

*Instituto Superior de Engenharia de Lisboa*

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Programação em Dispositivos Móveis

Relatório do 3º Trabalho

Docente: Pedro Pereira   
Discentes: Ana Correia e Diogo Cardoso

Junho 2012

Conteúdo

[Introdução 3](#_Toc327206872)

[Notificação assíncrona 4](#_Toc327206873)

[Activities 5](#_Toc327206874)

[Comunicação entre activities 6](#_Toc327206875)

[Controlo 8](#_Toc327206876)

[Controladores 9](#_Toc327206877)

[Bounded services clients 9](#_Toc327206878)

[TimelineServiceController 10](#_Toc327206879)

[Serviços 11](#_Toc327206880)

[Content providers 12](#_Toc327206881)

[Application 13](#_Toc327206882)

[Conclusão 14](#_Toc327206883)

# Introdução

Este relatório aborda as soluções adotadas na realização do terceiro trabalho da unidade curricular *Programação em Dispositivos Móveis.* O objetivo do trabalho era a finalização da aplicação iniciada nos outros trabalhos, sem qualquer restrição de funcionalidades da infraestrutura.

Nesta fase grande parte do módulo de negócio foi alterado para utilizar os mecanismos da plataforma *android*, nomeadamente ao que diz respeito a comunicação entre serviços e controlo dos mesmos.

A aplicação desenvolvida está dividida em três módulos singulares, os mesmos que o padrão *Model View Controller* (*MVC*), este documento encontra-se dividido de forma a explicar as opções tomadas em cada um dos módulos.

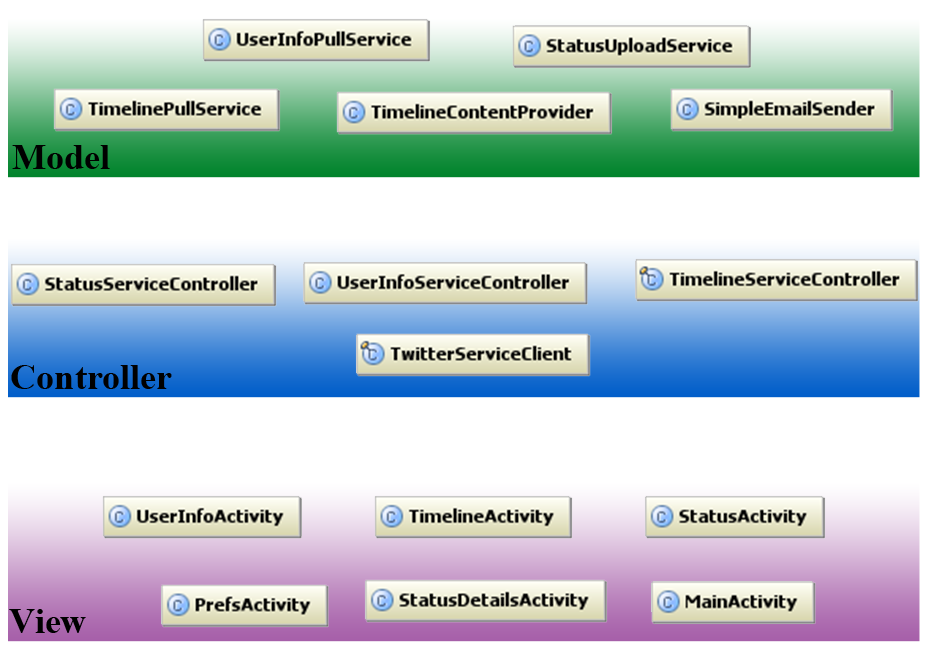


Figura - Arquitetura Geral da aplicação.

## Notificação assíncrona

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

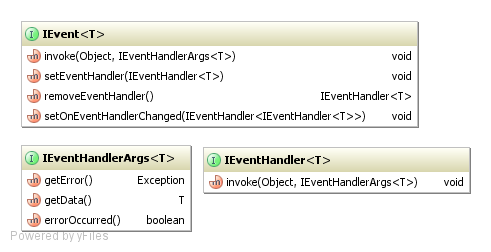


Figura - Diagrama de classes do esquema de notificação assíncrona.

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.

A motivação para a criação deste mecanismo foi que para cada operação assíncrona que exista no sistema, exista um evento associado a esta, evento que deverá ser chamado quando a operação for concluída. Um exemplo da utilização deste mecanismo está representado na Listagem 1.

class SomeClass

{

public IEvent<SomeReturnObject> someOperationCompleted;

public void someOperationAsync()

{

(...)

}

}

Listagem - Exemplo de utilização do mecanismo de notificação assincrona.

# Activities

As activities representam a componente visual da aplicação, foi tomado especial cuidado em não colocar qualquer lógica aplicacional dentro de activities sendo as suas responsabilidades as seguintes:

* Controlar os elementos visuais da aplicação.
* Iniciar outras activities por ordem do utilizador.
* Utilizar o mecanismo de notificação assíncrona descrito anteriormente para realizar operações de negócio.

Na Figura 3 é possível visualmente identificar as activities que realizam operações assíncronas devido à sua implementação explicita da interface *IEventHandler*.

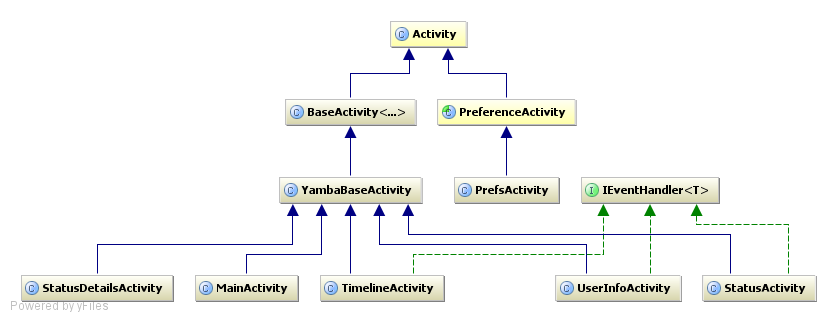


Figura - Diagrama de classes das activities.

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.

## Comunicação entre activities

A plataforma *android* tem alguns mecanismos para facilitar a comunicação entre activities, nomeadamente a possibilidade de serialização de instâncias de tipos primitivos. A serialização de um tipo complexo não é aconselhada neste tipo de plataformas pelas suas características físicas, portanto desenhou-se uma solução que consiste na utilização de um contentor de objetos referenciados por uma chave, chave essa que será um tipo primitivo. O facto de a chave ser um tipo primitivo permite a utilização dos mecanismos presentes no *android* para "parametrizar" uma *activity* com informação, nomeadamente o método *putExtra* da classe *Intent*.

Foi necessário então de criar e publicar o contentor num local único e acessível a todas as *activities*. Uma instancia estática do contentor foi criada na classe *BaseActivity* e dada permissão de acesso a todas as *activities* derivadas. A classe NavigationMessenger representa este contentor e o seu diagrama encontra-se na Figura 4.

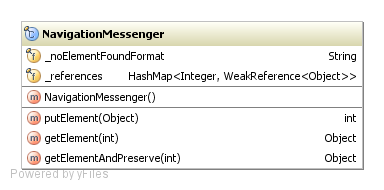


Figura - Diagrama da classe NavigationMessenger.

Na implementação da classe foi tido em conta o tempo de vida dos objetos, já que se uma instancia estática desta classe tiver qualquer referencia para um objeto este "só" irá ser "colhido" pelo *garbage* *collector* quando a aplicação terminar. Por isso a estrutura de dados utilizada uma *WeakReference*, permitindo a plataforma "colher" o objeto quando não existir mais referencias para este.

A justificação da utilização deste contentor aos outros mecanismos que a plataforma android contém para passar objetos complexos entre activities deve-se a questões de eficiencia e de aplicabilidade neste projeto. Uma solução poderia ser utilizar classes que implementam *Parcelable*, criadas para suportar a passagem de objetos complexos entre activities (nomeadamente entre processos). Nesta aplicação todas as activities se executam dentro do mesmo processo sendo o *overhead* de serialização e deserialização uma desvantagem para a performance da aplicação. Outra razão é o incomodo de criar classes *Parcelable*, além das regras não contratuais necessárias, a implementação da interface *Parcelable* "suja" a interface publica dos objetos de domínio passados pelas activities.

# Controlo

A estratégia utiliza no desenvolvimento do módulo de controlo da aplicação passou por ter controlos singulares a cada funcionalidade, significando que existe um controlador para cada operação ou conjunto de operações que um dado serviço possa realizar. De forma a simplificar futuras alterações dos controladores, as suas funcionalidades são expostas através da classe *TwitterServiceClient*, Figura 5.

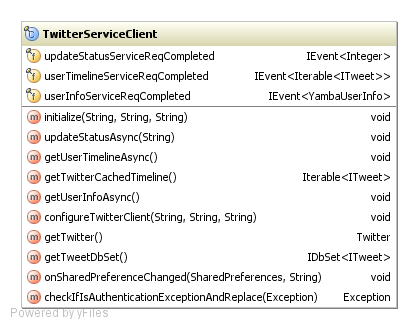


Figura - Diagrama da classe TwitterServiceClient.

Na Figura 5 encontra-se a classe central de todo o controlo da aplicação. Esta tem as seguintes responsabilidades:

* Fazer de router para os outros controladores de forma a realizar os pedidos das activities.
* Registar-se em eventos de sistema como a disponibilidade de rede e a alterações das preferências de forma a notificar os controlos dessas alterações.
* Disponibilizar eventos de finalização para todas as operações assíncronas que possam ocorrer dentro de outros controladores.

De salientar que a classe *TwitterServiceClient* apenas funciona como *facade* a operações que comuniquem de alguma forma com o twitter/yamba.

## Controladores

As classes *controllers* são as únicas que sabem como comunicar com os serviços externos. Estas servem como clientes e devem ter toda a lógica necessária para realizar pedidos e obter respostas.

As classes *controllers* são ainda responsáveis por chamar os eventos da classe *TwitterServiceClient* sempre que uma *activity* peça explicitamente que alguma operação em concreto seja realizada ou sempre que tal faça sentido.

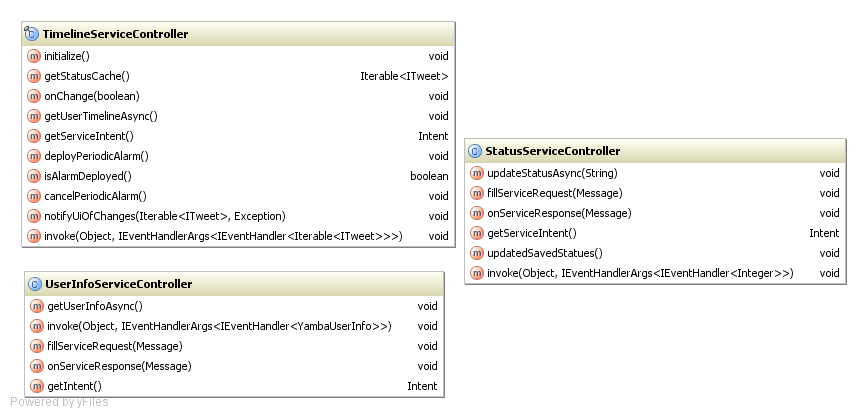


Figura - Diagrama de classes dos controladores.

Na Figura 6 encontram-se os três *controllers* criados para gerir os três serviços que a aplicação contém. Todos eles são pelo menos *bounded services*, devido à necessidade de comunicação de pedido e resposta entre os *controllers* e os serviços.

### Bounded services clients

Como referido todos os serviços são *bounded* *services* logo é necessário fazer a gestão dos pedidos e resposta destes. Os *controllers* são responsáveis por garantir que os eventos declarados em *TwitterServiceClient* são executados no contexto da *thread main*, bem como de notificar qualquer erro que ocorra no pedido realizado ao serviço.

A gestão do tempo de vida da conexão ao serviço também é responsabilidade dos *controllers*. Uma vez que os *controllers* não comunicam diretamente com as activities (nem as activities com os *controllers*) a gestão de tempo de vida das conexões aos serviços são controladas pela existência de um *handler* para o evento alvo de dada operação.

Na Figura 2 está ilustrada a interface de um evento, existe um método que permite o registo de uma *handler* que irá ser invocado quando o evento for configura com um *handler* ou esse mesmo for retirado. Através deste mecanismo e com o compromisso que as activities irão retirar o *handler* de *callback* quando já não tiverem interessadas em receber respostas assíncronas da infraestrutura é possível fazer *unbind* da conexão do serviço libertando eventualmente os seus recursos.

### TimelineServiceController

Ao contrário dos outros dois *controllers*, o *TimelineServiceController* recebe notificações assíncronas de outro local, do *Content Provider* de Status. A necessidade de ser um *observer* deve-se ao facto de o serviço de *Timeline* ser igualmente um *started service* (para a invocação periódica). Ao ser *observer* este *controller* pode receber atualizações de status sem que uma *activity* os tenha pedido.

# Serviços

Existem três serviços, cada um com a suas funções:

* UserInfoPullService - serviço responsável por retornar as informações de um dado utilizador.
* StatusUploadService - responsável por
  + Realizar o upload de um novo *status* para o serviço *yamba*.
  + Preservar *status* quando não existe conectividade.
* TimelinePullService - responsável por
  + Descarregar os *status* de um utilizador do serviço *yamba*.
  + Guardar os *status* descarregados no devido *content provider*.

## D:\FAC\LEIC\PDM\Trabalhos\working-copy\docs\trab3\pics\services.png

Figura - Diagrama de classes dos serviços.

A Figura 7 ilustra os três serviços, como já referido todos eles são *bounded services* mas os serviços de *Status* e *Timeline* são igualmente do tipo *started*.

A justificação ser todos serem *bounded* é pelo facto de ser necessário existir uma reposta explicita do serviço quando este é realizado por uma *activity*, de forma a que esta possa indicar ao utilizador que a operação acabou ou os resultados da operação.

Os serviços serem *started* igualmente deve-se à necessidade de algumas operações não necessitarem de retornar resposta **diretamente**. No caso do *StatusUploadService* este é invocado quando existe conexão para possibilitar o *upload* de status realizados pelo utilizador quando a plataforma não tinha conexão. O utilizador pode não estar na *activity* de *status* ou até pode estar noutra aplicação não faz sentido que seja reportado diretamente da finalização desta operação.

A justificação é semelhante para o caso do *TimelinePullService* já que este inicia o descarregamento após notificação de um *AlarmManager*, preservando-os posteriormente num *content provider* implementado para o caso. Se por acaso o utilizador estiver na *activity* de *timeline* o controller deste serviço tratará de mostrar os novos status ao utilizador caso contrário apenas serão preservados em memória não volátil.

## Content providers

Existem dois *content providers* na aplicação um para preservar os status descarregados do servidor *yamba* e outro para preservar os status que a aplicação não conseguiu realizar o seu *upload*.

O *content provider* da *timeline* utiliza uma base de dados SQLite para preservar os *status* e tem como produtores os serviços de *Timeline* e *StatusUpload* (quando este realiza com sucesso um *upload*). O unico consumidor é o *TimelineServiceController*.

O *content provider* dos status utiliza um ficheiro como forma de preservar os dados de forma não volátil e o único produtor e consumir é o serviço *StatusUpload*.

# Application

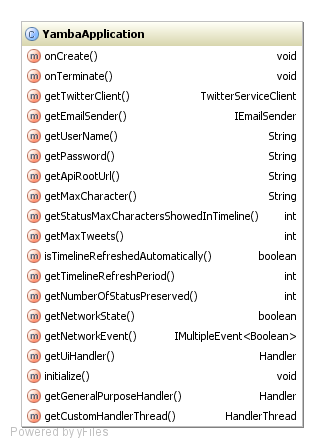


Figura - Diagrama da classe YambaApplication.

A classe *YambaApplication* é um contentor de dados uteis para toda a aplicação:

* Acesso tipificado a todas as preferências.
* *Handlers* para a *thread main* e uma *worker thread* criada para realizar os despachos assíncronos de serviços e *observers*.
* O evento de comutação de network, onde os interessados em saber alterações ao estado de rede se devem registar.
* O acesso à instancia de *TwitterServiceClient*.
* O acesso à instancia do serviço de enviar emails.

# Conclusão

Nas outras fases do trabalho toda a comunicação entre serviços e controlo era realizado com o mecanismos de eventos introduzido na introdução. O uso deste mecanismo permitiu que as activities não fossem praticamente alteradas desde a primeira fase. Apesar de nesta fase final do trabalho terem sido utilizados os mecanismos disponíveis pela plataforma android crê-se que este sistema é interessante para acesso tipificado de pedidos assíncronos intra-processo. A atual utilização do mecanismo de eventos nas *activities* podia ser alterada para o uso de *messengers* e *messages*, mas crê-se que o acesso tipificado e facilidade de utilização do mecanismo bem como das funcionalidades deste que prevalecem sobre o que é disponibilizado no android, claro que à custa da comunicação inter-processo.

Com a realização desta aplicação os intervenientes de grupo adquiriram alguns conhecimentos essenciais para o desenvolvimento nesta plataforma.