XPages: Make your PhaseListener Thread Safe

December 18, 2013

When implementing a JavaServer Faces (JSF) based framework for your XPages, you may decide to use a PhaseListener, or two, or three, or any number you want. PhaseListeners can be very powerful tools to help perform centralized processing for your application. Before we dive into the main topic of this article, it may be helpful to provide a base understanding of PhaseListeners and how to define one. If you already know how to use a PhaseListener then feel free to jump the next section.

What is a PhaseListener and how would I use it?

As JSF processes requests it fires specific events. These events are called Phase Events. These are fired in conjunction with the JSF Life Cycle. The Life Cycle consists of 6 phases:

- Restore View
- Apply Request Values
- Process Validations
- Update Model Values
- Invoke Application
- Render Response

JSF will fire an event both before and after every phase in the Life Cycle.

Using a PhaseListener allows you to capture the processing before and after each of the phases, and run your custom logic. The PhaseListener won't allow you to interact directly with the phase as it fires though; only before and after.

Caveat: It is possible that not all phases will get fired. JSF will skip phases depending on what is happening. For example, if there is no query data to process, JSF skips from Restore View directly to Render Response (bypassing the four phases in between). Or if validation fails during Process Validations, JSF skips directly to Render Response and does not update the model or invoke the application (the method bound to the UI actions). When determining when to fire your

PhaseListener, you should have a good handle on what phases will/could be fired by JSF based on what beans are being processed at that time and the potential state of the objects bound to the bean.

So, you may be wondering how you would take advantage of a PhaseListener. Well, lots of ways!

For example:

- A login control PhaseListener. Imagine you have an application that implements custom login functionality unrelated to the standard Domino ACL authentication (remember, JSF is not Domino specific). You may have some pages that require the user to be "authenticated" before they can access the information. Or there may be some pages that only require login under certain conditions. In these security scenarios a PhaseListener is a perfect solution. Simply have the PhaseListener check the view ID against a list of known secure IDs and test a bean (or two or three) for whatever condition requires authentication, and if required, redirect the user to the login page.
    - *XPages Caveat:* It is actually a bit more "standard" to use a servlet filter for this task, but Domino does not provide this functionality for our applications. However, in the Domino world, since it is not possible to add standard servlets anyway, the lack of a servlet filter is more of a nuisance than issue. We just have to be creative when moving code that would normally go in a filter, to a PhaseListener.
- A navigation handler for custom business rules. Imagine this scenario. You have a web site that tracks profiles of users (or customers). You add a new required piece of information to the user profile. As people login you warn them by some specific date the user will have to have updated their profile and fill out the additional information. Then the date arrives. How do you force them to fill out the information before using your website? PhaseListener to the rescue. Simple create a PhaseListener that tests the bean with the user data. If the loaded profile is missing the required data, force a redirect to their profile page. You can create the PhaseListener to constantly force them to their profile no matter what. You don't have to add code to every single page of your site testing the bean. The single PhaseListener does the job for you.
- Debug. A PhaseListener is one of the best tools you will have in your debug arsenal as you develop your JSF based applications. Because they can be fired before and after every single phase, it is easy to create a utility PhaseListener that performs some debug for you. Testing bean values and outputting messages (System.out or some advanced logging) will help you find out what your code is doing through the phases vs. what you expected your code to be doing through the phases.

As you read through the rest of this article you will not have to ever have created a PhaseListener to understand the Java Concurrency (threading) issues that arise when using PhaseListeners. All you have to remember is that a PhaseListener, if implemented, will always fire at the phase(s) you define (within the standard JSF Life Cycle rules).

On to the heart of this article.

Threads and Scopes and Singletons

Java code (including JSF) processes in threads. For our JSF applications, when a page is processed, the application server (Domino in our case) creates one (or more depending on your code) processing threads. Every user has a unique set of threads specific to their session during the processing.

In JSF, we have become accustomed to hearing about bean scopes. It helps to understand scopes when thinking about thread safety:

- Application. Objects in this scope can be seen and potentially modified by all users of the application (depending on how you wrote the scoped item). Items in this scope survive the entire life of the application while the HTTP server is running. You can re-create items and implement custom code to change the items, but the items are available across all sessions.
- Session. Objects in this scope are specific to each user and cannot be seen by other users; unless you code something to move objects between users. These objects survive the entire time the user is active on your website and only disappear if you explicitly destroy (invalidate) the user's session or the session expires due to inactivity.
- View. Objects are specific to the users session and survive only for as long as the user stays on (or reloads) the same web page. Once a user navigates to a new page, objects from the previous view are destroyed by the application server.
- Request. Objects are specific to the user's session and survive only the request – response cycle. Once the response is complete, the objects are destroyed by the application server.

Java, along with other code (ex. C++), implement a design pattern called the Singleton. The simplest definition we can provide (courtesy of Wikipedia) is:

"The singleton pattern is a design pattern that restricts the Instantiation of a class to one object".

If properly implemented, once a singleton object is instantiated, it is impossible to have additional objects of the same class instantiated. All subsequent attempts will refer to the same object in memory.

Here is an example of a simple singleton:

```
public class MySingleton {

	private static MySingleton mySingleton;

	private String myValue = "Initial Value";

	/**
	 * Only allow class to instantiate itself
	 */
	private MySingleton() {
		//Do nothing
	}

	/**
	 * Get the instance of the singleton
	 * @return The singleton instance
	 */
	public static MySingleton getMySingleton() {
		//Only create it if not already created, otherwise return the single created object
		if (mySingleton == null) {
			mySingleton = new MySingleton();
		}
		return mySingleton;
	}

	public String getMyValue() {
		return myValue;
	}
```

```
        public void setMyValue(String myValue) {
                this.myValue = myValue;
        }

}
```

Look at the method "getMySingleton". It only creates a new instance one time (if none already existed). Subsequent calls to the method will return the first created instance and never again create a new instance. All clients of this class get the single one and only MySingleton that was created at the first call of the method getMySingleton. And because it holds an instance reference to itself, it never gets garbage collected because it is always using itself. Hence, there is only ever one single object that never gets recreated for the entire life of the application in memory (aka, a Singleton).

If you look deeper into the class, you see some public getters and setters. Being a singleton, all client classes that use this object will all see the same value for the field "myValue". For example, if client object A and client object B both call "getMySingleton", then each initially see the value "Initial Value" for the field "myValue" if they invoke its getter method. Then, if client object B, changes the value to "New Value", client A will now see "New Value" when calling the getter method. Furthermore, if another client C object comes along and calls the "getMySingleton" method followed by "getMyValue", client C will get the value "New Value" NOT the value "Initial Value". This is because there is only one (single) object ever in memory because "getMySingleton" only ever creates one item and the private constructor will not allow client code to create new objects.

Note: Private variables created inside a method are automatically thread safe. Unlike class level fields of a Singleton, class methods of a Singleton are processed within each thread separately and do not share values. So any lists, strings, etc… created for client A inside a method that are only method level variables, are not visible to client B, C, D, etc…

So you are now wondering what does thread safety have to do with scopes and singletons and PhaseListeners? Well, everything. That's why we are writing this blog article for you.

Why do I care about threads, scopes, singletons, and thread safety in my PhaseListener?

Knowing all of the other topics previously covered in this article; if you remember anything about a PhaseListener remember the following sentence:

A PhaseListner is an application wide Singleton!

Yes that's correct, all users of your application only ever see the very first instantiation of each PhaseListener and all users will see the same value of class level fields as they change. PhaseListeners are NOT session specific. To be clear, a PhaseListener is not specifically application scope as we know it. PhaseListeners are never stored in the application map, but instead they are instantiated by the application server running the JSF framework implementation and held in memory for access by all code (across all sessions). It behaves like an application scope object, but does not exist in the application map.

Hopefully bells and whistles are going off in your head. Maybe you are thinking things like:

- If I obtain a bean from the session map, and store it in a class level field, other sessions can now see and manipulate that bean. And, the same idea goes for all the other scopes.

- Any secure information my code writes to a class level field of a PhaseListener is now visible to other user's sessions.
- Basically, anything I put in a class level field is exposed to all sessions!

All those thoughts, and more, should be alarming and force you to analyze how you implement your PhaseListener.

This does not mean PhaseListeners are bad. Not in any way. They are great. The singleton approach simply means a PhaseListner is simultaneously powerful and dangerous. You just really have to know what you are doing.

Make my PhaseListener Thread Safe

Java concurrency (asynchronous processing across multiple threads) is a huge topic. There are many excellent books that are dedicated to and/or devote chapters to, the idea of concurrency. There is no way we can cover all of that content here (a lot of theory, best practice, etc...). But what we can do is provide some general guidance and tips on how to implement PhaseListeners which keep your threads safe from each other.

Let's take a look at a thread "unsafe" PhaseListener.

```
public class MyPhaseListener implements PhaseListener {

	private static final long serialVersionUID = 1L;

	private List myObjectsToProcess;

	/*
	 * (non-Javadoc)
	 * @see javax.faces.event.PhaseListener#beforePhase(javax.faces.event.PhaseEvent)
	 */
	@Override
	public void beforePhase(PhaseEvent event) {
		myObjectsToProcess = new ArrayList();

		//Retrieve the users session bean from the session map
		MySessionBean mySessionBean = (MySessionBean) event.getFacesContext()
				.getExternalContext().getSessionMap().get("mySessionBean");

		/*
		 * Add objects to be processed after the phase is complete
		 * but before the values are applied because we want to process the objects based on
		 * the values before the the user updated them on the UI
		 */
		for (String processingRule : mySessionBean.getProcessingRules()) {
			if (StringUtils.isNotBlank(processingRule) && processingRule.equalsIgnoreCase("process objec
				myObjectsToProcess.add(new MyObject());
			}
		}

	}

	/*
	 * (non-Javadoc)
	 * @see javax.faces.event.PhaseListener#afterPhase(javax.faces.event.PhaseEvent)
	 */
```

```
        @Override
        public void afterPhase(PhaseEvent arg0) {
                if (myObjectsToProcess.size() > 0) {
                        //Perform custom business logic to process the objects for the current user
                        //.....custom logic here

                        //Clear the list when done
                        myObjectsToProcess.clear();
                }
        }



        /*
         * (non-Javadoc)
         * @see javax.faces.event.PhaseListener#getPhaseId()
         */
        @Override
        public PhaseId getPhaseId() {
                return PhaseId.APPLY_REQUEST_VALUES;
        }

}
```

This particular example is set to fire for the Apply Request Values phase. Before the phase begins, there is code to load up a list with some custom objects. After the phase completes, there is additional code to process those objects. The code illustrates a scenario where the bean contains some processing rule values that may change upon the submission of a page. So the listener is meant to process the rules based on the values before the request values are applied, but also may need to use some of the changed values after the request values are applied. So the rules are captured before the phase begins and the processing occurs after the phase completes. The example here is not of any specific use except to illustrate how unsafe this listener is.

Hopefully what stands out is the vulnerability of the list. That list is currently mutable. And because a PhaseListner is a singleton, the list is exposed to every user across all threads.

Watch the value of the list as the listener fires for two users simultaneously:

1) User A has 3 processing rules in its list of strings from “getProcessingRules()”.

2) User B has 5 processing rules in its list of strings from “getProcessingRules()”.

3) When the beforePhase fires for User A, the list “myObjectsToProcess” will be loaded with 3 objects to process.

4) When the beforePhase fires for User B, the list will be reset and 5 objects to process will be loaded.

5) Next, the PhaseListener fires the afterPhase for User A. But instead of 3 MyObjects in the list, there are the 5 MyObjects because User B had fired the beforePhase before User A had a chance to process the afterPhase. Not only are there 5 instead of 3, if there was code that manipulated data based on specific user characteristics, User B would be reading and processing User A data. To make matters worse, what if the code stored the manipulated data back into the MyBean and displayed it to the user? User B would suddenly see secure information from User A. Ouch!

Some of you may be thinking, *what are the odds that these two threads will be processing so close together that this happens. I mean, the code will process in nanoseconds*. Well, we are providing you this article not based on a hypothetical scenario, but in fact, a real scenario were we spent a lot of time debugging an issue where our session beans were suddenly losing values. The culprit…someone had implemented a PhaseListner that used a class level mutable field and multiple threads were in it at the same time.

Ok, so this is a very unsafe PhaseListener, how do we make it safe?

Techniques for Thread Safety

As mentioned earlier, Thread Safety and Java Concurrency is a broad topic. But in respect to PhaseListeners, we can narrow down the conversion to best practice approaches.

Option 1 – make your class fields "final" if you can

If you have a class level field, declaring it as final (immutable) will insure the value never changes. After its first value is assigned it is not possible to have any code change it for the life of the Singleton. But for our scenario above, that won't work since we need our list to be volatile (the information changes based on the user).

Caveat: Making a field final only marks the object reference as unchangeable (remember Java stores the reference to the object in the field, not the values of the object). The contents of the object (its getters, setters, etc…) are not final unless you add code to make the object final. And for lists, sets, maps, etc…, even though you mark the list as final, content can be added, removed, and cleared, because only the reference to the list is final, not the contents of the list.

Option 2 – synchronize method level block code (but not really good for busy PhaseListeners)

When your data must be volatile (mutable) and making it final is not a solution, the Java approach is to synchronize methods and / or block code inside of methods. Synchronizing the code tells Java that all threads that share the code will basically get in line and wait for the code to be available.

Ok, earlier I mentioned that for a Singleton, method code is processed separate from each thread and variables inside the methods are automatically thread safe (each thread does not see the other). But by synchronizing the code, you force all threads to see that another thread is currently running the code. The variables inside the code are still thread safe, but all threads are now aware that another thread is currently processing that block of code and must wait until the code is free. This is useful when the block of code has class level fields visible to all threads.

To synchronize a block of code inside a PhaseListener, you would surround the block with:

```
synchronized(this) {

}
```

All code you put between the braces will be synchronized and only one thread at a time can process that code, the other threads are waiting (thread wait state). This means that a thread can manipulate a class level field without interfering with another thread because the other thread can't access that code to do its own manipulation, until the first thread is complete. *Caveat Alert!*

Ok, the "but…"

First, for our scenario, that does not work. We manipulate the list in two different places. So when a thread is done with beforePhase, the next thread waiting will immediately start processing once the first thread is complete (and before the afterPhase fires).

Second, and even more importantly, synchronizing code in a PhaseListener is only good for applications with a small amount of users and code that processes quickly. Synchronizing code means that only one thread at a time can process the code. For a large application or public website, that means all of the users suddenly get in line (first come first serve) and wait for the code. Imagine 1000+ simultaneous users suddenly all having to merge into one long line. Another way to think about it would be a huge highway that had hundreds of lanes suddenly all converging onto a single lane road…the traffic jam would be epic!

Ok, so more often than not the chances are you can't synchronize the code in a PhaseListener.

Option 3 – Set a context attribute or work directly with the session map

Remember, our application is a web application and the point of this article is to help make your PhaseListener thread safe. So let's see what we have available…the underlying HttpServlet.

Every FacesContext is unique to the thread at process time with its own HttpServletRequest. So we can simply move our class level fields to method level variables and then store them to a request attribute like this (in the beforePhase):

```
List myObjectsToProcess = new ArrayList();
//…processing code here
HttpServletRequest req = (HttpServletRequest)event.getFacesContext().getExternalContext().getRequest();
req.setAttribute("myObjectsToProcess", myObjectsToProcess);
```

Then in the afterPhase, we can get the list back and process it:

```
HttpServletRequest req = (HttpServletRequest)event.getFacesContext().getExternalContext().getRequest();
List myObjectsToProcess = (List)req.getAttribute("myObjectsToProcess");
```

That is 100% thread safe and will isolate the list to the user's request in their session.

Furthermore, if you have to manipulate temporary data across PhaseListners, just think JSF! You can store anything you want in the session map with code at runtime. You don't have to rely on your faces configuration to define what's going to be in the map. After all, the session map really comes from the underlying architecture and is not JSF specific. It is a regular Java map like every other Java map (map of strings to objects). Just get it and add/remove from it. To get the session map, call:

```
event.getFacesContext().getExternalContext().getSessionMap();
```

That's it. You have modified the thread unsafe PhaseListener to be thread safe and cleared up all those phantom issues because data was passing between sessions or being modified by other sessions. We hope this helps. Check back again to find out what other tips or tricks we have to share with you!

Share:


December 18, 2013 [http://www.pipalia.co.uk/xpages-make-phaselistener-thread-safe/] | by Gary Glickman | in jsf, phaselistener, thread safety, xpages .

4 Comments

4 thoughts on "XPages: Make your PhaseListener Thread Safe"

Cameron Gregor
December 18, 2013 at 11:15 pm

Great article once again, this has answered a few questions I had in my head about thread safety! the articles on this fantastic.

Minor point: I think the example simple singleton is not currently instantiable at present? It needs it's MySingleton class member to be static and the getMySingleton method to be static too?

Gary Glickman Post author
December 19, 2013 at 12:58 am

Hi Cameron,

Thanks for taking the time to read the article. I am glad you enjoyed.

And yes you are correct! Great catch. Sometimes when writing these articles a lot of attention is given to the content and the code samples get sacrificed

Thanks again,
Gary

Stephan H. Wissel
December 19, 2013 at 9:56 am

Actually your Singleton would need a little work. You still could duplicate it using the clone(); method inherited from Object(); Currently the “cleanest” way to implement a Singleton would be:

public enum MySingleton {
INSTANCE;

private String myValue = “start value”;

public String getMyValue() {
return myValue;
}

}

Nice idea with the session map

Gary Glickman Post author
December 19, 2013 at 1:41 pm

Hi Stephan,

Thanks for taking the time to read the article. I’m glad you found the session map idea useful.

On your note about the singleton being cloneable, you may want to look into that a little bit. The singleton is not cloneable as written. The clone() method is not visible nor has it implemented the cloneable interfce. Further, you can’t extend the class because its constructor is private.

If you execute the following code using the singleton class in the article, you would get a compile error at the attempt to clone:

```
public static void main(String[] args) {
MySingleton mySingleton = MySingleton.getMySingleton();

System.out.println(mySingleton.getMyValue());

mySingleton.setMyValue("I changed the value");

System.out.println(mySingleton.getMyValue());

mySingleton.clone();
```

}

Have a look at this Javadoc from Oracle: http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html#clone%28%29

As far as an enum, I agree when implementing a Singleton that is the newer standard. But the point of the article is to show the hole in threading with the PhaseListener implementation of a Singleton, which does it the traditional way.

Thanks again for looking though!
Gary