# RecursiveComponents

NOTE: This is outdated information that applies only to Tapestry 4. Tapestry 5 also does not support recursive components. See

https://issues.apache.org/jira/browse/TAP5-739

A component whose template calls itself again causes a stack overflow (due to an infinite loop) when the Tapestry parser encounters it. This is well documented many places. A couple of pages talk about a workaround using Block and RenderBlock, but they have a couple of problems. This page contains a complete example I have working with Tapestry 4.1 combining ideas from the two main examples found elsewhere, but with one major addition to work around a side effect they don't mention.

## Inspirational Examples

1. http://www.behindthesite.com/blog/C1931765677/E923478269/index.html

This example shows the basic idea, but isn't really contained and reusable.

2. http://blogs.encodo.ch/news/view_article.php?id=15 This has an elegant, simple component definition containing both a starting Render{{`Block and the Block it renders that then contains another Render}}`Block to render itself. This is the primary example I based my solution on.

## Problems, Caveats, and Shortcomings

Example 1 has the major drawback of requiring inclusion of the block(s) that define the body of the recursive component in the body of each page that uses it - this isn't really a reusable component design. Attempts to put the block in a dummy page referenced by the component still caused stack overflows for me.

Example 2 is cleaner, but not a working example - their code has typos, and doesn't mention a very important caveat. This example uses a component level property to hold the child object For component iteration value. This appears fine, but the For component does (or doesn't do) something not documented - it doesn't remember and reset the state of the property used for the value and index bindings before and after iterating over the source values. This means that any recursive Block body contents that reference this component level property after the recursive For component has used it for it's value binding will see not the value of the property in the expected scope.

This is quite confusing, and took me a while to figure out. The problem relates to how OGNL evaluates references and the fact that the sample needs a component property to store the child For value object passed to the recursive `RenderBlock` call. Each level of Block rendering looks for it's input data via `getProperty()`, which is defined in terms of the For component value binding, which is set to the component property. There is only one instance of the component property for the whole recursive rendering, and the default behavior of the For component results in that property remaining set to the last value found in the last iteration of the lowest level of recursion once the recursive calls start to unwind.

Thus components in the Block body after the recursive For component do not see the Block input parameter for that block's original call, but the last value the underlying component property was set to by the last iteration of the lowest recursion.

# Solution Code

This example is for a basic Java{{`Bean called }}MessageNode{{ containing a Collection of }}MessageNode{{ objects (}}getChildren(){{) and a Collection of String messages (}}getMessages(){{). Rendering is as nested unordered lists, with messages as list item elements. Messages are rendered after children, which is where we encounter the }}For` component value problem.

### Template

```
<span jwcid="$content$">
        <span jwcid="@RenderBlock" block="ognl:components.treeBlock" node="ognl:root"/>

        <span jwcid="treeBlock@Block">
                <span jwcid="forChildren">
                        <span jwcid="@RenderBlock" block="ognl:components.treeBlock" node="ognl:child" />
                </span>
                <ul jwcid="forMessages">
                        <li jwcid="messageItem">Inserted message text</li>
                </ul>
        </span>
</span>
```

### JWC Spec

```
<component-specification class="MessageTree" allow-body="no" allow-informal-parameters="no">
	<description>
		This renders a tree of MessageNode objects as nested lists
	</description>

	<parameter name="root" required="yes">
		<description>
			Root MessageNode object to render
		</description>
	</parameter>

	 <property name="message"/>
	<property name="child"/>

	<component id="forChildren" type="For">
		<binding name="renderTag" value="false"/>
		<binding name="source" value="blockNode.children"/>
		<binding name="value" value="child"/>
	</component>

	<component id="forMessages" type="For">
		<binding name="renderTag" value="true"/>
		<binding name="source" value="blockNode.messages"/>
		<binding name="value" value="message"/>
	</component>

	<component id="messageItem" type="Insert">
		<binding name="renderTag" value="true"/>
		<binding name="value" value="message.toString()"/>
	</component>

</component-specification>
```

**Component Java Class**

```
public abstract class MessageTree extends BaseComponent {
	// root parameter from spec
	public abstract MessageNode getRoot();

	// This needs to be in the class so OGNL evaluation when parsing the spec doesn't cause an infinite
loop!
	public MessageNode getBlockNode() {
		return (MessageNode) ((Block) getComponent("treeBlock")).getParameter("node");
	}
}
```

## Analysis

This example looks pretty simple and straight forward at first, but when executed, what you get is the messages for the last leaf node on each branch of the object tree rendered as the messages for each parent node of that leaf. This is because of the aforementioned problem with the `For` component.

## Solution: A custom ForContext Component

To solve the problem, we need a component that acts like the standard `For` component but has the added behavior of remembering it's value and index initial state and reseting those after rendering it's iterations. This is similar to what the `Block` component does with it's `invoker` property, and why the `Block` component can be used recursively as we are doing here.

The custom code is quite simple - just copy the For.jwc spec file (changing the class of course), and create a component class like this:

```
public abstract class ForContext extends ForBean {
	/**
	 * Render the same as the standard ForBean, but remember initial index and value binding values if any,
	 * and reset them when done rendering.
	 *
	 * @see org.apache.tapestry.components.ForBean#renderComponent(org.apache.tapestry.IMarkupWriter,
	 *      org.apache.tapestry.IRequestCycle)
	 */
	@Override
	protected void renderComponent(IMarkupWriter writer, IRequestCycle cycle) {
		Object value = null;
		int index = 0;

		IBinding valueBinding = getBinding("value");
		if (valueBinding != null) value = valueBinding.getObject();

		IBinding indexBinding = getBinding("index");
		if (indexBinding != null) index = ((Integer) indexBinding.getObject()).intValue();

		try {
			super.renderComponent(writer, cycle);

		} finally {
			if (indexBinding != null) indexBinding.setObject(new Integer(index));

			if (valueBinding != null) valueBinding.setObject(value);

		}
	}
}
```

## Conclusion

Now we can replace the `For` components in the component spec above with `ForContext`, leaving everything else the same. When we render a tree of messages, we now get the proper message lists at every level, because the new component remembered and reset the state of the property used for the value and index bindings (if any) when it was done.

I think this is really a bug with the `ForBean` class, but changing that would probably break something that was depending on it's current behavior, expecting to see the component or page property still referencing the last collection value after the `ForBean` was done rendering.

## Additional Problem with "If" Component

If you want to enhance your recursive block with additional template logic, perhaps optionally rendering additional content for some iterations, you will want to use the `If` component. However, there is a bug in the template parsing code when it is used in this context. The condition binding is improperly cached between recursive calls, with the net result of the condition not being reevaluated for the first child iteration after the top-level call.

The solution is to use a custom `.jwc` spec file, calling the component something else (I used "IfContext"). In this file, add the `cache="no"` attribute to the `condition` and `conditionValue` parameters to force Tapestry to evaluate their expressions on each access.