

Supporting Content Type Display on Container Lists

This topic describes how you can get instances of your content type to show up in UI set aside for them in containers such as spaces, projects, and social groups.

Note: The content type API is still a new feature that might change as developers provide feedback about it.

These views on content give people a way to narrow their view of information so they can more quickly get to what they need. Your content type can have its own tab in these user interfaces. This topic is divided into two main sections: one for implementing support of formal containers -- spaces, projects, and social groups, for example -- and one for support of a more virtual container -- user profiles.

Supporting Display on Container Lists

When someone visits a space, project, or social group, they're able to get a sense what it contains. By default, there's a tab for each of the kinds of content they can find there. They can click the tab to view a list of the content (or some other kind of summary). In your content type, you can support having your instances displayed as a tab in containers such as spaces, projects, and social groups. You do something similar for user profiles, but they're a little different, as you'll see in what follows.

The way you implement support for display on a tab in a container is pretty much the same for spaces ("communities" in the API), projects, and social groups. Profiles (which are similar to containers) work a little differently. This section starts by describing how you implement support for the main container types, then describes how you can do it for profiles.

Here's the high-level process for most containers. The following sections will give more information about how to do this for display in a space as the container, but the code for projects and social groups can be very similar.

- Implement the containable type info provider corresponding to the container you want to support.
- Implement `ContainableTypeInfoProvider` to return the info provider for those containers in which instances should be listed.
- Configure Spring to inject dependencies as needed.
- Write the FreeMarker template that will render the UI for listing instances of your content type.
- Add a `<component>` stanza to your `plugin.xml` to have your tab show up in the UI and provide a URL to execute when it's clicked.

You might notice that one thing missing from this process is implementing and configuring a Struts action. In this case, there's already an action available, behind the scenes, that you can use to get access to code in the execution context -- including your info provider. This is a convenience feature, as you'll see in what follows.

Providing Information About Support for Display in the Container

When implementing an info provider specific to the container you're supporting, you give the application a path to the FreeMarker file you're using for user interface. You can add other code as you like to support your goals, much as you would if you were implementing an action.

- Create the class by implementing the info provider interface for the container you're supporting. Add accessors to support Spring dependency injection and interface implementation.

```
/**
 * Provides methods required for rendering a space's "Memos" tab-body.
 *
 * See community-content.ftl and {@link com.jivesoftware.community.action.CommunityAction}.
 * @see com.jivesoftware.community.objecttype.CommunityContainableInfoProvider
 */
public class MemoCommunityContainableInfoProvider implements CommunityContainableInfoProvider {

    // Fields to hold instances injected by Spring.
    private ObjectPackageManager objectPackageManager;
    private CommunityManager communityManager;

    /**
     * Called by Spring to inject an object type manager instance.
     *
     * @param objectPackageManager The instance to be injected by Spring.
     */
}
```

```

*/
public void setObjectTypeManager(ObjectTypeManager objectTypeManager) {
    this.objectTypeManager = objectTypeManager;
}

/**
 * Called by Spring to inject a community manager instance.
 *
 * @param communityManager The instance injected by Spring.
 */
public void setCommunityManager(CommunityManager communityManager) {
    this.communityManager = communityManager;
}

// Code follows.
}

```

- Add code that will be called to get your FTL file, as well as other code you'll need from within the UI. Here, FTL file is community-memos.ftl. There's also a method, called from inside the FTL file, to get the list of memo instances for the space that's displaying them.

```

/**
 * Gets the path to the FTL file that should be used to render a list of
 * memos in the space.
 *
 * @return The FTL URL.
 */
public String getMainBodyFtl() {
    return "/plugins/memo-type-example/resources/community-memos.ftl";
}

/**
 * Gets the memos in the specified space.
 *
 * @param The community that's the container for memos.
 * @return An iterator with the memo instances.
 */
public Jivelerator<JiveContentObject> getMemos( Community community ) {
    ResultFilter memoFilter = ResultFilter.createDefaultContentFilter();
    MemoObjectType memoType = (MemoObjectType) objectTypeManager.getObjectType(MemoObjectType.MEMO_TYPE_ID);
    ContainableType[] types = new ContainableType[1];
    types[0] = memoType;
    return communityManager.getCombinedContent(community, memoFilter, types);
}

```

- Implement ContainableTypeInfoProvider to return the info provider for those containers in which instances should be listed. Also, you'll want to implement getTagViewID to have a hook into your UI from the tab you add via your plugin.xml. The following example omits a great deal of code in order to keep things simple. Be sure to look at the sample for a more complete view.

```

public class MemoContainableTypeInfoProvider implements ContainableTypeInfoProvider {

    // A field to hold the instance injected by Spring.
    private CommunityContainableInfoProvider communityContainableInfoProvider;
}

```

```

/**
 * Gets the view ID value that will be appended to a URL that requests
 * the UI to display on the content type's tab in the container. This value
 * is used, for example, in the memo's plugin.xml file, where it appends
 * the view URL with "view-memo". Similar URLs would be used for the view
 * in spaces, projects, and social groups.
 *
 * @return The ID value.
 */
public String getTabViewID() {
    return "memo";
}

/**
 * Called by the application to get the info provider instance.
 *
 * @return The instance.
 * @see com.jivesoftware.community.ContainableTypeInfoProvider#getCommunityContainableInfoProvider()
 */
public CommunityContainableInfoProvider getCommunityContainableInfoProvider() {
    return communityContainableInfoProvider;
}

/**
 * Called by Spring to inject the info provider instance.
 *
 * @param communityContainableInfoProvider The injected instance.
 */
public void setCommunityContainableInfoProvider(CommunityContainableInfoProvider communityContainableInfoProvider) {
    this.communityContainableInfoProvider = communityContainableInfoProvider;
}

// Omits many of the other methods required by the ContainableTypeInfoProvider interface.
}

```

- Edit your spring.xml to configure Spring to inject the dependencies you'll need.

```

<bean id="memoCommunityContentInfoProvider" class="com.jivesoftware.clearspace.plugin.test_dynamic.MemoCommuni
    <property name="communityManager" ref="communityManager"/>
    <property name="objectManager" ref="objectManager"/>
</bean>

<bean id="memoContainableTypeInfoProvider" class="com.jivesoftware.clearspace.plugin.test_dynamic.MemoContainabl
    <property name="communityContainableInfoProvider" ref="memoCommunityContentInfoProvider"/>
    <!-- Properties omitted. -->
</bean>

```

Creating a User Interface for Displaying an Instance List

The FreeMarker template you'll write needs to display your summary of your content type's instances. That summary can look pretty much however you like. You can emulate the

lists of documents, discussions, and blog posts that are included by default. You can do something different if it makes more sense to do so. For example an image-oriented content type might display image previews instead. The following simple example emulates the default behavior.

- Write the FreeMarker template that will render the UI for listing instances of your content type. Notice in the sample code that you have access to an action instance behind the scenes. Through that instance, you can retrieve your info provider instance to get data that you'll need for your UI. Here, the action is used to get the list of instances for the space by passing the current community identifier into the info provider's getMemos method. The FTL files for projects and social groups in this sample do something similar.

```
<div class="jive-content-block-container jive-content-block-docapproval">
  <!--
    Display a heading such as "All Memos in Human Resources"
  -->
  <h3 class="jive-content-block-header"><@s.text name="community.tab.memo.header"/> <@s.text name="global.in"/> $
  <!-- Create a variable from the list of memos provided by the action. -->
  <#assign memos = action.containableInfoProvider.getMemos(action.community) >
  <!--
    If there are memo instances to list, then use the
    jiveContentList macro to list them here. That macro
    assumes that your content type implements the RecentContentInfoProvider
    interface to return the action URL for viewing an instance.
  -->
  <#if memos?exists && memos.hasNext()>
    <@ jive.jiveContentList content=memos />

  <!--
    If there aren't any memo instances to list, then display a
    way to create one. The jiveEmptyContentList macro is designed
    for cases when, yes, the content list is empty. The macro
    assumes that your Jive object type implements ContentObjectType
    and ContentObjectTypeInfoProvider. That interface's
    getCreateNewFormRelativeURL method returns the action URL for
    creating a new instance of your content type.
  -->
  <#else>
    <!-- BEGIN content list -->
    <div class="jive-content-block">

      <!-- BEGIN content results -->
      <div id="jive-content-results">
        <!-- Provide a link for creating a new instance. -->
        <@jive.jiveEmptyContentList container=community showTypeExclusively="memo"/>

      </div>
      <!-- BEGIN content results -->

    </div>
  </div>
```

```

<!-- END content list -->
</#if>
</div>

```

Add a Tab for the Content Type

You've got code that executes to display a view of your instances in various containers -- now you need a way for the user to get that view. To do that, you'll update your plugin.xml file to add a tab to each of the containers you're supporting.

The following snippet shows a <component> stanza that adds a content type tab to the set of tabs displayed when viewing a space. The <name> and <description> elements pull strings from the plugin's properties file. The <when> element ensures that your view is only available if the content type is supported in the current community. The <url> element gives the URL that should be invoked when someone clicks the tab. Noticed that the URL includes the view parameter whose value you specified in the getTabViewID method of your info provider class.

```

<component id="community-tabs">
  <tab id="memo" cssClass="jive-icon-med jive-icon-memo-med">
    <name><![CDATA[<@s.text name="comnty.tab.memo" /> <span class="jive-link-count">({statics[com.jivesoftware.clear
    <description><![CDATA[<@s.text name="comnty.tab.memo.desc" />]]></description>
    <when><![CDATA[ObjectTypeUtils.isTypeEnabledForContainer(community, statics[com.jivesoftware.clearspace.plugin.test_dy
    <url><![CDATA[<@s.url value='${JiveResourceResolver.getJiveObjectURL(community)}' />?view=memo<#if tagSet?exist
    </tab>
  </component>

```

Supporting Display in User Profile Lists

User profiles are a bit like containers (spaces, projects, social groups) in that they display summaries of content within a particular context -- in this case, the person whose profile it is. But they differ in that a content type instance isn't really inside the profile -- one doesn't create an instance inside the profile. The profile view is instead an aggregation based on one data point -- the user. Profiles also differ in that they offer two views of similar information -- a view provided to the user whose profile it is, and a view for other users who want to see that user's activity.

Here are the high-level steps:

- Implement an action to retrieve values that will be used on the user profile page.
- Write a FreeMarker template for giving a view of the user's instances.
- Configure Struts to execute your action.
- Implement the UserProfileInfoProvider interface to return hooks into the UI and action you're creating.

Supporting Content Type Display on Container Lists

- Implement `ContentObjectTypeInfoProvider` to return your `UserProfileInfoProvider`.
- Implement `ContentObjectType` to return your `ContentObjectTypeInfoProvider`.

Supporting

- Implement an action to retrieve values that will be used on the user profile page. The easiest way to do this is to extend `BaseViewProfileContent`, from which you can easily get the ID for the current user. In your implementation, you'll provide the data that will be needed by your user interface. Note that this is a little different from other actions you might have written. The UI that this action connects to includes an FTL file that wraps your own template. The containing FTL renders the UI that's outside the view of your instance view. As a result, it has its own requirements that must be addressed by this action, as noted in the following code.

```
/**
 * An action to display the memos for a user.
 */
public class MemoViewProfileAction extends BaseViewProfileContent {

    // The user's memos.
    protected JiveIterator<JiveContentObject> memos;

    // A field for the Spring-injected instance.
    private CommunityManager communityManager;

    /**
     * Gets the FeedInfo instance matching a property in the containing
     * profile FTL file. There's no feed info for memos, but the property
     * has to be satisfied, so return null here.
     *
     * @return The instance.
     */
    public FeedInfo getFeedInfo() {
        return null;
    }

    /**
     * Gets the title of the page that will display the view of memos.
     *
     * @return The page's title.
     */
    public String getPageTitle() {
        return "Profile Memos";
    }

    /**
     * Called by Spring to inject a community manager instance.
     *
     * @param communityManager The injected instance.
     */
    public void setCommunityManager(CommunityManager communityManager) {
        this.communityManager = communityManager;
    }
}
```

```

}

/**
 * Gets the injected community manager instance.
 *
 * @return The instance.
 */
public CommunityManager getCommunityManager() {
    return communityManager;
}

/**
 * Gets a list of the memos for the current user.
 *
 * @return
 */
public JiveIterator<JiveContentObject> getMemos() {
    if ( memos == null ) {
        // Set a result filter to filter on the current user. The userID
        // is inherited from the base class.
        ResultFilter memoFilter = ResultFilter.createDefaultContentFilter();
        memoFilter.setUserID( userID );
        MemoObjectType memoType =
            (MemoObjectType) objectTypeManager.getObjectType(MemoObjectType.MEMO_TYPE_ID);
        ContainableType[] types = new ContainableType[1];
        types[0] = memoType;
        memos =
            communityManager.getCombinedContent(communityManager.getRootCommunity(),
                memoFilter, types );
    }
    return memos;
}
}

```

- Write a FreeMarker template for giving a view of the user's instances. The UI rendered by this template will be contained by the UI rendered by another template included with the application, view-profile-content-type.ftl. This template also uses two macros included with the application to render a list of the instance or to provide alternative UI if there aren't any to list.

```

<div class="jive-content-block-container">
  <h3 class="jive-content-block-header"><@s.text name="profile.tab.memo.header" /></h3>
  <#--
    If there are memos to display, assign the memos retrieved from the action as
    content for the jiveContentList macro.
  -->
  <#if memos?exists && memos.hasNext()>
    <@jive.jiveContentList content=memos />
  <#else>
    <!-- BEGIN jive-table -->
    <!-- BEGIN content list -->
    <div class="jive-content-block">

      <!-- BEGIN content results -->

```

```

<div id="jive-content-results">
    <!--
        If there aren't any user-specific memos, use the jiveEmptyContentList
        to provide a way to create one.
    -->
    <@jive.jiveEmptyContentList container=communityManager.rootCommunity showTypeExclusively="memo"/>
</div>
<!-- END content results -->
</div>
<!-- END content list -->
</#if>
</div>

```

- Configure Struts to execute your action. You'll pass this action name back to the application through your info provider class (see the code that follows). Also, notice here that the FTL file you're using for a successful result is one you didn't write; it's the template that contains yours. In later code, you'll tell the application how to find yours.

```

<action name="view-profile-memos" class="com.jivesoftware.clearspace.plugin.test_dynamic.action.MemoViewProfileAct
    <result name="success">/template/global/view-profile-content-type.ftl</result>
</action>

```

- Implement the UserProfileInfoProvider interface to return hooks into the UI and action you're creating. The getMainBodyFtl method is where you tell the application where to find the template you're using to render the view of the user's instances. The getActionName method should return the name of your action as you defined it in your struts.xml file.

```

/**
 * An info provider to support displaying memo-related information on a user profile.
 */
public class MemoUserProfileInfoProvider implements UserProfileInfoProvider {

    /**
     * Gets a path to the FTL file that represents the profile view of a user's
     * memos.
     *
     * @return The path to the FTL file.
     */
    public String getMainBodyFtl() {
        return "/plugins/memo-type-example/resources/view-profile-memo-body.ftl";
    }

    /**
     * Gets the name of the action executes to support user interface for viewing
     * memos on a profile.
     *
     * @return The name of the action as it is given in the struts.xml file.
     */
    public String getActionName() {
        return "view-profile-memos";
    }
}

```

Supporting Content Type Display on Container Lists

- Implement `ContentObjectTypeInfoProvider`. Here's where the application gets your user profile info provider.

```
/**
 * An object type info provider for the memo.
 */
public class MemoContentObjectTypeInfoProvider implements ContentObjectTypeInfoProvider {

    private UserProfileInfoProvider userProfileInfoProvider;

    /**
     * Gets the info provider that describes how memos support user
     * profiles.
     *
     * @return The info provider instance.
     */
    public UserProfileInfoProvider getUserProfileInfoProvider() {
        return userProfileInfoProvider;
    }

    /**
     * Called by Spring to inject a user profile info provider instance.
     *
     * @param userProfileInfoProvider The injected info provider instance.
     */
    public void setUserProfileInfoProvider(UserProfileInfoProvider userProfileInfoProvider) {
        this.userProfileInfoProvider = userProfileInfoProvider;
    }

    // Code omitted.
}
```

- Implement `ContentObjectType` to return your `ContentObjectTypeInfoProvider`.

```
public class MemoObjectType implements ContentObjectType
{
    private ContentObjectTypeInfoProvider contentObjectTypeInfoProvider;

    public void setContentObjectTypeInfoProvider(ContentObjectTypeInfoProvider contentObjectTypeInfoProvider) {
        this.contentObjectTypeInfoProvider = contentObjectTypeInfoProvider;
    }

    public ContentObjectTypeInfoProvider getContentObjectTypeInfoProvider() {
        return contentObjectTypeInfoProvider;
    }

    // Code omitted.
}
```