YambaPDM

1º Parte

Programação em Dispositivos Móveis

32766 – João Silvestre

????? – Ricardo Nunes

YambaPDM

# Introdução

# Acesso ao Yamba – TwitterAsync

A comunicação com o serviço é feita através da UI, logo, toda esta interação será executada na *thread* da UI. A comunicação com um serviço *online* deve ser sempre considerada uma ação lenta que, quando executada na *thread* da UI, irá dar ao utilizador a sensação de bloqueio, tornando a sua experiência de utilização má. De forma a evitar isto deve-se usar uma outra *thread* para executar estas ações, o Android oferece mecanismos, tais como as AsyncTask, para facilitar este processo.

De forma a tornar todo este processo transparente na UI foi criada a classe TwitterAsync que serve de *proxy* para a classe Twitter, que implementa um modelo assíncrono para aceder ao serviço. Este permite que sejam registados *listeners* para diversas ações, sendo estes chamados na *thread* da UI pela tarefa correspondente. Por exemplo, o *listener* StatusPublishedListener é chamado quando a tarefa acaba de enviar um estado.

## Primeira Versão

Esta secção tem como objetivo descrever a versão inicial da API de acesso ao Yamba.

### TwitterAsyncTask

Existe um aspeto em comum, partilhado por todas as tarefas, que implementam uma determinada ação do Yamba API, e este aspeto é o facto de todas precisarem de um objeto Twitter para poderem efetuar a ação. Desta forma, a TwitterAsyncTask garante que esse objeto exista e seja o mais atual possível, ou seja, com as configurações atuais, pois o utilizador pode mudar de utilizador e/ou serviço.

## Services

Numa segunda versão foram implementados serviços para obter a *timeline* e publicar estados. Esta troca permitiu uma maior segurança quanto à execução das tarefas em questão, isto pois o android pode a qualquer momento decidir terminar *activities*, seja por falta de memória ou um outro qualquer critério.

Tendo tarefas de longa execução, associadas a uma atividade, estas corriam sempre o risco de serem terminadas a meio, isto porque o android pode a qualquer momento terminar uma tarefa caso necessite de mais recursos. Um serviço tem uma maior resiliência a estes processos de terminação automáticos, isto pois foi determinado que primeiro serão sempre terminadas as atividades, principalmente se não estiverem a ser mostradas.

A troca de AsyncTasks para serviços não teve qualquer impacto nas atividades, isto porque estas fazem uso da API TwitterAsync para aceder ao serviço.

### TimelinePullService

O serviço “TimelinePullService” corre sempre em background desde o início da aplicação até que esta seja terminada, isto permite manter a *timeline* sempre atualizada mesmo quando o utilizador não se encontra na vista. Este serviço vai ainda guardar a timeline numa base de dados SQLite, permitindo uma consulta rápida nas próximas vezes que o utilizador volte à aplicação, ao invés de ficar a espera que se carreguem os novos pode ver imediatamente os últimos guardados.

No caso de não haver acesso à internet é parado o timer que atualiza a timeline, na eventualidade do acesso à internet voltar com a aplicação aberta é enviada uma notificação ao serviço, o qual irá arrancar o timer novamente, permitindo que a timeline volte a ser atualizada.

### UserInfoPullService

O serviço “UserInfoPullService” é responsável por obter a informação do utilizador, este serviço corre num processo separado o que faz com que se tenha que ter alguns cuidados com a sua implementação. A informação obtida do serviço tem que ser encapsulada num bundle e enviada através de um Messenger, de forma a permitir ultrapassar a barreira dos processos.

## Tratamento de Exceções

Dado que todas as ações são executadas em background é necessário permitir que a UI possa receber estas exceções, e trata-las como desejar. O mecanismo implementado, embora não sendo o ideal, permite que isto aconteça através de um *listener* especial, registado de forma estática, que é chamado sempre que uma exceção acontece.

Mas quem chama este *listener*? Esta é mais uma das responsabilidades da TwitterAsyncTask, que apanha todas as exceções e notifica quem estiver registado no *listener*, usando a *thread* da UI.

# DatabaseManager

A API de acesso à base de dados foi dividida em três partes:

* Classe DatabaseManager, o ponto de entrada para a base de dados, onde é gerido todo o processo de criação e atualização da mesma
* Package Versions, especificação de todas as versões da base de dados, isto permite ir criando versões progressivas que o DatabaseManager sabe usar para actualizar uma qualquer versão até à mais actual
* Package Model, onde existem os objetos do modelo divididos em duas partes
  + <Tabela>.class, classe simples onde contém propriedades para as colunas da tabela
  + <Tabela>DataSource.class, classe que permite operações CRUD sobre a tabela em questão

## Estrutura da Base de Dados

Existe apenas uma tabela na base de dados, esta tabela é onde são armazenados todos os estados.

**Timeline (ID INT, SERVERID INT, MESSAGE TEXT, PUBLICATION\_DATE TEXT, REPLY\_TO INT, PUBLISHED INT)**

A coluna ID é uma coluna identity local permitindo identificar unicamente o estado, a coluna SERVERID é o identificador do lado do serviço Yamba. As colunas MESSAGE, PUBLICATION\_DATE e REPLY\_TO são a mensagem, a data de publicação e, caso tenha sido uma resposta, o identificador do estado ao qual serviu como resposta, respectivamente.

A coluna PUBLISHED é um indicador interno se o estado já foi enviado para o serviço, quando não existe conexão esta coluna permite identificar quais os estados a submeter quando a conexão for recuperada.

## Versionamento

Por cada versão da base de dados deve ser criado uma classe que estende de DatabaseVersion, esta força a criação de scripts de criação desta versão e de upgrade da versão anterior. Esta versão deve ser depois registada no DatabaseManager, adicionando-a no fim do array DATABASE\_VERSIONS e alterando a referência para a versão atual (CURRENT\_VERSION) para esta.

O DatabaseManager sabe então executar todas as alterações necessárias de forma a atualizar a base de dados da versão em que esta se encontra até à atual.

# Atividades

## TimelineActivity

Esta atividade faz uso do serviço TimelinePullService para a obtenção dos estados mais recentes, fazendo depois uso de uma lista para mostrar os mesmos. Nesta atividade é possível navegar para todas as outras atividade e é ainda possível enviar o conteúdo da *timeline* na forma de um email.

### Envio da timeline por email

O envio da *timeline* é feito recorrendo a um “intent chooser”, para o tipo “message/rfc822”, que permite escolher entre todos os clientes de email instalados no dispositivo. O email criado nesta versão é apenas uma versão “plain-text” dos estados, no entanto, facilmente poderia ser criado um email HTML para mostrar o conteúdo numa forma mais apelativa.

## DetailsActivity

Esta activity é mostrada quando é escolhido um tweet da lista da timeline, mostrando todos os dados de um tweet:

* O texto do Tweet
* Criador do Tweet
* Data de Criação
* O ID do Tweet

Esta atividade permite também o envio do estado por email, da mesma forma que é permitido o envio da timeline.

## StatusActivity

## PreferencesActivity

# Requisitos Opcionais

Não foi possível ao grupo implementar os requisitos opcionais, no entanto, vai-se tentar explicar como se poderia ter abordado este problema.

## AppWidget

A widget faz uso de *broadcast* para comunicar com a aplicação, seria necessário implementar um para receber notificações do widget, e um ContentProvider para fornecer os dados armazenados.

O “broadcast receiver” quando fosse notificado que o widget estava ativo teria que se arrancar o TimelinePullService, para que o widget mostra-se constantemente os estados mais recentes. Seria depois usado um ContentProvider (que se encontra implementado em pt.isel.pdm.yamba.content.YambaTweetsProvider.java) para a obtenção dos dados armazenados na base de dados.

## Notificações na StatusBar

### Estados por ler

Para a implementação destas notificações bastava notificar o utilizador sempre que se recebesse novos estados, isto pois temos os últimos estados guardados e sabe-se quando se recebem estados mais recentes. Poder-se-ia, adicionalmente, guardar o último estado lido e fazer uso deste para quando o utilizador visita-se a aplicação, por via da notificação, mostrar automaticamente o último estado por ler, caso este ainda estive-se visível na aplicação.

### Sucesso ou Insucesso da submissão offline dos estados

Sempre que efetuada a ação de submissão dos estados offline, lançar uma notificação com base no resultado desta submissão. Em caso de falha, poderia eventualmente mostrar qual o estado no qual ocorreu o erro.

# Conclusão

Este trabalho permitiu aprender a melhor utilizar o sistema operativo Android, este faz uso de diversos conceitos para implementar, corretamente, uma aplicação. O modelo de programação do Android é bastante versátil permitindo implementar de várias formas a mesma funcionalidade, cada forma com as suas vantagens e desvantagens.

No entanto, em algumas situações o grupo ficou preso em bugs criados por especificidades do modelo de programação do android que, muitas das vezes, induz o programador em erro. Um destes caso é, por exemplo, forçar algumas chamadas a métodos da base, algo que poderia ter sido facilmente evitado, criando um ambiente mais seguro para o programador da aplicação.