ERIMAdaptor Framework

Overview

ERIMAdaptor provides an Instant Messenger interface to your application using standard WOComponents. The architecture of this adaptor is somewhat different than the normal HTTP adaptor, because of inherent limitations in the IM interfaces like AOL IM:

- WOSessions are tracked by buddy name. When a buddy contacts the server's IM account, a Conversation is initiated. A Conversation associates a buddy name with the WOSession ID. Conversations have a configurable expiration, but the default is to expire after 5 minutes of inactivity. The WOSession will have the same expiration as they normally would, but the connection between the buddy name and that session will go away, effectively expiring the session's usefulness.
- Different IM networks may impose additional restrictions on your components. For instance, most networks limit the number of bytes that can be transferred in a message. Currently the framework doesn't auto-split, but that's an obvious enhancement that will probably be added in.
- Some features of WORequest and WOResponse may not behave exactly like the HTTP counterpart. For instance, there are no normal HTTP headers in the WORequest.
- The buddy name and message appear as form values and in the request userInfo dictionary - use whichever is more convenient.
- The Conversation object is accessible via the request userInfo dictionary if you need direct access (to force expiration, etc).
- The implementation is TEMPORARILY single threaded for all conversations. This will change soon, I just haven't had time to test it.

Currently there are implementions of the IInstantMessengerFactory for AIM using the jaimbot and daim libraries. Jaimbot is available from http://jaimbot.sourceforge.net and daim is available from http://daim.dev.java.net if you're interested in getting more information on either library.

How to use it

ERIMAdaptor is a custom adaptor, so it can be added using the WOAdditionalAdaptors option on your WO:

```
-WOAdditionalAdaptors ({WOAdaptor="er.imadaptor.InstantMessengerAdaptor";})
```

Properties

There are several new settings that can (or must) appear in your Properties file as well:

- IMFactory (optional, default er.imadaptor.AimBotInstantMessenger$Factory) - the factory class for creating IM connections. If you want to provide a new type of IM network, you should implement the IInstantMessengerFactory interface for whatever class you provide here.
- IMScreenName (required) - the screen name of the IM account that the server should login with
- IMPassword (required) - the password of the IM account that the server should login with
- IMTimeout (optional, default 5 minutes) - the conversation timeout time in milliseconds
- IMAutoLogin (optional, default "true") - whether or not the adaptor should autologin to AIM. If false, you must call adaptor.connect()
- IMConversationActionName (optional, default "imConversation") - the name of the DirectAction to call when a Conversation is initiated, the default is that you must create a public WOActionResults imConversationAction() { .. } method in your DirectAction class
- IMWatcherEnabled (optional, default "false") - whether or not you want to have a second AIM account login and watchdog the first. Most of the AIM libraries can have issues with getting kicked off after a period of time. If you have two AIM accounts, they can keep eachother alive.
- IMWatcherFactory (optional, default same as IMFactory) - if IMWatcherEnabled, the factory class to use
- IMWatcherScreenName (optional, required if IMWatcherEnabled) - the screen name of the watcher IM account
- IMWatcherPassword (optional, required if IMWatcherEnabled) - the password of the watcher IM account

Request

In your code, the following request headers are available:

- IsIM - Boolean.TRUE if the current request is an IM request (you can call InstantMessengerAdaptor.isIMRequest(WORequest) also)
- IMConversation - the Conversation associated with this request
- BuddyName - the name of the buddy that initiated the request
- Message - the message sent by the user

The following form values are available in the request:

- BuddyName - the name of the buddy that initiated the request
- Message - the message sent by the user

DirectAction

To kick things off, you must have a direct action method that matches the value of IMConversationActionName (which defaults to "imConversation"). This method should return the the first "page" of the IM conversation.

For example:

```
public WOActionResults imConversationAction() {
   return pageWithName(SayHelloPage.class.getName());
 }
```

IMAction

For each request, the IM Adaptor needs to know what action to call when the subsequent request comes in. In a normal WOComponent, think of this as each page having a "Next" button on it that is clicked each time a request comes in. Instead of using a WOSubmitButton or a WOHyperlink, you instead use the IMAction element. This element has one attribute - "action" that points to the action method to call on your component. Because of inherent limitations in an instant messenging interface as an interaction method, only one IMAction can be triggered per-page (the last one that is evaluated on the page, specifically). This makes sense when you think of the user's response as the action being performed. There is no semantic equivalent of "multipleSubmit" on WOForm.

IMAction is the simplest action you can use, though there are several common response actions, and ERIMAdaptor provides default action implementations for them. Here is an example of a very simple conversation that uses IMAction (that will just continue to say the same thing over and over again):

Your Java component would look something like this:

```
public class IMComponent extends WOComponent {
   public boolean userResponded;

   ...
```

```
public String buddyName() {
     return InstantMessengerAdaptor.buddyName(context().request());
   }

   public WOActionResults processResponse() {
     userResponded = true;
     return null;
   }
 }
```

The HTML would be:

```
<webobject name = "FirstContactConditional">
   Hi <webobject name = "BuddyName"></webobject>!
 </webobject>
 <webobject name = "UserRespondedConditional">
   Welcome back <webobject name = "BuddyName"></webobject>!
 </webobject>
 <webobject name = "IMAction"></webobject>
```

And lastly the WOD file would contain:

```
FirstContactConditional : WOConditional {
   condition = userResponded;
   negate = true;
 }

 UserRespondedConditional : WOConditional {
   condition = userResponded;
 }

 BuddyName : WOString {
   value = buddyName;
 }

 IMAction : IMAction {
   action = processResponse;
 }
```

Here's what an example conversation transcript might look like:

```
Mike> Hey server!
 Server> Hi Mike!
 Mike> How's it going?
 Server> Welcome back Mike!
 Mike> Uh. OK.
 Server> Welcome back Mike!
 ...
```

IMConfirmationAction

IMConfirmation has a single binding "confirmed". If the response from the IM buddy matches any of a set of common "yes", "no", etc words, confirmed is set to the appropriate value. If neither a yes nor a no word is found, confirmed is set to null. You should bind this to a Boolean rather than a boolean so that you can detect the third state properly and re-ask the question. Note that this is not internationalized, so it currently has "yes", "y", "yep", "true", "no", "n", "nope", "nah" as known values.

Bindings

- confirmed = three-state Boolean, when Boolean.TRUE the response was affirmative, when Boolean.FALSE, the response was negative,

and when null, the response did not match one of the known true/false values
- action = the action method to execute when the response is received

Example

```
AreYouSureAction : IMConfirmationAction {
   confirmed = yesOrNo;
   action = nextStep;
 }
```

In the above example, the binding "yesOrNo" is a Boolean that will contain whether or not the user's response was a confirmation. After processing the response, the "nextStep" action will be called so that you can evaluate the "yesOrNo" value and respond accordingly.

IMSearchMessageAction

IMSearchMessageAction allows you to map substrings that appear in AIM message responses to other objects. For instance, you can pass in an options dictionary that maps the word "hi" to the object Greeting, or the word "bug" to the object BugReport. If the word "hi" appears in the aim response, it will return the matching object as its value.

Bindings:

- value = the binding to write the first matching value into
- values = (at least one of "value" or "values" must be specified, but you can use both as well) the binding to write the array of all matching values into
- optionsDictionary = a mapping of substrings to look for in the response to arbitrary objects those substrings are associated with
- optionsArray = a list of known substrings to look for
- optionKeyPath = (only applicable for optionsArray) instead of looking for the values in optionsArray, look for the value of this key path on each value in the optionsArray; for instance, optionsArray might contains Person objects, and optionKeyPath might be "firstName", so this would match the first names of the list of the people.
- quicksilver = match like Quicksilver matches ("NPE" matches "NullPointerException")
- action = the action method to execute when the response is received

Example

```
PickRequestTypeAction : IMSearchMessageAction {
   value = selectedOption;
   optionsDictionary = options;
   action = nextStep;
 }
```

In the above example, "optionsDictionary" contains a mapping where the keys are the substrings to look for in the response and the values are objects to associate them with. For instance, the dictionary from our bug tracking system looks like
{ "bug"=>BugReportType object, "request"=>ServiceRequestType object }
. "action" is the method to fire when the user's response comes back, and "value" will contain the value that matched the user's response.

When the user responds, the response will be interpreted by the provided action, the "selectedOption" field will contain whatever object matched their response, or null if there was no match, and the "nextStep" action method will be called. "nextStep" is a normal action method like any other, except it has access to the additional AIM-only request parameters as described in the Request section.

IMSearchOptionsAction

IMSearchOptionsAction, IMSearchMessageAction's evil twin, allows you to search your options for the AIM response that is received (as opposed to SearchMessage that searches the AIM response for your options -- the other way around). For instance, you can pass in an options dictionary that maps the word "Company XYZ" to the object CompanyXYZ, or the word "Company ABC" to the object CompanyABC. If the word "XYZ" is aimed, it will return the matching CompanyXYZ object as its value.

Bindings

- value = the binding to write the first matching value into
- values = (at least one of "value" or "values" must be specified, but you can use both as well) the binding to write the array of all matching values into
- optionsDictionary = a mapping of substrings to look for in the response to arbitrary objects those substrings are associated with
- optionsArray = a list of known substrings to look for
- optionKeyPath = (only applicable for optionsArray) instead of looking for the values in optionsArray, look for the value of this key path on each value in the optionsArray; for instance, optionsArray might contains Person objects, and optionKeyPath might be "firstName", so this would match the first names of the list of the people.
- quicksilver = match like Quicksilver matches ("NPE" matches "NullPointerException")
- action = the action method to execute when the response is received

Example

```
PickCompanyAction : IMSearchOptionsAction {
   optionsArray = companies;
   optionKeyPath = "name";
   value = company;
   values = companies;
   quicksilver = true;
   action = nextStep;
 }
```

The above example from our bug reporting system allows a user to select a company to associate with a bug. "optionsArray" contains an array of Company objects. "optionKeyPath" specifies that the response should be compared against the value of the "name" keypath on the Companies. The resultant first Company object that matched will put stored in the "company" binding (as defined by "value"), and if there were multiple matches, they would all appear in the "companies" list. Quicksilver matching is enabled above, and after the response is processed, and the bindings set, the "nextStep" action will be called.

IMPickListAction

A very common case is that you want to present the user with a list of options and have the user pick from that list. IMPickList provides a handy implementation of this behavior. Whereas the previous actions were "headless," IMPickList actuals renders the list on its own entirely. A typical use case is to continue rendering the IMPickList until the selection is whittled down to one by the user.

An example conversation might look like:

```
Server> Pick a company:
 Server> 1) Company XYZ
 Server> 2) Company VWX
 Server> 3) Company ABC
 User> X
 // note this matches #1 and #2
 Server> Pick a company:
 Server> 1) Company XYZ
 Server> 2) Company VWX
 User> V
 // Company VWX is now selected
 Server> ...
```

or

```
Server> Pick a company:
 Server> 1) Company XYZ
 Server> 2) Company VWX
 Server> 3) Company ABC
 User> 1
 // picking by number uniquely identifies Company XYZ
```

or

```
Server> Pick a company:
 Server> 1) Company XYZ
 Server> 2) Company VWX
 Server> 3) Company ABC
 User> XYZ
 // XYZ uniquely matches Company XYZ
```

Bindings

- displayStringKeyPath = The key path of the string to use to display each object in the list. The value should be in "quotes".
- list = an NSArray of the list of options to pick from
- quicksilver = whether or not to use Quicksilver-style matching on the user's response ("NPE" matches "NullPointerException")
- selections = the list of selections that matched
- selection = the single matching selection (if there was only one)
- action = the action method to execute when the response is received

Example

```
PickProductAction : IMPickListAction {
   list = products.@sort.name;
   selection = product;
   selections = products;
   displayStringKeyPath = "name";
   quicksilver = true;
   action = nextStep;
 }
```

The above example has the user selecting from a list of Products (from the "list" binding). The "displayStringKeyPath" binding specifies that the "name" keypath of the Product should be displayed in the list. When the user uniquely identifies a single Product, the Product object will be bound to the "product" variable. If there are multiple matches, the NSArray of the matches will be bound to the "products" variable. Quicksilver matching is enabled. After the response is processed, the "nextStep" method will be called. Notice in this example, the "products" array is both the "list" binding AND the "selections" binding. When the page is first loaded, the "products" NSArray is initialized with all the products. The nextStep method keeps returning the same page until there is only one product selected. Until that point, the list is reset with the products that matched the user's partial response (like in the example transcript of IMPickList above).

IMTextAction

IMTextAction is one of the simplest IMAction implementations. It stores the user's response in a String binding and calls an action method.

Bindings

- value = the variable to store the user's response into
- allowBlanks = if false, empty strings (response.trim.length == 0) are turned into nulls
- action = the action method to execute when the response is received

Example

```
AskNameAction : IMPickListAction {
   value = userName;
   allowBlanks = false;
   action = nextStep;
 }
```

The above example stores the user's response in the "userName" variable and does not allow blank values (i.e. they turn into nulls). The "nextStep" action method is called when the response is processed.