
Flex 3: Introducing Cairngorm

August 2008

Do Not Copy

Trademarks

Adobe, the Adobe logo, Acrobat, Acrobat Reader, Adobe Type Manager, ATM, XMP, Display PostScript, Distiller, Exchange, Frame, FrameMaker, and PostScript" are trademarks of Adobe Systems Incorporated, 1 Step RoboPDF, ActiveEdit, ActiveTest, Authorware, Blue Sky Software, Blue Sky, Breeze, Breezo, Captivate, Central, ColdFusion, Contribute, Database Explorer, Director, Dreamweaver, Fireworks, Flash, FlashCast, FlashHelp, Flash Lite, FlashPaper, Flex, Flex Builder, Fontographer, FreeHand, Generator, HomeSite, JRun, MacRecorder, Adobe, MXML, RoboEngine, RoboHelp, RoboInfo, RoboPDF, Roundtrip, Roundtrip HTML, Shockwave, SoundEdit, Studio MX, UltraDev, and WebHelp are either registered trademarks or trademarks of Adobe Systems Incorporated and may be registered in the United States or in other jurisdictions including internationally. Other product names, logos, designs, titles, words, or phrases mentioned within this publication may be trademarks, service marks, or trade names of Adobe Systems Incorporated or other entities and may be registered in certain jurisdictions including internationally.

Third-Party Information

This guide contains links to third-party websites that are not under the control of Adobe, and Adobe is not responsible for the content on any linked site. If you access a third-party website mentioned in this guide, then you do so at your own risk. Adobe provides these links only as a convenience, and the inclusion of the link does not imply that Adobe endorses or accepts any responsibility for the content on those third-party sites.

Copyright © 1997-2008 Adobe Systems Incorporated

All rights reserved.

The software described in this manual is provided by Adobe Systems Incorporated under a Adobe Systems Incorporated agreement. The software may be used only in accordance with the terms of the agreement. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, photocopying, manual, optical, recording, or otherwise, outside the license agreement accompanying these materials, without the prior written permission of Adobe Systems Incorporated

Adobe Systems Incorporated claims copyright in this program and documentation as an unpublished work, revisions of which were first licensed on the date indicated in the foregoing notice. Claim of copyright does not imply waiver of other rights of Adobe Systems Incorporated and its subsidiaries.

Information in this manual may change without notice and does not represent a commitment on the part of Adobe Systems Incorporated

NOTICE OF LIABILITY

The information in these training materials is distributed on an "AS IS" basis, without warranty of any kind, either express or implied. While every precaution has been taken in the preparation of these materials, neither Adobe Systems Incorporated nor its licensors shall have any liability to any person or entity with respect to liability, loss, or damage caused or alleged to be caused directly or indirectly by the instructions contained in these materials or by the computer software and hardware products described herein.

Third Edition: Aug 2008, Second Edition: June 2008, First Edition: March 2008

Adobe Systems Incorporated
345 Park Avenue
San Jose, CA 95110-2704
USA

Flex 3: Introducing Cairngorm

Introducing Cairngorm

Introducing Cairngorm.....	2
Understanding the purpose of Cairngorm	2
Reviewing the benefits of using Cairngorm	3
Learning about the components of Cairngorm	3
Using Cairngorm in Flex RIAs	4
Identifying roles of code	4
Refactoring code to Cairngorm layers and classes	5
Reviewing benefits of Cairngorm refactoring.....	7
Walkthrough 1: Using Cairngorm in a project	8
Implementing the ModelLocator.....	9
Example ModelLocator code.....	9
Using the ModelLocator	10
Walkthrough 2: Building the ModelLocator	11
Implementing the ServiceLocator	13
Example ServiceLocator code	13
Using the ServiceLocator	13
Walkthrough 3: Building the ServiceLocator.....	14
Implementing Cairngorm events.....	16
Introducing the concept of Business Events.....	16
Example Cairngorm event code.....	16
Using a Cairngorm event	17
Walkthrough 4: Building Cairngorm events	18
Implementing Commands.....	20
Example Command code.....	20
Using the Command	21
Walkthrough 5: Building Cairngorm Commands.....	22
Implementing the FrontController	27
Example FrontController code.....	27
Using the FrontController	28
Walkthrough 6: Building the FrontController	29
Implementing Delegates	31
Understanding the role of a Delegate	31
Implementing a Delegate	32
Implementing a Responder	32
Example Delegate code	32
Walkthrough 7: Building a Cairngorm Delegate	34
Using the Cairngorm components	36
Walkthrough 8: Modifying FStop to use Cairngorm	37

Lab:

Tasks	42
Create a new Cairngorm MVC Project.....	43
Create a form to request a patient visitation	45
Submit the Visitation Request	47
Create the Business Component to Process the Visitation Details	49

Do Not Copy

Introducing Cairngorm

In this module you will preview the use of Cairngorm for Flex. You will be introduced to concepts and components of Cairngorm; without the accompanying in-depth theory and details. You will also convert an existing Flex application to use Cairngorm.

Objectives

After completing this unit, you should be able to:

- Understand why Cairngorm is used in Flex development.
- Understand what are the primary components used in Cairngorm.
- Understand the use of custom events in Cairngorm.
- Understand the impacts of using Cairngorm on Flex applications.

Topics

In this unit, you will learn about the following topics:

- Introducing Cairngorm
- Using Cairngorm in Flex RIAs
- Implementing the ModelLocator
- Implementing the ServiceLocator
- Implementing Cairngorm events
- Implementing Commands
- Implementing the FrontController
- Implementing Delegates
- Using the Cairngorm components

Introducing Cairngorm



- Understanding Cairngorm purposes
- Review the benefits of using Cairngorm
- Learn about the components of Cairngorm

Understanding the purpose of Cairngorm

- Cairngorm is an approach to organizing and partitioning
 - code and packages
 - component functionality and roles
- Cairngorm is a “best-practice” methodology for Flex software design and development
- Cairngorm encourages developers to identify, organize, and separate code based on its roles/responsibilities.

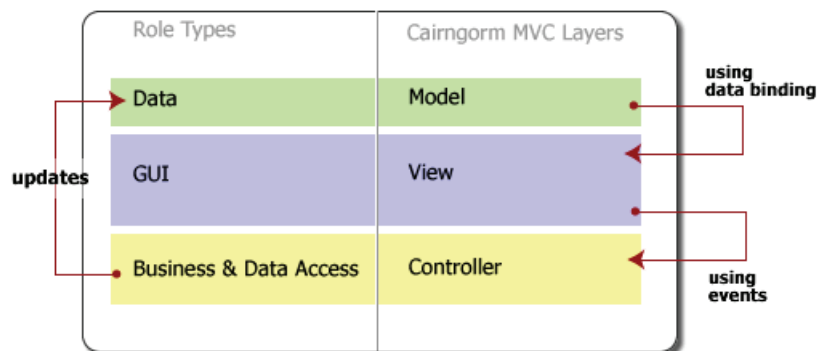


Figure 1: Mapping Functionality Roles to Cairngorm Layers

- Cairngorm encourages separation of concerns
 - Model holds data objects and the state of the data
 - Controller process business logic
 - Views render data and announce gestures with events
 - Views communicate with Controller using events
 - Views watch Model with data bindings
 - Views are graphical user interfaces or visual portions of the Flex application.
 - Views usually are composites of UI controls or other views.
 - Views can contain child views
 - Even `<mx:Application />` is a view

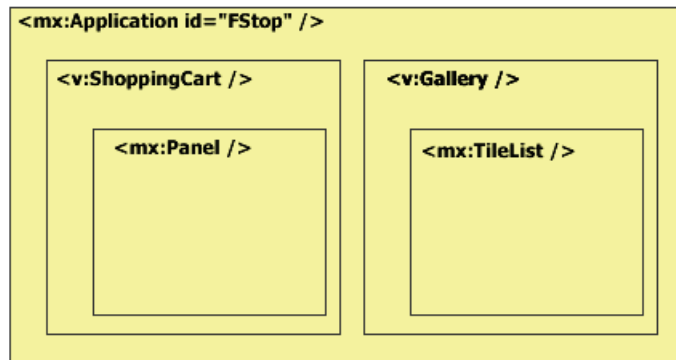


Figure 2: View Nestings within the FStop application

Reviewing the benefits of using Cairngorm

- Cairngorm allows designers, component developers, and data-services developers to work in parallel
- Cairngorm is best suited for medium to large-size applications
- Cairngorm is ideal for team development
- Cairngorm provides the following benefits
 - maintenance is easier
 - debugging is easier
 - feature additions and/or changes are easier
 - automated testing of business logic and data access is easier
 - using mock data-services is easier

Learning about the components of Cairngorm

- Cairngorm has 5 primary components that are used with Flex RIA solutions
 - ModelLocator: a repository for global data and global access
 - Services: a repository of pre-configured RDS components
 - Commands: non-UI components that process business logic
 - Events: custom events that trigger the business objects [Commands] to start processing
 - Controller: component needed to route business events to commands for processing.

Using Cairngorm in Flex RIAs



- Identify Roles of Code
- Refactor Code to Cairngorm MVC Layers
- Review Benefits of Cairngorm Refactoring

Identifying roles of code

- Within the source code for a Flex application identify code that has a role- or is responsible for data, GUI, business, or data services.

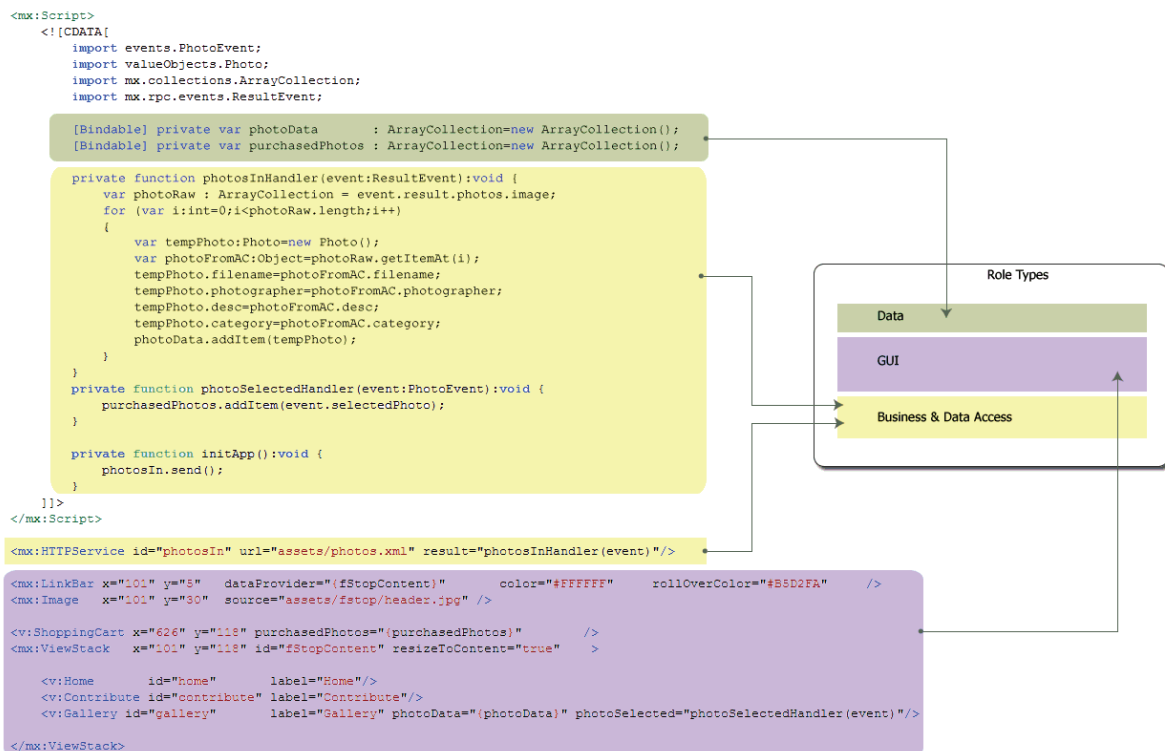


Figure 3: Identifying Roles of Code and Color by Stereotype

- Using the example illustration [from the FStop application] above, let's identify specific code “roles”:
 - Inside `<mx:Script />`
 - [Bindable] public var declarations are part of “Data” roles
 - The remaining is for either business or data access.
 - After `<mx:Script />`
 - These are controls and components instantiated using MXML tag notation.
 - `<mx:HTTPService />` is a non-UI controls in the “Data Access” role

- All other <mx: /> tags are view components associated with the “GUI” role
- <v: /> tags are custom view components with “GUI” roles

Refactoring code to Cairngorm layers and classes

- Refactoring code to use Cairngorm can be achieved by

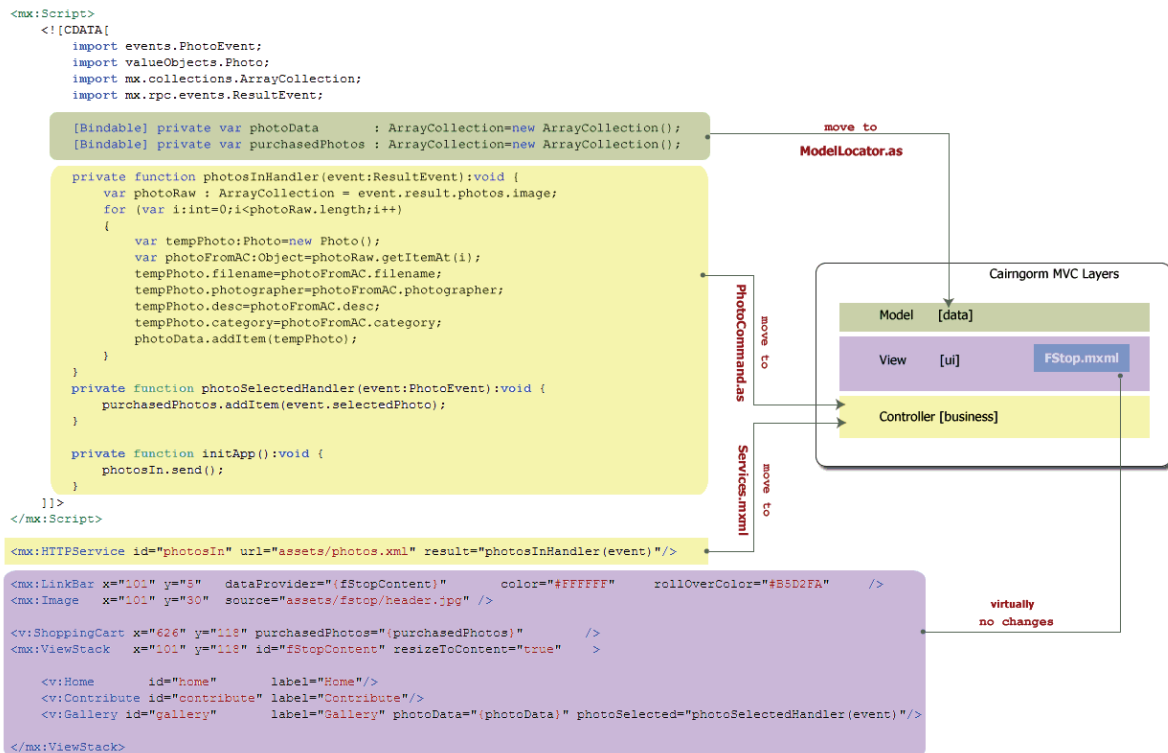
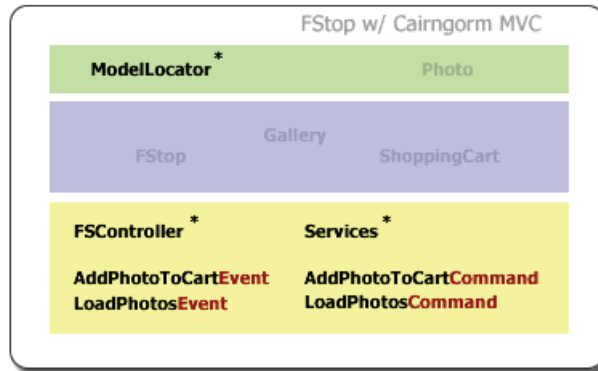


Figure 4: Mapping code to Cairngorm MVC Layers and Components

Tip: While the imports statements in the code above have not been highlighted, the imports will be modified accordingly to support code restructuring.

1. Isolate the “ui” tags in the FStop application file in the View layer
2. Move the “data” code to the ModelLocator in the Model layer
3. Create custom [business] events to communicate from the View layer to the Controller [business] layer.
 - AddPhotosToCartEvent
 - LoadPhotosEvent

4. Move the “business” code to the Controller layer
 - Business logic is moved to the `LoadPhotosCommand`
 - Business logic is moved to the `AddPhotoToCartCommand`
 - RDS data access is moved to the `Services` repository
5. Create `FSController` component to route business events to `PhotoCommand`



* Shows classes that will only EVER have one instance created.

Figure 5: Identifying New Classes Required for Cairngorm Refactoring

6. Update *View* components to databind to `ModelLocator` data
7. Update *View* components to dispatch business events.
8. Update *Application* to create the `FSController`, `Services` repository, and `ModelLocator` instances.

Note: Only one, unique instance of the `FSController`, `Services`, and `ModelLocator` are created for the application. In contrast, each business event will have its own business command class... this is a 1-to-1 mapping of business events to business commands.

Reviewing benefits of Cairngorm refactoring

- View code is concise and understandable and “renders” only.
- View classes do not know anything about Controller [Business] components; except through the use of business events
- *Model* data is stored and accessed from the ModelLocator.
- All ModelLocator variables support data binding.
- Business components do not know anything about the *View* classes.
- *View* components use data binding to render data cached in the ModelLocator [*Model*] layer

```
<mx:Script>
  <![CDATA[
    private function initApp():void {
      var event : LoadPhotosEvent = new LoadPhotosEvent();
      event.dispatch();
    }

    private function photoSelectedHandler(event:PhotoEvent):void {
      var request : AddPhotoToCartEvent = new AddPhotoToCartEvent(event.selectedPhoto);
      request.dispatch();
    }

    [Bindable] private var __model : ModelLocator = ModelLocator.getInstance();
  ]]>
</mx:Script>

<rds:Services          xmlns:rds="business.*" />
<router:FSController  xmlns:rds="business.*" />

<mx:LinkBar x="101" y="5"   dataProvider="{fStopContent}"   color="#FFFFFF"   rollOverColor="#B5D2FA" />
<mx:Image   x="101" y="30"  source="assets/fstop/header.jpg" />

<v:ShoppingCart x="626" y="118" purchasedPhotos="{__model.purchasedPhotos}" />
<mx:ViewStack x="101" y="118" id="fStopContent" resizeMode="true" >

  <v:Home      id="home"      label="Home"/>
  <v:Contribute id="contribute" label="Contribute"/>
  <v:Gallery id="gallery"      label="Gallery" photoData="{__model.photoData}" photoSelected="photoSelectedHandler(event)"/>

</mx:ViewStack>
```

Figure 6: Benefits Code Simplicity when using Cairngorm MVC

Walkthrough 1: Using Cairngorm in a project



In this walkthrough, you will perform the following tasks:

- Add the Cairngorm library to the FStop project

Steps

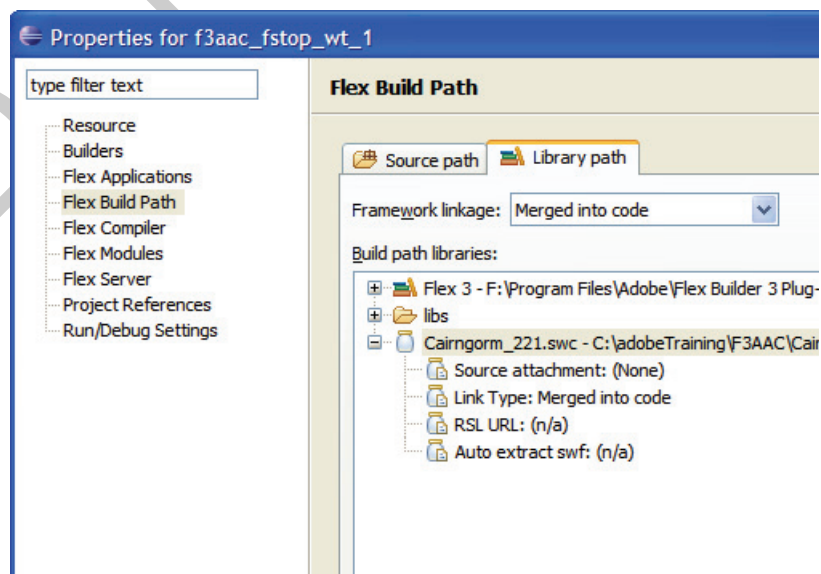
Set up the working environment

1. Unzip the file **f3ic_studentFiles_16Jun08.zip** to disk. It will automatically create an **f3ic** folder with supplied files in it.
2. Open Flex Builder.
3. Select **File > Import > Flex Project...**
4. Browse to `{installationLocation}/f3ic` and select the archive **f3ic_fstop.zip**.
5. Click **Finish**.

Note: In the same folder there are staged solutions archived for each each walkthrough . Simply import the archive to see the solution.

Add the Cairngorm library to the FStop project

6. Click the project just opened, then select **Project > Properties > Flex Build Path > Library Path**.
7. Select the **Add SWC...** button and add the Cairngorm library located at `{installationLocation}/f3ic/Cairngorm.swc`.



Implementing the ModelLocator



- The ModelLocator is a global singleton repository for shared or global data
 - Singleton means that only instance of the class should exist
- Does not load or persist data to a permanent data store
- Does not contain business logic
- Serves as caching and access location only
- Caches custom variables/data specific to application
- Supports data binding for auto-notifications of data changes to views

Example ModelLocator code

- Be sure the class implements the `getInstance()` method
- Also contains public variables that hold application data
 - In example code variables `photoData` and `purchasedPhotos`

```
package model
{
    import mx.collections.ArrayCollection;

    [Bindable]
    public class ModelLocator
    {
        static private var __instance:ModelLocator=null;
        public var photoData:ArrayCollection=new ArrayCollection();
        public var purchasedPhotos:ArrayCollection=new ArrayCollection();

        static public function getInstance():ModelLocator
        {
            if(__instance == null)
            {
                __instance=new ModelLocator();
            }
            return __instance;
        }
    }
}
```

Using the ModelLocator

- Access data stored in the ModelLocator via the `getInstance()` method

```
<![CDATA[
import model.ModelLocator;

private var __model:ModelLocator=ModelLocator.getInstance();
]]>
```

- Access data via the instance name of the ModelLocator

```
<mx:DataGrid dataProvider=__model.photoData/>
```

Do Not Copy

Walkthrough 2: Building the ModelLocator



In this walkthrough, you will perform the following tasks:

- Create the ModelLocator repository

Steps

1. In the FB Navigator view, open the `src` directory.

Create the ModelLocator Repository

2. Create a new package called `model`.
3. Create an ActionScript class called `ModelLocator.as`.
 - Uncheck the “Generate Constructor from super class” checkbox.
4. Create a `static, private` variable named `__instance`, data typed as `ModelLocator`, and set it equal to `null`.

```
static private var __instance:ModelLocator=null;
```

5. Create a global accessor function to allow only one ModelLocator instance to be accessed from anywhere. This function should be called `getInstance()`.

```
static public function getInstance():ModelLocator
{
    if (__instance == null)
    {
        __instance=new ModelLocator();
    }
    return __instance;
}
```

6. Open `FStop.mxml` application file.
7. **Copy** the business variables declared for `photoData` and `purchasedPhotos` model data [see lines 17-18].
8. Return to the `ModelLocator.as` class.
9. At the top of the class paste the copied code into the `ModelLocator`.
10. Modify the variables declarations to be `public`.
11. Remove the `[Bindable]` tags on each variable.
12. Make the `ModelLocator` class `[Bindable]`.
13. Since you copied in the variable declarations, you need to import the `ArrayCollection` class.

*Note: An easy way to do the import is simply place the cursor after the letter **n** from one of the new `ArrayCollection()` statements, then press **Ctrl-Space**.*

14. Be sure the ModelLocator appears as follows.

```
package model
{
    import mx.collections.ArrayCollection;

    [Bindable]
    public class ModelLocator
    {
        public var photoData:ArrayCollection=new ArrayCollection();
        public var purchasedPhotos:ArrayCollection=new ArrayCollection();
        static private var __instance:ModelLocator=null;

        static public function getInstance():ModelLocator
        {
            if(__instance == null)
            {
                __instance=new ModelLocator();
            }
            return __instance;
        }
    }
}
```

15. Save the file.

16. Close the ModelLocator file.

Implementing the ServiceLocator



- The ServiceLocator pattern is used to create a global, singleton registry to centralize all instances of Flex RDS components used in an application
 - HTTPService
 - WebService
 - RemoteObject
 - Custom RDS classes
- Supports easy lookup of services by name
- Should never be used outside the Control layer

Example ServiceLocator code

- Extends the Cairngorm ServiceLocator class
- Contains RDS tag instantiations

```
<?xml version="1.0" encoding="utf-8"?>
<rds:ServiceLocator xmlns:rds="com.adobe.cairngorm.business.*"
  xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:HTTPService id="photosIn" url="assets/photos.xml"/>

</rds:ServiceLocator>
```

Using the ServiceLocator

- Instantiate the object in MXML to configure and cache the services repository

```
<rds:Services xmlns:rds="business.*"/>
```

- Use the ServiceLocator by calling the getInstance() method and then use a defined service

```
private var __locator:ServiceLocator=ServiceLocator.getInstance();
var service:HTTPService=__locator.getHTTPService("photosIn");
```

Walkthrough 3: Building the ServiceLocator



In this walkthrough, you will perform the following tasks:

- Create the Services repository

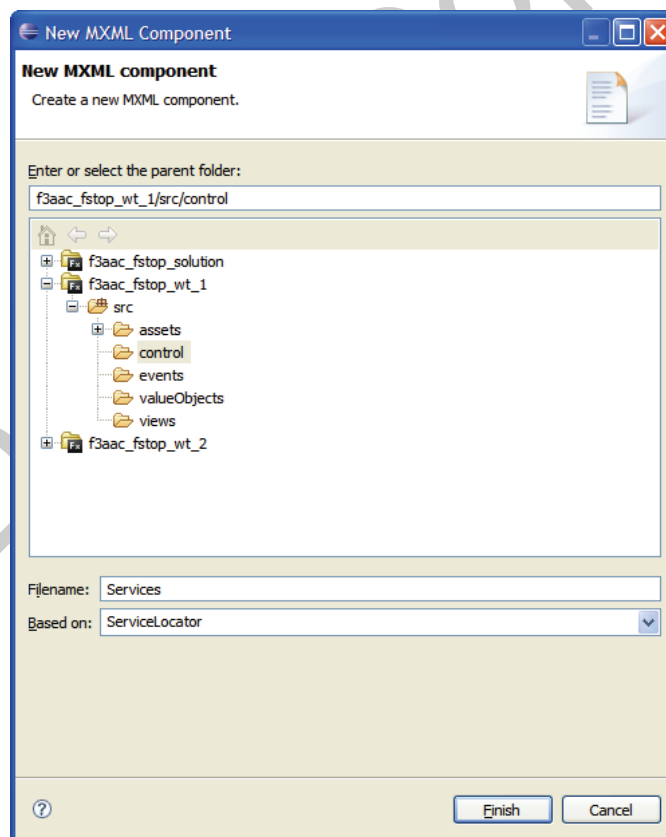
Steps

1. In the Navigator view, select the `src` directory.

Create the Services repository

2. Create a new package called `business`.
3. In the `business` package, create an **MXML Component** named `Services`. This class will extend `ServiceLocator`.

Note: The `ServiceLocator` option will not appear in the drop down list. Simply type it in.



-
4. Modify the default namespace entered, `xmlns="*"`, to properly import `com.adobe.cairngorm.business.*`. Use the *rds* prefix.

```
<rds:ServiceLocator xmlns:rds="com.adobe.cairngorm.business.*"
  xmlns:mx="http://www.adobe.com/2006/mxml">
```

```
</rds:ServiceLocator>
```

5. Open **FStop.mxml** application file.
6. **Copy** the tag declaration for the `HTTPService` [see line 46].
7. Paste the tag into the `Services.mxml` class. Paste the code as a child tag.
8. Remove the `result` event handler defined in the `HTTPService` tag.

```
<mx:HTTPService id="photosIn" url="assets/photos.xml" />
```

9. Check to be sure your `ServiceLocator` appears as follows.

```
<?xml version="1.0" encoding="utf-8"?>
<rds:ServiceLocator xmlns:rds="com.adobe.cairngorm.business.*"
  xmlns:mx="http://www.adobe.com/2006/mxml">
```

```
<mx:HTTPService id="photosIn" url="assets/photos.xml"/>
```

```
</rds:ServiceLocator>
```

10. Save and close the file.

Implementing Cairngorm events



- Many events take place in Flex applications, but Cairngorm is only concerned with Business Events

Introducing the concept of Business Events

- Flex uses an Event Delegation Model
 - A system where the *responses* and *reactions* to activity in one component are **delegated** to one or more different components
 - This delegation is achieved by dispatching events
- Flex has many categories of events
 - System events: initialize, creationComplete, showEffect
 - Framework event: click, mouseMove, keyDown
 - User events: hideContactDetails, showCatalog
 - Business events: GetAllPurchasedItems, LoadCatalog, SubmitOrder, LoginUser
 - Data Service events: ResultEvent, FaultEvent
- Custom events can be used to transport or package data to the recipient(s)
- User events are custom events that are delegated to other components in the View layer
- Business events are
 - Custom events delegated to components in the business [Control] layer
 - Used to announce gestures and activate Command processing

Example Cairngorm event code

- Each event should have a unique ID that is exposed statically and passed to the `super` constructor
- Simple event

```
package business.events
{
    import com.adobe.cairngorm.control.CairngormEvent;
    public class LoadPhotosEvent extends CairngormEvent
    {
        static public var EVENT_ID:String="loadPhotos";
        public function LoadPhotosEvent()
        {
            super(EVENT_ID);
        }
    }
}
```

-
- Event that uses a value object

```
package business.events
{
    import com.adobe.cairngorm.control.CairngormEvent;
    import valueObjects.Photo;

    public class AddPhotoToCartEvent extends CairngormEvent
    {
        static public var EVENT_ID:String="addPhotoEvent";
        public var photo:Photo=null;

        public function AddPhotoToCartEvent(photo:Photo)
        {
            super(EVENT_ID);
            this.photo=photo;
        }
    }
}
```

Using a Cairngorm event

- The View layer can create and dispatch Business Events
- The Control layer can create and dispatch Business Events
- Only the Control layer manages and processes Business Events
- Dispatching of Business Events should be considered 1-way to the business [Control] layer
- Cairngorm events dispatch themselves to the Controller

```
private function initApp():void
{
    var event:LoadPhotosEvent=new LoadPhotosEvent();
    event.dispatch();
}

private function photoSelectedHandler(event:PhotoEvent):void
{
    var addEvent:AddPhotoToCartEvent=
        new AddPhotoToCartEvent(event.selectedPhoto);
    addEvent.dispatch();
}
```

Walkthrough 4: Building Cairngorm events



In this walkthrough, you will perform the following tasks:

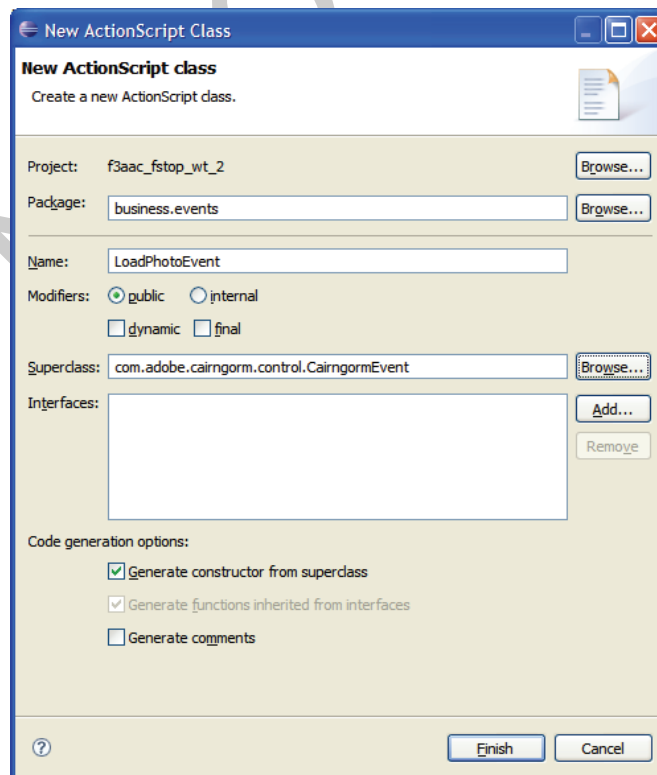
- Create the LoadPhotos and AddPhotoToCart business events

Steps

1. Select the **business** package in the Navigator.

Create the LoadPhotos and AddPhotoToCart Business Events

2. Create a new child package called *events*.
3. In the **events** package, create a new ActionScript class with the following properties
 - Filename is *LoadPhotosEvent*
 - Extends the class `com.adobe.cairngorm.control.CairngormEvent`



4. Modify the **LoadPhotosEvent** class so it conforms to a simple Cairngorm event.

```
package business.events
{
    import com.adobe.cairngorm.control.CairngormEvent;
    public class LoadPhotosEvent extends CairngormEvent
    {
        static public var EVENT_ID:String="loadPhotos";

        public function LoadPhotosEvent()
        {
            super(EVENT_ID);
        }
    }
}
```

5. Save the file.

6. Create another event ActionScript class with the following properties

- Filename is *AddPhotoToCartEvent*
- Extends the class *com.adobe.cairngorm.control.CairngormEvent*
- Exposes an *EVENT_ID* variable of *addPhotoToCart*
- Requires a *Photo* value object as a constructor argument

7. Modify the class so it conforms to a Cairngorm event.

```
package business.events
{
    import com.adobe.cairngorm.control.CairngormEvent;
    import valueObjects.Photo;

    public class AddPhotoToCartEvent extends CairngormEvent
    {
        static public var EVENT_ID:String="addPhotoToCart";
        public var photo:Photo=null;

        public function AddPhotoToCartEvent(photo:Photo)
        {
            super(EVENT_ID);
            this.photo=photo;
        }
    }
}
```

8. Save the file.

Implementing Commands



- The Control layer relies on Commands to manage business logic and respond to Business Events
- Command features
 - Each Command class represents a specific business feature with associated business logic and processing
 - Commands update the ModelLocator with new data or changes to existing data
 - Each Command class represents a specific business feature with associated business logic and processing
- All Commands have the *same* entry point to initiate or start business processing: `execute()`
- Command implementations have class names that are equivalent to Business Events
 - `LoadPhotosEvent` and `LoadPhotosCommand`

Example Command code

- Must contain an `execute()` method
- That method contains business logic

```
package business.commands
{
import business.events.AddPhotoToCartEvent;
import com.adobe.cairngorm.commands.ICommand;
import com.adobe.cairngorm.control.CairngormEvent;
import model.ModelLocator;

public class AddPhotoToCartCommand implements ICommand
{
private var __model:ModelLocator=ModelLocator.getInstance();
public function execute(event:CairngormEvent):void
{
__model.purchasedPhotos.
addItem((event as AddPhotoToCartEvent).photo);
}
}
}
```

Using the Command

- Used in the Controller where the events are listened for and corresponding commands are called
 - Controller built in the next section

```
addCommand(LoadPhotosEvent.EVENT_ID, LoadPhotosCommand);  
addCommand(AddPhotoToCartEvent.EVENT_ID, AddPhotoToCartCommand);
```

Do Not Copy

Walkthrough 5: Building Cairngorm Commands



In this walkthrough, you will perform the following tasks:

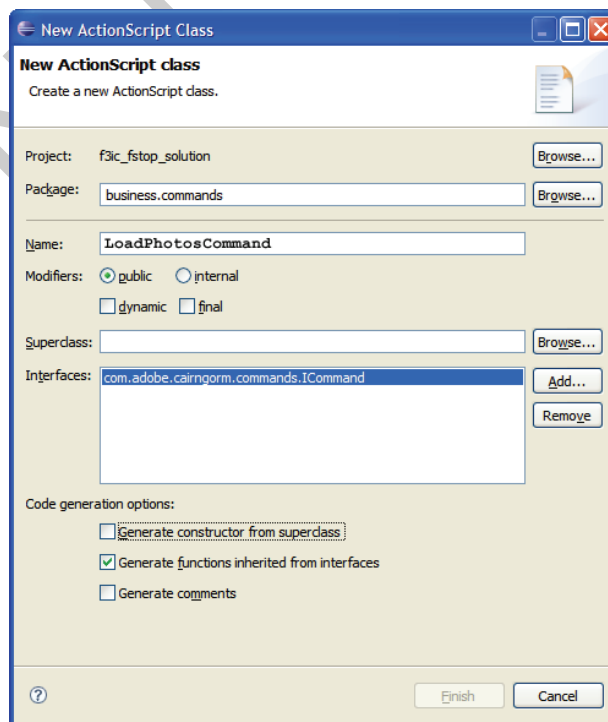
- Create the LoadPhotosCommand business component
- Create the AddPhotoToCart command

Steps

1. Select the **business** package in the Navigator.

Create the LoadPhotosCommand business component

2. Create a new child package called *commands*.
3. In the **commands** package, create a new ActionScript class with the following properties
 - Filename is *LoadPhotosCommand*
 - Implements the interface *com.adobe.cairngorm.commands.ICommand*
 - Uncheck the **Generate constructor from superclass** option



Note: This command will be using the Services repository for RDS and the ModelLocator to store the load photo data.

Create instances of the ModelLocator and ServiceLocator

4. In the class, create two alias references to the ModelLocator and ServiceLocator instances using the respective getInstance() methods.

```
private var __model:ModelLocator=ModelLocator.getInstance();
private var __locator:ServiceLocator=ServiceLocator.getInstance();
```

Note: If using tag help choose the ModelLocator class you have built, not the Cairngorm ModelLocator interface.

Modify the execute method

5. In the execute() function use the ServiceLocator to lookup a reference to the HTTPService instance called *photosIn*.

```
var service:HTTPService=__locator.getHTTPService("photosIn");
```

Note: If using tag help choose the mx.rpc.http.HTTPService class.

6. Manually specify the result handler for the asynchronous HTTPService call. Use addEventListener to add a result event handler to the RDS component instance.

```
service.addEventListener(ResultEvent.RESULT,onResults_loadPhotos);
```

Note: We are not adding a listener for the FaultEvent, because this FStop application will ignore any errors. Release-quality solutions must anticipate possible failures in the RDS calls and must plan for Fault event handlers.

7. Invoke the RDS call to the server to load the photo data for the FStop application.

```
service.send();
```

Create the result handler

8. Create another function in the class that is the skeleton for the result handler. Name the function *onResults_loadPhotos()* and datatype the function itself as void. The function should accept a parameter named *event*, data typed as ResultEvent

```
private function onResults_loadPhotos(event:ResultEvent):void
{
}
}
```

9. Open **FStop.mxml** application file.

10. **Copy** the method body of the `photosInHandler()` function, lines 22-32.

11. In **LoadPhotosCommands.as** paste the copied code into the method body for the `onResults_loadPhotos()` function.

12. Modify the `photoData.addItem()` call in the for loop to use the `photoData` reference cached in the `ModelLocator` repository.

```
__model.photoData.addItem(tempPhoto);
```

13. Be sure your function appears as follows.

```
private function onResults_loadPhotos(event:ResultEvent):void
{
    var photoRaw:ArrayCollection=event.result.photos.image;
    for(var i:int=0;i<photoRaw.length;i++)
    {
        var tempPhoto:Photo=new Photo();
        var photoFromAC:Object=photoRaw.getItemAt(i);
        tempPhoto.filename=photoFromAC.filename;
        tempPhoto.photographer=photoFromAC.photographer;
        tempPhoto.desc=photoFromAC.desc;
        tempPhoto.category=photoFromAC.category;
        __model.photoData.addItem(tempPhoto);
    }
}
```

14. Be sure the `ArrayCollection` and `Photo` classes are imported to support the copied code.

Note: The cursor at the end of class name/Ctrl-Space trick helps here.

15. Check to be sure your class appears as follows.

```
package business.commands
{
    import com.adobe.cairngorm.business.ServiceLocator;
    import com.adobe.cairngorm.commands.ICommand;
    import com.adobe.cairngorm.control.CairngormEvent;
    import model.ModelLocator;
    import mx.collections.ArrayCollection;
    import mx.rpc.events.ResultEvent;
    import mx.rpc.http.HTTPService;
    import valueObjects.Photo;

    public class LoadPhotosCommand implements ICommand
    {
        private var __model:ModelLocator=ModelLocator.getInstance();
        private var __locator:ServiceLocator=
            ServiceLocator.getInstance();

        public function execute(event:CairngormEvent):void
        {
            var service:HTTPService=__locator.getHTTPService("photosIn");
            service.addEventListener(ResultEvent.RESULT,
                onResults_loadPhotos);
            service.send();
        }
        private function onResults_loadPhotos(event:ResultEvent):void
        {
            var photoRaw:ArrayCollection=event.result.photos.image;
            for(var i:int=0;i<photoRaw.length;i++)
            {
                var tempPhoto:Photo=new Photo();
                var photoFromAC:Object=photoRaw.getItemAt(i);
                tempPhoto.filename=photoFromAC.filename;
                tempPhoto.photographer=photoFromAC.photographer;
                tempPhoto.desc=photoFromAC.desc;
                tempPhoto.category=photoFromAC.category;
                __model.photoData.addItem(tempPhoto);
            }
        }
    }
}
```

16. Save the file.

Create the AddPhotoToCartCommand

17. Create a new ActionScript class in the **commands** package with the following properties

- Filename is *AddPhotoToCartCommand*
- Implements the interface *com.adobe.cairngorm.commands.ICommand*
- Uncheck the **Generate constructor from superclass** option

Note: This command will be using the ModelLocator instance to add the photo to the photo purchase list.

18. In the class, create an alias reference to the ModelLocator instance using the getInstance() method.

```
private var __model:ModelLocator=ModelLocator.getInstance();
```

19. Modify the execute() function to add the photo to the purchase list

- Cast the event to an instance of AddPhotoToCartEvent

```
var photo:Photo=(event as AddPhotoToCartEvent).photo;  
__model.purchasedPhotos.addItem(photo);
```

20. Be sure class appears as follows.

```
package business.commands  
{  
    import business.events.AddPhotoToCartEvent;  
    import com.adobe.cairngorm.commands.ICommand;  
    import com.adobe.cairngorm.control.CairngormEvent;  
    import model.ModelLocator;  
    import valueObjects.Photo;  
  
    public class AddPhotoToCartCommand implements ICommand  
    {  
        private var __model:ModelLocator=ModelLocator.getInstance();  
  
        public function execute(event:CairngormEvent):void  
        {  
            var photo:Photo=(event as AddPhotoToCartEvent).photo;  
            __model.purchasedPhotos.addItem(photo);  
        }  
    }  
}
```

21. Save the file.

Implementing the FrontController



- The FrontController
 - Intercepts dispatched business events and forwards each event instance to the appropriate Command instance for processing
 - Is a registry of event-to-command mappings
 - Is the “router” for business events
- The FrontController pattern provides a solution that allows the View and Control layers to connect using event delegation
 - Views dispatch business events to the business layer
 - Views are never aware of the FrontController, Commands, or Delegates, or the Services repository
 - Views simply render data stored in the Model layer and announce user gestures
 - Dispatched business events are routed to appropriate business components

Example FrontController code

- In the FrontController’s constructor use the `addCommand()` method to link Business Events and corresponding Commands

```
package business
{
import business.commands.AddPhotoToCartCommand;
import business.commands.LoadPhotosCommand;
import business.events.AddPhotoToCartEvent;
import business.events.LoadPhotosEvent;
import com.adobe.cairngorm.control.FrontController;

public class FSController extends FrontController
{
public function FSController()
{
super();
addCommand(LoadPhotosEvent.EVENT_ID, LoadPhotosCommand);
addCommand(AddPhotoToCartEvent.EVENT_ID, AddPhotoToCartCommand);
}
}
}
```

Using the FrontController

- When Cairngorm events are dispatched they are automatically handled by the FrontController

```
private function initApp():void
{
    var event:LoadPhotosEvent=new LoadPhotosEvent();
    event.dispatch();
}
```

Do Not Copy

Walkthrough 6: Building the FrontController



In this walkthrough, you will perform the following tasks:

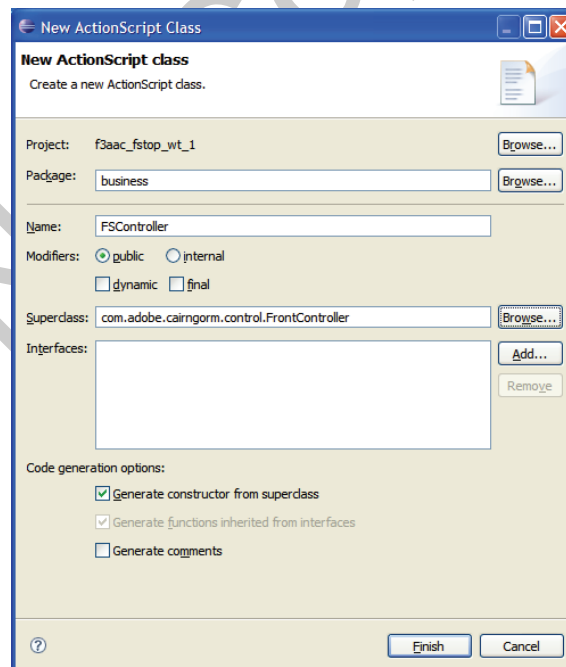
- Create the FSController to route the business events

Steps

1. Select the **business** package in the Navigator.

Create the FSController to route the business events

2. In the **business** package create a new ActionScript class with the following properties
 - Filename is *FSController*
 - Extends the class *com.adobe.cairngorm.control.FrontController*
 - Select the **Generate constructor from superclass** checkbox



3. Use the `addCommand()` method to register each business event with its corresponding command.

```
addCommand(LoadPhotosEvent.EVENT_ID, LoadPhotosCommand);  
addCommand(AddPhotoToCartEvent.EVENT_ID, AddPhotoToCartCommand);
```

4. Import the classes used.

```
import business.commands.AddPhotoToCartCommand;
import business.commands.LoadPhotosCommand;
import business.events.AddPhotoToCartEvent;
import business.events.LoadPhotosEvent;
```

5. Be sure the **FSController** appears as follows.

```
package business
{
    import business.commands.AddPhotoToCartCommand;
    import business.commands.LoadPhotosCommand;
    import business.events.AddPhotoToCartEvent;
    import business.events.LoadPhotosEvent;
    import com.adobe.cairngorm.control.FrontController;

    public class FSController extends FrontController
    {
        public function FSController()
        {
            super();
            addCommand(LoadPhotosEvent.EVENT_ID, LoadPhotosCommand);
            addCommand(AddPhotoToCartEvent.EVENT_ID,
                AddPhotoToCartCommand);
        }
    }
}
```

6. Save the file.

Implementing Delegates



- Each Delegate class
 - provides a local proxy for a remote service
 - serves as a contract between client and server development teams
 - may hide underlying implementation details including
 - Client-side service lookups
 - Server API method invocations
 - Formatting of method arguments
 - Data transformations
- Use of a Delegate allows mock services and dummy data to be used while server implementations are resolved
- Delegate classes should not be used outside the Control layer

Understanding the role of a Delegate

- Using Delegate classes hides the details of the Flex client invoking `WebService`, `RemoteObject`, `HTTPService`, or custom RDS components to connect to the server tier or the local shared object
- The Delegate is a client-side proxy for one or more remote server APIs
- Each Delegate class may group API functionality by context
 - `OrderDelegate` may proxy the API for all order functionality such as `addOrder`, `updateOrder`, `cancelOrder` etc.
 - `UserDelegate` may proxy the API for all user functionality such as `login`, `logout`, `registerUser`, `updateUser` etc.

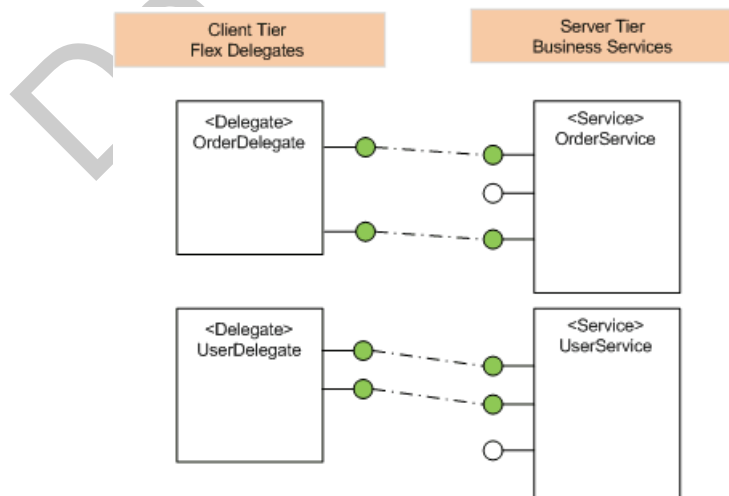


Figure 7: Conceptualization of Delegate API mappings to Server-Tier Business APIs

Note: Delegates are not required to match all the all business service APIs. Only those client-side proxies required should be matched.

Implementing a Delegate

- A Delegate
 - is created and invoked by a corresponding Command
 - defines and configures the Responder for each remote service call
 - is the only class to directly use the ServiceLocator
 - may perform data transformations required by a remote service
 - Outgoing data may be transformed and aggregated
 - Incoming data may be transformed, parsed, and validated
 - may manage queuing and result-routing for multiple server calls
- No specific Delegate interface or parent class is provided by the Cairngorm MVC framework
- Delegates are commonly identified using a Delegate" suffix

Implementing a Responder

- A `mx.rpc.Responder` object defines a method to be called when a remote service call returns a result, and when the remote call fails
- The Responder is either
 - an `mx.rpc.Responder` object, or
 - a class which implements the `mx.rpc.IResponder` interface

```
import mx.rpc.Responder;
...
var responder:Responder = new Responder(onResult, onFault);
private function onResult(event:ResultEvent):void { ... }
private function onFault(event:FaultEvent):void { ... }
```

Example Delegate code

```
package control.delegates
{
    public class PhotoDelegate
    {
        public function PhotoDelegate()
        {
        }
    }
}
```

- Commonly, a Command will define a Responder to specify how results from a remote call, made through a Delegate, are to be handled
- a Delegate constructor may allow the calling Command to specify a Responder, caching it for later use

```
package control.delegates
{
    import mx.rpc.IResponder;
    public class PhotoDelegate
    {
        private var __responder:IResponder;
        public function PhotoDelegate(responder:IResponder)
        {
            __responder = responder;
        }
    }
}
```

- a Delegate will use the ServiceLocator to access the specific RDS instance that will be used to execute the method call

```
private var __locator:ServiceLocator=ServiceLocator.getInstance();
```

- Delegates may implement accessors for specific services
- Delegates may have one or more methods that match the server API

Tip: The names of the methods should be present-tense verb forms such as *loadCatalog*, *updateUser*, *processOrder*, etc.

Walkthrough 7: Building a Cairngorm Delegate



In this walkthrough, you will perform the following tasks:

- Create the `PhotoDelegate` business component
- Modify `LoadPhotosCommand` to use `PhotoDelegate`

Steps

1. Select the **business** package in the Navigator.

Create the `PhotoDelegate` business component

2. Create a new child package called `delegates`.
3. In the **delegates** package, create a new ActionScript class with the following properties
 - Filename is `PhotoDelegate`
 - Leave the **Generate constructor from superclass** option checked
4. From the `commands` package, open `LoadPhotosCommand.as`
5. Move the definition and instantiation of `__service` from `LoadPhotosCommand` into `PhotoDelegate`, along with its required `import` statement.

```
import com.adobe.cairngorm.business.ServiceLocator;
...
private var __locator:ServiceLocator=ServiceLocator.getInstance();
```

6. Declare a private class property named `__service`, typed as `HTTPService`, and import `mx.rpc.http.HTTPService`.

```
import mx.rpc.http.HTTPService;
...
private var __service:HTTPService;
```

7. In the constructor, use the `ServiceLocator` to assign a reference to the `photosIn` service to `__service`.

```
__service = __locator.getHTTPService("photosIn");
```

8. Declare a private class property named `__responder`, typed to the `IResponder` interface, and import `mx.rpc.IResponder`.

```
import mx.rpc.IResponder;
...
private var __responder:IResponder;
```

-
9. Modify the constructor to accept a `responder` parameter, and assign it to `__responder`.

```
public function PhotoDelegate(responder: IResponder)
{
    __service = __locator.getHTTPService("photosIn");
    __responder = responder;
}
```

10. Declare a public method named `loadPhotos`, with a `void` return type, because the responder will be used to define the service result handler.

```
public function loadPhotos():void
{
}
```

11. In `loadPhotos()`, invoke the `send()` method on the `__service` reference, assigning the result to a local variable typed as `AsyncToken`.

```
var token:AsyncToken = __service.send();
```

12. Use the `addResponder()` method to assign `__responder` as the responder for this token's remote service call.

```
token.addResponder(__responder);
```

Modify LoadPhotosCommand to use PhotoDelegate

13. From the `commands` package, open `LoadPhotosCommand.as`.

14. Comment out all code in the `execute()` method.

15. In the `execute()` method, declare and create a local `mx.rpc.Responder` object, passing a reference to the `onResults_loadPhotos` method as its first parameter, and `null` as its second parameter.

```
var responder:Responder = new Responder(onResults_loadPhotos, null);
```

16. Next in this method, declare and create a local `PhotoDelegate` object, passing `responder` to its constructor.

```
var delegate:PhotoDelegate = new PhotoDelegate(responder);
```

17. Last in this method, invoke the `loadPhotos()` method on the `delegate`.

```
delegate.loadPhotos();
```

18. Save the file.

Using the Cairngorm components



- Once Cairngorm components have been built, you can use them to either build a new application or refactor an existing, non-Cairngorm application

Do Not Copy

Walkthrough 8: Modifying FStop to use Cairngorm



In this walkthrough, you will perform the following tasks:

- Modify the FStop application to use your new, custom Cairngorm components.

Steps

1. Open the **FStop.mxml** application file.

Modify FStop to use the new Cairngorm components

2. Since the **Services** repository now has RDS components, delete the **HTTPService** tag on line 46.
3. Since the **PhotoCommand** now has the business logic for the processing the response of the *LoadPhotos* RDS request, delete the function `photosInHandler()`.
4. Since the **ModelLocator** now has the variables for the *photos* and *purchase cart* business data, delete lines 17-18.
5. Just after the **Script** block, instantiate **Services** to configure and cache the repository.

```
<rds:Services xmlns:rds="business.*"/>
```

6. Below the **Services** instantiation, also instantiate the **FSController** to properly route business events to the **PhotoCommand** business component.

```
<router:FSController xmlns:router="business.*"/>
```

7. To dispatch the *LoadPhotos* business event, modify the body of the `initApp()` function. Simply dispatch an instance of the **LoadPhotosEvent** business event. Be sure the code appears as follows.

```
private function initApp():void
{
    var event:LoadPhotosEvent=new LoadPhotosEvent();
    event.dispatch();
}
```

Tip: Remember that business events can “self-dispatch” directly to the Controller layer.

-
8. To dispatch the *AddPhotoToCart* business event, modify the body of the `photoSelectedHandler()` function. Dispatch a new business event as an instance of `AddPhotoToCartEvent`. Be sure the code appears as follows.

```
private function photoSelectedHandler(event:PhotoEvent):void
{
    var addEvent:AddPhotoToCartEvent=
        new AddPhotoToCartEvent(event.selectedPhoto);
    addEvent.dispatch();
}
```

9. In the Script block just below the imports, declare a bindable variable to alias the instance of the `ModelLocator` repository.

```
[Bindable]
private var __model:ModelLocator=ModelLocator.getInstance();
```

Tip: This alias is a convenience solution to allow easy databinding referencing of business data from the *ShoppingCart* and *Gallery* tags.

10. Update the `ShoppingCart` tag to use the `purchasedPhotos` cached in the `ModelLocator` repository.

```
<v:ShoppingCart x="626" y="118"
    "purchasedPhotos="{__model.purchasedPhotos}" />
```

11. Update the `Gallery` tag to use the `photoData` cached in the `ModelLocator` repository.

```
<v:Gallery id="gallery" label="Gallery"
    photoData="{__model.photoData}"
    photoSelected="photoSelectedHandler(event)"/>
```

12. Save the changes to `FStop.mxml` application file.
13. Change the Flex Builder perspective to Flex Debugging.
14. Set breakpoints with your instructor's guidance.
15. Build and debug the application.

Summary



- Cairngorm encourages developers to organize source code by roles and layers.
- Cairngorm is easily implemented with 3 primary classes:
 - Business Event(s)
 - Command(s)
 - FrontController
- Cairngorm solutions may also employ 2 optional classes:
 - ModelLocator
 - ServiceLocator
- Using Cairngorm helps developer focus on the MVC mantra:
 - Views render data from Model and announce user gestures
 - Control layer focus on business logic and data persistence.
 - Control logic updates Model.
- Cairngorm solutions deliver applications where views can be changed independent of business logic and data access.
- Even simply applications can [and often do-] grow to large applications. As such, even simply applications can benefit from Cairngorm architecture implementations.
 - Cairngorm MVC is not appropriate for libraries or component-level architectures.
- Cairngorm solutions are easily maintained.
- Cairngorm solutions can scale in complexity in well-defined manners.

Do Not Copy

Lab: 1



Objectives

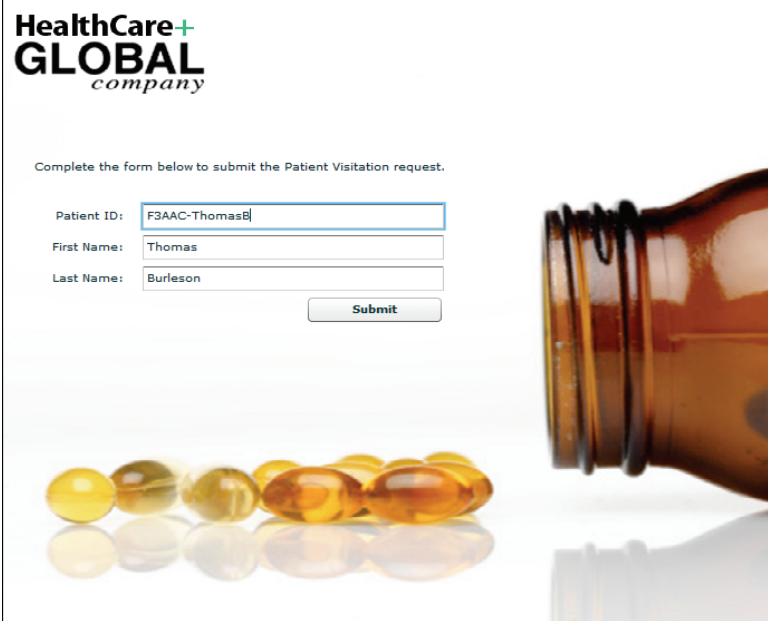
In this lab, you will perform the following tasks:

- Create a new *Cairngorm MVC* application from scratch.
- Create a form to request a patient visitation.
- Submit the patient visitation request using a business event.
- Create business components to process the visitation request.

Do Not Copy

Tasks

This lab exercise will ask you to build a Flex RIA application that allows patient visitation information to be submitted.



HealthCare+
GLOBAL
company

Complete the form below to submit the Patient Visitation request.

Patient ID:

First Name:

Last Name:

In this lab, you will learn how to do the following:

- Configure a blank *Cairngorm MVC* project
- Use **Business Events** to deliver view gestures and changes to the business layer.
- Build a **Control** layer for your business features
- Use the **Services** repository for *LiveCycle* data services
- Use the **FrontController** to connect business components to your views

Note: This RIA application will not use remote data services to save the visitation request to the server tier. That RDS feature will actually be completed in subsequent training where you will create a LiveCycle server process and configure the application to communicate with the LiveCycle data services.

Part 1: Create a new Cairngorm MVC Project

In this part of the lab, you will perform the following tasks:

- Configure project properties to use the Cairngorm library
- Create the MVC shell packages
- Create an empty Services repository
- Create an empty FrontController implementation
- Instantiate the Services and FrontController implementations

Steps

1. Close all open projects and then import the `f3ic_patientVisits` project from `{installLocation}/f3ic`.

Configure project properties to use the Cairngorm library

2. Select **Project > Properties > Flex Build Path > Library Path**
3. Select the **Add SWC...** option.
4. Browse to the `{installLocation}/f3ic` directory.
5. Choose the **Cairngorm.swc** file.

Note: If you wish to see the Cairngorm source files download the source from labs.adobe.com then use the Source Attachment option in the Library Path screen.

6. Click **OK** to close the Project Properties dialog.

Create the MVC shell packages

7. Move to the Flex Builder Navigator.
8. In the `src` folder, create the `com.patientvisitations.control` package. Follow these substeps:
 1. In the Flex Builder Navigator, **Right-Click** on the `src` folder,
 2. Select **New > Folder** from the popup menu.
 3. Specify `com/patientvisitations/control/`

Tip: This is a shortcut to create multiple, nested packages at one time.

4. Click **Finish**.
9. Create the other shell packages
 - `com.patientvisitations.control.commands`
 - `com.patientvisiations.control.delegates`
 - `com.patientvisitations.control.events`
 - `com.patientvisitations.view`
 - `com.patientvisitations.model`

Create an empty Services repository

10. **Right-Click** on the `delegates` package. Select the **New > MXML Component** option.
11. Specify a **Filename** of `Services`, a **Based on** value of `ServiceLocator`, and click **Finish**.

Note: The New MXML component wizard does not provide options for custom (non- mx Framework) classes. Type in `ServiceLocator` option for now. Next steps will manually fix the `xmlns` properly imports the `Cairngorm` class and the `Services` extends `ServiceLocator`.

12. Modify the default namespace entered, `xmlns=""`, to properly import `com.adobe.cairngorm.business.*`. Use the `rds` prefix.

```
<rds:ServiceLocator xmlns:rds="com.adobe.cairngorm.business.*"
  xmlns:mx="http://www.adobe.com/2006/mxml">
```

```
</rds:ServiceLocator>
```

13. Save the file.

Create an empty FrontController implementation

14. **Right-Click** on the `control` package. Select the **New > Actionscript Class** option.
15. Specify a filename of `PTController`.
16. Use the **Browse...** button to specify a superclass of `FrontController`.
17. Select **Finish**.
18. Save the file.

Instantiate the FrontController and Services Repository

19. Right-Click on the `src` folder.
20. Select **New > MXML Application**.
21. Specify the **Filename** `PatientVisitations` and the **Layout** to `vertical`.
22. Using the MXML tag notations, instantiate the `FrontController` and `Services` classes.

```
<control:PTController
  xmlns:control="com.patientvisitations.control.*" />
```

```
<rds:Services
  xmlns:rds="com.patientvisitations.control.delegates.*" />
```

23. Save changes.
24. Build the project and run it.

Part 2: Create a form to request a patient visitation

In this part of the lab, you will perform the following tasks:

- Create a view component to request a patient visit
- Use the view component in the application

Steps

Create a view component to submit the visitation request

1. Create a package `com.patientvisitations.view.visits`
2. Create a **New > MXML Component** called `VisitRequestForm`. Follow these substeps to build the form.
 1. Extend the `Form` class.
 2. Specify **width** and **height** of `100%`.

Tip: The following steps can be easily accomplished using the MXML Editor's Design mode and the Components view to drag controls onto the Form's stage area.

3. Add a **TextInput** field to the Form
 1. Specify a **TextInput** ID of `txtPatientID`
 2. Specify a **FormItem** label `Patient ID:`
4. Add a **TextInput** field to the Form
 1. Specify a **TextInput** ID of `txtFirstName`
 2. Specify a **FormItem** label `First Name:`
5. Add a **TextInput** field to the Form
 1. Specify a **TextInput** ID of `txtLastName`
 2. Specify a **FormItem** label `Last Name:`
6. Add a **Button** control to the Form
 1. Specify a **Button** label of `Submit`
 2. Add a click event handler to the button

```
<mx:Button label="Submit" click="onRequestVisit()"/>
```

-
7. Create a script section with an empty *onRequestVisit()* event handler.

```
<mx:Script>
  <![CDATA[
    private function onRequestVisit():void
    {

    }
  ]]>

</mx:Script>
```

8. Save changes.

Use the view component in the application

9. Open the **PatientVisitations.mxml** application file.
10. As the last child tag in `<mx:Application />`, insert a tag instance of the **RequestVisit** form.

```
<visits:VisitRequestForm xmlns:visits=
  "com.patientvisitations.view.visits.*" />
```

11. Save changes.
12. Build the project and run it.

Part 3: Submit the Visitation Request

In this part of the lab, you will perform the following tasks:

- Create a business event to deliver patient details to the business layer
- Gather the patient details from `VisitRequestForm`
- Dispatch the details to the business layer

Steps

Create a business event to deliver patient details to the business layer

1. Create a `com.patientvisitations.control.events.visits` package.
2. In the `visits` package, create a `SavePatientRequest` business event. Follow these substeps.
 1. Create a **New > Actionscript Class**.
 2. Specify a filename `SavePatientRequestEvent`.
 3. Select the `CairngormEvent` as the superclass.
 4. Select **Finish**.
3. In the source for this new event, create a static variable `EVENT_ID`.

```
static public var EVENT_ID:String="savePatientRequest";
```

4. Create a public variable for the patient's ID, firstname, and lastname

```
public var patientID:String="";  
public var firstName:String="";  
public var lastName:String="";
```

5. Modify the constructor to require `patientID`, `firstName` and `lastName` parameters. Also use the `EVENT_ID` in the constructor.

```
public function SavePatientRequestEvent(  
    patientID:String,  
    firstName:String,  
    lastName:String)  
{  
  
    super(EVENT_ID);  
  
    this.patientID=patientID;  
    this.firstName=firstName;  
    this.lastName=lastName;  
}
```

-
6. Save changes.

Gather the patient details from the RequestVisit form

7. Open the **VisitRequestForm.mxml** view component
8. In the `onRequestVisit()` method body, create and an instance of the `SavePatientRequestEvent` class.

```
var event:SavePatinetRequestEvent=null;
event=new SavePatinetRequestEvent(txtPatientID.text,
    txtFirstName.text,txtLastName.text);
```

Dispatch the visit details to the business layer

9. Invoke the self-dispatching feature [of business events] to transport the visit details to the business layer.

```
event.dispatch();
```

10. Save changes.

Do Not Copy

Part 4: Create the Business Component to Process the Visitation Details

In this part of the lab, you will perform the following tasks:

- Create a `SavePatientRequestCommand` business component to process the details of the patient visit request.
- Register the `SavePatientRequestCommand` with the `FrontController`.

Steps

Create a Command to process patient visit details

1. In the `com.patientvisitations.control.commands` package, create a Command implementation that will be used to process patient visit details. Follow these substeps.
 1. Create a **New > Actionscript Class**.
 2. Specify a filename `SavePatientRequestCommand`.
 3. Select **Add...** button to add `com.adobe.cairngorm.commands.ICommand` interface.
 4. Uncheck the option for **Generate constructor from superclass**.
 5. Select **Finish**.
2. In the method body for `SavePatientCommand::execute()`, forward the event to the `savePatientRequest()` method.

```
public function execute(event:CairngormEvent):void
{
    savePatientRequest(event as SavePatientRequestEvent);
}
```

3. Create a private method called `savePatientRequest()` that requires a `SavePatientRequestEvent` parameter.
4. In the `savePatientRequest()`, insert a `Alert.show()` statement to display the event details.

```
private function savePatientRequest
(event:SavePatientRequestEvent):void
{
    Alert.show(event.patientID + " " + event.firstName + " " +
        event.lastName + "is scheduled for a visit");
}
```

Tip: Be sure that you import the `mx.controls.Alert` class.

5. Save the file.

Register the Command with the FrontController

6. Open the **FrontController** implementation class `com.patientvisitations.control.PTController`.
7. Register the command with the `SavePatientRequestEvent` business event (and import accordingly).

```
public function PTController()  
{  
    addCommand(SavePatientRequestEvent.EVENT_ID,  
        SavePatientRequestCommand);  
}
```

8. Save the file.
9. Build project and run.
10. In the `PatientVisitations` application, complete the form and select the **Submit** button.

Do Not Copy