Développement d'un gestionnaire de contacts (CManager) avec ActionScript Foundry (AS Foundry) - Partie 1

par Wajdi Hadj ameur (ServeBox Open Source)

Date de publication : 2 Avril 2009

Dernière mise à jour :

Dans ce premier tutoriel, nous mettrons en évidence les concepts fondamentaux MVC et ceux de l'AS Foundry.

1 - Introduction

Il existe aujourd'hui un nombre important d'initiatives open-source destinées au développement d'applications Flex. Le framework AS Foundry a été conçu en 2005 puis proposé à la communauté open source en 2007 par l'équipe de développement de ServeBox.

Afin d'illustrer son fonctionnement, nous allons créer pas-à-pas une application de gestion de contacts. CManager est constitué d'un client Flex et d'un service Java/BlazeDS exécuté sous Tomcat. Basé sur une structure simple, CManager permet la gestion (ajout, modification, suppression) de fiches contacts récupérées depuis le service Java.

Dans ce premier tutoriel, nous mettrons en évidence les concepts fondamentaux MVC et ceux de l'AS Foundry. D'autres tutoriaux permettront d'enrichir le CManager et illustreront certaines des fonctionnalités avancées de la ToolBox AS Foundry.

*Vous avez un aperçu de CManager à l'adresse suivante : **http://cmanager.servebox.org***

*La documentation de l'API est disponible **ici**.*

2 - Prérequis

Environnement de développement :

- Eclipse 3.3.2 : **http://www.eclipse.org**
- Flex Builder Plugin 3.2 pour Eclipse : **http://www.adobe.com/products/flex/**

Environnement d'execution :

- JDK 1.5
- Tomcat 6.x : nous avons préparé des versions préconfigurées de Tomcat pour Windows et Linux, disponibles ci-dessous.

3 - Installation

3-1 - Installation du JDK 1.5

Télécharger et installer le **JDK 5.0 Update x**.

Créer la variable d'environnement correspondant à votre installation, par exemple :

```
${JAVA_HOME} = C:\Program Files\Java\jdk1.5.0_17
```

Rajouter la variable dans votre path :

```
Path= ... ;%JAVA_HOME%\bin
```

3-2 - Installation de Tomcat

Télécharger **apache-tomcat-6.0.16.zip**.

Dézipper l'archive dans un répertoire, par exemple : C:\Foundry\apache-tomcat-6.0.16

Créer la variable d'environnement :

```
${CATALINA_HOME} = C:\Foundry\apache-tomcat-6.0.16
```

Rajouter la variable dans votre path :

```
Path= ... ;%CATALINA_HOME%\bin
```

Votre serveur d'application Apache Tomcat est prêt, lancez le grâce à startup.bat (C:\Foundry\apache-tomcat-6.0.16\bin).

3-3 - Paramétrage de Flex Builder ou Eclipse

Téléchargez la structure de l'application **CManager** (format zip, 1 049 ko) et complétez le code source en suivant le tutoriel pas à pas.

Dans le fichier .actionScriptProperties de *flex-gui* remplacer l'attribut outputFolderLocation de la balise compiler :

```
outputFolderLocation="/C:/Foundry/apache-tomcat-6.0.16/webapps/contactApplication/flex-gui-debug"
```

Toujours la même manipulation pour l'attribut serverRoot de la balise flexProperties dans le fichier .flexProperties :

```
serverRoot="C:\Foundry\apache-tomcat-6.0.16\webapps\contactApplication"
```

Dans le fichier .project de *flex-gui* remplacer la balise location afin de pointer sur le dossier de déploiement dans Apache Tomcat :

```
<location>/C:/Foundry/apache-tomcat-6.0.16/webapps/contactApplication/flex-gui-debug</location>
```

Vous pouvez désormais importer le projet *flex-gui* dans votre workspace.

4 - Le projet

L'application CManager est composée de trois sous-projets :

- « flex-gui » pour l'interface graphique Flex
- « java-service » pour le service Java
- « web-app » qui permet l'assemblage de l'application Web déployée sous Tomcat. Si vous utilisez la version de Tomcat fournie avec cet exemple, vous n'aurez pas besoin de vous intéresser à cette partie pour réaliser ce tutoriel.

4-1 - Structure physique

Le projet « Java-service » contient les classes Java nécessaires à l'exécution du service pour la récupération des données. Il embarque une base de données HSQLDB qui s'installe automatiquement dans un répertoire temporaire du serveur Tomcat. Nous ne nous étendrons pas sur la partie service Java mais nous nous concentrerons sur le projet « flex-gui ».

Le dossier src/main/flex contient les sources.

Le dossier src/main/resources contient les ressources additionnelles utilisées par le compilateur : assets et feuilles de styles CSS.

4-2 - Structure logique : Modèle, vue et contrôleur

Basée sur différents design patterns, l'AS Foundry permet un cycle de développement réduit grâce à l'utilisation d'outils tels que la synchronisation de données modèles-vues, navigation inter-écrans, liste de contrôle d'accès, internationalisation et externalisation de labels, et bien d'autres !

Dans le pattern MVC, le modèle prend en charge les données de l'application, les vues servent à l'affichage de ces données et au support des interactions de l'utilisateur. Enfin, le contrôleur traduit les actions de l'utilisateur en modifiant les données du modèle ou en déclenchant des appels aux services distants.

Pour plus d'information sur MVC, consultez l'article ***« The MVC Framework »*** *de l'ASFoundry.*

Les principaux packages AS Foundry sur lesquels nous allons intervenir sont les suivants :

- view : layout graphique
- helper : logique des vues
- control : contrôleur
- business : services distants
- model : modèle de données

5 - Initialisation

5-1 - Création du modèle

La partie modèle du framework AS Foundry est représentée par la classe AbtractModel. Cette classe est une implémentation de IObservable (sujet dans le design pattern Observer). Le rôle d'un Observable est de transmettre les modifications à une ou plusieurs instances d' « Observer » (dans notre cas les vues) en utilisant un mécanisme de notification.

Modèle du framework AS Foundry

Pour créer notre modèle, nous étendons la classe AbstractModel fournie par le framework.

```
//model/ContactModel.as
...
public class ContactModel extends AbstractModel
{
 public function ContactModel()
 {
  super();
 }
}
```

Pensez à bien vérifier que tous les imports de classe sont faits.

ContactModel va gérer la liste de contacts. Le getter et le setter permettent respectivement de récupérer et de modifier cette liste de contacts. L'utilisation de la méthode notifyObservers permet aux observateurs d'être informés des modifications intervenant sur la liste de contacts. Les observateurs (nos vues) pourront s'abonner au modèle et être notifiés des modifications intervenant sur nos données.

```
//model/ContactModel.as
...
```

```
private var _contacts : ArrayCollection;
...
public function getContacts(): ArrayCollection
{
 _contacts.source.sortOn("lastName");
 return _contacts;
}
public function setContacts( ar : ArrayCollection ): void
{
 _contacts = ar;
 // Notification des observateurs éventuels
 notifyObservers( new ContactListNotification( null ) );
}
...
```

Pensez à bien vérifier que tous les imports de classe sont faits.

5-2 - Création du contrôleur

Le contrôleur permet de traiter l'ensemble des actions réalisées par l'utilisateur depuis les vues en faisant appel au modèle ou aux services distants. Pour créer le contrôleur de l'application, on étend simplement la classe AbstractController.

Ce contrôleur étant le point d'entrée principal de l'application :

- il doit implémenter le design pattern singleton afin que l'on puisse toujours obtenir la même instance ;
- il est chargé d'initialiser les modèles et les délégués des services distants (voir 7.3 Création du BusinessDelegate) : pour ce faire, nous surchargeons les méthodes initializeModels et initializeDelegates.

Le code suivant initialise ces méthodes pour créer l'instance de MainController. On peut noter que notre modèle est enregistré auprès du ModelLocator. ModelLocator implémente un Singleton, cette classe enregistre des références aux modèles de l'application.

```
//control/MainController.as
...
public class MainController extends AbstractController
{
 private static var _mainControllerInstance : MainController;
 ...
 /**
 * MainController Singleton
 * */
 public static function getInstance() : MainController
 {
  if(_mainControllerInstance == null)
  {
   _mainControllerInstance = new MainController();
  }
  return _mainControllerInstance;
 }
 ...
 /**
 * Initialisation des Modèles
 * */
 override public function initializeModels():void
 {
  ModelLocator.getInstance().registerModel( getQualifiedClassName(ContactModel), new ContactModel() );
 }
 ...
 /**
 * Initialisation des services distants
 * */
 override public function initializeDelegates():void
```

```
 {
  //Initialisation des services distants
 }
 ...
```

Pensez à bien vérifier que tous les imports de classe sont faits.

Maintenant que votre Modèle et votre Contrôleur sont créés, nous pouvons initialiser l'application puis commencer à créer nos vues.

5-3 - Initialisation de l'application

Dans le cas d'une application Flex basique, la racine est une instance de mx.core.Application. Dans le cas de l'AS Foundry, l'application doit effectuer l'initialisation de notre contrôleur. Nous étendons donc la classe AbstractApplication en redéfinissant la méthode *getControllerClass*.

```
//control/MainApplication.as
...
package org.servebox.sample.contactapplication.control
{
 import org.servebox.foundry.control.AbstractApplication;
 public class MainApplication extends AbstractApplication
 {
  public function MainApplication()
  {
   super();
  }
  override protected function getControllerClass():Class
  {
   return MainController;
  }
 }
}
```

Notre application est prête, il suffit maintenant de la déclarer comme composant racine, dans le fichier main.mxml.

```
<?xml version="1.0" encoding="utf-8"?>
<control:MainApplication
 xmlns:mx="http://www.adobe.com/2006/mxml"
 xmlns:control="org.servebox.sample.contactapplication.control.*"
 xmlns:view="org.servebox.sample.contactapplication.view.*"
 >
 <mx:Style source="/../resources/css/main.css"/>
</control:MainApplication>
```

6 - Créer une vue

Pour créer une vue, on ajoute un fichier MXML en utilisant comme balise racine, un des composants disponibles de l'AS Foundry (CanvasView, HBoxView, VBoxView).

L'application est constituée d'une vue avec plusieurs « State » (MainView.mxml) :

- BASE_VIEW_STATE permettant la consultation et la modification d'un contact.
- ADD_CONTACT_VIEW_STATE permettant l'ajout d'un contact.

Nous n'allons pas nous attarder sur le code source de la vue, qui ne contiendra que des composants graphiques (le code source de la vue est disponible dans le package view). Nous allons plutôt nous intéresser au ViewHelper qui va permettre à notre vue d'interagir avec le reste de l'application.

6-1 - Le ViewHelper

La seule utilisation de MXML pour décrire les vues n'est pas conseillée, même si cela peut paraître plus facile au premier abord. Le code source devient vite difficile à maintenir car le de layout et le code de gestion sont mélangés.

Le framework AS Foundry propose une séparation entre la présentation et la logique fonctionnelle. Chaque vue est associée à un ViewHelper, contenant:

- la logique fonctionnelle
- les appels au contrôleur
- les événements
- la gestion des notifications provenant du modèle et les « data binding »

Ainsi la vue peut facilement être décrite en utilisant les balises MXML. La logique fonctionnelle étant séparée, le code sera plus facile à maintenir, réutiliser ou encore étendre.

Nous créons donc la classe MainViewHelper qui contient un « binding » vers la liste des contacts. En tant qu'implémentation de l'interface IObserver, notre ViewHelper peut s'enregistrer auprès de ContactModel et être notifié en surchargeant la méthode *update*.

```
//helper/MainViewHelper.as
package org.servebox.sample.contactapplication.helper
{
import org.servebox.sample.contactapplication.model.ContactModel;
import flash.events.Event;
import flash.utils.getQualifiedClassName;
import org.servebox.foundry.model.ModelLocator;
import org.servebox.foundry.observation.IObservable;
import org.servebox.foundry.observation.Notification;
import org.servebox.foundry.view.ViewHelper;
import org.servebox.sample.contactapplication.control.MainController;
import org.servebox.sample.contactapplication.model.notification.ContactListNotification;
import org.servebox.sample.contactapplication.service.value.vo.ContactVO;
import org.servebox.sample.contactapplication.view.MainView;
public class MainViewHelper extends ViewHelper
{
...
 /**
 * S'enregistre comme un observateur auprés de ContactModel
 * et permet de recevoir les notifications depuis ContactModel
 * */
 override public function registerToModels():void
 {
  var model : IObservable = ModelLocator.getInstance().getModel(getQualifiedClassName( ContactModel ));
  model.registerObserver(this);
 }
 /**
 * Déclenché aprés une notification de ContactModel
 * */
 override public function update( o:IObservable, n:Notification) :void
 {
  // réalise une action aprés avoir reçu une notification depuis ContactModel
  dispatchEvent( new Event("contactListChange") );
 }
}
...
```

Pensez à bien vérifier que tous les imports de classe sont faits.

6-2 - Ajout de la vue à l'application

Avant d'ajouter la vue à l'application, il faut lui associer son « helper ». Il suffit d'ajouter simplement cette balise dans le code MXML:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- view/MainView.mxml -->
<view:HBoxView
 xmlns:view="org.servebox.foundry.view.*"
 xmlns:mx="http://www.adobe.com/2006/mxml"
 width="100%"
 height="100%"
 horizontalAlign="center"
 horizontalCenter="0">
 <!-- association de la vue avec MainViewHelper.as -->
 <helper:MainViewHelper id="helper" autoRegisterBasedOnQualifiedClassName="true"/>
...
</view:HBoxView>
```

On ajoute la vue à notre application, en la plaçant dans un « ViewStack » par exemple :

```
<?xml version="1.0" encoding="utf-8"?>
<control:MainApplication
 xmlns:mx="http://www.adobe.com/2006/mxml"
 xmlns:control="org.servebox.sample.contactapplication.control.*"
 xmlns:view="org.servebox.sample.contactapplication.view.*"
 >
 <mx:Style source="/../resources/css/main.css"/>
 <mx:ViewStack id="myViewStack" width="1024" height="720">
  <view:MainView id="mainView"/>
 </mx:ViewStack>
</control:MainApplication>
```

7 - Connecter une vue au service

Pour charger nos contacts, l'application va déclencher la séquence d'opérations suivantes :

- 1 - Au chargement de l'application, le ViewHelper est appelé pour récupérer la liste des contacts.
- 1' - Il appelle la méthode getContactList dans le contrôleur.
- 1" - Le contrôleur crée une instance du Responder (expliqué ci-dessous).
- 2 - Le contrôleur demande au BusinessDelegate de commencer l'appel au service distant.
- 2' - Le BusinessDelegate appelle le service et récupère un ACT (Asynchronous Completion Token).
- 2" - Le BusinessDelegate renvoie l'ACT à l'instance du Responder.
- 3 - Le service répond et le Responder associé à l'ACT est appelé.
- 3' - Le Responder gère la réponse du service en modifiant la valeur dans le modèle.
- 4 - Une notification est envoyée aux observers
- 4' - Le ViewHelper peut alors récupérer les données et déclencher la mise à jour de la vue.

Asynchronous Completion Token (ACT): les outils de connectivité Flex utilisent ce design pattern, on associe les actions et les états de l'application avec des réponses qui indiquent que les opérations asynchrones sont terminées. Pour chaque opération asynchrone, on crée un ACT qui identifie les actions et les états requis pour le résultat de l'opération. Quand le résultat est renvoyé, on peut utiliser cet ACT pour le différencier des résultats des autres opérations asynchrones. Le client utilise l'ACT pour identifier l'état requis pour gérer le résultat.

7-1 - Traiter la réponse du service (responder)

Le rôle principal d'un Responder est de gérer les réponses d'une méthode spécifique provenant du service. Chaque Responder doit implémenter l'interface IbusinessResponder. Les méthodes *fault* et *result* sont destinées respectivement à gérer les erreurs systèmes (erreur réseau par exemple) ou propager les résultats. Dans le cas présent, la méthode *résult* de *ContactListResponder* fait appel au setter de *ContactModel*.

```
//business/responder/ContactListResponder.as
...
public class ContactListResponder implements IBusinessResponder
{
 public function ContactListResponder()
 {}
 public function result(data:Object):void
 {
  // Ici on gère la réponse du service distant
 }
 public function fault(info:Object):void
 {
  Alert.show("Error getting contact list !");
 }
}
```

7-2 - Création du BusinessDelegate

Avant d'ajouter la fonction au contrôleur, nous devons créer notre « businessDelegate ». Lorsque vous utilisez l'AS Foundry, il faut créer une instance de BusinessDelegate, afin de permettre les interactions entre le service distant et le design pattern MVC. L'un de principaux avantages réside dans le fait que les changements sur le service distant ne poseront de conflits que sur le BusinessDelegate, et aucun autre composants de l'application. Il y a d'autres avantages à utiliser les « businessDelegate », pour avoir plus d'informations lisez **http://www.servebox.org/actionscript-foundry/actionscript-foundry-documentation/remote-procedure-call/**.

Il faut donc créer *MainBusinessDelegate* dans le package com.servebox.sample.contactapplication.business. Puis on crée la méthode getContactList, à l'intérieur de cette méthode on relie l'ACT (asynchronous completion token) à une instance de IbusinessResponder. Le « responder » gère la réponse du service.

```
//business/MainBusinessDelegate.as
...
public class MainBusinessDelegate extends AbstractBusinessDelegate implements IBusinessDelegate
{
 public function MainBusinessDelegate( service : Object=null )
 {
  super(service);
 }
 /**
 * Appel la requete getContactList() du service Java
 * */
 public function getContactList( responder : ContactListResponder ) : void
 {
  linkResponderToCallToken( getService().getContactList(), responder );
 }
...
```

On initialise le « businessDelegate » en surchargeant la méthode initializeDelegates dans le contrôleur. L'appel au serveur distant est réalisé par le contrôleur en utilisant la classe ContactListResponder (que nous allons créer par la suite) en tant que « responder ».

```
//control/MainController.as
...
```

```
/**
* Récupère le service Java
* */
private function get service() : RemoteObject
{
 var service : RemoteObject = new RemoteObject("java-service");
 service.channelSet = ChannelSetProvider.getInstance().getDefaultChannelSet();
 return service;
}
/**
* Initialisation du service distant
* */
override public function initializeDelegates():void
{
 var bd : MainBusinessDelegate = new MainBusinessDelegate( service );
 registerBusinessDelegate( bd, getQualifiedClassName( MainBusinessDelegate ) );
}
/**
* Appel à la fonction getContactList du MainBusinessDelegate
* */
public function getContactList() : void
{
 getMainBusinessDelegate().getContactList( new ContactListResponder() );
}
...
```

Pensez à bien vérifier que tous les imports de classe sont faits.

7-3 - Afficher les données

Nous commençons par créer une méthode dans MainViewHelper pour appeler la méthode *getContactList* dans le contrôleur.

```
//helper/MainViewHelper.as
...
 /**
 * Appel la méthide getContactList dans MainController
 * */
 public function retrieveContactList() : void
 {
  MainController.getInstance().getContactList();
 }
...
```

On crée le « binding » sur le getter dans MainViewHelper, pour alimenter la liste. Si vous n'êtes pas familier avec le binding, lisez ce document: **http://www.adobe.com/devnet/flex/quickstart/using_data_binding/**.

```
//helper/MainViewHelper.as
...
 /**
 * Renvoi un tableau des contacts stockés dans le model
 * */
 [Bindable("contactListChange")]
 public function get getContactList() : Array
 {
  return getContactModel().getContacts().source;
 }
 /**
 * Accesseur de ContactModel
 * */
 public function getContactModel() : ContactModel
 {
  return ContactModel(ModelLocator.getInstance().getModel(getQualifiedClassName( ContactModel )));
 }
```

```
...
```

Pensez à bien vérifier que tous les imports de classe sont faits.

Nous mettons à jour *MainView* et son *MainViewHelper*. Il suffit ensuite d'ajouter la propriété DataProvider au composant List et un bouton pour récupérer la liste des contacts.

```
//view/MainView.mxml
...
 <HBoxView
  xmlns="org.servebox.foundry.view.*"
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:comp="org.servebox.sample.contactapplication.component.*"
  width="{MainConf.SCENE_WIDTH}" height="{MainConf.SCENE_HEIGHT}"
  xmlns:form="org.servebox.toolbox.form.*"
  xmlns:elements="org.servebox.toolbox.form.elements.*"
  xmlns:helper="org.servebox.sample.contactapplication.helper.*"
  horizontalAlign="center"
  horizontalCenter="0"
  horizontalScrollPolicy="off"
  verticalScrollPolicy="off"
 >
...
 <mx:Button
  id="getBtn"
  icon="{EmbeddedMedia.getInstance().updateContactImg}"
  click="{helper.retrieveContactList()}"
 />
...
 <mx:List
  width="300"
  height="648"
  id="contactList"
  dataProvider="{helper.getContactList}"
 />
...
```

7-4 - Gérer les réponses du service

getContactList nous renvoie un TransferObject de type ContactTO:

```
//service/value/transfer/ContactTO.as
package org.servebox.sample.contactapplication.service.value.transfer
{
 import mx.collections.ArrayCollection;
 import org.servebox.foundry.transfer.TransferObject;
 [Bindable]
 [RemoteClass(alias="org.servebox.sample.contactapplication.service.value.transfer.ContactTO")]
 public class ContactTO extends TransferObject
 {
  private var _contacts : ArrayCollection;
  public function set contacts( value : ArrayCollection ) :void
  {
   _contacts= value;
  }
  public function get contacts() : ArrayCollection
  {
   return _contacts;
  }
 }
}
...
```

La classe *ContactVO* est une représentation de notre contact :

```
//service/value/vo/ContactVO.as
package org.servebox.sample.contactapplication.service.value.vo
{
 import org.servebox.foundry.value.AbstractValueObject;
 [Bindable]
 [RemoteClass(alias="org.servebox.sample.contactapplication.service.value.vo.ContactVO")]
 public class ContactVO extends AbstractValueObject implements IContactVO
 {
  private var _firstName : String;
  private var _lastName : String;
...
```

Dernière étape, il faut gérer le résultat renvoyé par le service dans ContactListResponder :

```
//business/responder/ContactListResponder.as
...
public function result(data:Object):void
{
 var contactTO : ContactTO = ResultEvent( data ).result as ContactTO;
 MainController.getInstance().getContactModel().setContacts( contactTO.contacts );
}
...
```

Lors de l'enregistrement de la liste de contact dans le modèle, une notification est envoyée à tous les « observers » et les vues sont mis à jour. En cliquant sur le bouton pour récupérer les contacts, vous voyez désormais votre liste de contacts.

7-5 - Consultation des informations d'un contact

Lors de la sélection d'un contact dans la « list », il faut afficher l'ensemble des données de ce contact dans le formulaire. On crée un « binding » qui nous renvoie le contact sélectionné dans la « list » :

```
//helper/MainViewHelper.as
...
/**
* Au déclenchement de l'événement "selectedContactChange",
* récupère l'item sélectionné en tant que ContactVO
* */
[Bindable("selectedContactChange")]
public function get selectedContact():ContactVO
{
 return ContactVO( getCurrentView().contactList.selectedItem );
}
...
```

L'événement selectedContactChange se déclenche lors du changement de l'élément sélectionné dans la « list » :

```
//view/MainView.as
...
<mx:List
 styleName="listBox"
 width="300"
 height="648"
 id="contactList"
 rowHeight="{Math.floor( contactList.height / 9 )}"
 dataProvider="{helper.getContactList}"
 itemRenderer="{new ClassFactory( ContactRenderer )}"
 change="helper.dispatchEvent(new Event('selectedContactChange'))">
...
```

Dans la prochaine étape, nous mettons à jour *MainView* : on alimente la propriété source aux composants « SmartForm » et on appelle le getter *selectedContact*.

```
//view/MainView.as
...
<form:SmartForm
 id="leftSmartForm"
 styleName="form"
 width="50%"
 source="{helper.selectedContact}">
...
<form:SmartForm
 id="rightSmartForm"
 styleName="form"
 width="50%"
 source="{helper.selectedContact}">
...
<form:SmartForm
 id="bottomSmartForm"
 styleName="form"
 width="50%"
 source="{helper.selectedContact}">
...
```

En utilisant le SmartForm, la propriété ID de chaque élément du formulaire est liée aux propriétés de l'objet récupéré par la méthode selectedContact. Le SmartForm AS Foundry permet de faciliter l'entrée et l'envoi de données via les formulaires. Le smartForm utilise des copies d'objets plutôt que des références, ce qui permet par exemple de proposer à l'utilisateur d'annuler une modification. Le smartForm permet de passer facilement d'un mode écriture à un mode lecture.

Pour avoir plus d'information sur le SmartForm, lisez cet article **"Using SmartForm"**.

Si vous avez bien suivi tout le tutoriel, vous devriez avoir le même code source que l'archive **suivante** (format zip, 1 052 ko)

Dans la prochaine partie de ce tutoriel, nous verrons comment ajouter / modifier / supprimer des contacts et nous mettrons en place le moteur de recherche.