Flex Application Design for Cairngorm

Overview

This document will walk a developer through the process of designing a simple Cairngorm based Flex application that addresses a real world business need. The intended audience is a developer who is at least somewhat familiar with Flex, has built a few Flex applications, is at least somewhat familiar with the Cairngorm micro architecture, and wishes to learn more about how to plan a Cairngorm application properly, right from the start of their development effort.

Although this document addresses visual mockups and suggests the use of the Digimmersion Flex 2 RIA Visio stencil, it is not intended solely as a sales pitch; rather, it is intended to teach planning practices that allow for rapid development of Cairngorm applications. The use of the Digimmersion Visio stencil is obviously encouraged, but there are certainly other ways to build your visual mockups. Regardless of what tool you choose for your mockups, pre-visualizing your application is an important part of application development.

If you have questions about this document or any of the topics discussed here you may contact the author using the Digimmersion Contact Us page. If you have specific or detailed questions about Cairngorm, Flex, or ActionScript you will want to participate in the FlexCoders Yahoo! Group or access the Cairngorm Wiki on Labs.Adobe.com.

Business Challenge

This real world business challenge pertains to directory services. Most large companies maintain an Active, Novell, or LDAP Directory containing user accounts for that network. In my "day job" I work for an organization comprised of 6000+ employees and 2000+ affiliates, all requiring access to our network. Network accounts are centralized in our Novell eDirectory (eDir), accessible via LDAP.

Employee accounts reside in our Employee container and are linked to our PeopleSoft HR system with dirXML drivers so that changes in hire status, such as "Leave Of Absence" and "Terminated" are automatically migrated from PeopleSoft to eDir, and account privileges are enabled and disabled appropriately. Our Affiliates container holds user accounts for users who are *not* in our PeopleSoft system. Affiliate office staff, volunteers, vendors, and anyone who may require network access, yet are not managed or controlled by our HR department.

Since Affiliates control their own hiring & firing, and our eDir is not linked to any external HR systems, it's done manually, so our eDir is not always updated appropriately when there are staffing changes within those external organizations. They are quick to notify us that a new hire requires a new network account, but not so quick to let us know someone has left their organization. That process cannot be automated at this time and therefore leaves active accounts in our eDir for users who no longer work for these external affiliates. Not only are these lingering network accounts a security risk, but they increases the size of our eDir which adds overhead and requires additional administrative resources.

Solution

Build an application comprised of two parts:

- Part I: A scheduled background task (executable, web page, service, etc.) that runs nightly and walks our eDirectory tree collecting information on questionable accounts, department managers, last login dates/times, etc., and storing that information in a small, simple, SQL compliant database.

- Part II: A web based client used to interact with that database, providing the ability to warn our managers about questionable accounts, and give our managers the ability to select a status (LOA, Vacation, Terminated, etc) for each questionable account.

This application will provide a safety net for times when our affiliates fail to inform us of status changes with their staff.

Since we want the client to be accessible remotely a web based solution is required, and Adobe Flex has been chosen as the technology for the client.

Software Requirements Specification

First thing is to determine what the application will do and how it will behave. Follow whatever standard requirement gathering procedures you or your company uses to determine the requirements for this application, for our purposes we've kept it simple. Our application will feed off a simple database that is populated each night by an automated process. We won't detail the automated process here, just the requirements pertaining to the actual client interface itself:

- Login function
- Two available user roles: Manager and Admin
- All users are Managers
- Certain users can be defined as Administrators
- Administrators can…
    - See all questionable accounts
    - Filter to see only questionable accounts not yet assigned to a Manager
    - Select one or more accounts
    - Assign a Manager to selected accounts
    - Remove Managers assigned to selected accounts
    - Switch to Manager role
- Managers can…
    - See all accounts assigned to this Manager
    - Filter to see only accounts not yet assigned a status
    - Select one or more questionable accounts
    - Select a status
    - Select a return date for certain status conditions
    - Assign a status (and date if applicable) to selected accounts
    - Remove a status from selected accounts
    - Switch to Admin role if they have admin privileges
- Help & instructions for all interface functions
- Logout function

Overview of Cairngorm

Although this document attempts to cover the basics of planning a Cairngorm application, it does not go into any deep framework detail. For further information and code example on Cairngorm see Steven Webster's blog, the Cairngorm Wiki on Adobe Labs, or the Yahoo! FlexCoders group.

What is Cairngorm? Cairngorm is fundamentally a methodology for breaking up your application code by logical functions; by data, by user views, and by the code that controls everything. This is routinely referred to as MVC, or Model, View, and Control.

The Pieces of Cairngorm

- **Model Locator:** Stores all of your application's Value Objects (data) and shared variables, in one place. Similar to an HTTP Session object, except that its stored client side in the Flex interface instead of server side within a middle tier application server.

- **View:** One or more Flex components (button, panel, combo box, Tile, etc) bundled together as a named unit, bound to data in the Model Locator, and generating custom Cairngorm Events based on user interaction (clicks, rollovers, drag-n-drop.)

- **Front Controller:** Receives Cairngorm Events and maps them to Cairngorm Commands.

- **Command:** Handles business logic, calls Cairngorm Delegates and/or other Commands, and updates the Value Objects and variables stored in the Model Locator

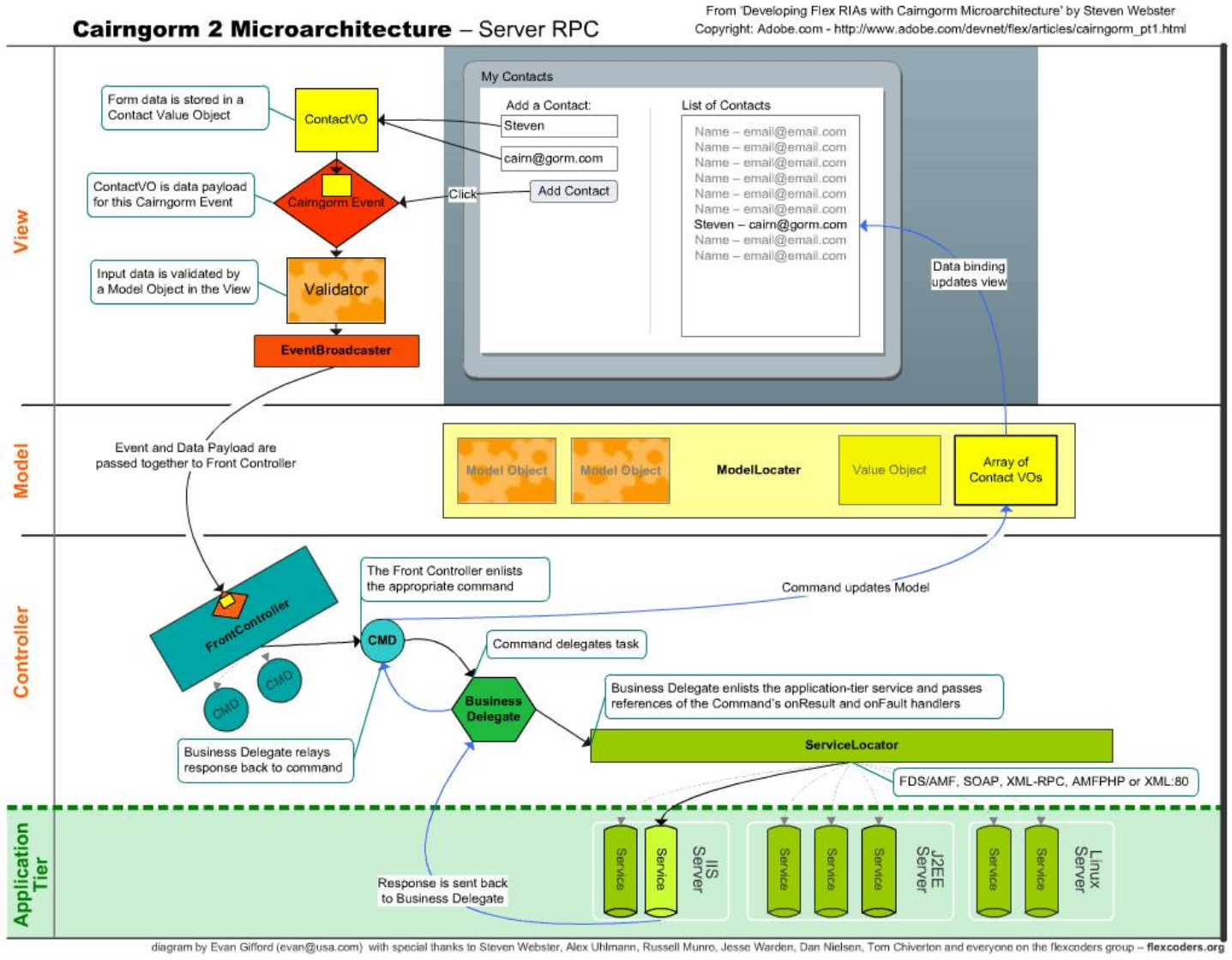
- **Delegate:** Created by a Command, they instantiate remote procedure calls (HTTP, Web Services, etc) and hand the results back to that Command.

- **Service:** Defines the remote procedure calls (HTTP, Web Services, etc) to connect to remote data stores.

How the Pieces Fit Together

Cairngorm basically works like this: Your client interface is comprised of Views. The Views use Flex binding to display data contained in the Model Locator. The Views generate Events based on user gestures such as mouse click, button press, and drag & drop. Those Events are "broadcast" and "heard" by the Front Controller, which is a map of Events to Commands. Commands contain business logic, create Delegates to perform work, handle responses from Delegates, and update the data stored in the Model Locator. Since Views are bound to the data in the Model Locator the Views automatically update when the Model Locator data is changed. Delegates call Services and hand results back to Commands, and are optional but recommended. Services make remote data calls and hand the results back to Delegates.

The diagram on the following page provides a simple overview of the processing flow in a Cairngorm application.


Cairngorm 2 Microarchitecture – Server RPC
From 'Developing Flex RIAs with Cairngorm Microarchitecture' by Steven Webster
Copyright: Adobe.com - http://www.adobe.com/devnet/flex/articles/cairngorm_pt1.html
View
Model
Controller
Application Tier
My Contacts
Add a Contact:
Steven
cairn@gorm.com
Click
Add Contact
List of Contacts
Name – email@email.com
Steven – cairn@gorm.com
Form data is stored in a Contact Value Object
ContactVO
ContactVO is data payload for this Cairngorm Event
Cairngorm Event
Input data is validated by a Model Object in the View
Validator
EventBroadcaster
Data binding updates view
Event and Data Payload are passed together to Front Controller
Model Object
Model Object
ModelLocater
Value Object
Array of Contact VOs
FrontController
CMD
CMD
CMD
The Front Controller enlists the appropriate command
Command updates Model
Command delegates task
Business Delegate
Business Delegate enlists the application-tier service and passes references of the Command's onResult and onFault handlers
Business Delegate relays response back to command
ServiceLocator
FDS/AMF, SOAP, XML-RPC, AMFPHP or XML:80
Service
IIS Server
J2EE Server
Linux Server
Response is sent back to Business Delegate
diagram by Evan Gifford (evan@usa.com) with special thanks to Steven Webster, Alex Uhlmann, Russell Munro, Jesse Warden, Dan Nielsen, Tom Chiverton and everyone on the flexcoders group – flexcoders.org

Based on these Cairngorm pieces and parts we now know some things about how to build our application.

- We need to understand our data first
- We'll use our data to determine our Views
- We'll use the Views to determine the possible user gestures
- User Gestures will be converted (mutated) into Cairngorm Events
- Events will be mapped to Commands
- The Commands may be mapped to Delegates
- Delegates map to Services
- Commands will use the results from Services to update the Model Locator

Using Mockups to Reveal the Cairngorm Components

Our Views need to display the data we're collecting nightly, so we'll need to identify at least a minimal data set to start with so we know how to approach our Views. Since Cairngorm is modular we can always add to the data model later as we find additional data we need to track and store. We'll start with some quick data in the Model Locator.

Cairngorm: Model Locator

On some system somewhere we're going to create a scheduled process that walks our eDir tree and finds user accounts that are active and enabled, but have not logged in for a pre-determined number of days. We'll refer to these as Questionable Accounts. We're also going to walk the tree and collect accounts that are managers. We'll refer to these as Manager Accounts. Our application will need to display both types of accounts, but in different Views. After retrieval we'll store all accounts in the Model Locator. We could try and create one generic AcctVO value object to hold either type of account, but the two will be different enough so that we'll use two different types of value objects; QAcctVO for a Questionable Account, and MAcctVO for a Manager Account. We'll also have multiples of each account type, so we'll store these accounts in two lists, one for Managers – MVOList - and one for Questionable accounts - QVOList. The two lists will reside inside the Model Locator. Our Views will bind to the data in these lists. We'll also need to track other information within our application, such as the person logged in and some state information, so we'll create a LoginVO object to track the user's information, and the application state will be stored as public variables. If the application was large and complex we might store application state and constants inside their own arrays or value objects to keep things organized, but in this case it will be easy enough to just create some individual public variables and constants.

*Note: just so there's no confusion, use of the term "state" denotes variable values such as in "Logged in" or "Initialised" and NOT the use of <mx:State> tag. To control Views this application will primarily use state variables as the selectedIndex for one or more ViewStacks.

Cairngorm: Views

Views can be broken up in many ways; by the data to be displayed, by the functions to be performed, by user role, by usability features, by states, etc. RIA provides almost limitless possibilities when designing a user interface, and we couldn't

possibly get into every user design pattern, usability feature, state, animation, effect, etc. available in Flex. For the sake of simplicity we're going to break the Views up by both the data to be displayed and by state (and by state I mean we'll need one View for the Questionable accounts, but that View will look slightly different for Managers than it will for Administrators, so that one View will have two states.)

We want to stay agile, so here is a rough draft for breaking up our Views. We'll refine it as we progress...

- **LoginView** – a dialog or form that accepts user authentication information.
- **ControlView** – a menu that allows users to shift from Manager to Admin roles (based on privileges) and provide a logout or close application function.
- **QAcctsView** – a grid that will list all of the Questionable accounts. It will display the same columns regardless of role, but Administrators will be able to filter their grid differently than Managers.
- **AssignmentView** – area where managers and status are selected and assigned to Questionable Accounts. When the user is an Administrator this view will allow them to assign a Manager to one or more accounts. When the user is a Manager this view will allow them to assign a status to one or more accounts.
- **HelpView** – an area designated solely for the display of help text. Since this application will potentially be used by anyone in the organization regardless of computer experience, clear and concise help information must be displayed automatically as users mouse over their different interface choices.

Mockups

As you can see, we have started to verbally describe the application interface, but it is not totally clear and not really detailed enough to start coding. We need some quick visual mockups, and here's why:

- Even though this isn't a terribly complex application, if we were to start coding at this point we would inevitably run across things we forgot or didn't realize we needed. It could really slow us down if we had to go back and add or replace large chunks of code or entire sections of the application. Agile is good, but it's also smart to plan at least some of that now.
- Flex applications can employ many layers of nested components and layout; A Canvas inside of a Panel inside of an HBox inside of a VBox inside of another Panel, and so on. With only a verbal description of the application it's very likely we'll have to code, test, recode, test, etc. fifty times trying to get a Flex layout that we like, that works properly, and that fulfills the application requirements. Mockups can really help save time here.
- If this is an application you're designing for a customer, you'll want something visual you can show them, either to get signoff or at the very least generate some excitement about the application you're building for them.

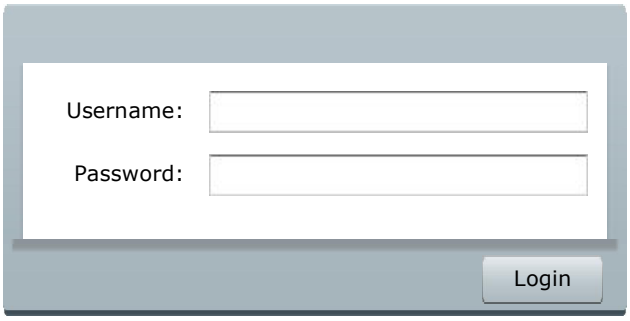
They don't need to be 100% correct and show the absolute final product (unless you want them to) but they can provide a solid foundation to start with.

Mockups can be done using any number of tools, but the one used here is the Digimmersion Flex 2.0 RIA stencil for Visio, which was created to fill a huge void in our own application development. It contains stencil masters for all the Flex 2.0 objects as well as several additional masters that will guarantee your mockups look as close to a finished Flex application as possible.

The following mockups were created with the stencil, and are shown along with the Model Locator state variables that decide which state each view will be in at any given time.

LoginView – a dialog or form that accepts user authentication information.

This View will be visible when…
ModelLocator.LoginState = "Not Authenticated"



ControlView – a menu that allows users to shift from Manager to Admin roles (based on privileges) and provide a logout or close application function.

ModelLocator.LoginState = "Authenticated"

In addition, we will track the user's role in the Model Locator and use its value to alter the look, or state of the view. The main View won't significantly change, just certain small details in the View.

ModelLocator.RoleState = "Administrator"

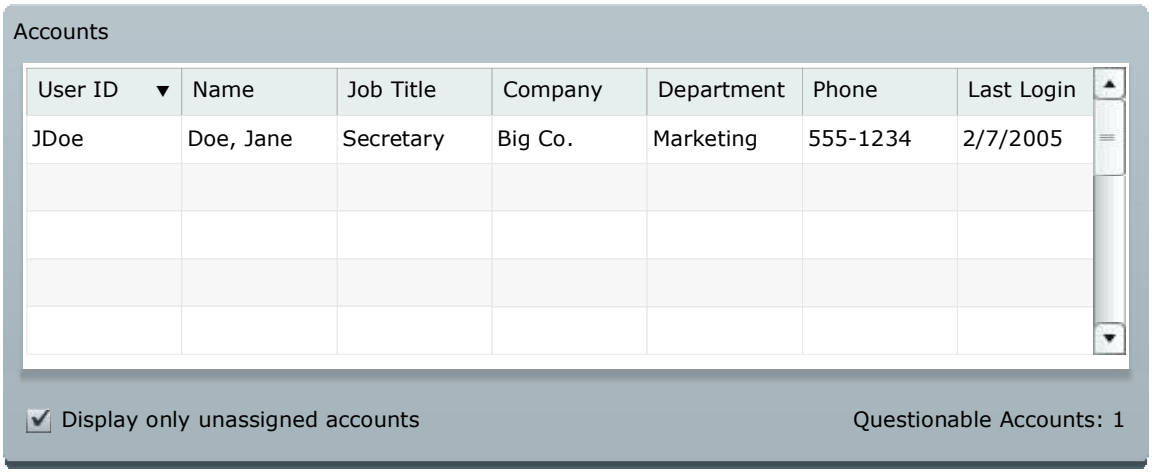
ModelLocator.RoleState = "Manager"

* An Administrator will see both "Administrator" and "Manager" as choices in their Role combo box. Managers will only see "Manager" unless they have Administrator privileges.

As you can see, there is only one View, the ControlView, but the combo box choices change based on ModelLocator.RoleState. If you find a View needs to change drastically between states then you'll need to consider breaking your View into two entirely different Views. In this case the change between states is very small (no added/removed controls or layout) so it stays one View

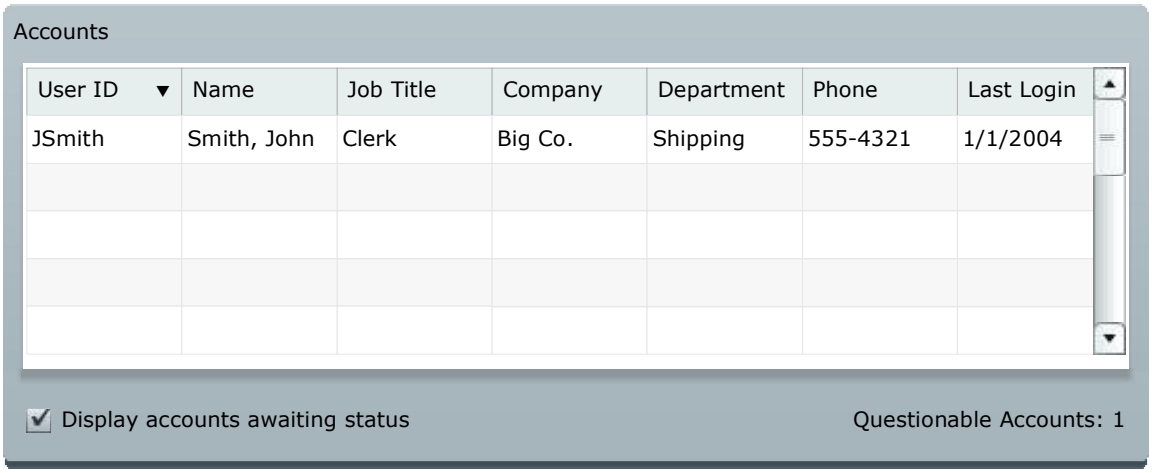
QAcctsView – a grid that will list all of the Questionable accounts. It will display the same columns regardless of role, but Administrators will be able to filter their grid view differently than Managers.

ModelLocator.LoginState = "Authenticated"
ModelLocator.RoleState = "Administrator"
ModelLocator.FilterState = "Filtered"

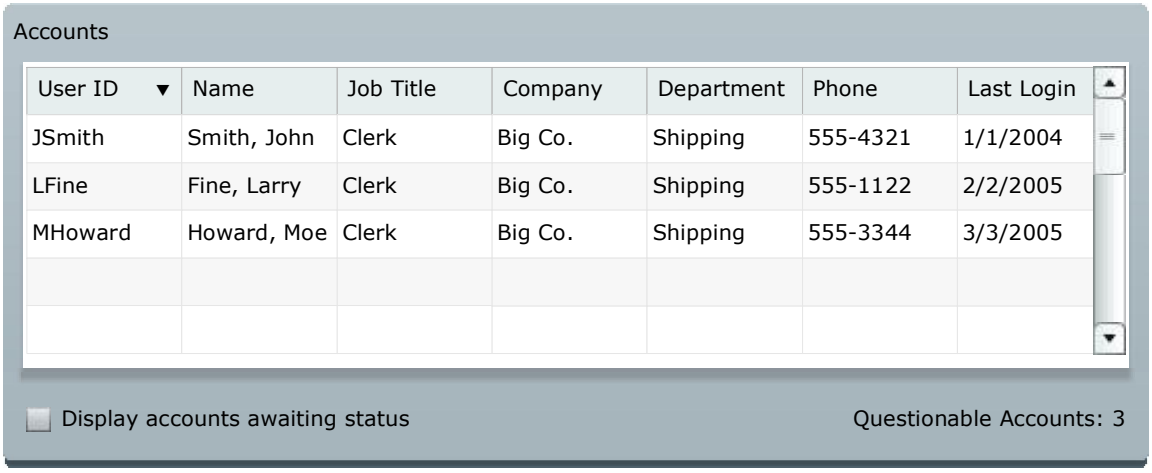


ModelLocator.LoginState = "Authenticated"
ModelLocator.RoleState = "Administrator"
ModelLocator.FilterState = "Unfiltered"

ModelLocator.LoginState = "Authenticated"
ModelLocator.RoleState = "Manager"
ModelLocator.FilterState = "Filtered"



ModelLocator.LoginState = "Authenticated"
ModelLocator.RoleState = "Manager"
ModelLocator.FilterState = "Unfiltered"

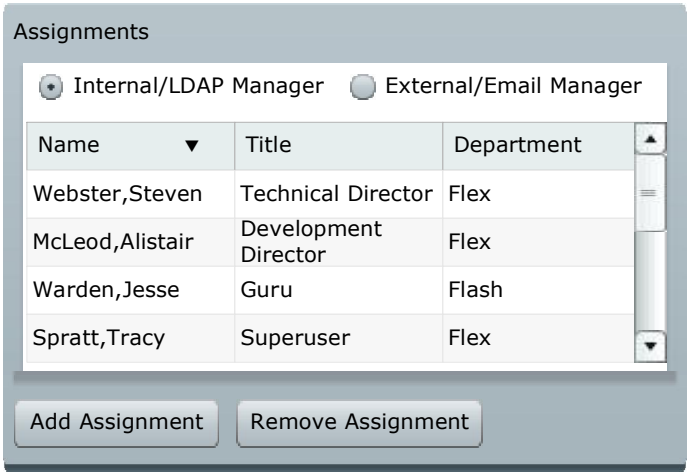


* You can see that the View is the same in each case, but as the states change (RoleState, FilterState) the data displayed by the View changes as does the filter message. Again, one View with minor changes based on the value of state variables.

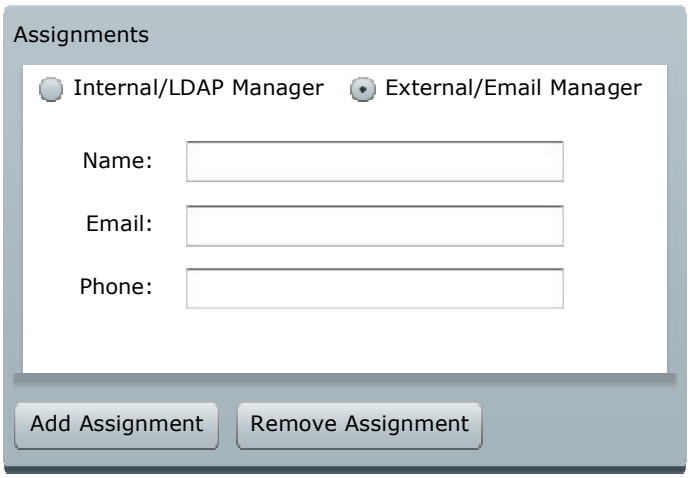
AssignmentView – area where managers and status are selected and assigned to Questionable Accounts. When the user is an Administrator this View will allow them to assign Managers to one or more accounts. When the user is a Manager this View will allow them to assign a status to one or more accounts.

The AssignmentView will have many states, causing several different layout variations to be displayed. This is an example of a View that could be broken into separate views, possibly by Role, but in this case will be left as one moderately complex View that changes based on role, type of manager, leave status, etc.

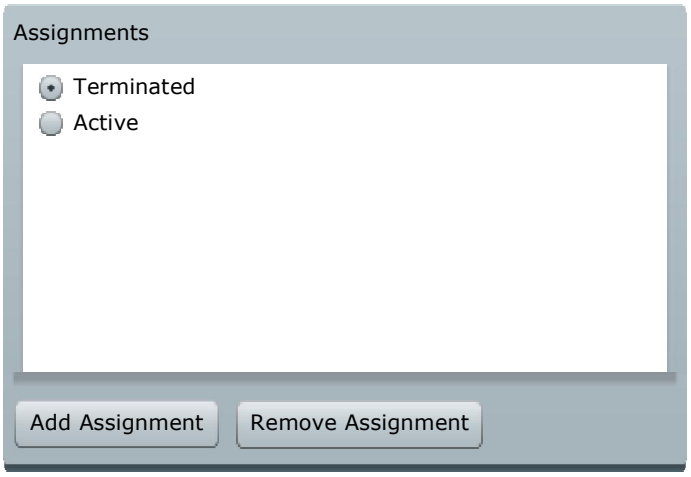
ModelLocator.LoginState = "Authenticated"
ModelLocator.RoleState = "Administrator"
ModelLocator.ManagerState = "Internal"



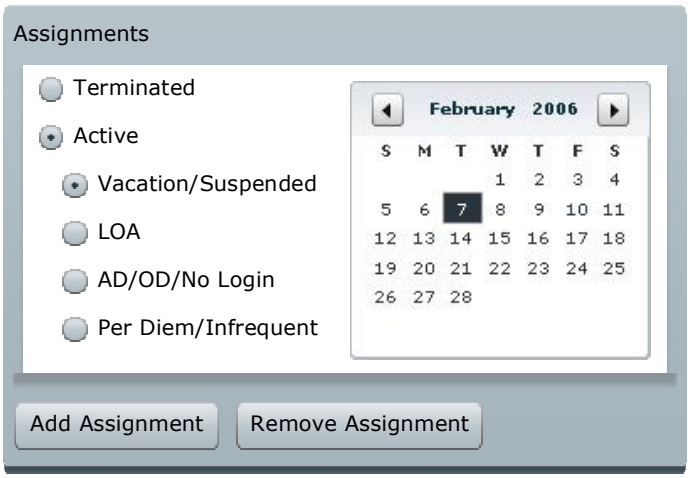
ModelLocator.LoginState = "Authenticated"
ModelLocator.RoleState = "Administrator"
ModelLocator.ManagerState = "External"



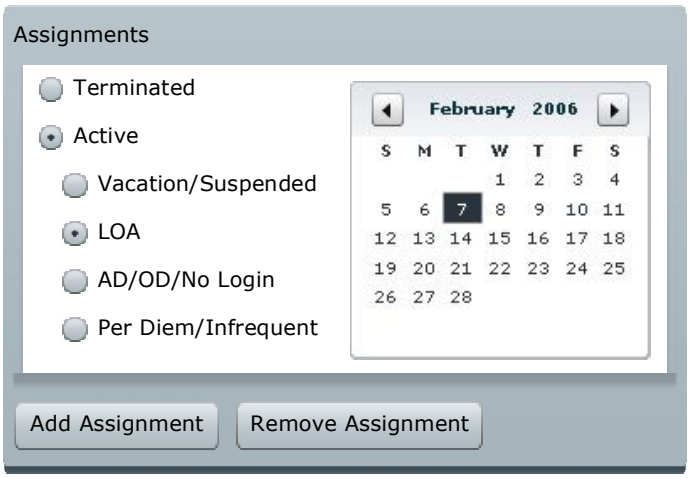
ModelLocator.LoginState = "Authenticated"
ModelLocator.RoleState = "Manager"
ModelLocator.EmployeeState = "Terminated"



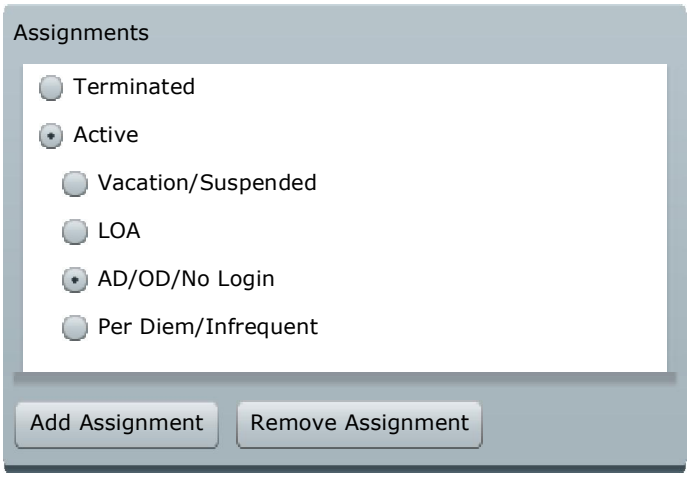
ModelLocator.LoginState = "Authenticated"
ModelLocator.RoleState = "Manager"
ModelLocator.EmployeeState = "Active"
ModelLocator.ActiveState = "Vacation / Suspended"



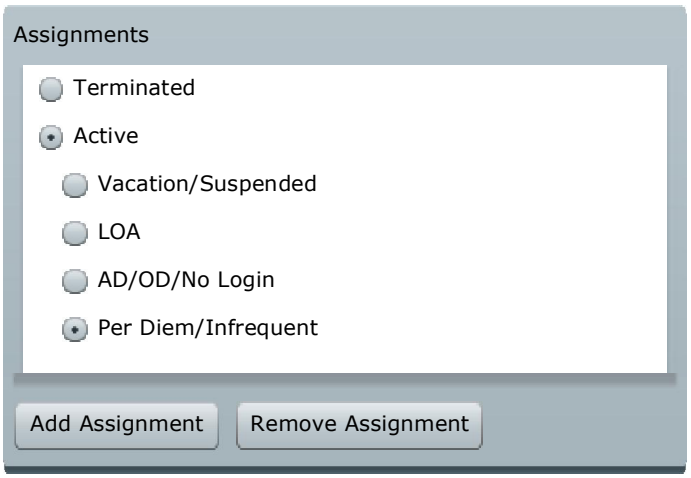
ModelLocator.LoginState = "Authenticated"
ModelLocator.RoleState = "Manager"
ModelLocator.EmployeeState = "Active"
ModelLocator.ActiveState = "LOA"



ModelLocator.LoginState = "Authenticated"
ModelLocator.RoleState = "Manager"
ModelLocator.EmployeeState = "Active"
ModelLocator.ActiveState = "AD / OD / No Login"

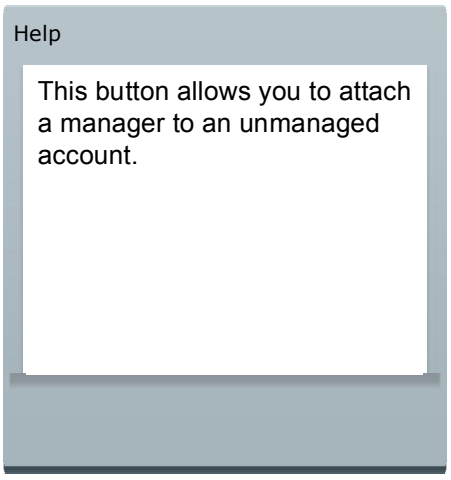


ModelLocator.LoginState = "Authenticated"
ModelLocator.RoleState = "Manager"
ModelLocator.EmployeeState = "Active"
ModelLocator.ActiveState = "Per Diem / Infrequent"



HelpView – an area designated solely for the display of help text. Since this application will potentially be used by anyone in the organization regardless of computer experience, clear and concise help information must be displayed automatically as users mouse over their different interface choices.

This View is not dependent on a state, rather it will display different text blocks of information when the user passes their mouse over different areas and components. For example, the following message might be displayed when the user moused over the Add Assignment button in the AssignmentView.

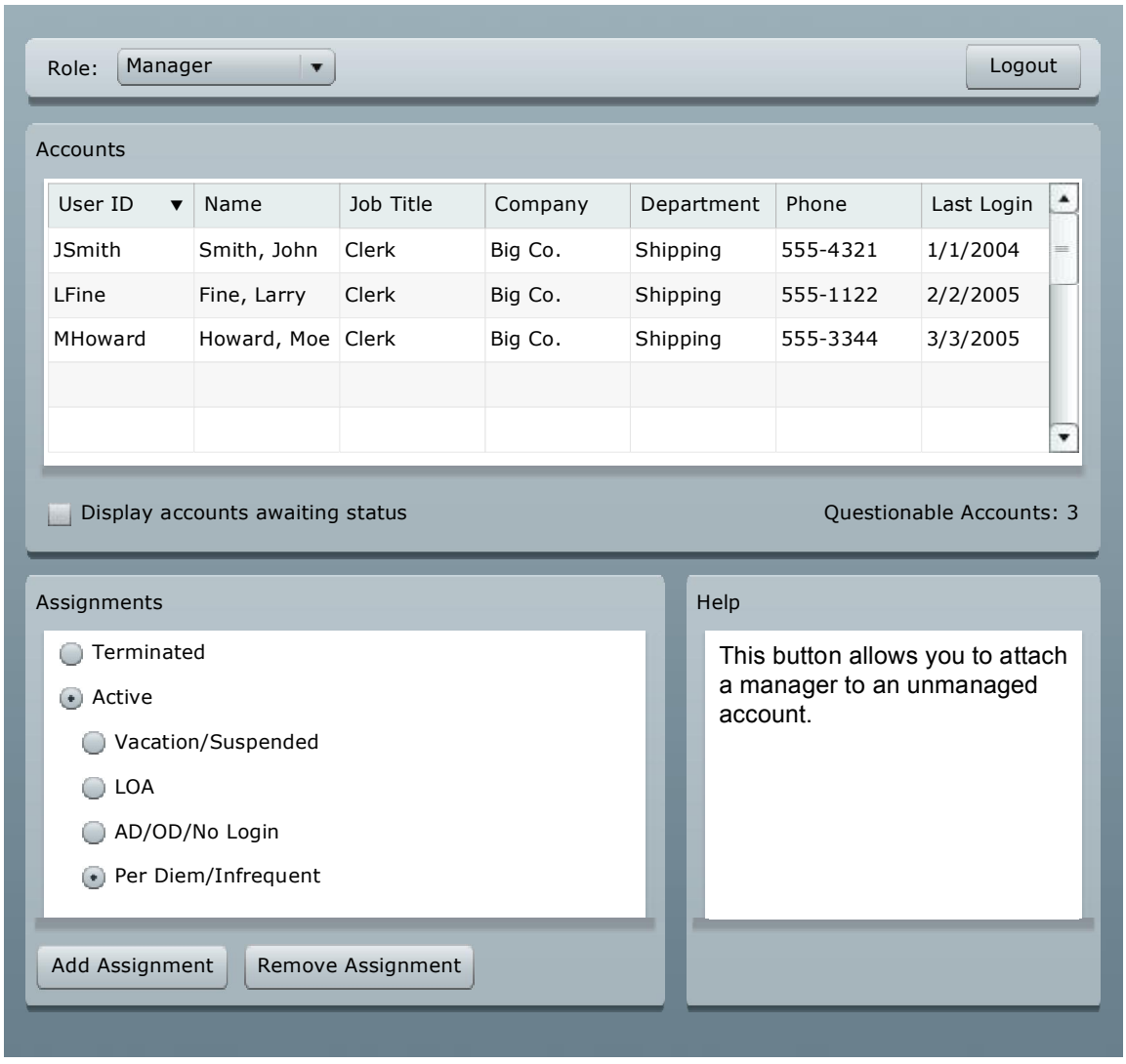


* Since we are simply designing the UI at this point we will not specify every detailed message now, instead we're just concerned with the layout and any functionality available to the user so we can determine potential user gestures that might later map to commands.

Fully Assembled Application

Creating fully assembled mockups for the whole application in every state would be time consuming and fairly redundant now that we've mocked up each individual View and it's associated states. But, just to give the reader (co-developer, manager, customer, etc) an overall picture or snapshot of the entire application, mockups can be created to show the entire application in one given state, for instance:

ModelLocator.LoginState = "Authenticated"
ModelLocator.RoleState = "Manager"
ModelLocator.FilterState = "Unfiltered"
ModelLocator.EmployeeState = "Active"
ModelLocator.ActiveState = "Per Diem / Infrequent"

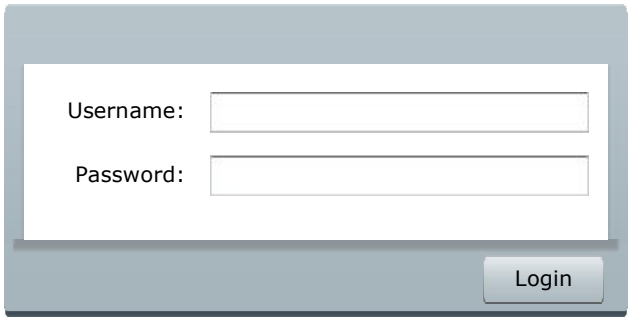


* At this point in development it's very likely you would run your mockups past stakeholders to garner some kind of approval before continuing to the next step. At this point changes to mockups can be made quickly and easily without affecting any code.

Cairngorm: Events

Now that we have Views to work from we can go through each one and determine which user gestures, or actions, a user can perform for each View in each state. We'll list the actions and then give them names that make sense in our application's context. For instance, the "Add Assignment" button's click is mutated into an assignManagerEvent. This step is actually very simple, you just need to examine the View and determine what user interaction, if any, will perform a function. For example…

LoginView



In this case, typing into the text boxes does not trigger any Commands or processes. The Login button, on the other hand, needs to trigger some processing, so…

The Login button click="" event is mutated…

LoginButton click event → LoginCommand

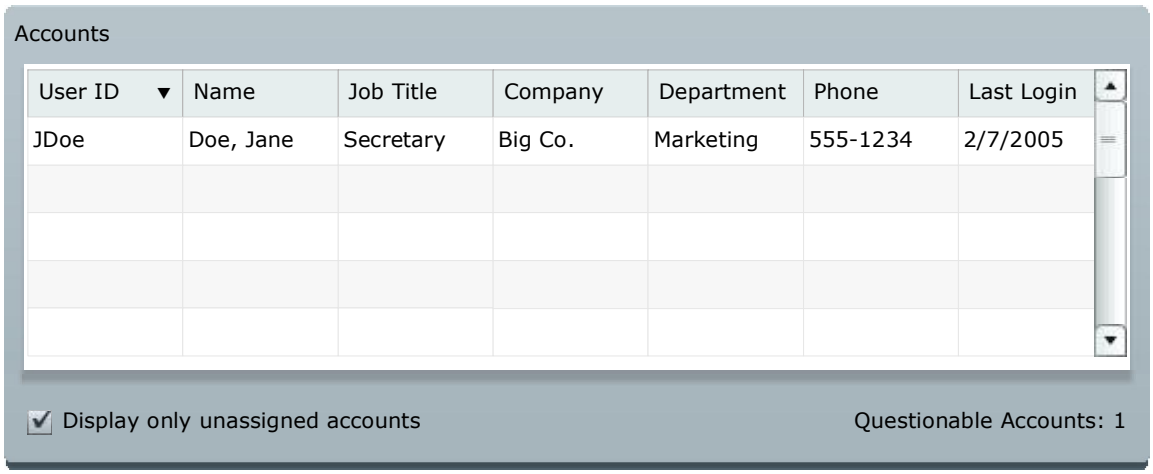
Now, follow the same reasoning for the rest of the Views. Keep in mind that these events may or may not be the final events you use in your code, but… at least it's a starting point and can somewhat clarify the application a little further.

ControlView

RoleComboBox change event → ChangeRoleEvent
RoleComboBox rollover event → DisplayInfoEvent

LogoutButton click event → LogoutEvent
LogoutButton rollover event → DisplayInfoEvent

QAcctsView



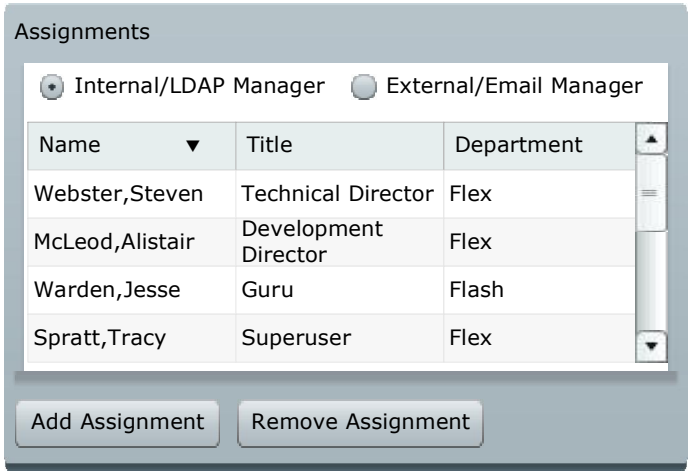
AcctDataGrid click event → SelectAcctEvent
AcctDataGrid rollover event → DisplayInfoEvent

FilterCheckbox change event → ToggleFilterEvent
FilterCheckbox rollover event → DisplayInfoEvent

* it is not necessary to list the events for this View in every state because changing state (e.g Filtered, Unfiltered) doesn't reveal any new or different buttons, checkboxes, etc. and therefore reveals no new user gestures. If components are added/subtracted or different user gestures are exposed in different states you'll need to list the events for each – as seen in the next example.

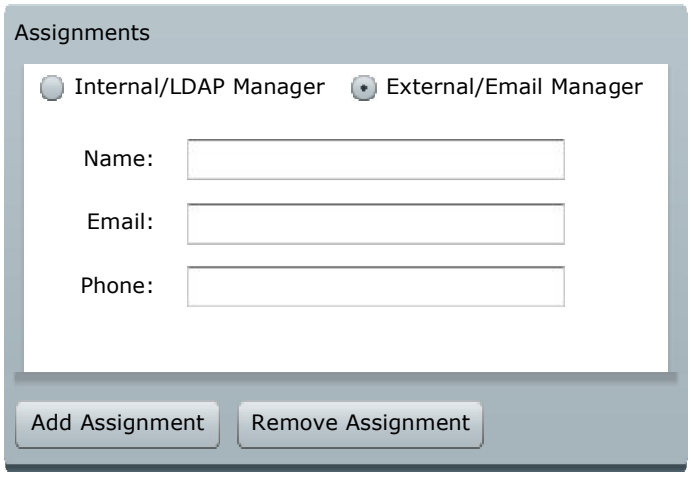
AssignmentView

ModelLocator.RoleState = "Administrator"
ModelLocator.ManagerState = "Internal"



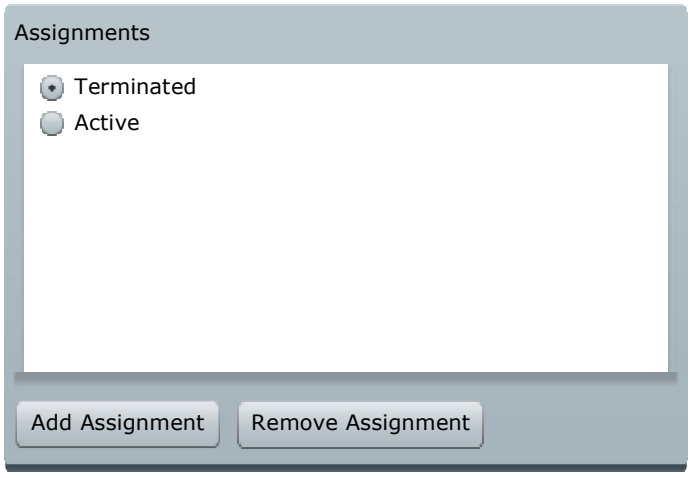
InternalRadioButton rollover event →	DisplayInfoEvent
ExternalRadioButton select event →	ToggleIntExtMgrEvent
ExternalRadioButton rollover event →	DisplayInfoEvent
ManagerDataGrid select event →	SelectManagerEvent
ManagerDataGrid rollover event →	DisplayInfoEvent
AddAssignButton click event →	AddAssignmentEvent
AddAssignButton rollover event →	DisplayInfoEvent
RemoveAssignButton click event →	RemoveAssignmentEvent
RemoveAssignButton rollover event →	DisplayInfoEvent

ModelLocator.RoleState = "Administrator"
ModelLocator.ManagerState = "External"



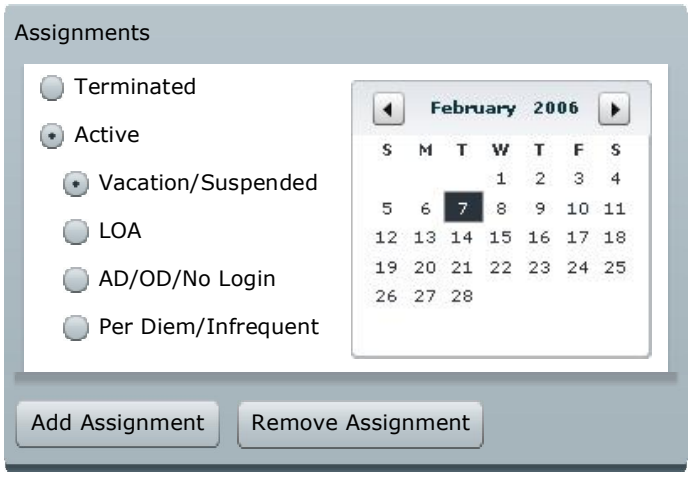
ExternalRadioButton rollover event →	DisplayInfoEvent
InternalRadioButton select event →	ToggleIntExtMgrEvent
InternalRadioButton rollover event →	DisplayInfoEvent
AddAssignButton click event →	AddAssignmentEvent
AddAssignButton rollover event →	DisplayInfoEvent
RemoveAssignButton click event →	RemoveAssignmentEvent
RemoveAssignButton rollover event →	DisplayInfoEvent

ModelLocator.RoleState = "Manager"
ModelLocator.EmployeeState = "Terminated"



TerminatedRadioButton rollover event →	DisplayInfoEvent
ActiveRadioButton select event →	ToggleEmployeeStatusEvent
ActiveRadioButton rollover event →	DisplayInfoEvent
AddAssignButton click event →	AddAssignmentEvent
AddAssignButton rollover event →	DisplayInfoEvent
RemoveAssignButton click event →	RemoveAssignmentEvent
RemoveAssignButton rollover event →	DisplayInfoEvent

ModelLocator.RoleState = "Manager"
ModelLocator.EmployeeState = "Active"



ActiveRadioButton rollover event →	DisplayInfoEvent
TerminatedRadioButton select event →	ToggleEmployeeStatusEvent
TerminatedRadioButton rollover event →	DisplayInfoEvent
VacaRadioButton rollover event →	DisplayInfoEvent
LOARadioButton select event →	ChangeActiveStatusEvent
LOARadioButton rollover event →	DisplayInfoEvent
ADODRadioButton select event →	ChangeActiveStatusEvent
ADODRadioButton rollover event →	DisplayInfoEvent
PDRadioButton select event →	ChangeActiveStatusEvent
PDRadioButton rollover event →	DisplayInfoEvent
AddAssignButton click event →	AddAssignmentEvent
AddAssignButton rollover event →	DisplayInfoEvent
RemoveAssignButton click event →	RemoveAssignmentEvent
RemoveAssignButton rollover event →	DisplayInfoEvent

* if all possible user gestures are identical in this View's other ActiveStates then there is no need to list them. If other ActiveStates result in new components or different functionality then you would list them. In this case selecting a different Active Status only adds/subtracts the date chooser, but since we're not mutating any date chooser events there's no new mutated events to list, therefore no new commands to list.

Cairngorm: Front Controller

So, now that we've used our mockups to determine user gestures that need to trigger some business logic, we can now list these events all in one place and eliminate duplicates. That leaves us with the following Events:

1. LoginEvent
2. ChangeRoleEvent
3. DisplayInfoEvent
4. SelectAcctEvent
5. ToggleFilterEvent
6. ToggleIntExtMgrEvent
7. SelectManagerEvent
8. ToggleEmployeeStatusEvent
9. ChangeActiveStatusEvent
10. AddAssignmentEvent
11. RemoveAssignmentEvent
12. LogoutEvent

What looked like a large number of events now becomes just a handful shared by the entire application. We're starting to see the power of planning ahead with our mockups! Instead of coding things several times and then realizing we could have combined them into shared functions, we now have all the functions we need listed out before we've even written a line of code.

Also at this point we may decide we need other Events that trigger background processing, like an InitializeEvent, or maybe we see where one Event needs to chain to another Event, e.g. LoginEvent may need to trigger the InitializeEvent upon successful completion.

After generating this draft list of Events the Front Controller is the next step, and it's really just a simple piece of the Cairngorm framework. The job of the Front Controller is to listen for our custom Events and trigger associated Commands. In most cases this will be a 1 to 1 mapping.

In other words, actionscript code that says:

When the LogoutEvent is broadcast → run LogoutCommand

Use all the Events listed above to draft out your Front Controller file, then go on to Commands.

Cairngorm: Commands

Cairngorm Commands are where the majority of the action takes place. This is where you code your business logic, check states in the Model Locator, branch code based on those state values, create Delegates that call Services, update the Model Locator, etc. We won't be getting into specific code for each Command, but we'll need to create at least one Command for each Event listed above. In some cases you may need to create Commands that are triggered by other Commands or non-user events, such as an InitializeCommand to be run automatically at application startup.

Here's the draft list of Commands we'll need to code:

1. LoginCommand
2. ChangeRoleCommand
3. DisplayInfoCommand
4. SelectAcctCommand
5. ToggleFilterCommand
6. ToggleIntExtMgrCommand
7. SelectManagerCommand
8. ToggleEmployeeStatusCommand
9. ChangeActiveStatusCommand
10. AddAssignmentCommand
11. RemoveAssignmentCommand
12. LogoutCommand

It's as simple as that. 1 to 1 mapping of the Events listed previously. Within each Command class you'll add the business logic that creates Value Object's, gets state stored in the Model Locator, creates Delegates that will talk to Services to make RPC calls, check results, return data, make changes to the data in the Model Locator, set new state information, etc.

Cairngorm: Delegates

Delegates in their simplest form are nothing more than go-betweens. If a Command needs to call a web service for some data it will create a Delegate to make the call. A Command creates a named Delegate… that Delegate calls a named data Service… the Service returns a result to the Delegate… the Delegate returns the results to the Command.

Delegates are not 100% required or necessary, but they are helpful when dealing with test & production environments. It's easier to remap a Delegate to a test Service than it is to use find & replace to change all prod references to test references in Command code.

Since Delegates are basically just simple classes that map to a Service we'll just need one Delegate for each Service. We'll need to determine our Services first, and then create a matching Delegate that calls it.

For instance, typically if you have a Service named GetAccountsService then you'd have a corresponding Delegate called GetAccountsDelegate. Since your ChangeRoleCommand will be the Command that gets the list of user accounts your ChangeRoleCommand would create a GetAccountsDelegate, which would then call GetAccountsService.

Cairngorm: Services

Services are somewhat similar to the Model Locator in that all your Remote Procedure Calls are contained in one place. In the Services file you could organize things any way you wish, for instance…

Service1
    Operation1
    Operation2

Or

Service1
    Operation
Service2
    Operation

This particular Flex application will be using Web Services, but Flex handles other Service types, such as Remote Objects and HTTP calls, but because of the way Cairngorm handles basic RPC Services it doesn't really matter which type you use or how many you have. Each RPC Service is called by a Delegate, makes the remote data call, and hands the result back to the Delegate.

Determining the Services needed for this application requires a look at our Commands (from the draft list we made above.) Each Command may have data (arguments) it needs to pass in and we'll need Services that can handle those.

We'll walk through each command and review it's basic functions to see what Services might be needed by each. Sounds tricky, but as you read you'll see it's actually very easy. Again, trying to stay agile we use quick and plain language to describe what each Command will do, and that should hopefully give us enough information to start with when we begin coding.
.

LoginCommand – will need to pass a uid and password to a Service, which in turn will validate the information against the directory of your choice, and return a success or failure response. In addition, administrators will be defined so we'll need a Service that can determine if the user is an administrator or not.

- AuthenticateUserService (uid,pwd)
- IsAdminService (uid)

InitializeCommand – we'll need to use our role to populate our account grid. If the user has Administrator right's we'll also want to populate the manager grid. Getting the LDAP accts will be one simple call, but the accounts shown in the larger grid will depend on role, so we'll need to call a Service and pass it a role so it knows what data to return.

- GetAcctsService (role)
- GetLDAPMgrsService()

ChangeRoleCommand – this command will need to refresh the accounts grid with a new list based on the newly selected role. We already have that Service listed above, so we'll just use it.

- GetAcctsService (role)

DisplayInfoCommand – this command would only need to make a Service call if the help information was going to be stored remotely. For simplicity we're electing to hard-code the messages into the application, so this Command won't need to retrieve any remote data.

- N/A

SelectAcctCommand – when an account is selected only some state information is changed in the application, no remote data is read or written, so this Command will not need to retrieve any data.

- N/A

ToggleFilterCommand – this Command will change the number of records displayed in the grid by changing attributes in the account VO's, it will not make return trips to the database for different data, so it will not need to retrieve any data.

- N/A

ToggleIntExtMgrCommand – this command changes the AssignmentView from displaying the LDAP grid to displaying several text input boxes. No remote data is required.

- N/A

SelectManagerCommand – when a manager is selected in the manager grid we're only going to change some application state and selected objects, no Services are needed.

- N/A

ToggleEmployeeStatusCommand – again, when switching from active / terminated only application state changes, no Services are needed.

- N/A

ChangeActiveStatusCommand – only application state changes, no remote data needed.

- N/A

AddAssignmentCommand – this Command will need to assign a selected manager from the manager grid to one or more selected accounts in the questionable accounts grid.

- AddAssignmentService (MAcctVO, QAcctVO[])

RemoveAssignmentCommand – Same as above, except the selected accounts will be looped and an attempt will be made to remove the previously assigned manager from each.

- RemoveAssignmentService (QAcctVO[])

LogoutCommand – just changes state and clears the Model Locator, no remote data necessary.

- N/A

So, after reviewing all Commands and determining needed Services, it seems we only have need for four:

- AuthenticateUserService (uid,pwd)
- IsAdminService (uid)
- GetAcctsService (role)
- GetLDAPMgrsService()
- AddAssignmentService (MAcctVO, QAcctVO[])
- RemoveAssignmentService (QAcctVO[])

Now that we've determined the Services we'll need and the parameters to be passed in we can stub out the Cairngorm Services file and build our server-side code (Web Services) using our application server of choice.

* Note: for simplicity sake this application does not secure or encrypt any web service information to validate or authenticate the client and server communications. To add this type of functionality you would want to pass additional params in each Service call containing tokens, encrypted keys, sessions, etc. to be received and validated within the web service prior to returning data.

Code the application

Now that we've drafted our Events, Commands, and Services we can start coding. At this point hopefully you can see how much easier the coding phase will be. The Events, Front Controller, and Delegates are all basically cut & paste with minor editing, and the Services file should be a snap as well, needing only one Web Service comprised of six Operations (methods). The majority of your work will be done building the mxml Views and writing the business logic into the Commands. Even the definitions for the three Value Objects shouldn't pose too much of a problem.

All this planning and use of mockups sounds like a real drag, I know, and it's acknowledged that it can certainly cause a problem and slow things down if it's overdone… but… when it comes to Cairngorm use of mockups and pre-visualizing can actually save you time later on.

Fine Tuning

Want to add new functionality to an existing Cairngorm application after it's completed? Not really a big deal. You simply identify and create the View that will display the new data, use that View to determine any new Events (user gestures), map those Events to Commands, determine if the Commands need new remote Services or can use existing ones, and create them if so, and then create a Delegate for each Service.

For instance, say you wanted to add a button to the interface that deleted accounts. You might decide to add a delete button to the accounts View.

1. That button possesses a click event that you would mutate into a DeleteCompletedAccountEvent.

2. You'd then add another line to your Front Controller that mapped the event to a Command.

3. The command would contain some business logic, create a delegate to call a Service, and assuming the Service call succeeded update the Model Locator by removing that account.

4. A new Service (or method) would also need to be created.

In Conclusion

Hopefully this tutorial has been helpful and has shown not only the benefits of designing parts of your Cairngorm Flex app ahead of time, but the benefits of using quality mockup design tools and the Cairngorm micro architecture. Planning your app properly from the start and staying organized with a design pattern like Cairngorm can help you code faster, complete projects sooner, test and debug easier, and more easily scale your application later, when it become a big hit and the suits ask it to do more than you ever thought it would have to do.

Thanks, and good luck!