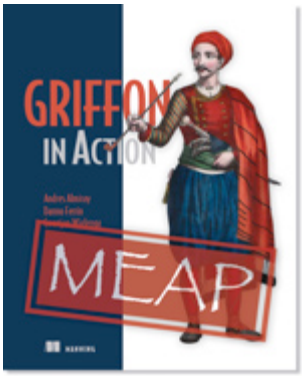
MANNING PUBLICATIONS

Groovy SwingBuilder and Threading

Excerpted from



Griffon in Action

EARLY ACCESS EDITION

Andres Almiray and Danno Ferrin
MEAP Began: March 2009
Softbound print: Summer 2010 | 375 pages
ISBN: 9781935182238

This article is taken from the book Griffon in Action. *Groovy SwingBuilder has a lot to offer in terms of threading goodness, and it makes the job of properly handling threading on a Swing application much easier. This article explains why a lack of proper threading in your applications is a problem, shows a possible solution, and then takes a closer look at SwingBuilder's threading options. While not Griffon-centric, this article's implications are, nevertheless, very important.*

If you've had any experience with Groovy SwingBuilder, then you know SwingBuilder reduces visual clutter and increases readability. But relying on SwingBuilder alone doesn't mean you're safe from shooting yourself in the foot. Groovy can work its magic, but you have to give it a few nudges in the right direction from time to time.

In this article, we'll journey into the threading capabilities that SwingBuilder exposes. To start, let's delve into the problems caused when an application lacks proper threading.

Groovy Swing without threading

Consider the following Java code for a simple file viewer (see listing 1).

Listing 1 Java version of SimpleFileViewer

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.*;

public class SimpleFileViewer extends JFrame {                    1
  public static void main(String[] args) {
    SimpleFileViewer viewer = new SimpleFileViewer();             5
  }

  private JTextArea textArea;
  private static final String EOL = System.getProperty("line.separator");
```

```
  public SimpleFileViewer() {
    super("SimpleFileViewer");
    buildUI();
    setVisible(true);
  }

  private void buildUI() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JButton button = new JButton("Click to select a file");
    textArea = new JTextArea();
    textArea.setEditable(false);
    textArea.setLineWrap(true);
    button.addActionListener(new ActionListener(){                    2
      public void actionPerformed(ActionEvent event) {
        selectFile();                                                 3
      }
    });
    getContentPane().setLayout( new BorderLayout() );
    getContentPane().add(button, BorderLayout.NORTH);
    getContentPane().add(new JScrollPane(textArea), BorderLayout.CENTER);
    pack();
    setSize( new Dimension(320, 240) );
  }

  private void selectFile() {
    JFileChooser fileChooser = new JFileChooser();
    int answer = fileChooser.showOpenDialog(this);
    if( answer == JFileChooser.APPROVE_OPTION ) {
      readFile(fileChooser.getSelectedFile());
    }
  }

  private void readFile( File file ) {                                 4
    try {
      StringBuilder text = new StringBuilder();
      BufferedReader in = new BufferedReader(new FileReader(file));
      String line = null;
      while( (line = in.readLine()) != null ) {
        text.append(line).append(EOL);
      }
      textArea.setText(text.toString());
      textArea.setCaretPosition(0);
    } catch( IOException ioe ) {
      ioe.printStackTrace();
    }
  }
}
```

#1 Creates a subclass of javax.swing.JFrame
#2 Defines an event handler on a javax.swing.JButton
#3 Executed inside EDT
#4 Does not care about EDT
#5 Instantiated outside EDT

Now, consider the same simple file viewer, but let's switch to Groovy (see listing 2). The code's design is pretty much the same, apart from using composition instead of inheritance from the get-go.

Listing 2 Groovy version of SimpleFileReader

```
import groovy.swing.SwingBuilder
import javax.swing.JFrame
import javax.swing.JFileChooser

public class GroovyFileViewer {
  static void main(String[] args) {
    GroovyFileViewer viewer = new GroovyFileViewer()                  1
  }
```

```
  private SwingBuilder swing

  public GroovyFileViewer() {
    swing = new SwingBuilder()
    swing.fileChooser(id: "fileChooser")
    swing.frame( title: "GroovyFileViewer",
                 defaultCloseOperation: JFrame.EXIT_ON_CLOSE,
                 pack: true, visible: true, id: "frame" ) {
      borderLayout()
      button("Click to select a file", constraints: context.NORTH,
             actionPerformed: this.&selectFile)                          2
      scrollPane( constraints: context.CENTER ) {
        textArea( id: "textArea", editable: false, lineWrap: true )
      }
    }
    swing.frame.size = [320,240]
  }

  private void selectFile( event = null ) {
    int answer = swing.fileChooser.showOpenDialog(swing.frame)
    if( answer == JFileChooser.APPROVE_OPTION ) {
      readFile(swing.fileChooser.selectedFile)
    }
  }

  private void readFile( File file ) {
    swing.textArea.text = file.text                                      3
    swing.textArea.caretPosition = 0
  }
}
```

#1 Instantiated outside EDT
#2 Conversion of method into closure
#3 Executed inside EDT

Cueballs #1-3 in code and text

Comparing lines of code, listing 2 is 20 lines shorter than listing 1, but it still lacks proper threading support. The application is still being instantiated in the main thread (#1), which means UI components are also being instantiated and configured on the main thread. (#2) shows a shortcut to obtain a closure from an existing method. This is not a problem; it is a handy alternative to declaring an inline closure. Back to the application□s business end (#3), the file is read in the current executing thread (that would be the EDT again) and the textArea is updated in the same thread.

Before we get into SwingBuilder threading proper, let's take a look at what the Groovy language alone has to offer.

Groovy Swing with threading

The ability to use closures in Groovy is a big selling point to many developers, but the ability to use Groovy closures with plain Java classes is perhaps the most compelling reason to switch.

Consider the Java implementation for a revised version of our simple file viewer (see listing 3), which now implements threading.

Listing 3 Java version of RevisedSimpleFileViewer with threading taken into account

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.*;

public class RevisedSimpleFileViewer {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {                          1
```

```
      public void run() {
        RevisedSimpleFileViewer viewer = new RevisedSimpleFileViewer();
      }
    });
  }

  private JTextArea textArea;
  private JFrame frame;
  private static final String EOL = System.getProperty("line.separator");

  public RevisedSimpleFileViewer() {                                    2
    frame = new JFrame("RevisedSimpleFileViewer");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().setLayout(new BorderLayout());
    frame.getContentPane().add(buildUI(), BorderLayout.CENTER);
    frame.pack();
    frame.setSize( new Dimension(320, 240) );
    frame.setVisible(true);
  }

  private JPanel buildUI() {
    JPanel panel = new JPanel(new BorderLayout());
    JButton button = new JButton("Click to select a file");
    textArea = new JTextArea();
    textArea.setEditable(false);
    textArea.setLineWrap(true);
    button.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent event) {
        selectFile();                                                   3
      }
    });
    panel.add(button, BorderLayout.NORTH);
    panel.add(new JScrollPane(textArea), BorderLayout.CENTER);
    return panel;
  }

  private void selectFile() {
    JFileChooser fileChooser = new JFileChooser();
    int answer = fileChooser.showOpenDialog(frame);
    if( answer == JFileChooser.APPROVE_OPTION ) {
      readFile(fileChooser.getSelectedFile());
    }
  }

  private void readFile( final File file ) {
    new Thread(new Runnable(){                                          4
      public void run() {
        try {
          final StringBuilder text = new StringBuilder();
          BufferedReader in = new BufferedReader(new FileReader(file));
          String line = null;
          while( (line = in.readLine()) != null ) {
           text.append(line).append(EOL);
          }
          SwingUtilities.invokeLater(new Runnable() {                   5
            public void run() {
              textArea.setText(text.toString());
              textArea.setCaretPosition(0);
            }
          });
        } catch( IOException ioe ) {
          ioe.printStackTrace();
        }
      }
    }).start();
  }
}
```

#1 Runs the code inside EDT

#2 Uses a composed JFrame
#3 Executed inside EDT
#4 Reads the file outside EDT
#5 Updates the UI inside EDT

Take a moment to try to visualize the behavior of this code behind the verbosity of its implementation. Ready? Now let's look at the Groovy version of the same code. Listing 4 shows a rather different picture.

Listing 4 Groovy-enhanced readFile()

```
private void readFile( File file ) {
  Thread.start {                                                    1
    String text = file.text
    SwingUtilities.invokeLater {                                    2
      swing.textArea.text = text
      swing.textArea.caretPosition = 0
    }
  }
}
```

#1 Spawns a new thread
#2 Runs the code inside EDT

Cueballs #1-2 in code and text

We can almost hear you shouting in a fit of anger and disbelief: "Not Fair!" (That is, if you are one of the countless developers that have been bitten by Swing threading problems in the past.) Otherwise, you might be saying, "I didn't know you could do that!"

Among the several tricks Groovy has in its arsenal, one is very helpful with threading. You see, Groovy is able to translate a closure into an implementation of Java interface that defines a single method. In other words, Groovy can create an object that implements the `Runnable` interface using a closure as implementaion. The behavior of the runnable's `run()` method is determined by the closure's contents.

Back to listing 4, a new thread is created at (#1), the file's contents are read inside that thread, then we□re back into the EDT at (#2) to update the textArea's properties.

This article would be over right now if these were the only things Groovy had to offer regarding threading, but luckily that is not the case. SwingBuilder, too, has some tricks up its sleeve.

Threading with SwingBuilder

Listing 5 shows the full rewrite of our enhanced simple file viewer using SwingBuilder's threading facilities.

Listing 5 SwingBuilder threading applied to GroovyFileReader

```
import groovy.swing.SwingBuilder
import javax.swing.JFrame
import javax.swing.JFileChooser

public class RevisedGroovyFileViewer {
  static void main(String[] args) {
    def viewer = new RevisedGroovyFileViewer()                      1
  }

  private SwingBuilder swing

  public RevisedGroovyFileViewer() {
    swing = new SwingBuilder()
    swing.edt {                                                     2
      fileChooser = fileChooser()                                   3
      frame( title: "RevisedGroovyFileViewer",
             defaultCloseOperation: JFrame.EXIT_ON_CLOSE,
             pack: true, visible: true, id: "frame" ) {             4
        borderLayout()
```

```
        button("Click to select a file", constraints: context.NORTH,
               actionPerformed: this.&selectFile)
        scrollPane( constraints: context.CENTER ) {
          textArea( id: "textArea", editable: false, lineWrap: true )
        }
      }
      frame.size = [320,240]
    }
  }

  private void selectFile( event = null ) {
    int answer = swing.fileChooser.showOpenDialog(swing.frame)          5
    if( answer == JFileChooser.APPROVE_OPTION ) {
      readFile(swing.fileChooser.selectedFile)
    }
  }

  private void readFile( File file ) {
    swing.doOutside {                                                   6
      String text = file.text
      doLater {                                                         7
        textArea.text = text
        textArea.caretPosition = 0
      }
    }
  }
}
```

#1 Instantiated outside EDT
#2 Conversion of method into closure
#3 Executed inside EDT
#4 Frame is set in the builder's context
#5 FileChooser is set on the builder's context
#6 Execute the code outside EDT
#7 Execute the code inside EDT

Cueballs #1-7 in code and text

Judging by (#1), the application is still being instantiated on the main thread. This would mean UI components are also being instantiated in that thread, but (#2) says otherwise. At that particular line, SwingBuilder is being instructed to run the code by making an synchronous call to the EDT, thus assuring that UI building is done in the correct thread. Notice at (#3) and (#4) that because fileChooser and frame variables are tied to the builder□s context there is no longer a need to define external variables to access those components if you keep a reference to the builder (which we do). This design choice is taken into account at (#5) where the builder□s instance is used to get a reference to both fileChooser and frame.

The interesting bits—besides (#2)—can be found at (#6) and (#7). Calling `doOutside{}` on SwingBuilder has pretty much the same effect as calling `Thread.start{}` except there is a slight but important difference: the SwingBuilder instance is used as the closure□s delegate. This means you□ll be able to access any variables tied to that particular instance and any SwingBuilder methods inside the closure□s scope. That is why the next call to a SwingBuilder threading method (#7) doLater{} does not need to be prefixed with the swing variable.

But wait—we can go a bit further. We can combine both `selectFile()` and `readFile()` in a single method, and we can also encapsulate the method body with a special Groovy construct (see listing 6).

Listing 6 Simplified version of selectFile() and readFile()

```
  private void selectFile( event = null ) {
    swing.with {
      int answer = fileChooser.showOpenDialog(frame)
      if( answer == JFileChooser.APPROVE_OPTION ) {
        doOutside {
          String text = fileChooser.selectedFile.text
          doLater {
```

```
                textArea.text = text
                textArea.caretPosition = 0
              }
            }
          }
        }
      }
```

This is quite the trick. By using the with{} construct you are instructing Groovy to override the closure's delegate. In this case, its value will be the SwingBuilder instance our little application is holding. This means that the closure will attempt to resolve all methods and properties not found in `RevisedGroovyFileViewer` against the SwingBuilder instance. That is why the references to fileChooser and name, as well as method calls to `doOutside{}` and `doLater{}` need not be qualified with the SwingBuilder instance. Sweet!

Now let's take a moment to further inspect SwingBuilder's threading facilities, shall we?

Synchronous calls with edt{}

We have already established that code put in `edt{}` is executed directly in the EDT. That is, no event is posted to the EventQueue. It appears there is no difference whatsoever between `edt{}` and explicitly calling `SwingUtilities.invokeAndWait{}`. However, in reality, there are three very important things to consider.

- The first convenient improvement found in `edt{}` is that the current SwingBuilder instance is set as the closure's delegate, so you can call any SwingBuilder methods and nodes directly, without needing to qualify them with an instance variable.

- The second convenience has to do with a particular rule of executing code in the EDT. You see, once you are executing code inside the EDT, you can't make an explicit call to `SwingUtilities.invokeAndWait{}` again, or a nasty exception will be thrown. Let's see what happens when we naively make a synchronous call inside the EDT when we are already executing code inside that very thread. Listing 7 shows a trivial example. Remember that the button's ActionListeners are processed inside the EDT.

Listing 7 Violating EDT restrictions

```
import groovy.swing.SwingBuilder
import javax.swing.SwingUtilities

def swing = new SwingBuilder()
swing.edt {
  frame(title: "Synchronous calls #1", size: [200,100], visible: true) {
    gridLayout(cols: 1, rows:2)
    label(id: "status")
    button("Click me!", actionPerformed: {e ->
       status.text = "attempt #1"
       SwingUtilities.invokeAndWait{ status.text = "attempt #2" }       1
    })
  }
}
```

#1 Synchronous call inside EDT

Cueballs #1 in code and text

We were so infatuated with Groovy threading that we forgot for a moment that `SwingUtilities.invokeAndWait{}` cannot be called inside the EDT. When running the application, and after hitting the button—BAM! An ugly exception like the following is thrown:

```
Caused by: java.lang.Error: Cannot call invokeAndWait from the event dispatcher thread
        at java.awt.EventQueue.invokeAndWait(EventQueue.java:980)
        at javax.swing.SwingUtilities.invokeAndWait(SwingUtilities.java:1323)
        at javax.swing.SwingUtilities$invokeAndWait.call(Unknown Source)
```

But if we rely on `SwingBuilder.edt{}` to make the synchronous call at (#1), we get a different result: a working, bug-free application! Check out listing 8.

Listing 8 EDT restriction are not violated anymore

```
import groovy.swing.SwingBuilder

def swing = new SwingBuilder()
swing.edt {
  frame(title: "Synchronous calls #2", size: [200,100], visible: true) {
    gridLayout(cols: 1, rows:2)
    label(id: "status")
    button("Click me!", actionPerformed: {e ->
       status.text = "attempt #1"
       edt{ status.text = "attempt #2" }                              1
    })
  }
}
```

#1 Synchronous call inside EDT

Cueballs #1 in code and text

Much better. The name change is not that hard to remember, is it?

- The third and final thing to consider is that `edt{}` is smart enough to figure out whether it needs to make a call to `SwingUtilities.invokeAndWait{}`. If the currently executing code is already on the EDT then it will simply continue to be executed inside the EDT; `edt{}` makes a call to `SwingUtilities.isEventDispactchThread()` to figure that out.

Making asynchronous calls to the EDT is next.

Asynchronous calls with doLater{}

Posting new events to the EventQueue can be done by calling `SwingUtilities.invokeLater{}`. These events will be processed by the EventQueue the next time it gets a chance, which is why we name this asynchronous calls. The code posted this way may take a few cycles to be serviced depending on the currently executing code inside the EDT and the EventQueue□s state.

Parallel to what we described in the previous section, a call to `doLater{}` is like a call to `SwingUtilities.invokeLater{}` but you can guess there is a small difference. As with `edt{}`, the `doLater{}` method will make sure that the current SwingBuilder instance is set as the closure□s delegate. Again, SwingBuilder methods and properties will already be in scope, so no need to qualify them.

Because this threading facility always posts a new event to the EventQueue, there is nothing much else to see here. There are no calling EDT violations to worry about as in the previous section.

There is one aspect of SwingBuilder threading left to discuss—that of executing code outside of the EDT.

Outside calls with doOutside{}

You may be picking up on a trend here: for every Groovy threading option, there is a SwingBuilder-based alternative that adds a bit of spice, namely the ability to register the current SwingBuilder instance as the closure□s delegate, saving you from typing a lot of repeated identifiers. Well, the trend continues in this case, which means that calling `doOutside{}` has the same effect as `Thread.start{}` with the added benefit of a proper delegate set on the closure.

This threading facility mirrors `edt{}` in the sense that when `doOutside{}` is invoked, it will spawn a new thread if and only if the current thread is the EDT. Otherwise it will call the code in the currently executing thread. This behavior was introduced in Groovy 1.6.3, and previous versions will spawn a new thread regardless of the currently executing thread.