8

Faces Extensions in the Bridge Environment

It is common for Faces applications to run in an extended Faces environment. I.e. one where the applications relies on function not provided in the base (reference) implementation but rather is provided by an implementation using one of the many extensions mechanisms Faces defines. As the portlet bridge also is in part built as a Faces extension consideration needs to be given by these other Faces extension developers to ensure proper execution in a portlet environment..

8.1 Configuration

Because its the bridge's implementation that abstracts the details of running in a portlet environment its important that the bridge extensions execute as close as possible to the base layer as possible. This is why the bridge's `FacesContextFactory` is configured as a service provider. To run correctly all other application extensions that aren't defined in the application's `faces-config.xml` should be configured in the `META-INF/faces-config.xml` in their jar file. They should not be registered as a service provider.

Unfortunately, the Faces extensions implemented by the bridge aren't limited to those that are configured via the service provider mechanism. The bridge also extends the `ViewHandler` and `StateManager`[6.2 and 6.3]. The `ViewHandler` and `StateManager` are configured in its `faces-config.xml` that its in the `META-INF` directory of its `jar` file. As Faces doesn't define a set precedence order when like configuration exists in other `faces-config.xml` files that are also in a `jar`'s `META-INF` directory, one can't control or determine whether or where in the `ViewHandler` or `StateManager` chains the bridge's implementation will be called. To avoid obscuring other extensions, the bridge delegates where ever possible and then augments the result. When this is not feasible, the bridge ensures it at least duplicates the behavior defined by the RI for this function. Other extensions are expected to be implemented in a similar manner to avoid obscuring necessary bridge behavior. I.e. they either delegate to ensure the bridge gets a chance to execute or they implement the bridge's function directly. Sections 8.5 and 8.6 discusses this in detail.

8.2 Initializing Extensions

If an extension subsystem needs to perform one time initialization, it is recommended it rely on providing a `ServletContext` listener as defined in the Servlet

specification.

If an extension subsystem needs to perform per request initialization, it is recommended it provide its own implementation of `FacesContextFactory` and perform this initialization in the appropriate activation calling sequence.

Commonly such initialization involves wrapping either the request or response objects to support extended semantics on these objects. In the Servlet environment extensions have two options, they can wrap the request or response object before delegating the creation of the `FacesContext`/`ExternalContext` or they can wrap later and replace the request/response object managed by the `ExternalContext` using the `setRequest` or `setResponse` calls. Because the portlet API doesn't directly support such wrapping, extensions must only use the later mechanism when wrapping a portlet request or response. I.e. so that the bridge can properly dispatch Faces requests it must be constructed with the original portlet request and response objects.

Some Faces extensions use servlet filters to manage they per request initializations/processing. Because filters aren't supported in the portlet environment, this will not work. Instead its recommended the extension use the "provide a `FacesContextFactory`" technique discussed in this section.

8.3 Portlet Considerations

There may be times when an extension needs to determine if its running in a portlet vs. a servlet environment. Generally the check needs to be made without regard to whether the bridge and/or portlet container are actually deployed in the environment. The simplest technique relying on `instanceof javax.servlet.ServletRequest(Response)` is insufficient because some portlet containers may have extended the underlying servlet request/response object in implementing `PortletRequest`/`PortletResponse`. Instead the extension should look for the existence of the bridge's request attribute which holds the current portlet phase. The attribute name is `javax.portlet.faces.phase`. If this attribute exists it is a portlet request.

```
boolean isPortletRequest = externalContext.getRequestMap().containsKey("javax.portlet.fac
```

When the bridge can be assumed to be deployed in the application, the utility method `javax.portlet.faces.BridgeUtil.isPortletRequest()` may be used instead.

Additionally, if the Faces extension utilizes a `Lifecycle` listener, it likely needs to be modified to run properly in a portlet environment. A portlet application can contain many portlets each relying on their own instance of the bridge. The Faces `Lifecycle` however is application scoped, hence any `Lifecycle` listeners are also application scoped. This means a `Lifecycle` listener registered on a

Lifecycle in one portlet will actually be registered and called for all portlets receiving requests for the duration of the registration. To avoid doing inappropriate or duplicate work, the `Lifecycle` listener code should include a check to only operate if its operating within the current `FacesContext`. I.e. Surround the listener function with a condition check that ensures the `FacesContext` passed in the event parameter to the method is the same as the `FacesContext` in the current thread (`FacesContext.getCurrentInstance`).

8.4 General Considerations

Extensions should:

1. avoid interacting directly with container specific objects. Instead rely on the `ExternalContext` abstraction.
2. when interacting directly with container specific objects, provide support for both the `Servlet` and `Portlet` containers.
3. avoid completely overriding any method. Where its necessary to do so, the extension must provide equivalent function as specified in this specification.
4. avoid caching the `FacesContext` and other dependent objects in your own variables. In the portlet model the action and render phases span requests. A likely bridge implementation creates/destroys the `FacesContext` for each request. This means a different `FacesContext` (and dependent) instance exists during the action phase then in the render phase. If you cache such instances you will be referencing the wrong, destroyed instance.

8.5 Cohabiting with the Bridge's ViewHandler

Because of limitations in the Faces configuration environment there is no way to order extensions that are defined (exclusively) in the `faces-config.xml`. This applies to the Faces `ViewHandler`. This means when running in an environment where other subsystems also introduce a `ViewHandler`, one can't predict whether the bridge's `ViewHandler` will take precedence or not. The bridge is designed to cohabit with these other `ViewHandlers` regardless of the actual precedence order. However the same may not be true of these other `ViewHandlers`. To work correctly in a portlet environment with the bridge, other `ViewHandler` extensions are expected to:

- Not run any (servlet) non-portlet container specific code when executing a portlet request.
- If the extension's `ViewHandler.createView()` returns its own `UIViewRoot` whose `Class.getName` isn't `javax.faces.component.UIViewRoot` then the extension should have its `UIViewRoot` implement the same semantics for portlet namespacing as described in section 6.6. I.e. the

implementation class of its `UIViewRoot` should be annotated to indicate it implements the portlet namespacing semantics and also provide such an implementation.
- If the extension's `ViewHandler.renderView()` handles its own form of rendering then the extension should ensure the same style of response interleaving as provided by the default `renderView` and the bridge.

8.6 Cohabiting with the Bridge's StateManager

Because of limitations in the Faces configuration environment there is no way to order extensions that are defined (exclusively) in the `faces-config.xml`. This applies to the Faces `StateManager`. This means when running in an environment where other subsystems also introduce a `StateManager` one can't predict whether the bridge's `StateManager` will take precedence or not. The bridge is designed to cohabit with these other `StateManagers` regardless of the actual precedence order. However the same may not be true of these other `StateManagers`. To work correctly in a portlet environment with the bridge, other `StateManager` extensions are expected to:

- Not generate any output directly within the `writeState(FacesContext context, Object state)`. I.e. the extension should rely/override other methods that are called when `writeState` is executed to generate any output. If this is not feasible, the extension developer should document the issue to inform portlet application developers that when using the extension in a bridge environment they need to move the `StateManager` configuration from the bridge jar's `faces-config.xml` and put it into the applications (`WEB-INF`) `faces-config.xml`. This ensures the bridge's `StateManager` runs before the extensions and hence isn't obscured by it.
- Delegate `writeState(FacesContext context, Object state)` to the next `StateManager` in the chain. Note: a common practice when implementing a `StateManager` (in earlier Faces versions) was to extend `javax.faces.application.StateManager` and to delegate calls to this super class. To work properly with the bridge such implementations need to change to delegate to the next `StateManager` in the chain. This is most easily done by reimplementing the `StateManager` as an extension of `javax.faces.application.StateManagerWrapper`.

8.7 Excluding Attributes

When executing an extension may store temporary state in the request scope. Because the bridge extends the life of this scope, extensions should explicitly exclude those attributes which do not need to be preserved. I.e. are recreated as needed in each (portlet) request. The preferred way to exclude such an attribute is to annotate the class of the attribute's value

with `javax.portlet.faces.annotation.ExcludeFromManagedRequestScope`. If this is not feasible, attributes can be excluded by declaring them in the extension jar's `faces-config.xml` as described in 5.1.2.1. When using this technique care should be taken to only exclude attributes that are uniquely namespaced as other portlets in the same web application may rely on other extensions and/or attributes that might collide with a non-unique name and be mistakenly excluded. In such a situation when non-uniquely namespaced attributes are used, the extension developer should merely document for the portlet developer her need to configure such an exclusion in the application's portlet.xml for each portlet that utilizes the extension.

Previous

Portlet 2.0 Bridge for JavaServerTMFaces 1.2 – November 14th, 2010

Next