com este dispositivo, antes mesmo do usuário efetuar o *download* e instalação de sua aplicação, e novamente, teste a disponibilidade deste *hardware* antes da utilização;
- **Nem sempre a conexão é boa.** Seu cliente, usuário da aplicação de Carteira Eletrônica, nem sempre terá uma conexão 3G em

velocidade máxima, com 100% de disponibilidade, em todos os locais. Procure utilizar pacotes pequenos e compactos para transmissão de dados, sem esquecer que estes pacotes devem ser codificados ou criptografados. Armazene localmente, no *smartphone*, o máximo possível de informações não confidenciais, tais como textos e vídeos tutoriais, imagens de logotipos e navegação, tabelas técnicas, arquivos de *log* ou telemetria;

- **Forneça mensagens claras.** Se existe um problema durante o processamento de sua aplicação de Carteira Eletrônica, informe o cliente sobre a existência do problema, qual o problema, e qual a alternativa, mesmo que seja simplesmente continuar aguardando ou tentar novamente. Evite utilizar mensagens que sejam vagas em demasia, como “ocorreu um problema com sua solicitação, tente mais tarde”, e nunca utilize mensagens técnicas ou complexas demais, tais como “o servidor retornou erro 500, erro em *PeerTransaction.java* linha 525: null object reference”. Procure dosar o conteúdo das mensagens da aplicação, de forma que sejam explicativas, mas também fáceis de entender;
- **Abuse de Threading e Queuing.** Jamais execute chamadas assíncronas dentro da *Thread* principal de sua aplicação, seja qual for a plataforma ou linguagem utilizada, conforme o **BOX 1**. As *Threads* podem ajudar a exibir notificações e indicadores de progresso enquanto sua aplicação executa uma operação crítica.

**Não perca tempo
reinventando a roda!**

COBREBEMX

**Componente completo para sua
Cobrança por Boleto Bancário
e Débito em Conta Corrente**

**Mais de 40 exemplos
em diversas linguagens
de programação**

**Geração e leitura de arquivos
(remessa e retorno) nos padrões
FEBRABAN e CNAB**

**Testes e Downloads
gratuitos em nosso site**

**ACESSE E CONHEÇA O COMPONENTE EM:
WWW.COBBREM.COM**

Já o *queueing*, ou enfileiramento, pode ser bastante útil quando sua aplicação carrega longos volumes de dados, como listas e imagens. A única exceção para utilização do enfileiramento dá-se durante uma chamada para uma transação financeira. Transações financeiras devem ocorrer *on-line* e em tempo real, e em caso de erros, nunca submeta novamente a transação sem a autorização do usuário. E, mesmo que o usuário autorize que a transação seja efetuada novamente, certifique-se de que a operação já não fora executada em *server-side*.

BOX 1. Threads em aplicações mobile

Uma chamada assíncrona – tais como chamadas de conexão ou manipulação de dados no storage, as quais é impossível saber se, e quando, retornarão uma resposta – executada na Thread principal da aplicação causa travamentos e congelamento da aplicação.

Agregue valor à sua Carteira Eletrônica

Uma Carteira Eletrônica em um *smartphone* é muito mais do que uma aplicação que armazena cartões de crédito e valores, para efetuar pagamentos. Agregar valor não se resume a exibir saldos e extratos. Lembre-se, a Carteira Eletrônica deve ser segura, fácil e rápida de utilizar. Estas características imprescindíveis não são necessariamente um obstáculo para agregar valor, e podem ser vistas como oportunidades.

Primeiramente, a segurança. Além de segurança, credibilidade. Sua Carteira Eletrônica não será considerada confiável para seus clientes enquanto você mesmo – ou sua empresa – não a considerarem confiável. Passar uma imagem de credibilidade e segurança além de atrativo aos clientes, também é atrativo para a empresa que opera a plataforma, não apenas por mitigar os danos à imagem, como também prevenir fraudes. Se sua empresa já possui um relacionamento estável e confiável com os clientes, antes de oferecer-lhe a Carteira Eletrônica, ótimo, este é o cenário ideal. Caso esteja longe do cenário ideal, não desanime, muitas alternativas estão ao seu alcance para manter a credibilidade e confiabilidade de sua plataforma móvel de pagamentos.

Uma Carteira Eletrônica pressupõe alta tecnologia, e principalmente, ambiente *web* e *on-line*. Algo muito diferente e distante de enviar *faxes* e cartas com cópias de documentos. Mas, ainda assim, podemos construir uma plataforma segura. Caso não solicite comprovantes de endereço, envie ao endereço informado pelo cliente um telegrama contendo um código de acesso à sua plataforma. Caso deseje verificar o telefone celular informado, envie um SMS com um código de verificação. Ao cadastrar um novo cartão de crédito, use se possível o *soft descriptor* com um código a ser exibido na fatura do cartão de crédito, ou em alternativa, débito inicialmente um valor pequeno e “quebrado”, em centavos, para que o cliente consulte na fatura qual foi o valor debitado, e informe este valor em sua plataforma. Não se esqueça de devolver este saldo para utilização. Estas ações podem parecer um tanto burocráticas, mas transmitem ao seu cliente uma sensação de segurança e credibilidade, além de evitar significativamente a chance de fraudes.

Outra ação que pode ser adotada em sua Carteira Eletrônica é a de antecipar-se às ações do cliente. Por exemplo, consultar o saldo disponível ou restante, é uma opção interessante, porém não espere seu cliente efetuar esta consulta na Carteira Eletrônica; faça a exibição ao entrar na aplicação ou logo após um pagamento. Exibir extratos e históricos também é um valor agregado não disponível nas carteiras convencionais – obviamente – porém pode ser automaticamente sugerido ao usuário após atingir certo ponto de utilização ou saldo remanescente. Aprenda quais são as ações mais comumente utilizadas por seus clientes na Carteira Eletrônica, e as coloque em destaque, ou em sugestão.

Lembre-se de que sua Carteira Eletrônica está rodando em um *smartphone*, que por mais simples que seja, possui grande poder de processamento e muitos recursos de *hardware*, as possibilidades de agregar valor são inúmeras. Utilize a geolocalização (GPS) para localizar estabelecimentos comerciais e demais clientes de carteiras eletrônicas que estejam próximos. Processe, armazene e utilize, os dados de utilização da carteira. Construa gráficos e imagens dinamicamente. Incremente a segurança com criptografia avançada. Enfim, faça uso do poder de processamento dos *smartphones* para agregar valor ao produto.

Inovando, antes de lançar sua Carteira Eletrônica

Inovação é algo subjetivo, depende da percepção e da criatividade de quem desenvolve um produto. Todavia, nunca esqueça que uma Carteira Eletrônica em um *smartphone* é muito mais do que uma aplicação que armazena cartões de crédito e valores, para efetuar pagamentos. A inovação não pode ser determinada, mas alguns caminhos podem ser sugeridos para uma Carteira Eletrônica de sucesso.

Faça um exercício simples: tire sua carteira do bolso, abra-a, retire dela os cartões de crédito, as folhas de cheque, as notas de dinheiro. O que sobrou dentro de sua carteira? Estes demais itens não podem ingressar também na carteira eletrônica?

Certamente, um dos itens que restaram dentro de sua carteira, para começar, foram seus documentos pessoais. Até que ponto eles não podem estar presentes em sua Carteira Eletrônica? Podemos tanto digitalizar documentos pessoais, ou mesmo fotografá-los, e armazenar em uma carteira eletrônica, quanto podemos fazer uso de certificados digitais.

Outro item que talvez faça parte de sua carteira, e tenha restado dentro dela, são bilhetes com anotações importantes e até mesmo senhas. Sua Carteira Eletrônica não pode incorporar este tipo de informação? Mais um item, talvez não muito presente em muitas carteiras, são os comprovantes de compras. Mas eles existem, mesmo que a maioria das pessoas prefira rasgá-los e jogá-los ao lixo. Sua Carteira Eletrônica não poderia armazená-los? O que mais podemos encontrar em sua carteira, além de pequenas fotos de família? Bilhetes de transporte? Crachá da empresa? Cartões de clubes e afiliações? Moeda estrangeira? Todos estes itens abrem novas possibilidades. A tecnologia está disponível.

Agora, deixe de lado a carteira convencional, e verifique seu *smartphone*. Notou uma câmera? As câmeras de celulares e

smartphones atualmente são capazes de capturar e processar códigos de barras, todos os produtos à venda em lojas físicas possuem código de barras. As contas e boletos no Brasil possuem códigos de barras. Inversamente, um *smartphone* também é capaz de exibir em tela um código de barras que pode ser lido por uma caixa registradora, ou um leitor de bilhetes e ingressos, ou ainda um caixa eletrônico, um terminal de autoatendimento. Além de exibir na tela um código que pode ser lido pela câmera de outro *smartphone*. Considere também, se possível, a utilização da tecnologia NFC nos *smartphones*, ou mesmo a tecnologia *Android Beam*. Pode-se executar transações e troca de dados entre dois aparelhos dotados desta tecnologia, seja entre dois smartphones, ou com um moderno terminal de pagamentos por aproximação. Ou mesmo, bilhetagem, e utilização em leitores de cartões do transporte coletivo.

Conclusão

Planejar a criação de uma Carteira Eletrônica deve obrigatoriamente considerar uma hierarquia de implementação, sendo nesta ordem, a segurança dos dados, a facilidade de utilização, a estabilidade do *software*, incluir funcionalidades que agreguem valor e sejam inovadoras. Considerando que uma Carteira Eletrônica deve ser tão fácil e rápida de utilizar, e segura de carregar, quanto uma carteira convencional destas que carregamos em nossos bolsos ou bolsas, a Carteira Eletrônica pode e deve, agregar valor à carteira convencional.

Agregar valor e inovações não devem ser vistas apenas como uma obrigação, mas sim como um diferencial imprescindível, dada a quantidade atual de diferentes *softwares* de carteiras eletrônicas, e principalmente, considerando as especificidades e complexidades da indústria de meios de pagamentos. A inovação e o valor agregado, assim como em quase tudo no capitalismo, são características especialmente necessárias e requeridas nas soluções de carteiras eletrônicas.

A inovação depende de criatividade, tanto para o desenvolvimento do produto quanto para o planejamento tecnológico, porém alguns caminhos estão disponíveis e já podem ser trilhados. A exemplo, o NFC, e certificados digitais. Estas novas tecnologias ainda são recentes, mas são rapidamente incorporadas aos novos modelos de *smartphones*, mesmo os mais baratos. São tecnologias que permitem integrar novas funcionalidades às carteiras, tais como uso em transporte público, identificação pessoal e confiável, acesso físico a ambientes e catracas, comprovações de transações com valor jurídico, dentre muitas outras funcionalidades possíveis.

Os *smartphones* já possuem a tecnologia necessária para a implementação de uma Carteira Eletrônica segura, fácil, e rápida. As portas estão abertas à inovação neste meio de pagamento.

Autor



Renato Galdino Machado

macrenato@gmail.com

Possui 14 anos de experiência em tecnologia e programação, sendo os últimos 10 anos dedicados à pesquisa e desenvolvimento de aplicações mobile para o mercado financeiro. Participou de projetos inovadores, como o primeiro mobile payment homologado da América do Sul, além de projetos internacionais de adquirência financeira em mobile. Atualmente é colaborador em uma empresa de adquirência em mobile payment, atuando na pesquisa e desenvolvimento de novos meios de captura em mobile, tais como MagReaders, Chip&PIN, micro-impressoras, dentre outros.



Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/webmobile/feedback

Ajude-nos a manter a qualidade da revista!



Analisando o comportamento de aplicações Android

Conheça o Flurry para análise de eventos e crash report

De acordo com os dados do GSMA Intelligence, vide **BOX 1**, existe por volta de 6.3 bilhões de celulares espalhados pelo mundo das mais diversas tecnologias, modelos, fabricantes e operadores de telefonia móvel.

Diante deste número tão impressionante é fácil perceber a popularização do celular e a sua modernização tanto no que diz respeito ao hardware quanto ao software.

Segundo Andy Rubin, cofundador do Android, não existe nada que o usuário possa acessar de seu computador tradicional que não possa acessar de seu celular. No cenário atual, esta com certeza é uma afirmação verdadeira. A complexidade e a variedade de funcionalidades e recursos que podem ser explorados hoje por uma aplicação mobile são enormes.

Infelizmente a qualidade de muitas aplicações disponibilizadas em lojas virtuais de aplicativos é pobre, sem contar os inúmeros bugs que são encontrados após atualizações.

Criar uma aplicação mobile que tenha aceitação pelos usuários de smartphones é o sonho de qualquer desenvolvedor nos dias de hoje. Porém, criar uma solução de sucesso não é uma tarefa fácil. Não basta ter uma boa ideia, tem que transformar esta ideia em produto e, num produto de qualidade, pois caso contrário há grandes chances de perder a popularidade em um intervalo curto de tempo.

Criar um produto que tenha os mesmos recursos e comportamentos em plataformas mobile distintas é um desafio sem sombra de dúvida, tendo em vista uma série de variáveis a serem controladas nas principais plata-

Resumo DevMan

Porque esse artigo é útil:

Neste artigo iremos apresentar a plataforma Flurry para análise de Crash Reports e análise de eventos. O Flurry além de disponibilizar um SDK enxuto e objetivo, oferece também um ambiente online para visualização e análise dos dados coletados de uma aplicação mobile. Este tipo de ferramenta é bastante útil no cenário atual, que é extremamente competitivo no que se refere ao mundo de aplicativos para smartphones. Neste cenário, a discussão deste tema é útil para profissionais da área de desenvolvimento mobile que queram obter mais informações sobre a utilização de sua solução para obter um diferencial competitivo.

formas mobile de mercado como: modelos, fabricantes, versões do sistema operacional etc.

Posso afirmar que é impossível prever todos os pontos de falha durante o desenvolvimento de um produto mobile, por mais experientes que sejam os profissionais envolvidos.

Acredito que uma das principais perguntas a serem respondidas numa nova empreitada para criar um produto mobile é: Como minimizar os pontos de falhas da minha solução? Como identificá-los? Como dar uma resposta a um cenário de falha onde não é fácil reproduzi-lo?

E é neste cenário que as soluções de crash reports ajudam e muito a equipe a identificar pontos de falhas e tentar dar uma resposta rápida com base na análise de dados fornecidos por estas ferramentas.

Somente com as informações fornecidas pelas lojas de aplicativos não é possível conduzir uma análise focando em pontos de falhas e utilização do aplicativo.

Ter sucesso em um produto não é receita de bolo, porém o sucesso é fruto de informações precisas e respostas rápidas a fim de não afetar a imagem do produto.

No decorrer do artigo iremos apresentar o Flurry, um produto bastante interessante no seguimento de ferramentas para crash reports e análise de eventos das funcionalidades do aplicativo.

BOX 1. GSMA Intelligence

O GSMA Intelligence é uma base de dados que concentra dados de telefonia móvel em todo mundo, utilizando estas informações para análise e previsões mercadológicas utilizando seus dados como métricas para gerar informações mais precisas para indústria de telecomunicações.

Possui uma base de mais de 800 clientes em todo mundo. Entre os principais clientes são encontradas operadoras de telefonia móvel, vendedores e fabricantes de equipamentos.

Entendo melhor as ferramentas de Crash Reports

As principais plataformas de mobile possuem suas próprias lojas de aplicativos, assim como um dashboard para os desenvolvedores disponibilizarem seus produtos e obterem algumas informações sobre versões, resenhas de usuários e uma análise sobre o uso dos aplicativos.

Conduzir uma análise sobre a qualidade do produto utilizando apenas os dados fornecidos pelas lojas e tendo que responder perguntas como as apresentadas a seguir é muito complicado: Quais versões do meu produto que apresentaram problemas? Em quais dispositivos? Em quais versões de sistema operacional? É praticamente impossível identificar o foco do problema.

Para ter uma pró-atividade maior e conduzir uma análise mais assertiva, é bastante evidente que são necessárias mais informações sobre os problemas relatados pelos usuários.

Focado nesta carência de informações não fornecidas pelas lojas de aplicativos, surgiu a necessidade de ferramentas que agreguem valor ao desenvolvimento do produto mobile coletando estas informações, armazenando, classificando e quantificando estas informações.

No mercado existe uma grande quantidade de opções de ferramentas que oferecem este tipo de solução, tanto ferramentas pagas quanto gratuitas. Acredito que o mais importante no momento de escolher uma ferramenta é o quanto esta ferramenta está em sintonia com as principais plataformas de mobile do mercado.

Se estamos desenvolvendo um produto que irá possuir versões para Android, iPhone e Windows Phone, por exemplo, é muito interessante que seja utilizada a mesma ferramenta para captura das informações. Este é um fator muito importante, pois utilizar ferramentas diferentes para plataformas diferentes

pode gerar mais trabalho na hora de iniciar uma análise olhando para informações coletadas por elas.

Nem todas as ferramentas possuem os mesmos recursos ou disponibilizam as informações e o acesso a estas da mesma forma. Portanto, escolha bem antes de decidir por uma ferramenta A ou B.

Levando em considerações todas as ferramentas de Crash Reports e análise de eventos que trabalhei, duas me chamam bastante atenção.

O Google Analytics, que disponibiliza uma integração para produtos mobile nas plataformas Android e iPhone e, o Flurry, que para mim é a melhor opção, por disponibilizar integrações para quase todas as plataformas mobile conhecidas no mercado.

O Flurry é uma ferramenta bastante completa, disponibilizando um SDK para cada plataforma e um excelente Dashboard online para análise destas informações.

A seguir será apresentando um pouco dos primeiros passos para ter acesso a ferramenta, e um breve passo a passo para criar um mecanismo de monitoramento para uma aplicação Android fictícia.

Apresentando o Flurry

O Flurry é uma ferramenta que possui opções gratuitas e pagas. O produto que iremos apresentar neste artigo é o Flurry Analytics que é gratuito.

Para ter acesso ao SDK do Flurry é necessário criar uma conta e uma chave de identificação desta aplicação.

Para criar sua conta, acesse o link apresentado na seção links e procure pelo link *SIGN UP*, no topo direito da página principal do site. Siga os passos informados pelo site, que é bastante simples, e rapidamente você criará uma conta para ter acesso aos recursos do Flurry Analytics.

Após ter criado sua conta, na **Figura 1** temos a primeira visão dos recursos oferecidos pela plataforma Flurry. Como não temos ainda nenhuma aplicação integrada, o próprio Dashboard disponibiliza um link, *Create your first application*, para criar nossa primeira chave de integração para uma plataforma mobile.

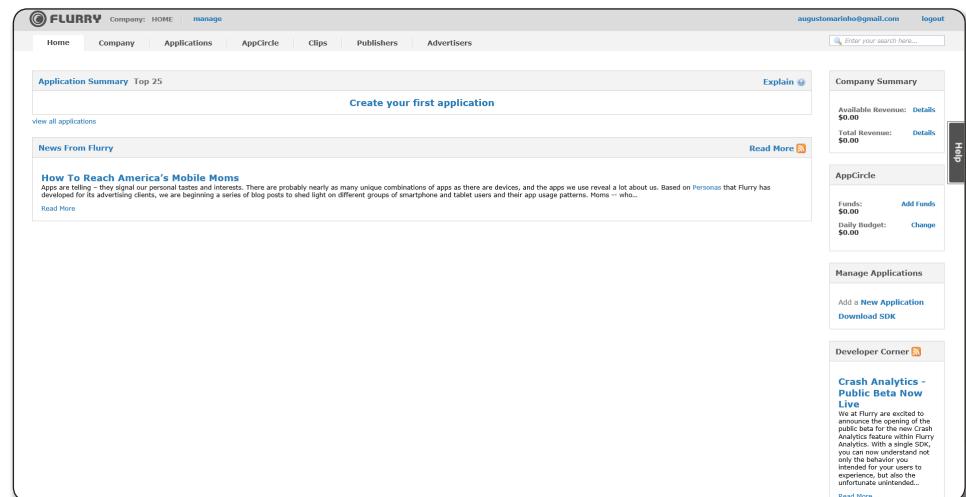


Figura 1. Primeira Visão do Flurry após criar uma conta

Analisando o comportamento de aplicações Android

Após clicarmos no link, somos solicitados a selecionar a plataforma mobile que será integrada, conforme **Figura 2**.

Percebem que, como citado anteriormente, estas são as opções de plataformas que o Flurry disponibiliza integração: Android, iPhone, JME, Black Berry e API para in-

tegração com aplicações Web. Em nosso exemplo foi selecionada a opção Android.

Após termos selecionado a plataforma Android, conforme **Figura 2**, somos encaminhados para informarmos o nome da aplicação e uma categoria, conforme apresentado na **Figura 3**.

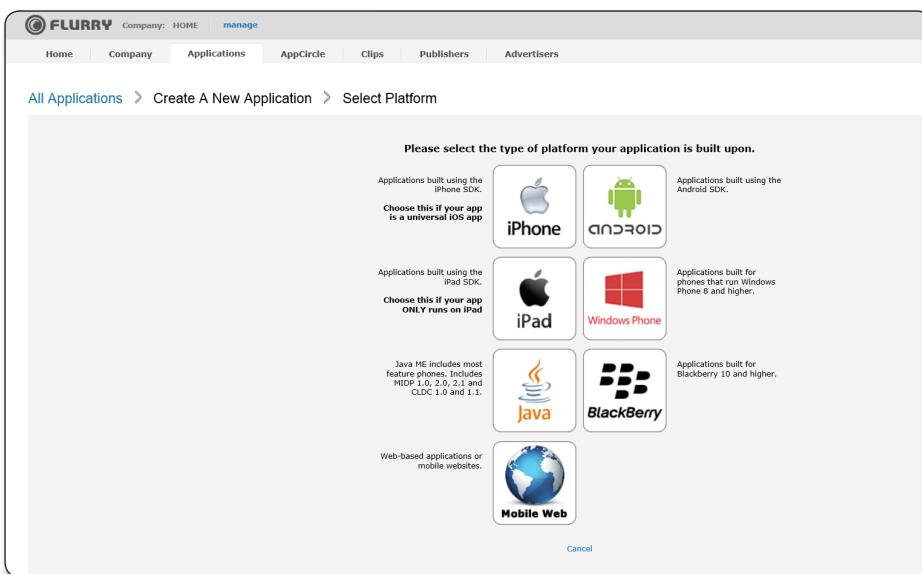


Figura 2. Selecionado a plataforma mobile

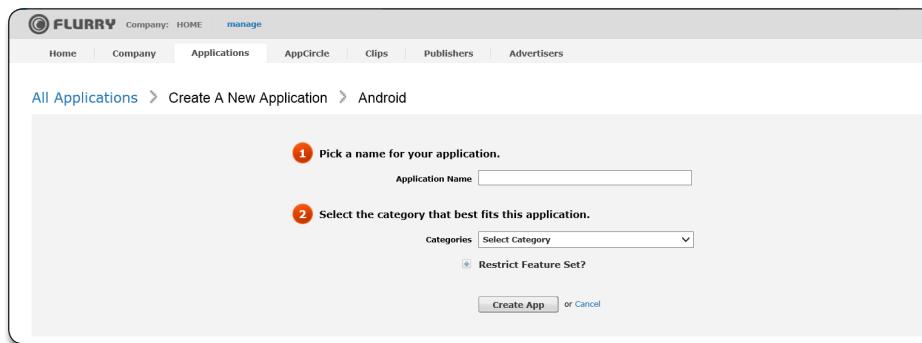


Figura 3. Maiores detalhes sobre o foco da aplicação

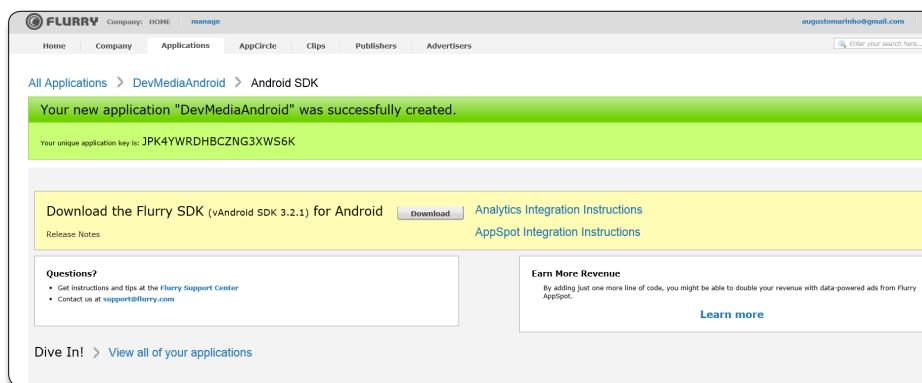


Figura 4. Criação da chave do aplicativo

Após termos informado a categoria, a plataforma Flurry cria um API Key, que deverá ser utilizado durante o desenvolvimento desta aplicação Android, conforme pode ser visto na **Figura 4**.

Ainda nesta mesma página, é disponibilizado um link para ser feito o download do SDK da plataforma Android. Junto com o SDK, que basicamente é um único JAR, temos alguns PDFs e uma documentação estilo JavaDoc.

Ao clicarmos na opção *Home*, no topo da página apresentado na **Figura 4**, voltamos à página principal do Flurry e já podemos perceber, conforme **Figura 5**, que a plataforma já está configurada para receber informações coletadas pelo SDK do Flurry para aplicação Android nomeada de **DevMediaAndroid**.

Flurry Analytics e o SDK para Android

OSDK do Flurry para plataforma Android se resume um único JAR, *Flurry_3.2.1.jar*, que para ser integrado com uma aplicação, basta copiá-lo para dentro do diretório / *lib* do projeto.

Depois de copiado o JAR, é necessário configurar o arquivo *AndroidManifest.xml* adicionando as permissões *android.permission.INTERNET* e *android.permission.ACCESS_STATE*, conforme apresentado nas linhas 11 e 12 na **Listagem 1**.

As permissões *android.permission.INTERNET* e *android.permission.ACCESS_STATE* são obrigatórias e sem elas será lançado um *SecurityException* ao executar a aplicação.

Ainda sobre a **Listagem 1**, existem mais duas permissões configuradas nas linhas 14 e 15, *android.permission.ACCESS_COARSE_LOCATION* e *android.permission.ACCESS_FINE_LOCATION*, porém estas são opcionais e apenas são necessárias se sua aplicação desejar ter maiores informações sobre a localização dos usuários que estão utilizando seu aplicativo. Caso estas permissões não sejam utilizadas, a única informação de localização que o Flurry irá prover é o país de origem.

É possível desabilitar as informações de localização, porém iremos comentar sobre estas configurações um pouco mais a frente.

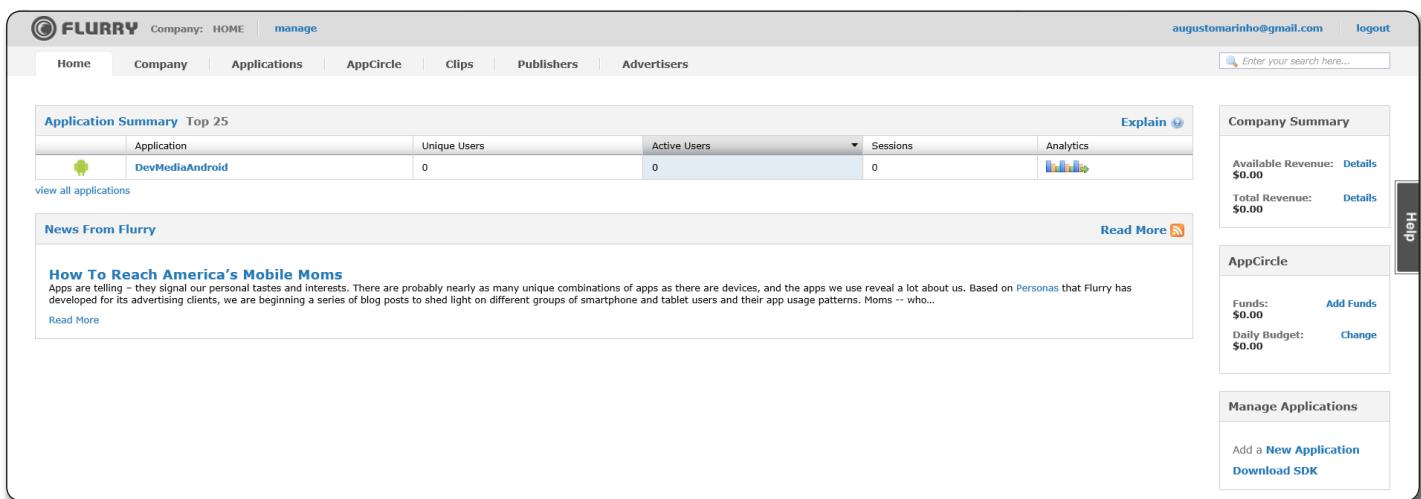


Figura 5. Primeira visão após criação da chave

Listagem 1. Configurando Flurry no AndroidManifest.xml.

```

01 <?xml version="1.0" encoding="utf-8"?>
02 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
03   Package="com.devmediaandroid.flurry"
04   Android:versionCode="1"
05   Android:versionName="1.0"
06   <uses-sdk
07     Android:minSdkVersion="8"
08     Android:targetSdkVersion="17"/>
09   <uses-permission android:name="android.permission.ACCESS_COARSE_
LOCATION"/>
10  <uses-permission android:name="android.permission.ACCESS_FINE_
LOCATION"/>
11  <uses-permission android:name="android.permission.INTERNET"/>
12  <uses-permission android:name="android.permission.INTERNET"/>
13
14 <application
15   Android:allowBackup="true"
16   Android:icon="@drawable/ic_launcher"
17   Android:label="@string/app_name"
18   Android:theme="@style/AppTheme">
19   <activity
20     Android:name="com.devmediaandroid.flurry.MainActivity"
21     Android:label="@string/app_name">
22     <intent-filter>
23       <action android:name="android.intent.action.MAIN"/>
24       <category android:name="android.intent.category.LAUNCHER"/>
25     </intent-filter>
26   </activity>
27 </application>
28 </manifest>
```

O Flurry envia as informações para o serviço em lote dentro de um escopo de sessão. Uma sessão para o Flurry é iniciada quando é realizada a chamada **FlurryAgent.onStartSession(Context, String)**, onde o **Context** passado deve ser a própria Activity e o segundo parâmetro deve ser a **API KEY** criada pelo Flurry para esta aplicação.

A API KEY pode ser encontrada dentro do dashboard do Flurry, ou juntamente com o SDK após ter realizado o download. A chave vem dentro do arquivo compactado do SDK num arquivo de texto. O Flurry sabe disponibilizar esta informação

junto como SDK, pois antes de realizar o download, deve ser selecionada a plataforma/aplicação.

Para finalizar uma sessão basta efetuar a chamada **FlurryAgent.onEndSession(Context)**, onde o **Context** deve ser a mesma **Activity** passada para o método que criou a sessão.

É recomendado que a chamada **FlurryAgent.onStartSession(Context)** seja realizada no método **onStart()** de uma Activity, e a chamada **FlurryAgent.onEndSession(Context)** seja realizado no método **onStop()** da Activity.

Se a chamada **FlurryAgent.onStartSession(Context, String)** for executada e não for realizada uma chamada para finalizar a sessão, a sessão do Flurry continuará ativa.

Se uma nova Activity for criada e o método de início de sessão for chamado novamente dentro de um intervalo de 10 segundos, sem uma chamada de finalização de sessão, a mesma sessão será utilizada para capturar as informações ao invés de ser criada uma nova sessão.

Este cenário pode ser um problema ou não, depende muito de como você queira visualizar os dados no Dashboard do Flurry.

Utilizando apenas estas duas chamadas em cada uma de suas Activities, ou na Activity que deseja obter informações, já é possível analisar os dados enviados pelo Flurry como: modelo de dispositivos, versões do sistema operacional, localização dos usuários, etc.

Como citado anteriormente, o Flurry captura informações de localização dos usuários. É de bom tom não ser muito intrusivo com relação à localização, e caso a sua aplicação não queira capturar a localização dos usuários de seu aplicativo, deve ser realizada uma chamada **FlurryAgent.setReportLocation(false)** antes de ser feita uma chamada ao método que inicia uma sessão.

Caso queira enviar para plataforma Flurry dados específicos, deverá ser feito uso de alguns de seus outros recursos para criar seus próprios eventos e analisar estes dados posteriormente.

O tempo de uma sessão do Flurry vem definido por padrão e caso seja necessário reduzir ou aumentar este intervalo de envio de dados para a plataforma Flurry, é possível alterar este dado através

da chamada `FlurryAgent.setContinueSessionMillis(long milliseconds)` antes de ser executado o método de início de sessão.

Recursos opcionais do Flurry Analytics

Além das informações básicas sobre a plataforma mobile, versão, modelo de aparelho, localização etc., o Flurry ainda viabiliza a criação de eventos específicos da sua aplicação para que seja possível criar pontos de controle mais refinados conforme a necessidade de cada aplicativo.

Para criar registro de eventos específicos no Flurry, o seu SDK disponibiliza as seguintes chamadas:

- `FlurryAgent.logEvent(String eventId);`
- `FlurryAgent.logEvent(String eventId, boolean timed);`
- `FlurryAgent.logEvent(String eventId, Map<String, String> parameters);`
- `FlurryAgent.logEvent(String eventId, Map<String, String> parameters, boolean timed).`

Existem quatro sobrecargas do método `FlurryAgent.logEvent()` para realizar o registro de eventos específicos na plataforma Flurry.

Através destes métodos é possível obter informações da frequência de cada evento no aplicativo, a ordem em que estes eventos são registrados, o intervalo de tempo entre os eventos etc.

Cada chave registrada no Flurry para um aplicativo/plataforma pode registrar no máximo 300 eventos classificados pelo nome e cada identificador de evento um mapa de eventos chave/valor, onde o valor de cada parâmetro pode conter no máximo 255 caracteres.

Cada evento que informe uma mapa de eventos pode conter apenas 10 parâmetros. O mapa de eventos é um parâmetro opcional e pode ser passado nulo para o método.

Uma informação importante é que cada sessão do Flurry pode registrar no máximo 200 eventos.

Caso seja necessário registrar eventos de possíveis exceções lançadas no aplicativo, o SDK do Flurry disponibiliza a chamada `FlurryAgent.onError(String errorId, String message, Throwabe exception)`. É permitido o reporte de 10 eventos de erro por sessão.

Com relação às exceções, o Flurry por padrão envia crash de exceções de tempo de execução, que são aquelas exceções em que o aplicativo é encerrado inesperadamente. Porém, caso queira que esta funcionalidade seja desabilitada, basta fazer uma chamada ao método `FlurryAgent.setCaptureUncaughtExceptions(false)` antes da chamada de um método de início de sessão.

O Flurry também nos permite fazer um contador de acesso a páginas do seu aplicativo e registrar estas informações separadas do registro de eventos. Para registrar um acesso a uma página basta fazer uma chamada ao método `FlurryAgent.onPageView()`.

Por padrão, os registros de eventos são enviados por HTTP. Porém, caso seja trafegada alguma informação sensível de seu aplicativo é possível solicitar ao SDK que as mensagens sejam enviadas por HTTPS, fazendo uso do método `FlurryAgent.setUseHttps(true)`.

Para finalizar os recursos opcionais, é possível coletar informações um pouco mais específicas, caso o seu aplicativo solicite estas informações ao usuário é claro, como idade, sexo e login. Estes registros podem ser realizados respectivamente através de chamadas aos métodos `FlurryAgent.setAge(int)`, `FlurryAgent.setGender(byte)` e `FlurryAgent.setUserID(String)`.

A idade do usuário pode ser informada no intervalo de 1 a 109. O sexo do usuário pode ser informado pelas constantes `Constants.MALE` ou `Constants.FEMALE`. O login do usuário é livre.

Pequeno exemplo de integração Flurry com aplicações Android

Como já citado neste artigo, para criar uma aplicação Android que disponibilizará informações de Crash Reports e análise de eventos para plataforma Flurry, basta copiar o arquivo JAR para o diretório `/lib` do projeto da aplicação Android e adicionar as permissões `android.permission.INTERNET` e `android.permission.ACCESS_STATE` no arquivo `AndroidManifest.xml`.

Na **Listagem 2** é apresentada a classe `MainActivity`, que é Activity principal da aplicação de demonstração. Com apenas estas duas chamadas já será possível disponibilizar para o Flurry informações básicas dos usuários de sua aplicação.

Listagem 2. Pequena demonstração de código

```
01 package com.devmediaandroidflurry;
02
03 import android.app.Activity;
04 ...
05
06 public class MainActivity extends Activity{
07
08     @Override
09     protected void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.activity_main);
12     }
13
14     @Override
15     protected void onStart(){
16         super.onStart();
17         Log.i("FLURRY","Criando Sessao Flurry");
18         FlurryAgent.onStartSession(this,"JPK4WRDHBC2NG3XW56K");
19     }
20
21     @Override
22     protected void onStop() {
23         super.onStop();
24         Log.i("FLURRY","Finalizando Sessao Flurry");
25         FlurryAgent.onEndSession(this);
26     }
27 }
```

Na **Listagem 2** temos uma pequena implementação do registro de eventos específicos que podem ser criados conforme necessário nos aplicativos. Em específico, criamos um evento com o nome **"Primeiro Log específico Flurry!"**. Observe que estes eventos são registrados em log nas linhas 20 (para o `onStart`) e 27 (para o `onStop`).

Mesmo que o usuário não esteja conectado à Internet no momento de execução da aplicação, o Flurry cria uma fila destas informações e guarda um cache temporário no aplicativo para ser enviado assim que o telefone do usuário estiver conectado a Internet. Todos estes passos são executados de forma transparente para o usuário, não impactando em nada a experiência do usuário.

Na **Listagem 3** podemos visualizar uma evidência do Flurry enviado informações para o servidor de forma transparente, bastando apenas que o usuário entre no aplicativo. Observe também que um dos eventos enviados é justamente o que programamos na **Listagem 2**.

Listagem 3. Evidência de envio de informações para o Flurry.

```
107-11 14:51:0... 12105 com.devmediandroidflurry system.out waiting for
debugger to settle...
107-11 14:51:0... 12105 com.devmediandroidflurry system.out waiting for
debugger to settle...
107-11 14:51:0... 12105 com.devmediandroidflurry system.out waiting for
debugger to settle...
107-11 14:51:0... 12105 com.devmediandroidflurry system.out waiting for
debugger to settle...
107-11 14:51:0... 12105 com.devmediandroidflurry system.out waiting for
debugger to settle...
107-11 14:51:0... 12105 com.devmediandroidflurry system.out waiting for
debugger to settle...
107-11 14:51:0... 12105 com.devmediandroidflurry system.out debugger has
settled (1496)
107-11 14:51:0... 12105 com.devmediandroidflurry FLURRY Criando Sessao
Flurry
107-11 14:51:0... 12105 com.devmediandroidflurry dalvikvm threaded=1: still
suspended after undo (sc=1 dc=1)
107-11 14:51:0... 12105 com.devmediandroidflurry dalvikvm threaded=1: still
suspended after undo (sc=1 dc=1)
107-11 14:51:0... 12105 com.devmediandroidflurry dalvikvm GC_CONCURRENT
freed 239k, 4% free 9374K/9671K, paused 4ms+2ms
107-11 14:51:0... 12105 com.devmediandroidflurry FlurryDa... --onReport...
HTTP response...
```

Os dados enviados para o Flurry não são disponibilizados em tempo real nos dashboards, existe um tempo de processamento destas informações para estarem disponíveis para visualização. O Flurry não precisa o intervalo de tempo necessário, diz apenas que em algumas horas as suas informações estarão disponíveis, porém durante a escrita deste artigo, os dados de testes foram disponibilizados vinte e quatro horas após o envio, porém não é possível tomar este tempo como absoluta verdade.

Na **Figura 6** é apresentando uma evidência dos dados enviados e posteriormente processados pelo Flurry e apresentado no Dashboard de novos usuários.

Na **Listagem 2** apresentamos uma pequena implementação do registro de eventos específicos que podem ser criados conforme necessário nos aplicativos. Neste exemplo, criamos um evento com o nome “**Primeiro Log específico Flurry!**”.

Na **Figura 7** é possível visualizar o registro deste evento no Dashboard de eventos do Flurry.

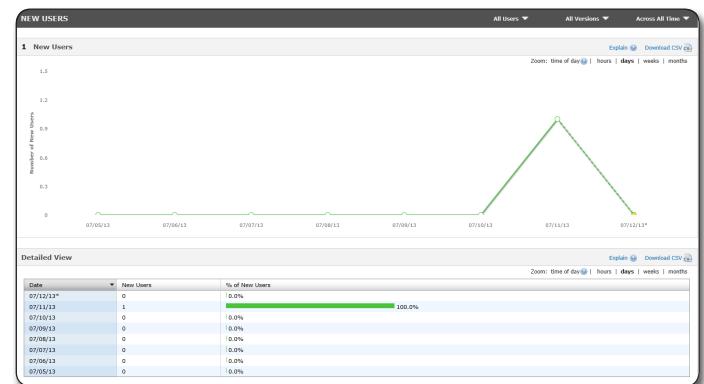


Figura 6. Dashboard novos usuários

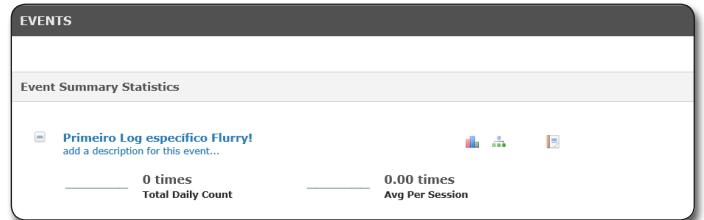


Figura 7. Registro de eventos no Flurry

Consulta a dados de aplicações integradas com o Flurry

O Flurry possui uma integração baseada em XML e JSON para viabilizar a exportação dos dados capturados para outros sistemas. Este serviço não é habilitado por padrão na plataforma Flurry, para habilitar este recurso, vá até a página do Flurry, ver seção links, e faça autenticação.

Depois de autenticado, clique no link *manage*, no topo a direita da página principal. Ao carregar a página, no rodapé haverá um botão *Enable API Access*, conforme apresentado na **Figura 8**.

Após habilitar a integração, será gerada uma chave para ser utilizada nas chamadas HTTP, conforme apresentado na **Figura 9**.

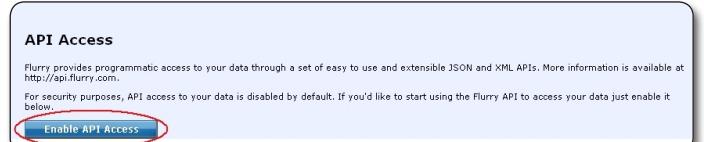


Figura 8. Habilitar integração para acesso aos dados



Figura 9. Chave de integração para acesso aos dados

De posse da chave de acesso à integração, já é possível realizar chamadas à plataforma Flurry para obter os dados de integração. Esta integração viabiliza o acesso às métricas padrão da aplicação

capturadas pelo Flurry. Estas métricas são aquelas providas unicamente pela chamada de `FlurryAgent.onStartSession(Context, String)` e `FlurryAgent.onEndSession(Context)`.

A integração também fornece informações referentes aos eventos especificamente criados na aplicação, assim como informações sobre versões da aplicação, plataformas, países em que a aplicação é utilizada entre outras.

A API que disponibiliza as chamadas é bastante extensa e seria fruto de outro artigo para explicar e demonstrar todas as chamadas possíveis. Para maiores detalhes e aprofundamento sobre a integração acesse o link disponibilizado na seção links.

Uma informação importante sobre esta API disponibilizada pelo Flurry é o seu controle de acesso à integração. É permitida apenas uma quantidade limitada de chamadas por minuto. Caso o intervalo entre uma chamada e outra seja considerado intrusivo, o intervalo de resposta às consultas pode ser ampliado podendo até ocasionar a rejeição das chamadas.

Conclusão

O uso de ferramentas de Crash Reports e análise de eventos com certeza é uma possibilidade que, se bem utilizada, pode proporcionar muitos benefícios para os desenvolvedores de soluções mobile.

O Flurry é uma ferramenta bastante completa que oferece muitos recursos para analisar as informações capturadas, disponibilizando alguns dashboards padrões e ainda permitindo que sejam criadas dashboards personalizados.

Um fator bastante interessante é que o Flurry Analytics é gratuito e pode ser utilizado nas principais plataformas mobile de mercado.

Autor



Augusto Marinho

augustomarinho@gmail.com

Analista de Sistemas formado pela UNESA atua como desenvolvedor de software para o mercado de pagamentos móveis e serviços de valor agregado na área de telecomunicações na empresa M4U utilizando principalmente Java EE. Em paralelo, atua com desenvolvimento para as principais plataformas mobile do mercado.



Links:

Abertura de conta Flurry

<http://www.flurry.com>

Desenvolvedor flurry

<http://dev.flurry.com>

Projeto flurry

<http://wiki.flurry.com>

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/webmobile/feedback

Ajude-nos a manter a qualidade da revista!



VOCÊ É ASSINANTE **MVP?**

ENTÃO PODE TER CERTEZA:
**O MERCADO
PREFERE VOCÊ!**



**DESENVOLVEDOR ATUALIZADO É O MAIS
VALIOSO DO MERCADO!**

Muitos profissionais estão acomodados, não se atualizam, não leem livros e não vão a eventos. Acham que é a empresa que deve mantê-los atualizados. Gastam dinheiro com tudo, mas acham caro investir menos de R\$60,00 por mês em suas carreiras.

Mas você não! Você é MVP! Parabéns!

TENHA ACESSO A:

 + DE **260** CURSOS ONLINE

 **09** REVISTAS MENSais

 **7.850** VÍDEO-AULAS

**POR APENAS 59,90
MENSais**



QUEM TEM ESTÁ TRANQUILO.



Acesse: www.devmedia.com.br/mvp

Desenvolva um programa de cadastro usando SQLite e Adapters

Aprenda a manipular dados e adapters na prática

O uso dos smartphones cresceu muitos nos últimos anos. Isso acarretou em um grande esforço das fabricantes em melhorias no hardware e das gigantes de software um reposicionamento quanto aos sistemas operacionais para estes tipos de dispositivos. E isso gerou uma bola de neve que trouxe grandes e constantes inovações no setor, refletindo cada vez mais em usuários e desenvolvedores mais satisfeitos com as novas tecnologias, que os permitem ficar mais tempo conectados.

Isso também teve outra consequência direta: os usuários passaram a armazenar seus dados não só nos computadores Desktop, mas também em seus smartphones. Logo, estes dispositivos passaram a receber uma quantidade maior e crescente de dados. Principalmente dados de áudio e mídia.

Logicamente, as plataformas móveis perceberam que teriam que fornecer métodos de persistência de dados mais sofisticados que o tradicional armazenamento baseado em arquivos, como acontecia com o RMS (Record Management System), este utilizado pelos programadores Java ME/MIDP para a persistência dos dados. Hoje em dia, todas as principais plataformas oferecem mais de uma maneira de salvar e recuperar informações produzidas por nossos aplicativos.

Com a plataforma Android não poderia ser diferente. Este fornece diferentes métodos de persistência. Alguns deles são através de armazenamento interno (no smartphone) e externo (cartão de memória). Ambos podem ser muito úteis para fazer armazenamento de imagens, por exemplo. No caso de armazenamento em cartão, deve-se atentar para não utilizá-los no armazenamento de informações essenciais para o correto funcionamento do aplicativo, pois a mídia pode ser removida a qualquer momento, e o aplicativo deixará de ser executado.

Resumo DevMan

Porque esse artigo é útil:

Neste artigo será apresentada uma forma simples de utilizar o banco de dados SQLite na plataforma Android. Para isso, serão apresentadas as principais classes do Android no uso do SQLite: SQLiteOpenHelper e SQLiteDatabase e como utilizá-las dentro de um Activity. Para isso, construiremos um aplicativo completo para gerenciamento de artigos publicados. Nesta aplicação iremos considerar as quatro operações básicas no banco de dados, chamadas conceitualmente de CRUD (Create, Read, Update e Delete).

O leitor deste artigo estará apto a fornecer uma arquitetura correta de persistência de dados através de banco de dados relacionais no Android. Este é um quesito muito importante para qualquer profissional que deseja ser visto com bons olhos pelo exigente mercado mobile.

Além disso, a plataforma também fornece as classes chamadas Shared Preferences, sendo esta uma forma de persistir pares de chave/valor para todos os tipos primitivos Java e mais o tipo String. Algo semelhante à utilização de Properties na programação Java tradicional. Seu uso e funcionamento são muito simples, podendo ser muito útil para persistências simples, como a pontuação de um jogo, por exemplo.

Porém, o Shared Preferences não é indicado para grande quantidade de dados ou conjunto de informações com grande complexidade. Nesse momento entram em ação os bancos de dados relacionais, utilizados há alguns anos em outros ambientes, como web e desktop, por exemplo. No Android o banco de dados mais utilizado é o SQLite.

Uma característica interessante da plataforma é que o SQLite já está disponível na plataforma Android, não havendo necessidade de instalá-lo. Além disso, o Android oferece suporte completo

ao banco, através de uma API com um rico conjunto de classes e métodos que abstraem as complexidades dos códigos SQL. Assim, não precisamos montar a cláusula SQL inteira para atualizar uma linha na tabela, ou ainda, para fazer uma pesquisa na mesma. O Android nos fornece um método, onde passando alguns parâmetros obtemos um apontador para os dados retornados, podendo navegar pelo resultado como se estivéssemos escolhendo uma folha em um arquivo.

Assim, o objetivo deste artigo é apresentar uma forma simples de utilizar o banco de dados SQLite na plataforma Android. Aqui apresentaremos as principais classes do Android no uso do SQLite: SQLiteOpenHelper e SQLiteDatabase e como utilizá-las dentro de um Activity.

Para uma compreensão melhor, o leitor construirá um aplicativo completo para gerenciamento de artigos publicados. Desta forma, veremos como fazer as quatro operações básicas no banco de dados, chamadas conceitualmente de CRUD (Create, Read, Update e Delete) e ao final do artigo serão apresentadas as transações exclusivas.

0 projeto

Todo escritor gosta de arquivar suas publicações e organizar quais estão pendentes, quais estão sendo revisados, quais estão prontos para publicações e quais já estão nas bancas. Então, o projeto que vamos criar neste estudo de caso é um gerenciador de artigos, este chamado de “ManagerArticles”.

O funcionamento é bem simples. Na primeira tela teremos uma lista com os artigos cadastrados e a opção de inserir um novo registro, conforme **Figura 1**. Tanto o botão adicionar como a opção de editar apresentará a tela mostrada na **Figura 2**. A diferença é que no caso da edição os campos são preenchidos com o conteúdo armazenado no banco de dados.

Construindo a aplicação sem o banco de dados

Inicialmente vamos construir os elementos e as telas da aplicação, sem nos preocuparmos com o banco de dados.

A classe utilitária Artigo

O primeiro passo será a construção de uma classe Artigo. Ela será utilizada em diferentes pontos do nosso aplicativo. Isso porque o nosso banco de dados terá as ações CRUD (Create, Read, Update e Delete) que envolvem os mesmos atributos, relacionados a mesma entidade. Desta forma, a classe Artigo será uma “visão” da tabela Artigo em um banco de dados relacional tradicional. Veja a **Listagem 1**.

Perceba que os atributos desta classe são os mesmos apresentados na **Figura 2**. Isso porque ao inserir um novo artigo temos que definir o nome, a revista e edição onde foi/serão publicados, seu status e se o mesmo já foi pago ou não ao autor.

O atributo *serialVersionUID* é utilizado porque a classe implementa a interface *Serializable*, assim, se fez necessário este atributo para que possamos enviar uma instância da mesma como um extra de uma *Intent*.

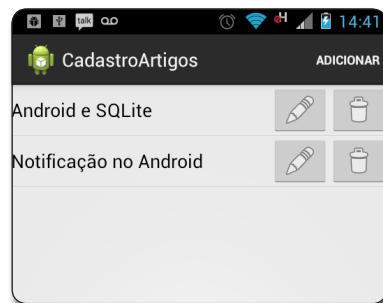


Figura 1. Tela principal do aplicativo

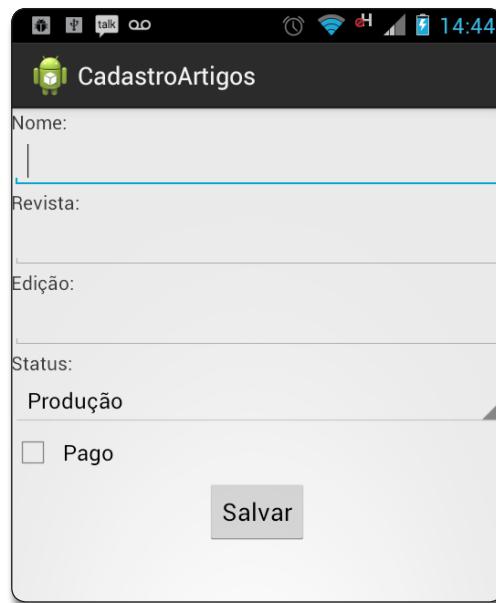


Figura 2. Tela para cadastro/edição de artigos

Listagem 1. Artigo.java – Classe responsável pelo armazenamento dos campos da tabela.

```
01. import java.io.Serializable;
02.
03. public class Artigo implements Serializable {
04.     private static final long serialVersionUID = 1633833011084400384L;
05.     int id;
06.     String revista;
07.     String nome;
08.     String edicao;
09.     int status;
10.     int pago;
11. }
```

Layouts das telas

O segundo passo é criar os arquivos XML que representam as duas telas da aplicação, estas apresentadas nas **Figura 1 e 2**. A **Listagem 2** apresenta o código da tela responsável para inclusão e alteração de um registro.

Na primeira linha temos o cabeçalho XML. Na linha 2 temos o TextView que servirá de contêiner a todos os outros elementos de UI que serão apresentados na tela. Como desejamos um

Desenvolva um programa de cadastro usando SQLite e Adapters

Listagem 2. novo_edicao.xml – Tela para incluir ou alterar um registro.

```
01.<?xml version="1.0" encoding="utf-8"?>
02.<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03.    android:layout_width="match_parent"
04.    android:layout_height="match_parent"
05.    android:orientation="vertical">
06.
07.    <TextView
08.        android:id="@+id/textView1"
09.        android:layout_width="wrap_content"
10.        android:layout_height="wrap_content"
11.        android:text="Nome:"/>
12.
13.    <EditText
14.        android:id="@+id edtNome"
15.        android:layout_width="match_parent"
16.        android:layout_height="wrap_content"
17.        android:ems="10" />
18.    </EditText>
19.
20.    <TextView
21.        android:id="@+id/textView2"
22.        android:layout_width="wrap_content"
23.        android:layout_height="wrap_content"
24.        android:text="Revista:"/>
25.
26.    <EditText
27.        android:id="@+id edtRevista"
28.        android:layout_width="match_parent"
29.        android:layout_height="wrap_content"
30.        android:ems="10" />
31.
32.    <TextView
33.        android:id="@+id/textView3"
34.        android:layout_width="wrap_content"
35.        android:layout_height="wrap_content"
36.        android:text="Edição:"/>
37.
38.    <EditText
39.        android:id="@+id edtEdicao"
40.        android:layout_width="match_parent"
41.        android:layout_height="wrap_content"
42.        android:ems="10"
43.        android:inputType="number" />
44.
45.    <TextView
46.        android:id="@+id/textView4"
47.        android:layout_width="wrap_content"
48.        android:layout_height="wrap_content"
49.        android:text="Status:"/>
50.
51.    <Spinner
52.        android:id="@+id/spEstado"
53.        android:layout_width="match_parent"
54.        android:layout_height="wrap_content"
55.        android:entries="@array/status" />
56.
57.    <CheckBox
58.        android:id="@+id/cbPago"
59.        android:layout_width="wrap_content"
60.        android:layout_height="wrap_content"
61.        android:text="Pago"/>
62.
63.    <Button
64.        android:id="@+id/button1"
65.        android:layout_width="wrap_content"
66.        android:layout_height="wrap_content"
67.        android:text="Salvar"
68.        android:onClick="salvar"
69.        android:layout_gravity="center_horizontal" />
70.</LinearLayout>
```

layout onde um componente é posicionado abaixo do outro, utilizamos o *LinearLayout*. A orientação padrão do mesmo é horizontal, por isso que na linha 5 alteramos esse comportamento com o valor *vertical* para a propriedade *android:orientation*. Além disso, configuramos a largura (linha 3) e altura (linha 4) da mesma como *match_parent*. Deste modo, ele ocupará todo o espaço disponibilizado pelo seu pai, que neste caso é o próprio *display* do smartphone.

Na linha 6 temos os primeiros rótulo, representado aqui pela *UI Widget TextView*. Além de seu *id*, definimos sua largura e altura (linhas 9 e 10). O valor passado é o *wrap_content*, que define que o mesmo ocupará o espaço necessário para mostrar seu conteúdo e mais nada. Na linha 11 definimos o conteúdo textual da view.

Na linha 13 criamos o primeiro *EditText*. Este representa um componente de entrada de texto. Na linha 15 definimos *match_parent* como largura, ou seja, irá ocupar todo o espaço horizontal disponibilizado pelo seu pai, que neste caso é o *LinearLayout*. Como altura temos *wrap_content*.

O *TextView* criado na linha 20 e o *EditText* da linha 25 tem as mesmas propriedades das *UI Widgets* descritas nos dois parágrafos acima. Exceção é feita aos seus *id* (*android:id*) e ao conteúdo textual do rótulo.

Na linha 31 temos mais um *TextView* que também tem as mesmas propriedades dos apresentados anteriormente, alterando apenas seu texto. Já no *EditText* criado na linha 37 temos uma propriedade a mais. Na linha 42 definimos que este campo de entrada aceita somente números, definindo o valor *number* para a propriedade *android:inputType*.

O componente *Spinner* (linha 50) é apresentado na sequência. Agora vamos pular para a linha 56 da **Listagem 2**. Nela estamos criando um *CheckBox*, que recebe *wrap_content* para largura e altura (linhas 58 e 59). Na linha 60 definimos seu conteúdo textual. Além da definição de seu *id* na linha 57.

Na linha 62 estamos criando um botão. Na linha 63 definimos seu *id*, nas linhas 64 e 65 sua largura e altura, ambas com *wrap_content*. Na linha 66 o atributo *android:text* define seu conteúdo textual. Na linha 67 configuramos que o método *salvar* será chamado quando tivermos uma ação de clique nesta view. E na linha 68 estamos orientando o botão no centro horizontal em relação a seu pai, neste caso o *LinearLayout*.

Agora vamos voltar para a linha 44, onde criamos o rótulo do campo de estado do artigo. O mesmo tem atributos iguais aos outros *TextViews* utilizados, exceto pelo seu texto. Logo em seguida, na linha 50, criamos um *Spinner*, além das configurações de *id*, largura e altura já utilizadas aqui, também definimos a lista de itens com a propriedade *android:entries*. Perceba que é passado um valor começando com @, ou seja, está nos nossos recursos. E depois temos *array*, porque foi definido com *string-array* dentro de *res/values/strings.xml*. Veja o conteúdo deste arquivo na **Listagem 3**.

Na **Listagem 4** vamos estudar o layout definido para a tela inicial da aplicação, que como o leitor irá perceber, é bem mais simples que o anterior.

Listagem 3. strings.xml – Conteúdo apresentado no componente Spinner.

```
01. <?xml version="1.0" encoding="utf-8"?>
02. <resources>
03.   <string name="app_name">CadastroArtigos</string>
04.   <string name="hello_world">Hello world!</string>
05.   <string name="menu_add">Adicionar</string>
06.   <string-array name="status">
07.     <item>Produção</item>
08.     <item>Revisão</item>
09.     <item>Aguardando Publicação</item>
10.     <item>Publicado</item>
11.   </string-array>
12. </resources>
```

Listagem 4. activity_artigos.xml – Tela principal do aplicativo.

```
01. <ListView xmlns:android="http://schemas.android.com/apk/res/android"
02.   xmlns:tools="http://schemas.android.com/tools"
03.   android:layout_width="match_parent"
04.   android:layout_height="match_parent"
05.   android:id="@+id/list">
06. </ListView>
```

Como a tela será uma lista, temos apenas uma *UI Widget*. Na plataforma Android podemos mostrar uma listagem com um tipo especial de tela. Quando falamos em tela entende-se *Activity*. Esta extensão é a *ListAdapter*. Para que ela funcione corretamente devemos ter no seu xml uma *ListView* com um *android:id* contendo o valor “@*android:id/list*”, como definido na linha 5.

Activities

Agora que temos os layouts das telas prontas vamos criar a *Activity* referente a tela de inclusão/edição e a *ListActivity* referente a tela de principal.

Desta forma, a tela principal do aplicativo vai mostrar uma lista de artigos, com a opção de editar ou excluir cada um deles. Sendo assim, podemos utilizar uma especialização de *Activity*: a *ListActivity*. O primeiro passo para isso já fizemos, criando um layout em XML que faça uso da *ListView*, conforme **Listagem 4**. Agora é codificar a classe Java, conforme **Listagem 5**.

Na listagem criamos uma subclasse de *ListActivity*. Também sobrescrevemos o método *onCreate*, procedimento padrão em todos os Activities. Na linha 6 chamamos o método *setContentView*, passando como parâmetro o recurso *R.layout.activity_artigos*, este o arquivo XML que possui o layout da tela principal.

A **Listagem 6** é o código do *Activity* que mostra a tela de editar ou criar um novo registro. O código inicial também é bastante simples, somente sobrescrevendo o *onCreate* e definindo o conteúdo através de um recurso de layout.

O banco de dados

A arquitetura inicial foi construída, para seguirmos adiante vamos precisar listar os artigos na tela inicial, dar a opção de editá-los e excluí-los. A tela de adicionar também terá que acionar o botão e permitir o salvamento de novos dados. Perceba que tudo isso envolve interação com o banco de dados SQLite. Sendo assim, vamos tratar disso agora.

Listagem 5. ArtigosActivity.java – Classe que tratará a tela principal do aplicativo.

```
01. public class ArtigosActivity extends ListActivity {
02.
03.   @Override
04.   protected void onCreate(Bundle savedInstanceState) {
05.     super.onCreate(savedInstanceState);
06.     setContentView(R.layout.activity_artigos);
07.   }
08. }
```

Listagem 6. NovoEdicaoActivity.java - Classe que tratará a tela de incluir e alterar.

```
01. public class NovoEdicaoActivity extends Activity {
02.
03.   @Override
04.   protected void onCreate(Bundle savedInstanceState) {
05.     super.onCreate(savedInstanceState);
06.     setContentView(R.layout.novo_edicao);
07.   }
08. }
```

No projeto, vamos criar uma classe que terá todas as ações que envolvem diretamente o banco de dados SQLite. Desta forma, o projeto fica mais organizado e separamos a parte de modelo de dados das visões das aplicações (telas) e da sua lógica, conforme o *design pattern Model-View-Controller* instrui. Essa classe será chamada de *BancoDeDados* e é mostrada na **Listagem 7**.

Das linhas 2 a linha 7 definimos objetos *Strings* que contém o nome de todos os campos da tabela que será tratado pelo aplicativo. Na linha 9 temos uma variável com o nome do banco de dados, na linha 10 outra variável com nome da única tabela utilizada pelo aplicativo. Na linha 11 um *int* que define a versão atual do banco relacional.

Na linha 12 temos uma variável *String* com a cláusula SQL responsável pela criação da tabela no banco de dados. Perceba que utilizamos as variáveis criadas anteriormente para os nomes dos campos. No exemplo, utilizamos uma chave primária do tipo inteiro e com autoincrement, dois campos inteiros simples e três campos do tipo text.

Na linha 14 criamos uma variável *Context*, que será necessária na classe *MeuOpenHelper*, esta codificada posteriormente. Também foi declarada na linha 15 uma variável para o *MeuOpenHelper*. Na linha 16 temos uma variável de *SQLiteDatabase*. É esta classe que nos permite chamar métodos de CRUD diretamente no banco de dados. Para este exemplo, esta é a classe mais importante.

Na linha 18 codificamos o construtor da classe *BancoDeDados*, que recebe um *Contexto* e passa para sua variável. Também cria uma instância de *MeuOpenHelper*.

A classe *MeuOpenHelper*, codificada na linha 18, estende de *SQLiteOpenHelper*. Esta classe ajuda na criação e gerenciamento de versões do banco de dados. Quando fizemos esta herança, podemos implementar três métodos: *onCreate*, *onUpdate* e *onOpen*. Para este exemplo utilizamos os dois primeiros. Além disso, perceba que o construtor de *MeuOpenHelper* – linha 24 - chama o construtor da super classe. Para este passamos o contexto, o nome do banco de dados, uma instância de *CursorFactory* e uma versão do banco.

Desenvolva um programa de cadastro usando SQLite e Adapters

Listagem 7. Classe BancoDeDados.java – Código referente a utilização do SQLite.

```
01. public class BancoDeDados {  
02.     static String KEY_ID = "_id";  
03.     static String KEY_NOME = "nome";  
04.     static String KEY_REVISTA = "revista";  
05.     static String KEY_EDICAO = "edicao";  
06.     static String KEY_STATUS = "status";  
07.     static String KEY_PAGO = "pago";  
08.  
09.     String NOME_BANCO = "db_Revistas";  
10.    String NOME_TABELA = "artigo";  
11.    int VERSAO_BANCO = 1;  
12.    String SQL_CREATE_TABLE = "create table contacts " + "(" + KEY_ID + " integer  
primary key autoincrement," + KEY_NOME + " text not null," + KEY_REVISTA +  
" text," + KEY_EDICAO + " text," + KEY_STATUS + " integer," + KEY_PAGO + "  
integer);";  
13.  
14.    final Context context;  
15.    MeuOpenHelper openHelper;  
16.    SQLiteDatabase db;  
17.  
18.    public BancoDeDados(Context ctx) {  
19.        this.context = ctx;  
20.        openHelper = new MeuOpenHelper(context);  
21.    }  
22.  
23.    private class MeuOpenHelper extends SQLiteOpenHelper {  
24.        MeuOpenHelper(Context context) {  
25.            super(context, NOME_BANCO, null, VERSAO_BANCO);  
26.        }  
27.  
28.        @Override  
29.        public void onCreate(SQLiteDatabase db) {  
30.            try {  
31.                db.execSQL(SQL_CREATE_TABLE);  
32.            } catch (SQLException e) {  
33.                e.printStackTrace();  
34.            }  
35.        }  
36.  
37.        @Override  
38.        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  
39.            db.execSQL("DROP TABLE IF EXISTS contacts");  
40.            onCreate(db);  
41.        }  
42.    }  
43.  
44.    public BancoDeDados abrir() throws SQLException {  
45.        db = openHelper.getWritableDatabase();  
46.        return this;  
47.    }  
48.  
49.    public void fechar() {  
50.        openHelper.close();  
51.    }  
52.  
53.    public long insereArtigo(String nome, String revista, String edicao, int status,  
int pago) {  
54.        ContentValues campos = new ContentValues();  
55.        campos.put(KEY_NOME, nome);  
56.        campos.put(KEY_REVISTA, revista);  
57.        campos.put(KEY_EDICAO, edicao);  
58.        campos.put(KEY_STATUS, status);  
59.        campos.put(KEY_PAGO, pago);  
60.        return db.insert(NOME_TABELA, null, campos);  
61.    }  
62.  
63.    public boolean apagaArtigo(long id) {  
64.        return db.delete(NOME_TABELA, KEY_ID + "=" + id, null) > 0;  
65.    }  
66.  
67.    public Cursor retornaTodosArtigos() {  
68.        return db.query(NOME_TABELA, new String[] { KEY_ID, KEY_NOME,  
KEY_REVISTA, KEY_EDICAO, KEY_STATUS, KEY_PAGO }, null, null, null, null, null);  
69.    }  
70.  
71.    public boolean atualizaArtigo(long id, String nome, String revista, String edicao,  
int status, int pago) {  
72.        ContentValues args = new ContentValues();  
73.        args.put(KEY_NOME, nome);  
74.        args.put(KEY_REVISTA, revista);  
75.        args.put(KEY_EDICAO, edicao);  
76.        args.put(KEY_STATUS, status);  
77.        args.put(KEY_PAGO, pago);  
78.        return db.update(NOME_TABELA, args, KEY_ID + "=" + id, null) > 0;  
79.    }  
80.}
```

O método *onCreate*, sobrescrito na linha 29, será chamado quando o banco de dados é criado pela primeira vez. Logicamente, é nele que teremos que criar nossas tabelas, com o código apresentado na linha 31. O método recebe uma instância de *SQLiteDatabase*, que tem um método *execSQL*. Logo, passamos a variável que criamos anteriormente com o sql para a criação da tabela.

O *onUpgrade*, sobrescrito na linha 38, é chamado quando temos uma nova versão do banco de dados ou da aplicação. Se mudarmos a variável com a versão do banco de dados este método será chamado. Nele também chamamos o *execSQL*, passando um SQL que exclui a tabela e chama o *onCreate*, que criará novamente esta, ou seja, limpa o banco de dados.

Na linha 44 temos o método *abrir*. Nele, chamamos o método *getWritableDatabase* da *MeuOpenHelper*. Lembre-se que a mesma é filha de *SQLiteOpenHelper*. Também poderíamos ter chamado o método *getReadableDatabase*, que retorna a mesma instância, porém em modo *read-only*.

Assim como temos o método *abrir*, temos o método *fechar* na linha 49. Este só chama o método *close()* da instância de *MeuOpenHelper*. A sequência utilizada na maioria dos programas se resume a cada operação: abrir uma conexão com o banco; pegar a instância de *SQLiteDatabase*; executar o comando; fechá-la. Entretanto, se o programador executar um grande conjunto de operações no banco sequencialmente, pode-se abrir uma conexão no início do aplicativo e fechar somente quando o usuário sair do aplicativo.

Na linha 53 temos o método *insereArtigo*, que recebe por parâmetro o valor de todos os campos da tabela no banco de dados. Na linha 54 encontra-se uma instância de *ContentValues*, chamada de *campos*. Esta classe armazena pares de dados, com uma chave e seu valor. Veja que o nome coincide com o nome do campo na tabela, por isso utilizamos as variáveis *KEY_**. Os métodos *put* sempre recebem um texto para a chave e um tipo primitivo para o valor. Estamos fazendo isso das linhas 55 até a linha 59.

Na linha 60 chamamos o método *insert* da instância de *SQLiteDatabase*. O primeiro parâmetro é o nome da tabela, o segundo é um *ColumnHack*, que passa automaticamente um valor *null* para campos que não serão vazios, porém, como estamos confiando no nosso código, passamos *null* neste parâmetro. E por fim, uma instância de *ColumnValues*.

Na linha 63 temos o método *apagaArtigo*, que recebe o *id* da linha a ser removida. O que estamos fazendo é chamando o método *delete* de *SQLiteDatabase*. Seus parâmetros são o nome da tabela, a cláusula *where* e os argumentos desta mesma classe. Em vez de passar a cláusula *where* “*KEY_ID = id*”, como está no código, poderíamos passar “*KEY_ID = ?*”. Sendo assim, passaríamos o *id* como argumento do *where*, no último parâmetro.

Na linha 67 temos o método *retornaTodosArtigos*, este responsável pela busca no banco de dados, sem filtros. Para isso, estamos chamando o método *query* de *SQLiteDatabase*. Os parâmetros desta versão do método utilizada são:

- Nome da tabela;
- Campos a serem retornados;
- Cláusula *where*;
- Parâmetros da cláusula *where*;
- *Group By* do SQL;
- *Having* do SQL;
- *Order By* do SQL. Se quisermos buscar os artigos por ordem alfabética, basta mudar este parâmetro;
- *Limit* do SQL. Poderíamos querer listar somente os 10 primeiros artigos.

O método retorna uma instância da classe *Cursor*. Esta serve como um ponteiro para os resultados retornados da consulta. Podemos navegar pelas linhas com métodos como *moveToFirst*, *moveToLast* e *moveToNext*. Além disso, quando posicionamos o *Cursor* em uma determinada linha retornada, podemos usar métodos *get* para todos tipos primitivos, como *getInt*, *getFloat* e *getString*, recuperando assim o conteúdo do registro apontado.

Por fim, na linha 71, temos o método *atualizaArtigo*. Esse método se parece bastante com o método de inserção. Ele tem apenas um parâmetro a mais, que é o *id* da linha que será alterada. Assim, também criamos uma instância de *ContentValues* e, por fim, na linha 78, chamamos o método *update* de *SQLiteDatabase*, passando como parâmetro o nome da tabela, a instância *args*, a cláusula *where* e os parâmetro da cláusula (caso existirem).

Como este método retorna o número de linhas afetadas, já estamos usando um operador relacional para saber se ouve registros alterados. Caso afirmativo, retornamos *true*, caso contrário, *false*.

Criando o adapter para o *ListActivity*

Após codificar a classe utilitária que trabalha com o banco de dados, o próximo passo é codificar um Adapter, este será responsável por recuperar os dados do banco e apresentá-los na tela de lista. Nossa tela inicial é uma lista, que é representada por um *ListActivity* em Java. A *ListActivity*, por sua vez, precisa de um

adapter, que serve como um adaptador entre seu conteúdo, ou seja, os itens que a lista irá representar e a tela propriamente dita.

Para definir um *adapter* para uma lista é utilizado o método *setListAdapter*. Podemos passar alguns *adapters* prontos, como o *ArrayAdapter*, por exemplo. Ele mostrará uma lista baseada em um vetor (array). Porém, perceba na **Figura 1** que não queremos isso. Desejamos que seja mostrado o nome do artigo e mais dois botões, um para editar e outro para excluir esta linha. Sendo assim, precisamos criar um *adapter* personalizado, estendendo este de *BaseAdapter*.

E mais, neste *adapter* customizado podemos utilizar a *View* que desejarmos. Se quisermos que cada linha da lista tenha mais de 30 *UI Widgets*, é totalmente possível. Então vamos primeiro ao XML que cria a interface de cada linha da nossa lista, conforme **Listagem 8**.

Listagem 8. Arquivo item_artigo.xml – Conteúdo das linhas da nossa lista principal.

```
01. <?xml version="1.0" encoding="utf-8"?>
02. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03.     android:layout_width="match_parent"
04.     android:layout_height="match_parent"
05.     android:orientation="horizontal">
06.     <TextView
07.         android:layout_gravity="center_vertical"
08.         android:id="@+id/txtNome"
09.         android:layout_width="wrap_content"
10.         android:layout_height="wrap_content"
11.         android:text="Medium Text"
12.         android:layout_weight="1"
13.         android:textAppearance="?android:attr/textAppearanceMedium"/>
14.
15.     <ImageButton
16.         android:id="@+id/btnEditar"
17.         android:layout_width="wrap_content"
18.         android:layout_height="wrap_content"
19.         android:src="@android:drawable/ic_menu_edit" />
20.
21.     <ImageButton
22.         android:id="@+id/btnExcluir"
23.         android:layout_width="wrap_content"
24.         android:layout_height="wrap_content"
25.         android:src="@android:drawable/ic_menu_delete" />
26. </LinearLayout>
```

Na primeira linha temos a marcação padrão do XML. Como *View root* temos um *LinearLayout*. Utilizamos este porque queremos alinhar os elementos um ao lado do outro. Veja na linha 5 que definimos a orientação do mesmo como horizontal. Na largura e altura estamos passando *match_parent*, com isso, ele ocupará a todo o espaço disponível pelo seu pai, que é a largura e altura estabelecida para a linha na lista.

Na linha 6 temos o *TextView* que armazenará o nome do artigo. Na linha 7 definimos sua orientação em relação a seu pai (o *LinearLayout*). Passamos o valor *center_vertical*. Com isso, ele ficará alinhado ao centro, mesmo que os botões tenham uma altura maior. Na linha 8 temos o *id*. Na linha 9 e 10 definimos a largura e altura como *wrap_content*. Na linha 11 configuramos o texto inicial, que será alterado posteriormente.

Desenvolva um programa de cadastro usando SQLite e Adapters

Na linha 12 definimos o peso do *UI Widget* como 1. Isso faz com que o *TextView* ocupe todo o espaço restante horizontalmente. Ou seja, se o nome do artigo for pequeno, vai evitar que tanto o texto como os dois botões fiquem presos no lado esquerdo da tela, mas sim, que os botões sempre fiquem na extremidade direita. Por fim, na linha 13, definimos a aparência deste texto como média.

Na linha 15 criamos o primeiro botão, na verdade um *ImageButton*. Além do já tradicional *id*, largura e altura, também estamos definindo a imagem que o componente terá, através da propriedade *android:src*. Perceba que não estamos utilizando imagens do nosso projeto, mas sim a *ic_menu_edit*, disponibilizado pelo próprio sistema operacional Android.

Na linha 21 temos mais um *ImageButton*. As propriedades são basicamente as mesmas, até mesmo seus valores. Exceção é feita a *android:src*, onde passamos a utilizar a imagem *ic_menu_delete*, também disponibilizada pelo sistema operacional Android.

Com o layout criado no XML, podemos passar para a classe que estende *BaseAdapter*. Veja a **Listagem 9**.

Antes de começar a detalhar este código, vamos reforçar o ponto-chave: esta classe representa o adaptador com os itens da lista mostrados na *ListActivity*. Dito isso vamos para a linha 3, onde temos um *List* de *Artigo*. Desta forma, todos os elementos que serão apresentados na tela são elementos deste *List*.

Na linha 4 temos uma variável de *LayoutInflater*. Esta classe serve literalmente para inflar um layout especificado em um arquivo XML e transformar em uma instância da classe *View*. Veremos sua enorme utilidade ao final do código.

No construtor da classe (linha 7) recebemos um *Contexto* e a lista de artigos. O contexto é utilizado para criar a instância de *LayoutInflater*, pois é nele que existe o método *getSystemService*, utilizado na linha 7.

Na linha 11 encontramos o método *novosDados*, que recebe como parâmetro uma lista de artigos. Quando inserirmos, deletarmos ou editarmos um artigo, a lista da tela inicial deve ser atualizada. Para isso, devemos informar ao *adapter* toda alteração feita no banco de dados. Neste método mudamos a lista de artigos e chamamos o método *notifyDataSetChanged*. Com isso, o *adapter* sabe que precisa atualizar e fará isso.

Na linha 17 temos o método *getCount*, que deve ser sobreescrito de forma obrigatória pela classe que estende *BaseAdapter*. A função deste método é informar quantos elementos a lista vai ter, por isso, retornamos o número de elementos presentes na instância de *List* através de seu método *size*.

Na linha 22 outro método que devemos sobreescrver é o *getItem*. Este, por sua vez, retorna o dado associado com uma dada posição no conjunto de dados. Estamos retornando o objeto *Artigo* daquela posição.

Na linha 27 temos o método mais importante da classe que estende *BaseAdapter* – o método *getView*. Como a *ListActivity* vai mostrar uma lista, consequentemente temos que informar a ela que *View* ela deverá apresentar em cada índice, ou seja, em cada posição da lista. E é isso que esse método faz. Perceba que seu retorno é uma instância de *View* e seu primeiro parâmetro é um *int* que identifica a posição. Se a lista tiver 5 elementos, esse método será chamado 5 vezes.

Listagem 9. Classe ArtigosAdapter.java – Definindo o Adapter que será utilizado no menu principal.

```
01. public abstract class ArtigosAdapter extends BaseAdapter {  
02.  
03.     private List<Artigo> artigos;  
04.     private LayoutInflater inflater;  
05.  
06.     public ArtigosAdapter(Context context, List<Artigo> artigos){  
07.         this.inflater = (LayoutInflater)  
            context.getSystemService(context.LAYOUT_INFLATER_SERVICE);  
08.         this.artigos = artigos;  
09.     }  
10.  
11.    public void novosDados(List<Artigo> artigos){  
12.        this.artigos = artigos;  
13.        notifyDataSetChanged();  
14.    }  
15.  
16.    @Override  
17.    public int getCount(){  
18.        return artigos.size();  
19.    }  
20.  
21.    @Override  
22.    public Object getItem(int position){  
23.        return artigos.get(position);  
24.    }  
25.  
26.    @Override  
27.    public View getView(final int position, View convertView, ViewGroup parent) {  
28.        View v = inflater.inflate(R.layout.item_artigo, null);  
29.        ((TextView)(v.findViewById(R.id.txtNome))).setText(artigos.get(position).nome);  
30.  
31.        ((ImageButton)(v.findViewById(R.id.btnEditar))).setOnClickListener  
            (new View.OnClickListener() {  
32.                @Override  
33.                public void onClick(View v) {  
34.                    edita(artigos.get(position));  
35.                }  
36.            });  
37.  
38.        ((ImageButton)(v.findViewById(R.id.btnExcluir))).setOnClickListener  
            (new View.OnClickListener() {  
39.                @Override  
40.                public void onClick(View v) {  
41.                    deleta(artigos.get(position));  
42.                }  
43.            });  
44.  
45.        return v;  
46.    }  
47.  
48.    @Override  
49.    public long getItemId(int position) {  
50.        return 0;  
51.    }  
52.  
53.    public abstract void edita(Artigo artigo);  
54.    public abstract void deleta(Artigo artigo);  
55. }
```

Listagem 10. ArtigosActivity.java – Integração entre o Adapter e o banco de dados.

```
...
01. private BancoDeDados db;
02. private List<Artigo> artigos = new ArrayList<Artigo>();
03. private ArtigosAdapter artigosAdapter;
04. public static final int REQUEST_EDICAO = 0;

05. protected void onCreate(Bundle savedInstanceState) {
06.     super.onCreate(savedInstanceState);
07.     setContentView(R.layout.activity_artigos);
08.
09.     db = new BancoDeDados(this);
10.     lerDados();
11. }
12.
13. public void lerDados(){
14.     db.abrir();
15.     artigos.clear();
16.     Cursor cursor = db.retornaTodosArtigos();
17.     if(cursor.moveToFirst()){
18.         do {
19.             Artigo a = new Artigo();
20.             a.id = cursor.getInt(cursor.getColumnIndex(BancoDeDados.KEY_ID));
21.             a.nome = cursor.getString(cursor.getColumnIndex(BancoDeDados.KEY_NOME));
22.             a.revista = cursor.getString(cursor.getColumnIndex(BancoDeDados.
KEY_REVISTA));
23.             a.edicao = cursor.getString(cursor.getColumnIndex(BancoDeDados.KEY_EDICAO));
24.             a.status = cursor.getInt(cursor.getColumnIndex(BancoDeDados.KEY_STATUS));
25.             a.pago = cursor.getInt(cursor.getColumnIndex(BancoDeDados.KEY_PAGO));
26.             artigos.add(a);
27.         } while (cursor.moveToNext());
28.     }
29.     if (artigos.size() > 0){
30.         if (artigosAdapter == null){
31.             artigosAdapter = new ArtigosAdapter(this, artigos){
32.                 @Override
33.                 public void edita(Artigo artigo) {
34.                     Intent intent = new Intent(getApplicationContext(), NovoEdicaoActivity.class);
35.                     intent.putExtra("artigo", artigo);
36.                     startActivityForResult(intent, REQUEST_EDICAO);
37.                 }
38.             };
39.             @Override
40.             public void deleta(Artigo artigo) {
41.                 db.abrir();
42.                 db.apagaArtigo(artigo.id);
43.                 db.fechar();
44.                 lerDados();
45.             }
46.         };
47.     };
48.     setListAdapter(artigosAdapter);
49. } else {
50.     artigosAdapter.novosDados(artigos);
51. }
52. }
53. db.fechar();
54. }
55.
...
...
```

A primeira linha do método (linha 20) recupera a interface gráfica de cada elemento da lista. Assim, estamos utilizando a instância de *LayoutInflater* para inflar a *View* que criamos no xml *item_artigo.xml*, mostrado na [Listagem 8](#). O primeiro parâmetro é o próprio recurso e segundo é uma instância de *ViewGroup*, que podemos deixar como *null*, já que a *View* não pertence a nenhum grupo.

Esta *view* tem um texto com o nome do artigo e dois botões. Logo, teremos que recuperar o *TextView* e editar seu conteúdo. É isso que fazemos na linha 29. Com o parâmetro *position*, é possível recuperar o elemento da lista de *Artigos* que será apresentado na tela. Depois disso é só pegar o valor da variável *nome*, informação esta que corresponde ao nome do artigo.

Também precisamos recuperar os botões e adicionar o *listener* que escuta o evento de clique. É isso que fazemos nas linhas 31 e 38. A diferença está justamente no método que cada botão irá chamar em seu método *onClick*. No caso do *editar* chamamos o método *edita*, passando o objeto *Artigo* referenciado pela posição clicada na lista. O mesmo acontece para o método *deleta*, chamado quando o *btnExcluir* for clicado.

Finalmente, na linha 45, retornamos a instância de *view*, chamada de *v*, que foi inflada no início deste método e que corresponderá ao elemento de um item na tela principal.

Na linha 49 temos outro método que devemos sobrescrever obrigatoriamente, o *getItemId*. Com ele podemos retornar um *id* específico para cada elemento da lista. Para casos mais avançados é muito útil, porém, no nosso projeto podemos retornar sempre o mesmo *id*.

Finalmente, nas linhas 53 e 54 temos os métodos *edita* e *deleta*. Repare que ambos estão marcados como *abstract*, o que força a classe a ser abstrata também. Fizemos isso porque ao editar, por exemplo, outra tela terá que ser chamada. E isso só poderá ser feito da *ListActivity*, sendo assim, ela vai utilizar *ArtigosAdapter* e terá que fornecer a implementação destes dois métodos.

Integrando o Adapter e o Banco de Dados

Para realizar a integração entre o Adapter e o banco de dados, devemos alterar o código da classe *ArtigosActivity*. Veja na [Listagem 10](#) as primeiras mudanças.

Primeiramente criamos uma variável para conter a instância de *BancoDeDados*, que será chamada simplesmente de *db*. Em seguida temos a variável com a *List* de instâncias de *Artigo*. Na linha 3 temos a variável com a instância do *adapter*, que é a instância de *ArtigosAdapter*, e por fim, uma constante chamada *REQUEST_EDICAO*, sendo esta utilizada para chamar novas janelas (*startActivityForResult*).

Dentro do *onCreate*, na linha 9, criamos a instância de *db*, e na linha 10 chamamos o método *lerDados*.

O método *lerDados* é codificado a partir da linha 13. Na linha 14 chamamos o método *abrir* de *BaseDeDados*, este possui a função de iniciar uma conexão com o banco SQLite. Na linha seguinte, limpamos o *List* que apresentará os dados na tela. Na linha 16 chamamos o método *retornaTodosArtigos*, o qual recupera todos os registros armazenados no SQLite, armazenando seu retorno em uma instância de *Cursor*.

Desenvolva um programa de cadastro usando SQLite e Adapters

Na linha 17 tentamos mover o *cursor* para o primeiro elemento, se isso for possível, significa que a tabela não está vazia. Sendo assim, entramos no laço *do-while*, este iniciado na linha 18. A sequência de instruções vai da linha 19 até a 25, sendo criado um objeto *Artigo*, o qual é populado com os valores presentes na tabela. Perceba que em todos os casos estamos chamando ou o método *getInt* ou *getString* de *cursor*. Ambos recebem como parâmetro o índice da coluna desejada. Poderíamos passar um valor fixo, como 0, 1, 2, 3, 4 e 5. Porém não é o mais indicado. O que estamos fazendo é utilizar o método *getColumnIndex* do *cursor*, passando o nome da coluna, que retorna o índice da mesma. Na linha 26 adicionamos a instância de *Artigo* no *List*, e finalmente, na linha 27, tentamos mover o *cursor* para o próximo registro.

Ao finalizar o looping, o código da linha 30 é executado, onde verificamos se a busca trouxe algum registro e, consequentemente, mudou o tamanho da lista para algo maior que zero. Caso positivo, a linha 31 verifica se o *artigosAdapter* ainda está nulo. Se sim, significa que o método *lerDados* nunca foi chamado antes e o *adapter* deve ser instanciado.

Ao fazer isso, na linha 32, precisamos fornecer a implementação dos métodos *edita* (linha 34) e *deleta* (linha 41), que estão marcados como abstratos na *ArtigosAdapter*.

No método *edita* estamos criando uma *Intent* para mudar de tela, tendo como destino a *NovoEdicaoActivity.java*. Além disso, definimos um extra para esta *intent*, definido na linha 36. Como a classe *Artigo* foi definida como *Serializable*, podemos passar sua instância direta para outra tela, e na linha 37 chamamos *startActivityForResult* para acionar a intenção passando um código de requisição. Com isso sabermos se o retorno foi o clique no botão *Salvar* ou se o usuário não completou a ação de editar e não é preciso mudanças na lista.

Já o método *deleta*, na linha 41, abre a conexão com o banco através do método *abrir* da *BaseDeDados*. Logo em seguida, na linha 43, chama o método *apagaArtigo*, passando como parâmetro o *id* do artigo que deve ser excluído. Na linha 44 fechamos a conexão com o banco e chamamos novamente o método *lerDados* para atualizar o *List* e o *adapter*.

Após isso, chamamos o método *setListAdapter*, na linha 48, passando a instância da classe que herda de *BaseAdapter*.

Se o adaptador já havia sido criado, precisamos atualizar a sua lista de artigos para que a lista também sofra esta ação. Assim, na linha 50, dentro do *else*, chamamos o método *novosDados* passando por parâmetro o *List*.

Por fim, na linha 53 fechamos a conexão com o banco de dados através do método *fechar*.

A classe *ArtigosActivity.java* ainda não está finalizada, veremos agora mais algumas alterações necessárias na mesma, conforme apresentado na **Listagem 11**.

No início da **Listagem 11** temos o método *onCreateOptionsMenu*. Este é chamado quando o usuário aciona o botão menu, presente nas versões anteriores ao Android Honeycomb ou, quando a *Activity* é criada e o *ActionBar* também é iniciado. Neste último caso, nas versões 3.0 e acima, quando o desenvolvedor não remove este comportamento via codificação.

Listagem 11. ArtigosActivity.java – Criando um menu para o Activity principal.

```
01. @Override
02. public boolean onCreateOptionsMenu(Menu menu) {
03.     getMenuInflater().inflate(R.menu.activity_artigos, menu);
04.     return true;
05. }
06.
07. @Override
08. public boolean onOptionsItemSelected(MenuItem item) {
09.     if (item.getItemId() == R.id.menu_add){
10.         Intent intent = new Intent(this, NovoEdicaoActivity.class);
11.
12.         startActivityForResult(intent, REQUEST_EDICAO);
13.         return true;
14.     } else {
15.         return super.onOptionsItemSelected(item);
16.     }
17. }
```

Neste método recebemos uma instância de *Menu* e podemos adicionar *MenuItem*'s neste. Ou, podemos chamar o método *getMenuInflate* e, com seu retorno, chamar *inflate* passando como parâmetro o recurso de menu e o parâmetro *Menu* recebido. Finalmente, retornamos *true* para o método, na linha 4.

Falamos que estamos apontando para um recurso do aplicativo, que fica na pasta *res*, sub-pasta *menu*. Na **Listagem 12** mostramos o *activity_artigos.xml*. Seu único elemento tem um *id menu_add*, um valor para ordenação (linha 4), um texto (linha 6) e uma propriedade *showAsAction*. Ao passarmos o valor *ifRoom* o Android coloca este item de menu na *ActionBar* caso tenha espaço suficiente. Para verificar o local onde se encontra o *ActionBar* é só procurar pela literal Adicionar na **Figura 1** e veja que este é exatamente o comportamento do aplicativo quando utilizado.

Listagem 12. activity_artigos.xml – Menu presente na tela principal do aplicativo.

```
01. <menu xmlns:android="http://schemas.android.com/apk/res/android">
02.     <item
03.         android:id="@+id/menu_add"
04.         android:orderInCategory="100"
05.         android:showAsAction="ifRoom"
06.         android:title="@string/menu_add"/>
07.     </menu>
```

Voltando à **Listagem 11**, mais precisamente na linha 8, temos a implementação do método *onMenuItemSelected*. Este é acionado quando o usuário clicou em um item de menu. Na linha 9 estamos recuperando o *id* do *MenuItem* acionado e comparando com o nosso *menu_add*. Se for o mesmo, criamos uma *Intent* com destino para a tela *NovoEdicaoActivity* e lançamos a mesma com o método *startActivityForResult*, passando também um código de requisição igual a quando o usuário quer editar um artigo e chama a mesma tela.

Para finalizar a codificação do *ArtigosActivity.java*, devemos codificar o retorno dos *Activities* chamados por este. O código referente ao *onActivityResult* é apresentado na **Listagem 13**.

Listagem 13. Método `onActivityResult`.

```
01. public static final int REQUEST_SALVOU = 1;
02.
03. protected void onActivityResult(int requestCode, int resultCode, Intent data) {
04.     if (requestCode == REQUEST_EDICAO) {
05.         if (resultCode == REQUEST_SALVOU) {
06.             lerDados();
07.         }
08.     }
09. }
```

Como estamos chamando a tela de inserção/edição com o `startActivityForResult`, a `ListAdapter` receberá o retorno da `Activity` acionada. Desta forma, verificamos se este retorno é igual a requisição de ação utilizada para chamar as telas e, se além desse, foi retornado um código de resultado igual a `REQUEST_SALVOU`. Caso a resposta para os dois condicionais forem verdadeiras, significa que o usuário inseriu ou editou um artigo, sendo necessário atualizar a lista. Para isso, chama-se novamente o método `lerDados`.

Assim, a classe `ArtigosActivity.java` foi finalizada, agora vamos à codificação da classe `NovoEdicaoActivity.java`, conforme **Listagem 14**.

Listagem 14. `NovoEdicaoActivity.java` – Classe responsável para inclusão e alteração de registros.

```
...
01. private EditText edtNome;
02. private EditText edtRevista;
03. private EditText edtEdicao;
04. private Spinner spEstado;
05. private CheckBox cbPago;
06. private Artigo artigo;
07.
08. @Override
09. protected void onCreate(Bundle savedInstanceState) {
10.     super.onCreate(savedInstanceState);
11.     setContentView(R.layout.novo_edicao);
12.
13.     edtNome = (EditText) findViewById(R.id.edtNome);
14.     edtRevista = (EditText) findViewById(R.id.edtRevista);
15.     edtEdicao = (EditText) findViewById(R.id.edtEdicao);
16.     spEstado = (Spinner) findViewById(R.id.spEstado);
17.     cbPago = (CheckBox) findViewById(R.id.cbPago);
18.
19.     Intent intent = getIntent();
20.     artigo = (Artigo) intent.getSerializableExtra("artigo");
21.     if (artigo != null){
22.         edtNome.setText(artigo.nome);
23.         edtRevista.setText(artigo.revista);
24.         edtEdicao.setText(artigo.edicao);
25.         spEstado.setSelection(artigo.status);
26.         cbPago.setChecked(artigo.pago==1?true:false);
27.     }
28. }
```

Nas primeiras seis linhas criamos as variáveis que contêm os objetos `UI Widgets` que foram utilizados nesta `Activity`. Já na linha 6 criamos uma variável para a classe `Artigo`.

No método `onCreate`, além das duas primeiras linhas que são padrão, temos a instanciação das variáveis de elementos gráficos.

Para isso utilizamos o método `findViewById` de `Activity`, passando como parâmetro o `id` dos `UI Widgets`. Precisamos fazer um `cast` explícito porque o retorno deste método sempre será `View`.

Na linha 19 recuperamos a `Intent` que chamou esta tela. Precisamos fazer isso para, na linha 20, tentar recuperar o objeto `serializable` que foi passado como extra nesta intenção. Caso ele exista, a variável `artigo` não estará nula e, o teste da linha 21 terá sucesso. Neste caso, significa que o usuário está tentando editar um artigo, então, pré-configuramos os valores em todos os campos com os atributos da classe `Artigo`.

É bom ressaltar que na linha 25 estamos passando o índice que deve ser selecionado no `Spinner`. Isso porque é um inteiro que estamos salvando no banco de dados. A mesma lógica serve para a linha 26, onde verificamos se o atributo `pago` de `artigo` é 1 ou 0, porque foi salvo um valor inteiro na base. Com o operador ternário passamos esses valores para `true` ou `false`, valor exigido no método `setChecked`.

Ainda precisamos de um método na `NovoEdicaoActivity.java`. Quando criamos o xml de layout da mesma, inserimos um `Button` que tinha no atributo `android:onClick` o valor `salvar`. Sendo assim, esta `Activity` precisa fornecer a implementação deste método. Observe agora a **Listagem 15** e depois vamos à sua explicação.

Listagem 15. `NovoEdicaoActivity.java` – Código do botão salvar.

```
01. public void salvar(View v){
02.     BancoDeDados db = new BancoDeDados(this);
03.     db.abrir();
04.
05.     if (artigo != null){
06.         db.atualizaArtigo(artigo.id, edtNome.getText().toString(),
07.                         edtRevista.getText().toString(),
08.                         edtEdicao.getText().toString(),
09.                         spEstado.getSelectedItemPosition(), cbPago.isChecked()?1:0);
10.    } else {
11.        db.insereArtigo(edtNome.getText().toString(),
12.                        edtRevista.getText().toString(),
13.                        edtEdicao.getText().toString(),
14.                        spEstado.getSelectedItemPosition(), cbPago.isChecked()?1:0);
15.    }
16.    db.fechar();
17.    setResult(ArtigosActivity.REQUEST_SALVOU);
18.    finish();
19. }
```

Logo na linha 2 criamos uma instância de `BancoDeDados`, chamada de `db`. Na linha 3 chamamos o método `abrir` da referida classe. Na linha 5 verificamos se a variável `artigo` contém valor diferente de null. Caso afirmativo, significa que queremos editar um artigo, logo, devemos chamar o método `atualizaArtigo` passando como parâmetro o `id` do registro alvo e os valores de nome, revista, edição, estado e se foi pago ou não. Estes são recuperados dos elementos de UI.

Caso o teste lógico retorne falso, significa que o usuário está inserindo um novo registro. Então, o método a ser chamado da classe `BancoDeDados` é o `insereArtigo`, passando os mesmos valores de edição, exceto pelo `id` que não existe neste caso.

Desenvolva um programa de cadastro usando SQLite e Adapters

Na linha 11 fechamos o link com o banco de dados, configuramos o resultado da *Activity* na linha 12, para que a *ListActivity* saiba que precisa atualizar seu *adapter*, ou seja, o conteúdo da lista. A linha 13 tem a função de finalizar o *NovoEdicaoActivity.java* e retorna para o *Activity* anterior.

Com estas mudanças nosso projeto está pronto. Execute-o no emulador ou no dispositivo real para testar seu funcionamento.

Transações em SQLite

Uma questão muito importante na persistência de dados em bancos relacionais são as transações em modos exclusivos. Imagine o seguinte cenário, temos que inserir 10.000 registros em uma tabela, mas desejamos que o trabalho só seja realmente efetuado se todas as linhas forem inseridas com sucesso, caso contrário, queremos voltar ao estado anterior, ou seja, como estava a tabela antes de iniciar a inclusão.

Para facilitar este tipo de trabalho a API do Android para SQLite fornece três métodos muito importantes, o *beginTransaction*, o *setTransactionSuccessful* e o *endTransaction*. Vamos observar o esqueleto apresentado na **Listagem 16**.

Listagem 16. Utilizando transações em SQLite.

```
01. db.beginTransaction();
02. try {
03. ...
04. db.setTransactionSuccessful();
05. } finally {
06. db.endTransaction();
07. }
```

Na primeira linha iniciamos a transação em modo exclusivo. Dentro do *try*, nos três pontos, poderíamos ter centenas de linhas de código salvando registros, ou ainda, um laço que ficaria buscando informações em um web service e jogando-as diretamente no SQLite. A diferença é que queremos efetuar todas essas mudanças se não tivemos nenhum erro de uma só vez.

No final do *try* chamamos o método *setTransactionSuccessful*, indicando ao Android que todo o trabalho desejado foi feito com sucesso. Caso algum erro aconteça ao longo do código, a linha 04 não será executada e o fluxo do aplicativo vai direto para o *endTransaction*, no *finally*.

Neste momento, se a API percebe que as tarefas não foram marcadas com sucesso, com o *setTransactionSuccessful*, assim, a API fará um *roll back* em todas as alterações. Caso contrário, tudo é comitado no banco de dados relacional.

Perceba a facilidade que somente estes três métodos nos trazem. Para implementar esta funcionalidade no sistema proposto, basta antes de qualquer manipulação do banco de dados (incluir, alterar ou excluir) executar o comando *beginTransaction*. Após a execução do comando desejado, executa-se o *setTransactionSuccessful*, caso contrário, em um *catch*, utiliza-se o *endTransaction*.

Conclusão

Acredito que não precisamos falar sobre a importância da persistência de dados em qualquer aplicação no mundo moderno, seja ela para um simples aparelho celular, seja ela para um servidor de centenas de milhares de dólares. Isso também serve para banco de dados relacionais, que são utilizados com sucesso há alguns anos em outros ambientes. Com o advento dos smartphones e seus ambientes inteligentes de desenvolvimento, agora o desenvolvedor mobile também pode usufruir deste tipo de banco de dados através do SQLite.

Também acredito que o artigo mostrou o poder da API que a plataforma Android oferece para trabalhar com SQLite, fornecendo duas classes chaves (*SQLiteOpenHelper* e *SQLiteDatabase*) e métodos que auxiliam nas tarefas CRUD relacionadas a um conjunto de dados. Desta forma, o desenvolvedor pode se preocupar bem menos com a persistência de dados e focar todos seus esforços na lógica de negócios de seu software.

Sendo assim, o leitor deste artigo deverá estar apto a fornecer uma arquitetura correta de persistência de dados através de banco de dados relacionais no Android. Um quesito muito importante para qualquer profissional que deseje ser visto com bons olhos pelo exigente mercado mobile.

Autor



RobisonCrisBrito

robison@utfpr.edu.br

Mestre em Engenharia Elétrica e Informática Industrial pela Universidade Tecnológica Federal do Paraná, onde ministra aulas sobre tecnologia Java, Computação Móvel e Sistemas Distribuídos. Escreve artigos para as revistas JavaMagazine e WebMobile Magazine, onde também elabora vídeo aulas semanais sobre JavaME, foi palestrante de eventos como JavaOne, M3DDLA, FISL, WebMobile TechWeek, Webdays e no Just Java. Evangelista da tecnologia, procura ministrar palestras e minicursos em universidades e eventos.



Autor



Ricardo Ogliari

rogliariping@gmail.com

Graduado em Ciência da Computação, pós-graduando em Web: Estratégias de Inovação e Tecnologia. Instrutor Android na Globalcode. Autor de mais de cento e cinquenta publicações veiculadas em anais de eventos nacionais e internacionais, sites especializados e revistas. Palestrante em eventos como JustJava, Java Day, GeoLivre, ExpoGPS, FISL, FLISOL, FITE e The Developer Conference, sempre abordando temas relacionados à computação móvel.



Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/webmobile/feedback

Ajude-nos a manter a qualidade da revista!



Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia. Somos uma equipe composta de gente que entende e gosta do que faz, assim como você.



Estrutura

100% NACIONAL.
Servidores de primeira linha, links de alta capacidade.



Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.



Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.



1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
[Conheça!](#)



Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |
Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486