

*Instituto Superior de Engenharia de Lisboa*

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Programação em Dispositivos Móveis

Relatório do 1º Trabalho

Docente: Pedro Pereira   
Discentes: Ana Correia e Diogo Cardoso

Abril 2012

# Introdução

Este relatório trata de explicar as soluções adotadas na realização do primeiro trabalho da unidade curricular *Programação em Dispositivos Móveis.* O objetivo do trabalho era a familiarização na plataforma *android* através da realização de um simples cliente *twitter* (via *Yamba*) com suporte à visualização da *timeline* do utilizador, atualização da mesma e suporte a algumas preferências.

No desenho da aplicação foi importante criar uma interface programática simples e adaptável para o acesso ao serviço *Yamba*, já que de futuro poderá ser útil adicionar novas funcionalidades à aplicação à custa do serviço.

# Acesso ao serviço Yamba

A Figura 1 representa o diagrama da classe de acesso ao serviço *Yamba*.

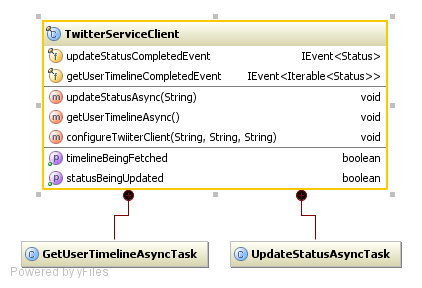
.

Figura - Diagrama da classe TwitterServiceClient.

Uma vez que o acesso ao serviço *Yamba* pode ser, realizado dentro da *thread* da *user-interface* (*UI*), todas as operações potencialmente bloqueantes foram transformadas em operações assíncronas, nomeadamente as operações *getUserTimeline* e *updateStatus*. A classe ainda expõe métodos de acesso ao estado corrente de pedidos realizados, apesar de não serem relevantes no acesso ao serviço, estes podem ser utilizados pelos controlos da UI de forma a decidirem que operações a realizar.

Por fim a classe expõe dois eventos um para cada operação assíncrona suportada. Estes eventos não são mais do que um sistema de notificação assíncrona.

## Notificação Assíncrona

A Figura 2 ilustra as interfaces base do esquema de notificação assíncrona criado.

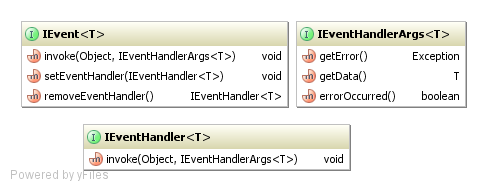


Figura - Diagrama de classes do esquema de notificação assíncrona.

A interface *IEvent* define uma "promessa" de um resultado futuro, para ser possível obter esse valor é necessário registar num objeto do tipo *IEvent* um *handler*, este irá ser invocado quando essa "promessa" for cumprida, ou seja, quando a operação assíncrona produzir um resultado. A interface *IEventHandlerArgs* serve apenas para que exista alguma forma de obter a exceção em caso de erro.

## Operações Assíncronas

A Figura 3 apresenta o diagrama de classes das operações assíncronas referidas anteriormente.

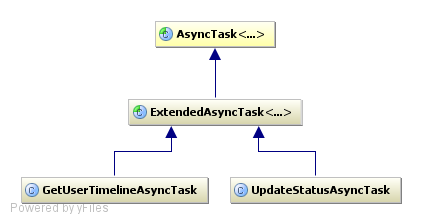


Figura - Diagrama de classes das operações assíncronas.

Ambas as operações são derivadas de *AsyncTask*, levantado dois problemas:

1. Se um *handler* de evento de uma destas operações for atualizar elementos da *UI*, esta operação **tem** de ser lançada no contexto da *thread* da *UI*.
2. A classe *AsyncTask* não é tolerante a falhas, significando que lançará exceção caso algum erro ocorra durante a operação.

O primeiro ponto é normal e comum na plataforma *android*, o segundo inviabiliza o sistema de notificação assíncrona criado já que não existe nenhuma maneira de resolver possíveis exceções. Assim foi criada a classe *ExtendedAsyncTask*, o único objetivo é capturar possíveis exceções que possam ocorrer na execução de uma *AsyncTask* e possibilitar o acesso às mesmas.

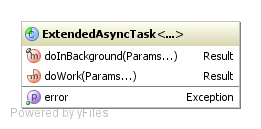


Figura - Diagrama da classe ExtendedAsyncTask.

Para capturar possíveis exceções o método *doInBackGround* de *AsyncTask* foi implementado nesta classe. Este invoca o método abstrato *doWork* de forma a executar código definido por subclasses. A Listagem 1 demonstra a utilização desta classe:

public SomeAsyncOperation extends ExtendedAsyncTask<Void,Void,Integer>{

@Override

protected Integer doWork(Void... params){

//Some intensive work.

}

@Override

protected void onPostExecute(Integer result){

Exception error = getError();

if(error == null)

//Handle the exception.

else

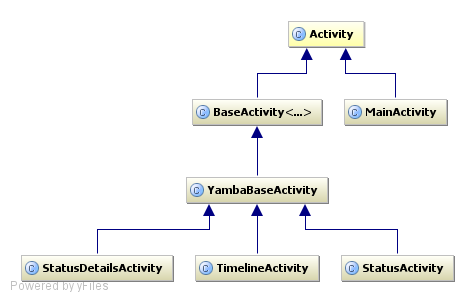
//Everything went smooth notify completation.

}

}

Listagem - Exemplo de utilização da classe ExtendedAsyncTask.

# Aplicação



Toda a gestão da *UI* e das ações realizadas pelo utilizador são controladas pelas diferentes *activities* contidas na aplicação. Para generalizar e evitar repetição de código foram criadas duas classes base:

* BaseActivity<T extends Application> - tem como principal objetivo aglomerar informação e rotinas que sejam de comum utilização pelas *activities* mas não dependam da aplicação em si. Neste caso contém o acesso tipificado à instância de *Application* e à instancia de *NavigationMessenger*.
* YambaBaseActivity - tem o mesmo objetivo que a classe BaseActivity com a diferença de aglomerar membros que sejam dependentes da aplicação como por exemplo o menu.

## Navegação

A navegação normal entre as *activities* é relativamente simples já que não existe passagem de informação entre estas. O mesmo não se aplica quando a *activity* *Timeline* necessita de navegar para a *StatusDetails*. A *activity* *Timeline* contém uma listagem de *status* (*Tweets*) e quando o utilizador carrega num, o comportamento esperado é que apareça uma nova view e que os detalhes desse *status* seja apresentado. O problema aqui é cm se passa obj para status

//Application

//Storage

//Conclusão