

PageAggregationDesign

Copyright 2004 The Apache Software Foundation

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

Page Aggregation Redesign Overview

Authors: Raphael Luta, David Sean Taylor

**** Goals ****

1. Refactor PSML Model and introduce a new Page model, fixing defects in old model
2. To reimplement the page aggregation features
3. Simplify storage and manipulation of all aggregation data structures
Clean separation of profiling information from page storage
4. Support storing page information in a relational database
5. Single threaded and multiple threaded aggregation
6. Provide simple and clear 'hooks' for other components to manipulate Page components
such as Page Selection, Layout Customization, and Content Caching)

**** Defects in J1 Page Aggregation ****

1. PSML couples preferences and user info with layout.
2. PSML has confusing terminology 'controls' 'controllers'
3. Aggregation process in Jetspeed-1 model was tied to Turbine framework via modules

**** Solution Summary ****

1. Separation of Layout from Preferences and User Information
The new model removes all preferences and user information from the design
Preferences and User Information are specified in the Java Portlet Spec

J2 stores preferences separately from layout

2. New Model Terminology Introduced
 - 'Controls' are now called 'Decorators'
 - 'Controllers' are now called 'Layouts'
 - 'Portlets' collections are now called 'Fragments'
 - 'Desktop' describes the entire Page layout comparable with Turbine layout+navigation
3. Jetspeed-2 no longer relies on Turbine.
 - Turbine modules and all Turbine dependencies are not in Jetspeed-2.
 - We now have the concept of a Desktop which has a layout component, a theme, and a current page.
 - A page is the aggregated content of portlets.

Page Aggregation Model

The new aggregation code is composed of the following elements:

1. pipeline stages in the main Jetspeed pipeline:
 - Profiler --> Layout --> Aggregator
2. New OM model under tentatively (org.apache.jetspeed.om.page)
3. New service for storing and retrieving pages (PageManager)
 - with two implementation (database/file system)
4. a PortletRenderer component

Pipeline in more Detail

1. Profiler valve
 - | -- find Desktop
 - | -- find Page

The Profiler finds a desktop and page dependent on the associated profiling rule associated with the current principal. A principal can have one profiling rule associated with it. Two rules are currently identified:

- a. Standard Profiling Rule - applies J1 profiling rules based on the standard J1 profiling criteria (user, mediatype, language, country, pagename, role, group)
- b. Role-based Fallback - applies J1 role-based fallback algorithm

Design goals for Profiling Rules:

- aa. pluggable algorithms

PageAggregationDesign

bb. generic engine based on Profiling Rule and Criteria model
cc. Support Profiling Types to generically map request parameters without programming

- standard
- request parameters, attributes, properties
- CCPP properties
- User properties

c. The profiler finds both Desktops and Pages which are then put into the request context to be used by the next pipeline valve

2. Layout valve

```
-- compile and build the layout for the Desktop  
-- compile and build the layout for the Desktop
```

The layout valve is the first pass of a two pass aggregation (layout, render)

Layout algorithm (Raphael Luta):

```
lStack = new Stack();  
push page root layout element on lStack  
while lStack is not empty  
do  
  cElement = pop lStack  
  if cElement is hidden  
    do nothing  
  else  
    put cElement in request attributes  
    find Portlet registered under cElement name  
    invoke Portlet in LAYOUT mode (through a sub-request to Portlet)  
    retrieve from portlet response attributes a list of portlet and  
    layout  
    elements to display  
    discard any other output from portlet  
    put only layout elements retrieved from response in lStack  
    for all retrieved portlet elements  
    do  
      if portlet element is visible  
        invoke PortletRenderer for retrieved portlet element with  
        current  
        request (asynchronous mode)  
      end if  
    done  
  end if  
done
```

The goal of this algorithm is to launch the rendering process in asynchronous mode but only for those elements that are going to actually be displayed in the aggregated page. The idea is thus to allow the layout elements that actually control the rendering to have a look at the page structure and return what elements they will actually render when they are invoked in RENDER mode. Using a Jetspeed specific LAYOUT Portlet mode is a way to

avoid defining Jetspeed specific Layout interfaces at the price of handling a sub-request within the main RENDER phase. It's also simply possible to define a PortletLayout interface that Portlet that want to be recognized as Layout elements need to implement and invoke a doLayout() method through this interface (this avoids having to manipulate the PortletRequest portletMode)

This is an important performance optimization in multi-threaded parallel aggregation mode to avoid rendering some components for nothing because they are not visible in the final page.

Also note that layout elements themselves are not sent to the Renderer in the layout phase because they will most likely not be able to output meaningful content without access the actual portlet content and should not cause a major performance issue if their rendering is serialized.

In summary:

- layout elements are never parallelized or cached
- visible portlet elements are sent to the rendering process in the Layout pipeline valve

3. Portlet Render valve

The next valve in the process is the Aggregate valve that simply retrieves the root element of the page and invoke the PortletRenderer service in synchronous mode with this element and request. This is enough to trigger the cascaded rendering all all elements that need to be displayed and aggregated. Exactly how a Layout Portlet is expected to access the other rendered elements is explained in more details in the PortletRenderer service description.

PortletRenderer service

This is the core of aggregation engine. It's responsible for rendering the Portlet themselves and make the rendered content available to other portlets for inclusion.

This service only provides 3 public methods:

```
/** Render the specified Page fragment using pInstance as the portlet to
    invoke and PortletRequest as the request to pass to pInstance. Result is
    returned in the PortletResponse.
    */

public void renderNow(Fragment fragment,
                    PortletInstance pInstance,
                    RenderRequest request,
                    RenderResponse rsp)

/** Render the specified Page fragment using pInstance and PortletRequest.
    The method returns before rendering is complete, rendered content can
```

PageAggregationDesign

```
be accessed through the
ContentDispatcher
*/

public void render(Fragment fragment,
                  PortletInstance pInstance,
                  RenderRequest request)

/** Retrieve the ContentDispatcher for the specified request
*/
public ContentDispatcher getDispatcher(PortletRequest request)

From an implementation perspective, the renderNow() method is pretty
straightforward but the multi-threaded
rendering is more involved.
Here are my current implementation thoughts (I've not yet fully
explored the
problem though because it would
probably require some working code running to check for all possible
bottlenecks and deadlocks):

The RendererService implementation needs to have:
- a rendering queue rQueue where are queued all the pending rendering
requests. This queue needs to provide
  a thread-safe element retrieval method like Stack.pop() and push() (in
fact I was planning to use the default
  Stack implementation even though it's not FIFO as this queue should
be)
- a pool of Worker threads that can render synchronously portlets
- a WorkerMonitor for controlling the worker threads and recycling them
into
the pool
- a ContentDispatcher that gives access to generated content. A
ContentDispatcher is basically a wrapper around
  a Hashtable of PortletResponses keyed to fragment Ids.

Each time render() is invoked it would create a new RenderingRequest
that
contains:
- a reference to the Fragment to render
- a reference to the Portlet Instance
- a thread-safe wrapper to the PortletRequest
- a simple PortletResponse implementation that writes to a buffer
It then adds the PortletResponse to the current session
ContentDispatcher
(creating the ContentDispatcher if no
dispatcher is found within the current session), keyed to the Fragment
ID.
It then puts the RenderingRequest in the rQueue and returns

The WorkerMonitor would continuously monitor the rQueue for new
elements to
render and whenever a new element
is added to the queue, it dispatches it to an idle worker if any. When
```

the Worker is done rendering, it explicitly commits the PortletResponse to signal to the ContentDispatcher that the content is available and complete.

Finally the ContentDispatcher simply provides an include(Fragment ID, PortletRequest, PortletResponse) method that copies its stored buffered content into the provided PortletResponse output stream. If content is not available when include() is called, the ContentDispatcher either pauses until the content is available (for example, by periodically polling the PortletResponse keyed to the ID is committed) or triggers a synchronous rendering of the Fragment (for example if no PortletResponse is currently keyed to this ID)

Layout portlets

The Layout portlets are simple JSR 168 portlets that implement 2 Jetspeed specific features:

- the LAYOUT mode and its associated request and response attribute names (or the LayoutPortlet interface if you think a custom mode is too cumbersome)
- they need to know how to get hold of the ContentDispatcher, either through a request attribute (possibly the simplest way) or through the RenderService.