

Continuação do tutorial Adobe Flex Builder a partir do zero
<http://msdevstudio.com/blog/2008/03/01/adobe-flex-a-partir-do-zero-parte-iii/>

6 - Componentes e Módulos

6.1-As diferenças entre componentes e módulos.

Inicialmente as diferenças podem não parecer muitas, mas na realidade faz toda a diferença, no menu do flex builder se clicarem em File -> New conseguem ver uma lista, de entre os quais existem "MXML Module" e também "MXML Component". A grande diferença destes 2 é que o Componente ao ser criado estará disponível para "juntar" ao nosso "stage" principal como elemento drag & drop na janela "Components" na pasta "Custom", e o Módulo não estará acessível dessa forma.

Algumas características dos Componentes:

- Podem ser baseados em componentes já existentes (Painel, Canvas, TitleWindow, ...)
- São controlados facilmente através do nosso "stage" já que passam a estar incorporados no nosso .swf principal
- Aumentam o tamanho do nosso swf final e por conseguinte afectam o desempenho da nossa aplicação.
- Em aplicações maiores podem causar alguma confusão no código.

Quanto aos componentes, estes são "Aplicativos" independentes ou seja, são criados como se pertencerem à nossa aplicação principal, mas na realidade estão fora dela própria. Cada módulo cria um .swf próprio, que é completamente externo à nossa aplicação, além de poder ser usado em futuras aplicações deixa o nosso código bem mais simples, e muito mais facilmente editável e organizado.

Algumas características dos Módulos:

- Leves, rápidos e compactos.
- Não aumentam o tamanho da nossa aplicação principal.
- Podem ser facilmente alterados sem ter que alterar a aplicação principal.
- Facilmente actualizáveis junto da nossa aplicação principal.
- Podem ser re-usados em qualquer outra aplicação.

Um dos grandes problemas para quem não tenha grande conhecimento de action script pode ser a comunicação da nossa aplicação principal com o nosso módulo, já que grande parte da nossa comunicação deverá ser feita através de eventos. Mas mais à frente mostrarei um exemplo de como chamar uma função do nosso módulo e tratar a resposta dessa função na nossa aplicação principal.

6.2-Criando um Componente e trabalhando com ele

Um componente como disse em cima não é nada mais que um componente personalizado, como se se trata-se de por exemplo um painel personalizado logo é tratado como componente na nossa aplicação.

Vamos então criar um Componente, tomando em conta que ainda continuamos a usar a nossa aplicação **olaMundo** que criamos anteriormente, e para isso vamos ao menu do flex builder e escolhemos:

File -> New -> MXML Component

Como "filename" escrevemos **dbConf** e na opção "based on" procuramos na lista o **TitleWindow** e deixamos o resto como está clicando no "finish". Se tudo correu bem, ter-vos-à aparecido um novo documento (dbConf.mxml) apenas com o código:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
width="400" height="300">

</mx:TitleWindow>
```

A diferença entre este código e o código que aparece ao criar-mos a nossa primeira aplicação é que este inicia logo com o <mx:TitleWindow> para indicar ao compilador que a sua base será a TitleWindow deverá ser tratado como tal na nossa aplicação principal.

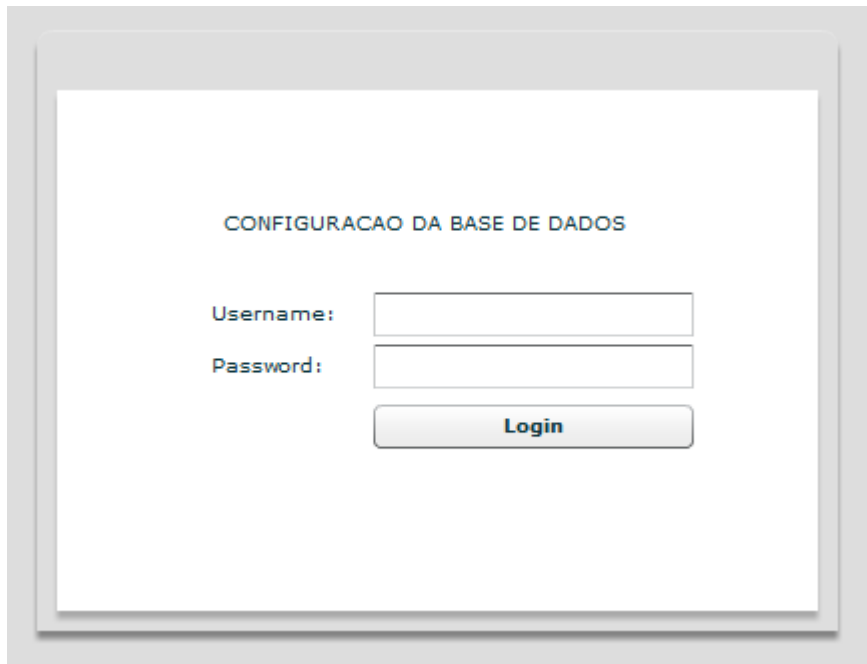
Se clicarem no modo de "design" verão que apenas é apresentado o nosso TitleWindow. Vamos aproveitar já para fazer deste componente o nosso menu para fazer um sistema de login com uma futura ligação a uma base de dados mysql, colocando 3 labels, 2 inputText (um deles como password) e 2 botões. Fica ao vosso critério a colocação dos labels "titulo", "username :", "password :" e dos campos inputText com os id's respectivamente iguais a "inputUser" e "inputPass". E o nosso botão com o id "btnLogin".

Notem que nas opções do inputText "password" devem seleccionar nas propriedades o "Display as password" colocando-o a "true".

Usando a minha interface e código:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:TitleWindow xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
width="400" height="300">
  <mx:Label x="80.5" y="58" text="CONFIGURACAO DA BASE DE DADOS"
width="219"/>
  <mx:Label x="75" y="103" text="Username:" width="75"/>
  <mx:Label x="75" y="129" text="Password:" width="80.5"/>
  <mx:TextInput x="158" y="101" id="inputUser"/>
  <mx:TextInput x="158" y="127" displayAsPassword="true" id="inputPass"/>
  <mx:Button x="158" y="157" label="Login" width="160" id="btnLogin"/>
</mx:TitleWindow>
```

Estará algo como:



Guardamos este componente e voltamos ao nosso exemplo olá mundo. Como a nossa aplicação olá mundo já está a ficar um tudo nada "non-friendly" ou seja, já está a ficar com muitas "inutilidades" vamos começar por eliminar alguns componentes que criámos...

como tínhamos um painel a ser adicionado por action script logo no inicio, vamos limpar um pouco a nossa aplicação; Um dos pontos que não tinha falado anteriormente era como remover o nosso painel criado por via de action script do nosso "stage", mas facilmente conseguiriam tira-lo ao criar um botão que executasse o seguinte código : `this.removeChild("id");` onde o id seria o nosso id criado pela função, neste caso `this.removeChild("panel3");` ("novo") ou então de uma maneira mais drástica: `this.removeAllChildren();` que elimina todos os nosso componentes adicionados com o `addChild`.

Vamos então desactivar o `creationComplete` na aplicação onde carregava os nosso outros componentes, (painel olá mundo, botão "olá" e "Adiciona panel"), para isso basta retirar do `<mx:Application ... >` o "trigger" `creationComplete="criaTudo()"`.

Ficamos apenas com a nossa `dataGrid` e os nosso efeitos. Vamos aproveitar para adicionar uma `ApplicationControllBar` que pode ser arrastada para o nosso stage, coloquem-na no topo do stage e adicionem-lhe um botão com o label "Ligar BD" de maneira a que no vosso código fique algo como:

```
<mx:ApplicationControlBar x="10" y="6" width="589">
  <mx:Button label="Ligar BD"/>
</mx:ApplicationControlBar>
```

vamos então ao nosso `mainScript.as` e fazemos as seguintes actualizações:

no topo a seguir aos nosso imports vamos colocar:

```
import dbConf;
import mx.managers.PopUpManager;
```

e no final a seguir à nossa ultima função a função seguinte:

```
private function abrePainelLogin(centrado:Boolean):void {
    var painel:dbConf = new dbConf();
    painel.showCloseButton=true;
    PopUpManager.addPopUp(painel, this, true);
    if(centrado==true) PopUpManager.centerPopUp(painel);
}

```

e no nosso dbConf.mxml colocamos na nossa tag <mx:TitleWindow ...> o seguinte: close="PopUpManager.removePopUp(this)" removedEffect="fechar"

(notem que adicionei um pequeno efeito para fechar (removedEffect) que vamos criar a seguir com o id "fechar") e teremos que fazer o import do PopUpManager senão vamos ter erros... basta adicionar antes do </mx:TitleWindow> o seguinte:

```
<mx:Script>
    <![CDATA[
        import mx.managers.PopUpManager;
    ]]>
</mx:Script>

```

de volta ao nosso olaMundo.as no botão que criamos dentro da nossa control bar no "trigger" click chamamos a nossa função (centrada=true):

```
<mx:ApplicationControlBar x="10" y="6" width="589">
    <mx:Button label="Ligar BD" click="abrePainelLogin(true)"/>
</mx:ApplicationControlBar>

```

e vamos criar os nossos efeitos chamados ao fechar o panel, para isso colocando o efeito a seguir à nossa applicationControlBar:

```
<mx:Parallel id="fechar">
    <mx:Zoom />
    <mx:Fade />
</mx:Parallel>

```

Vamos então a explicações. O que fiz foi importar o nosso dConf.mxml para o nosso stage para ser usado, e importei também o PopUpManager que será usado para fazer do nosso dbConf um popUp.

Criei a função para abrir o nosso dbConf como popup e adicionei um parâmetro que é recebido na função: centrado:Boolean que receberá apenas (true ou false) que serve para vos explicar também como enviar parâmetros para a função ao ser chamada. Dentro dessa função declarei a variável painel como "portador" do nosso dbConf.mxml e adicionei-lhe um botão de fechar:

```
painel.showCloseButton=true;
```

no nosso dbConf.mxml adicionamos o trigger "close" para remover o popup ao fechar no nosso dbConf.mxml e o removedEffect para executar o efeito que criamos ao ser fechado.

O nosso efeito com id="fechar" é um efeito paralelo, ou seja ambos os efeitos Zoom e Fade são executados ao mesmo tempo.

Podem guardar os vossos ficheiros CTRL+SHIFT+S, e corram a aplicação... se carregarem no Login BD aparecerá o nosso dbConf.mxml e se clicarmos no botão de fechar o nosso painel será removido com o efeito zoom+fade ("fechar").

Em si trabalhar com componentes é assim tão simples como importa-los e usa-los.

6.3 Enviando e recebendo dados de/para o nosso componente.

Para comunicarmos com o nosso componente, no caso de queremos enviar dados para ele, torna-se tão simples com o seguinte (suponhamos que no campo user do nosso dbConfig queremos colocar um user predefinido), basta na função "abrePainelLogin" colocarmos o seguinte:

```
painel.inputUser.text="teste";
```

a seguir ao `PopUpManager.createPopUp()`;

se escreverem `painel`. Aparece toda a lista de elementos do nosso painel, incluindo labels, buttons e inputs.

Para recebermos dados do nosso painel existem duas formas, ou adicionamos um `eventListener` para detectarmos a instrução de fechar, e acedermos às propriedades antes de remover o popup:

```
painel.addEventListener(CloseEvent.CLOSE, buscaDados);
```

```
private function buscaDados(event:CloseEvent):void{
```

```
    Alert.show(event.currentTarget.inputUser.text);
```

```
}
```

Este tipo de função e evento, não é muito usual e acaba por obrigar à duplicação de código...e apenas é disparado quando o popup é fechado, e por isso não é muito aconselhável em projectos grandes que necessitamos de usar o mesmo componente varias vezes em diferentes objectivos e de saber determinados valores sem estarmos dependentes dos events do componente.

A segunda forma é a mais lógica, que é disparar eventos personalizados e usar o `eventListener` para os escutar. Vamos por exemplo alertar o user que clicou no botão "Login" do nosso popup dbConfig.

O mais rápido seria colocar um evento no click do botão, mas vamos verificar isso com o nosso stage para termos a certeza que o nosso dbConf está a enviar dados/eventos para o nosso Stage.

Para isso basta criarmos uma função no nosso dbConf a seguir ao

```
import mx.managers.PopUpManager;
```

```
private function clickado():void {  
    this.dispatchEvent(new Event("clickado"));  
}
```

e vamos chamar esta função no botão login no "trigger" click:

```
<mx:Button x="158" y="157" label="Login" width="160" id="btnLogin"  
click="clickado()" />
```

se salvarem e correrem a vossa aplicação agora e chamarem o popup já aparece o campo user com o texto "teste" e se carregarem no login, nao acontece nada, mas na realidade ao ser clickado está a despachar o evento "clickado", mas nada acontece porque não o estamos a escutar em lado nenhum. Vamos então ao nosso `mainScript.as` e na função "abrePainelLogin" antes do `popUpManager.createPopUp` colocamos:

```
painel.addEventListener("clickado", lidaClickado);
```

e no mesmo `mainScript.as` criamos a seguinte função:

```

private function lidaClickado(event:Event):void{
    Alert.show("O botão Login do painel dbConf foi clicado!!\nO texto escrito
nos campos\n user: "+event.currentTarget.inputUser.text+"\npass:
"+event.currentTarget.inputPass.text+"\n\nCerto ??");
}

```

salvem os vossos ficheiros e corram a aplicação. Testem o botão Login com diferentes valores nos inputs user e pass. Como podem ver podemos criar eventos onde quisermos e sempre que quisermos, esta forma do dispatch event, é muito útil porque podemos usar o nosso componente onde quisermos e sem ter que o estar a alterar para adaptar a uma outra aplicação.

6.3 – Criando um Módulo e trabalhando com ele

A nível de módulos o sistema funciona quase da mesma forma que um componente como foi ditado em cima, excepto que no caso de módulos a comunicação entre o módulo e a nossa aplicação é mais elaborada.

Vamos criar um módulo com o mesmo componente que criamos antes (dbConf.mxml):

File->New->MXML Module

vamos dar-lhe o nome de modLogin e deixar os tamanhos como (250*200) e ao fundo vemos a opção de optimização, neste caso vamos escolher "Do not optimize..." para que possamos depois testar em outra aplicação que não o nosso olaMundo. Abre-se então o nosso recém-criado módulo:

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
width="250" height="200">

```

```

</mx:Module>

```

que funciona exactamente igual ao nosso dbConf ou olaMundo, com modo design activo também. Vamos então copiar o conteúdo do código do dbConf.mxml (exceptuando o <mx:TitleWindow> e </mx:TitleWindow>) e colar dentro do nosso modLogin:

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
width="250" height="200">
    <mx:Label x="15.5" y="10" text="CONFIGURACAO DA BASE DE DADOS"
width="219"/>
    <mx:Label x="10" y="54" text="Username:" width="75"/>
    <mx:Label x="10" y="93" text="Password:" width="80.5"/>
    <mx:TextInput x="80" y="52" id="inputUser"/>
    <mx:TextInput x="80" y="91" displayAsPassword="true" id="inputPass"/>
    <mx:Button x="80" y="139" label="Login" width="160" id="btnLogin"
click="clickado()" />
    <mx:Script>
        <![CDATA[
            //import mx.managers.PopUpManager;

            private function clickado():void {
                this.dispatchEvent(new Event("clickado"));
            }
        ]]>
    </mx:Script>
</mx:Module>

```

e vamos apagar/comentar a linha `import mx.managers.PopUpManager;` já que não necessitamos dela para nada neste caso. (Notem que eu alterei um pouco posições relativas (x,y) do nosso `mx:Module` para caber no nosso `stage`.)

Guardamos e o nosso módulo está criado, bem simples, vamos agora de volta ao nosso `olaMundo` e vamos inserir no código um `panel`, um `ModuleLoader` e uma `ProgressBar` (ambos `drag & drop`) mas por questões de ficar com o `layout` igual ao meu basta colarem o seguinte no código do `olaMundo` antes da tag `</mx:Application>`:

```
<mx:Panel x="10" y="47" width="354" height="310" layout="absolute">
    <mx:ModuleLoader x="10" y="10" width="314" height="226" id="loader">
    </mx:ModuleLoader>
    <mx:ProgressBar x="124" y="240" id="loaderBar"/>
</mx:Panel>
```

defini também o `id` da `progressBar` como `"loaderBar"` e do nosso `ModuleLoader` como `"loader"`

vamos adicionar o seguinte botão à nossa `applicationBar` a seguir ao nosso botão `Ligar BD`:

```
<mx:Button label="Carrega Modulo" click="carregaModulo()"/>
```

e no nosso `mainScript.as` a seguinte função:

```
private function carregaModulo():void{
    loader.url="modLogin.swf";
    loader.loadModule();
    loaderBar.source=loader;
}
```

aqui indicamos que o módulo a carregar no `loader` é o `modLogin.swf` (depois de a aplicação compilada no nosso `modLogin` é compilado num `swf` à parte com o nome de `modLogin.swf`) e que a `progressBar` `"loaderBar"` deve ter como `source` o nosso `moduleLoader` `"loader"` para acompanhar o seu carregamento.

Salvem e corram, cliquem no `Carrega Modulo` e verão o vosso módulo a ser carregado bem como o progresso do carregamento na `ProgressBar`, como no final do módulo e antes desse carregamento a `progressBar` deixa de ter sentido, vamos ocultá-la antes e depois do carregamento, mostrando-a apenas durante o carregamento do módulo. Para isso vamos colocar na tag `<mx:ProgressBar ..>` o seguinte: `visible="false"`

e na nossa função `carregaModulo`:

```
loaderBar.visible=true;
```

e para a ocultarmos, deveremos ainda colocar um `eventListener` para sabermos quando o módulo já foi carregado para que então possamos ocultar a nossa `progressBar`:

```
loader.addEventListener(ModuleEvent.READY, loadTerminado);
```

e criar a função para ocultar:

```
private function loadTerminado(event:ModuleEvent):void{
    loaderBar.visible=false;
    loader.removeEventListener(ModuleEvent.READY, loadTerminado);
}
```

salvem e corram o vosso aplicativo, agora apenas é apresentada a barra da progressBar enquanto o módulo está a ser carregado.

Quanto a acedermos e enviarmos dados para o nosso módulo, as coisas ficam um pouco mais complexas do que um componente.

Para receber dados ou obter dados, podemos usar o event dispatcher que criamos com o event "clickado", que vamos já passa a testar.

Na função loadTerminado vamos adicionar o seguinte:

```
loader.child.addEventListener("clickado", lidaClickado);
```

e aproveitamos a mesma função lidaClickado criada anteriormente, se correrem a aplicação verão que funciona na mesma.

Para enviarmos dados ou definir dados no módulo, temos duas formas, se os dados a receber form cruciais à aplicação, ou seja, forem sempre necessários, o que é aconselhado por uma questão de reutilização de código, é criarmos uma função no módulo que recebe o parâmetro/texto a definir. Se por outro lado a informação a receber no módulo for apenas esporádica, podemos aceder directamente ao componente, ou elemento que queremos afectar..

Vou passar a explicar as duas formas, tomando por exemplo que queremos colocar dados no campo inputUser do nosso modulo após o seu carregamento.

Vamos criar uma função no nosso modLogin, a seguir á função que despacha o nosso evento clickado, da seguinte maneira:

```
private function defineUser(user:String):void{  
    inputUser.text=user;  
}
```

voltamos então ao nosso mainScript.as a seguir à função loadTerminado e criamos a seguinte função:

```
private function define():void{  
    if((loader.getChildren()).length>0) (loader.child as  
modLogin).defineUser("TESTANDO");  
    else Alert.show("Modulo ainda Não carregado");  
}
```

esta função verifica se o nosso loader (ModuleLoader) já possui algum "child", ou seja, se o nosso modLogin já está disponível para ser utilizado, e caso esteja chama a nossa função defineUser do módulo.

Notem que ao colocar loader.child as modLogin estamos a indicar que o child/modulo carregado é uma "copia" do nosso modLogin para que o child do loader seja disponibilizado/utilizado com as funções e componentes do modLogin.

Para aceder ou definir qualquer dado do nosso module, este deve ser sempre tratado como disse em cima.

Esta é uma forma de passar parâmetros através de uma função, vamos então ve-la em acção, para isso vamos criar um botão no nosso olaMundo.mxml ao fundo do nosso painel, ao lado da progressBar "invisível" e em baixo do nosso moduleLoader.

Mais uma vez por questão de layout será melhor copiarem o código:

```
<mx:Button x="10" y="244" label="Define user" click="define();"/>
```

e colocarem-no em baixo a seguir ao:

```
<mx:ProgressBar visible="false" x="124" y="240" id="loaderBar"/>
```

Salvem a vossa aplicação e testem, se clicarem no botão "Define user" sem o modulo estar carregado vão receber um erro "Modulo ainda não carregado", mas se carregarem o módulo e voltarem a clicar nesse mesmo botão, o textInput do user passará a ser definido como "TESTANDO".

A outra possibilidade sem vez de utilizar uma função do módulo será aceder directamente ao campo inputUser e definir o seu texto, bastando substituir na função que criamos o chamar da função pelo campo, definindo aí o seu valor (.text). Ficaria algo como:

```
private function define():void{
    if((loader.getChildren()).length>0) (loader.child as
modLogin).inputUser.text="TESTANDO"
    else Alert.show("Modulo ainda Não carregado");
}
```

e teriam o mesmo efeito, mas a aceder directamente ao campo. Se necessitarem de usar bastantes vezes dados/funções do modulo, é altamente aconselhável definirem uma variável global para guardar o vosso modulo, para evitar estarem sempre a escrever (loader.child as modLogin), algo como definir no nosso mainScript.as, a seguir aos imports o seguinte:

```
public var modulo:*;
```

e na função loadTerminado colocarem:

```
modulo=(loader.child as modLogin);
```

e aproveitamos para testar, substituindo na função define o (loader.child as modLogin) por modulo e ficaríamos com algo como:

```
private function define():void{
    if((loader.getChildren()).length>0) modulo.inputUser.text="TESTANDO"
    else Alert.show("Modulo ainda Não carregado");
}
```

desta feita, podemos aceder ao nosso modLogin em toda a aplicação (olaMundo) através da variável modulo.

O trabalho com módulo pode passar por ser mais difícil, mas na hora da publicação da vossa aplicação e principalmente na hora da reutilização dos mesmos componentes por outras aplicações, torna-se muito mais simples e útil, já que nos pouca dezenas, se não, centenas de linhas de código além de que o nosso módulo pode ser distribuído e utilizado por outros utilizadores em outras aplicações de uma maneira simples e rápida.

7. Entendendo a comunicação com Objectos Remotos (php/mysql) via amfphp

O flex em si ainda não está completamente implementado com outras linguagens de programação, e embora já se possa criar uma aplicação com acesso a uma base de dados (Data->Create Application from database) o seu potencial é um bocado limitado comparando com as possibilidades de uma linguagem como o php, e como por vezes precisamos de dados de uma linguagem "server-side" aí o flex não tem mesmo qualquer potencialidade...

Por este e por muitos outros factores, existem vários "aplicativos" que servem de interface para que o flex possa comunicar com praticamente qualquer linguagem server side, tais como:

- PYamf, para Python
- AmfPHP para PHP
- FabreAMF para PHP5
- BlazeDS para Java
- Red5 para Java
- Fluorine para .Net

Basicamente o que estes sistemas fazem é transformar os pedidos do Action Script e transformarem-nos em pedidos para a linguagem destino, e devolver os resultados desse pedido à linguagem para um formato que o Action Script possa entender através de um processo chamado serialização.

Cada um dos sistemas em cima necessita de algumas configurações que podem encontrar junto do download ou na página desses mesmos sistemas.

O que vou falar chama-se amfPhp. Devem então fazer o download do mesmo em e podem também encontrar no site oficial (<http://www.amfphp.org/>) alguma documentação bem como alguns exemplos e tutoriais.

Depois do download feito, basta extrair o conteúdo do arquivo (.zip) para qualquer directoria e está pronto a usar. É bem simples o seu funcionamento como vamos ver mais à frente. Agora vamos proceder às configurações necessárias para poder ter a correr no nosso olaMundo um sistema que nos permita trabalhar com o php e por consequente o mysql.

Vamos copiar então a pasta "amfphp" para dentro da pasta do nosso projecto e de seguida abrir essa mesma pasta, se seguida abrir a pasta "services" onde podem encontrar um ficheiro que diz "place_services_here", ora estamos no sítio certo para criar o nosso serviço.

Um serviço é como se tratasse de uma interface dentro do próprio amfphp, é também o serviço que será identificado no flex.

Notem que para isto funcionar necessitam de ter o vosso projecto (olaMundo) dentro de um "servidor" web que tenha suporte a php. Existem alguns bastante simples de instalar e de colocar a funcionar, tal como o WAMP

(<http://www.wampserver.com/>), bastando depois instalar com os parâmetros por defeito e escolher o user e password da base de dados do mysql (vamos usá-los de seguida)

Este processo de trocar o nosso projecto (olaMundo) para uma directoria diferente da actual é mais complicado do que apenas mudar a localização, já que o flex não gosta muito disso e ainda por cima devemos trocar o nosso projecto para trabalhar com um servidor. O que aconselho, supondo que instalamos o wamp em c:/wamp é fazer o seguinte. Salvamos todos os nossos ficheiros do projecto (olaMundo) e fechamos o projecto (botão direito->Close Project).

Vamos depois copiar a pasta do nosso projecto para o servidor em c:\wamp\www\

Voltamos ao flex e no final do projecto fechado temos que o apagar (botão direito->Delete) e escolher "Do not delete Contents" já que o novo projecto a importar terá o mesmo nome.

No final vamos ao menu File->New->Flex Project.

No nome colocamos olaMundo, em projecto location, clicamos na "checkbox" para alterar e colocamos c:\wamp\www\olaMundo e em baixo escolhemos Application Server Type -> PHP

Clicamos em Next e deveremos colocar (se não foi colocado automaticamente)

Web root: c:\wamp\www\olaMundo

Root Url: <http://localhost/olaMundo>

Output Folder: bin-debug

clicamos em Validate Configuration e de novo em Next

em Main source folder: (aqui devemos verificar se na pasta olaMundo do nosso c:\wamp\www\olaMundo o olaMundo.mxml está na raiz da pasta ou na pasta src, já que na altura da criação do nosso projecto anterior eu defini o main source folder sem valor e logo foi criado o olaMundo.mxml na raiz, se usaram os valores por defeito na altura da criação devem deixar estar como "src")

Vamos supor que está na raiz, e limpamos este campo, em Main Application File devemos ter o olaMundo.mxml, clicamos em finish e o flex actualizará o nosso projecto e está pronto para usar de novo, mas desta feita no nosso servidor.

Voltando à pasta c:\wamp\www\olaMundo\amfphp\services\

Vamos criar uma pasta chamada **ola** e dentro dela criar um ficheiro de texto com o nome de **mondo** e guarda-lo com a extensão php, se o ficheiro não for guardado como .php deve aparecer o nome como mundo.php e ao abrir vi abrir no bloco de notas (é sinal que o Sistema Operativo está a ocultar extensões), neste caso, apaguem o ficheiro (mondo.php.txt) e voltem à pasta services, abram a pasta amfphp e copiem o ficheiro que ela contem (DiscoveryService.php) e copiem-no para a pasta ola, depois mudem o nome para mundo e abram esse ficheiro no bloco de notas e apaguem todo o seu conteúdo e escrevam apenas o seguinte a titulo de exemplo:

```
<?php
```

```
class mundo
```

```
{
```

```
    function teste($nome) {
```

```
        return "Olá ".$nome;
```

```
    }
```

```
    function mundo() {
```

```
        //Função principal do serviço, que poderá guardar variáveis
```

```
        //sempre disponíveis ao longo das nossas funções, tal como
```

```
        //uma ligação à base de dados
```

```
    }
```

```
}
```

?>

e guardem o ficheiro.

O amfphp tem um browser que permite escutar os nossos serviços e testa-los, logo vamos aproveitar para testar o serviço que criamos. Acedam com o vosso browser de internet a: <http://localhost/olaMundo/amfphp/browser> e vai-se abrir uma pagina a pedir informações sobre o gateway, que devem deixar estar como está por defeito clicando em save.

Do vosso lado esquerdo deve aparecer os serviços amfphp e o nosso recém criado ola, cliquem em cima dele e escolha o mundo. Do lado direito deve aparecer apenas a função "teste" com um campo a pedir a nossa variavel \$nome, vamos introduzir o vosso nome, exemplo joão, e carregar em call.

Se tudo correr bem ao fundo dessa pagina terão algumas tabs e na Results aparecerá o devolvido pelo return da função, neste caso olá joão, provavelmente irão aparecer uns caracteres esquisitos, já que o valor é devolvido com um padrão de caracteres diferente, para alterar isso voltamos ao nosso mundo.php e colocamos em vez de return "olá ".\$nome colocamos:

```
return "Olá ".utf8_decode($nome);
```

e testem o call de novo no browser. Agora deve devolver em Results a string "Olá joão" correctamente.

Neste momento o serviço está a funcionar correctamente, só temos que fazer com que o flex o teste com sucesso também. O flex por si não sabe o que é o amfphp e muito menos onde o ir buscar, por isso deveremos incluir um services-config.xml na altura da compilação. Podem fazer o download do meu site do services-config.xml já configurado para a nossa aplicação.

Download em: <http://www.msdevstudio.com/mywork/services-config.rar>

devem extrair o ficheiro services-config.xml para a raiz do projecto (ou na pasta src conforme definido antes no main source folder) (c:\wamp\www\olaMundo) e caso o caminho da aplicação no vosso servidor não seja o <http://localhost/olaMundo> devem editar este services-config.xml com o bloco de notas e ao fundo onde tem:

```
uri="http://localhost/olaMundo/amfphp/gateway.php"
```

devem alterar para o caminho correcto do vosso servidor e guardar.

Depois voltamos ao flex, ao nosso projecto olaMundo e vamos ao menu:

Project->Properties->Flex Compiler e no campo onde tem -locale en_US vamos escrever:

```
-locale en_US -services "services-config.xml"
```

e carregar em Apply, aguardem a actualização e cliquem em Ok.

Se aparecer um erro, cliquem em cima do projecto OlaMundo com o botão direito e depois Refresh e o erro deve desaparecer.

Se tudo correu bem, estamos prontos para o nosso primeiro teste com objectos remotos, neste caso o amfPhp, para isso no código do nosso olaMundo.mxml antes da tag </mx:Application...> colocamos os nosso serviço:

```

<mx:RemoteObject id="nossoObjecto" destination="amfphp" source="ola.mundo">
  <mx:method name="teste" result="{lidaTeste(event)}">
    <mx:arguments>
      <nome>
        {"António"}
      </nome>
    </mx:arguments>
  </mx:method>
</mx:RemoteObject>

```

e criar a seguinte função no nosso mainScript.as no final:

```

private function lidaTeste(evento:ResultEvent):void {
    Alert.show("Adivinha? o php respondeu à nossa chamada e
    devolveu:\n"+(evento.result as String));
}

```

e colocar junto dos imports:

```
import mx.rpc.events.ResultEvent;
```

O nosso Objecto está pronto, e a função para lidar com os resultados também... vamos adicionar um botão ao lado no nosso "Define user" no olaMundo.mxml, bastando no código a seguir a esse botão colocar:

```
<mx:Button x="110" y="244" label="Chama RO" click="nossoObjecto.teste.send()"/>
```

Guardem todos os vossos ficheiros vamos a explicações antes de lançar a aplicação.

Criamos o nosso RemoteObject com o id="nossoObjecto" que será utilizado para chamar qualquer função deste objectos, com o destination="amfphp" que é indicado no services-config.xml e com o source="ola.mundo" que são o nosso serviço:

pasta.ficheiro dentro da pasta services do amfphp.

Depois temos o <mx:method ..> onde o name é exactamente o nome da nossa função php, para cada função devemos ter um método. O result é a função que vai lidar com os resultados devolvidos pelo php.

A tag <mx:arguments> indica o inicio dos argumentos a passar, no nosso caso a função do php teste apenas recebe a variável \$nome (<nome>) e no nosso código colocamos o conteúdo da variável, neste caso "António".

Se tivermos mais variáveis a receber, devemos coloca-las sempre pela respectiva ordem... um exemplo, se a função php receber 3 variáveis, por exemplo:

function teste(\$nome, \$idade, \$local); devemos colocar no nosso remote object o seguinte:

```

<mx:arguments>
  <nome>
    {"António"}
  </nome>
  <idade>
    {"21"}
  </idade>
  <local>
    {"Lisboa"}
  </local>
</mx:arguments>

```

Logo depois temos a nossa função para "tratar" os dados recebido no `remoteObject`, a qual recebe um evento de resultado (`evento:ResultEvent`) no qual são incluídos os dados enviados pelo `amfphp` tratados como `evento.result`, ao fazer o `Alert.show()`; tratei-os como `string` (as `String`)

Basta então correrem a vossa aplicação depois de a ter salvo e carregarem no botão "Chama RO" e se tudo correu bem, têm a vossa aplicação a comunicar com o `php` via `amfphp remote object`.

Bem, por agora é tudo, na próxima parte do tutorial iremos utilizar este exemplo para fazer o login numa base de dados `mysql` e buscar dados para a nossa `dataGrid` presente no exemplo.