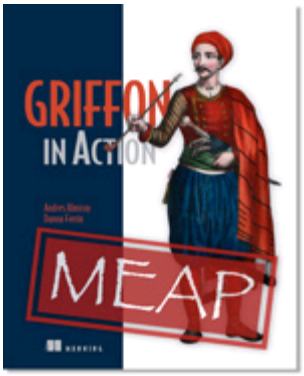
MANNING PUBLICATIONS

Creating a simple tracer plugin/addon

Article based on



[Griffon in Action](#)
EARLY ACCESS EDITION

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

This article is taken from the book Griffon in Action*. The authors discuss creating a plugin/addon as an event handler. They use an AOP-like approach to intercept controller actions and model properties, which allows the programmer to find out when and how the data has changed.*

Everybody knows that, as an application grows, it gets harder and harder to visualize data flows because the user interacts with it. Oftentimes during development, we rely on two techniques to keep track of the data flow: launch the application in debug mode, attach it to a debugger, place some breakpoints at the appropriate places, and see the data live; or, litter the code with `println` statements. However, there is a third alternative: dynamically intercept method calls. Seasoned Java developers may recognize this technique as applying an around advice or a before advice, as suggested by Aspect Oriented Programming (or AOP for short). AOP became very popular in the early 2000s and successfully penetrated the enterprise in tandem with the Spring framework[1]. If AOP is an alien concept to you, don't worry, Groovy greatly simplifies applying AOP-like techniques thanks to its extensive metaprogramming capabilities; this means you do not need to learn an AOP framework nor an AOP API in order to enhance your application.

In this article, we'll use an AOP-like approach to intercept controller actions and model properties. By intercepting a controller action, we'll know when it has been activated. Intercepting model properties will let us know when and how the data has changed.

[1] http://www.springframework.org

We'll package this new behavior with a plugin/addon combination. This way you can install it at any time and then uninstall it when it's no longer needed. This leaves your code untouched, relieving you from launching a debugger.

Bootstrapping the plugin/addon

The first step is to bootstrap the plugin and addon descriptors. You know what's coming, don't you? That's right; we'll use the Griffon command-line tools to create both descriptors. Let's name our plugin Trace, and execute the following commands in the console prompt:

```
griffon create-plugin Trace
cd trace
griffon create-addon Trace
```

You should now have a familiar set of files, similar to those shown in figure 1.

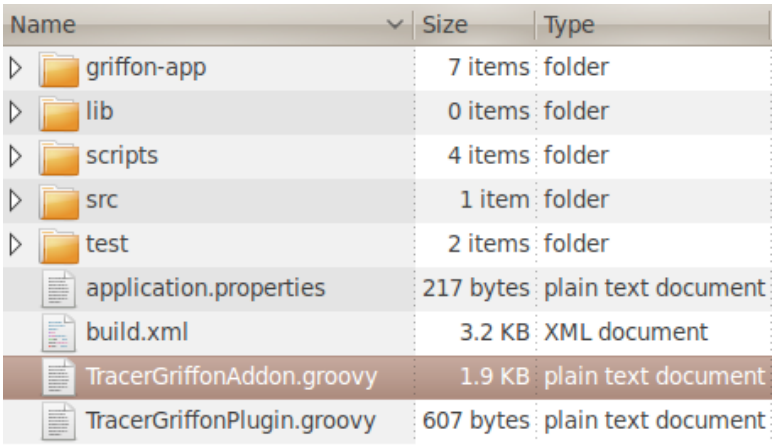


Figure 1 Contents of the Trace plugin. You can see the plugin and addon descriptor files created by the Griffon command line. The addon descriptor is selected.

We'll keep the plugin/addon simple, which means we won't include any external libraries or create additional files. All the behavior will be concentrated in the addon descriptor. Go ahead and modify the plugin descriptor as you see fit, then edit the addon descriptor by adding a skeleton of the behavior we must provide (listing 1).

Listing 1 Initial addon code

```
import java.beans.*
class TracerGriffonAddon {
    def events = [                                              1
        NewInstance: { klass, type, instance ->
                                                                A
        }
    ]

    void message(msg) {                                         2
        println msg
    }
}
```

#1 Event handlers
#2 Prints message to console
A New code to be implemented

This addon relies on an event handler (#1) to intercept and inject new behavior into a recently created instance. The new code is injected into every single instance created by the application regardless of its type or artifact. You can fine-tune this design choice, for example, by restricting the type to controller and model only.

Notice that (#2) defines a very generic message printing method. Alternatives to this implementation would be to use a proper logging mechanism and send the message to a file or even to a database. You have the last word on this design choice too; we're just presenting the very basics. Next, let's add some behavior to our addon.

Intercepting property updates

We'll rely on the fact that observable beans publish change events whenever one of their properties changes value. Java uses `PropertyChangeEvent` and `PropertyChangeListener` to enable change events and their handlers. Adding a `PropertyChangeListener` to the intercepted instance should be enough for now. Edit the addon descriptor once more, making sure its contents match listing 2.

Listing 2 Intercepting property updates on an observable instance

```
import java.beans.*
class TracerGriffonAddon {
    def events = [
        NewInstance: { klass, type, instance ->
            addPropertyChangeListener(instance)                                    #1
        }
    ]

    void addPropertyChangeListener(target) {
        MetaClass mc = target.metaClass
        if(mc.respondsTo(target, 'addPropertyChangeListener', [PropertyChangeListener] as
Class[])) {                                                                        #2
            target.addPropertyChangeListener({ evt ->
                message "${evt.propertyName}: '${evt.oldValue}' -> '${evt.newValue}'"
            } as PropertyChangeListener)                                           #3
        }
    }

    void message(msg) {
        println msg
    }
}
```

#1 Intercepts observable bean
#2 Verifies bean is observable
#3 Adds PropertyChangeListener

At (#1), we add a call to a helper method that will be responsible for inspecting the bean and applying the new behavior. You can see at (#2) that the helper method does not blindly assume that the instance is observable; it checks via the instance's metaclass if it responds to the `addPropertyChangeListener` method. This way we avoid a naughty exception from being thrown at runtime. Finally, at (#3), we define a closure and cast it to `PropertyChangeListener` using Groovy's `as` keyword. This is much better than defining an inline inner class[2], don't you think? We have enough behavior to try out the plugin.

Using the plugin

Before we use the plugin, we need to package it. This, again, is a simple task thanks to the Griffon command line. Type the following in your command prompt:

```
griffon package-plugin
```

After a few lines pass by, you should see a griffon-tracer-0.1.zip file in the current directory. The name of the plugin is computed by convention, and the version number is taken from the plugin descriptor; make sure to update the descriptor file each time you build a newer version.

We can install the plugin now that it has been packaged. You can specify the path to a plugin zip if the plugin is not available from a repository, which is precisely our case. Try installing the plugin on an existing application and

[2] Groovy supports inner class definitions since version 1.7; however, casting a closure is more concise.

see what happens when you run it. For illustration purposes, we'll use a simple yet effective calculator application (shown in figure 2). The application takes two inputs and calculates their sum after you click the Result button. Both inputs, the output, and the button's enabled state are bound to observable properties on the model.

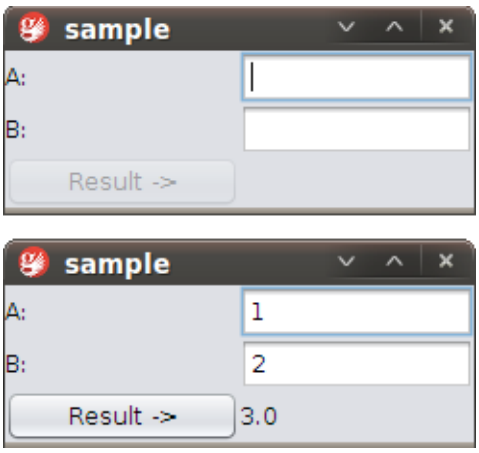


Figure 2 The calculator application. The first screen shows the application as it looks upon launching it. The second screen shows the application after the inputs have been entered and the button has been clicked.

For completeness, the application code is shown in listings 3, 4, and 5. We won't discuss the code thoroughly as our main concern is the Trace plugin. First comes the view, where you can see all bindings as they are set up.

Listing 3 The calculator's view

```
application(title: 'sample',
  pack:true,
  locationByPlatform:true,
  iconImage: imageIcon('/griffon-icon-48x48.png').image,
  iconImages: [imageIcon('/griffon-icon-48x48.png').image,
               imageIcon('/griffon-icon-32x32.png').image,
               imageIcon('/griffon-icon-16x16.png').image]) {
   gridLayout(cols: 2, rows: 3)
   label 'A:'
   textField(columns: 10,
             text: bind(target: model, targetProperty: 'inputA') )
   label 'B:'
   textField(columns: 10,
             text: bind(target: model, targetProperty: 'inputB') )
   button('Result ->', actionPerformed: controller.click,
          enabled: bind{ model.enabled })
   label text: bind{ model.output }
}
```

Next comes the model, shown in listing 4. You can appreciate all property definitions. A local event handler updates the model's `enable` property if both inputs have a value.

Listing 4 The calculator's model

```
import groovy.beans.Bindable
import java.beans.PropertyChangeListener
class SampleModel {
    @Bindable String inputA
    @Bindable String inputB
    @Bindable String output
    @Bindable boolean enabled = false

    SampleModel() {
        addPropertyChangeListener({ evt ->
```

```
            if(evt.propertyName in ['enabled', 'output']) return
            enabled = inputA && inputB
        } as PropertyChangeListener)
    }
}
```

Last comes the controller, shown in listing 5. Notice that it makes use of proper threading to calculate the output outside of the EDT before going back inside the EDT to update the view by setting the model's `output` property.

Listing 5 The calculator's controller

```
class SampleController {
    def model

    def click = {
        model.enabled = false
        String a = model.inputA
        String b = model.inputB
        doOutside {
            try {
                Number o = Double.valueOf(a) + Double.valueOf(b)
                edt { model.output = o }
            } finally {
                doLater { model.enabled = true }
            }
        }
    }
}
```

If the plugin works correctly, we should get a console output every time a model property is updated. Launching the calculator application yields:

```
inputA: 'null' -> ''
inputB: 'null' -> ''
```

Hold on a second! What just happened? Bindings take effect as soon as they are parsed on a view script. The value of each input changes from `null` to an empty string because that is the value of the respective source text field.

Entering values, as shown in figure 2, and clicking the Result button yields the following output in the console:

```
inputA: '' -> '1'
enabled: 'false' -> 'true'
inputB: '' -> '2'
enabled: 'true' -> 'false'
output: 'null' -> '3.0'
enabled: 'false' -> 'true'
```

Perfect! Now we know which values are held by model properties at a certain point during the application's execution without needing to modify the application's code. Perhaps adding a time reference on each output message would be in order. The logging alternative looks more appealing now.

Let's return to the Tracer plugin to see how we can intercept whenever a controller action is called.

Intercepting action calls

Go back to the plugin sources, and edit the addon descriptor. In listing 6, we'll add another intercept handler to the `NewInstance` event handler, similar to what we did previously.

Listing 6 Updated addon descriptor with additional behavior

```
import java.beans.*
class TracerGriffonAddon {
```

```
    def events = [
        NewInstance: { klass, type, instance ->
            addPropertyChangeListener(instance)
            injectActionInterceptor(klass, instance)                                    #1
        }
    ]

    void injectActionInterceptor(klass, target) {
        Introspector.getBeanInfo(klass).propertyDescriptors.each { pd ->
            def propertyName = pd.name
            def oldValue = target."$propertyName"                                       #2
            if(!oldValue?.getClass() || !Closure.isAssignableFrom(oldValue.getClass())) return
            def newValue = { evt = null ->                                              #3
                message "Entering $propertyName ..."
                oldValue(evt)                                                           #4
            }
            target."$propertyName" = newValue                                           #5
        }
    }
}
```

#1 Intercepting the bean
#2 Saving a reference to the action's old value
#3 Defining a new value for the action
#4 Invoking the old action behavior
#5 Assigning the new action value to the bean

The new handler requires both the instance and its class (#1) because we'll use the standard Java Beans inspection mechanism to figure out which properties can be intercepted. This is better than blindly assuming a specific property format or property definition. The handler then inspects the instance's class and iterates over all PropertyDescriptors that the class exposes. The handler performs some metaprogramming magic for each property whose value is a closure, and it skips those that do not match the required criteria. Notice that we save a reference to the current property value (#2). This step is important to be able to call the default behavior later on.

Next, we define the new value for a target action (#3), which turns out to be another closure. Make a note how it relies on the previously saved reference (#4) to the old behavior. This is how you can chain things together. Using Groovy's closures makes these steps a snap. You would have to jump through a few hoops if you used regular Java. The last step (#5) assigns the new behavior to the instance's property. We rely on Groovy's dynamic dispatch capabilities to resolve the actual property name.

But what about the actual intercepting code? It simply calls our `message()` method and then forwards the call to the old behavior. Known as an around advice in AOP terminology, we decorate the call before it is executed and then explicitly call the original behavior. Take note that you can actually remove the original behavior in this manner, either intentionally or inadvertently.

This concludes what we can do with this plugin. Let's test out the changes. Don't forget to repackage the plugin by issuing the following command:

```
griffon package-plugin
```

If, at any time, you feel you need to start over, at least in terms of packaging, then know that you can issue the `clean` command. This removes any compiled sources and the latest version of the plugin zip file.

Running the plugin again

Before we install the plugin, once more double-check that you have the latest version. Many headaches can be averted by a few seconds of carefully verifying that your tools, artifacts, and sources are in proper state. As before, install the plugin by pointing the install-plugin command target to the zip file's path in your file system.

Running the application and clicking the button, following the same steps as we did before, yields the following output in the console (the updated output is shown in bold):

```
inputA: 'null' -> ''
inputB: 'null' -> ''
inputA: '' -> '1'
```

```
enabled: 'false' -> 'true'
inputB: '' -> '2'
enabled: 'true' -> 'false'
Entering click ...
output: 'null' -> '3.0'
enabled: 'false' -> 'true'
```

There you have it! Of course, we showed but a small fraction of what addons can do. The most common usage for addons is to provide new node factories and metaclass enhancements besides event handlers, as we just saw. Be sure to browse the extensive list of plugins and addons that can be found at Griffon's plugin repository[3]. You'll find plenty of addons that provide node factories.

Summary

Perhaps the greatest strength found in the Griffon framework is its ability to be extended by plugins. This is an inherited feature from Grails. Both frameworks share a lot of traits in their plugin systems; for example, listing, installing, and uninstalling a plugin is virtually the same.

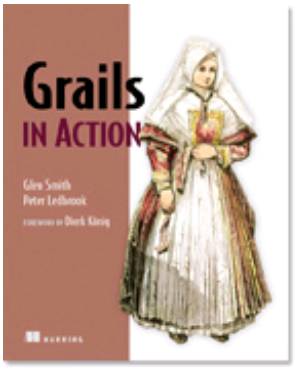
However, when it comes to developing a plugin, you'll start to notice a few differences. Griffon makes an explicit difference between build-time and runtime plugins, also known as addons. This difference allows Griffon to be very precise in the type of artifacts, libraries, and behavior that is exposed at build-time versus runtime.

A build-time plugin extends the framework's capabilities with new libraries and build-time events and scripts. Good examples of this kind of a plugin are testing-related (FEST, Easyb) as they never affect the running application.

An addon can add not only new libraries, but also other runtime aspects, such as application event handlers (featured in this article), metaclass enhancements, and node factories. Addons expose another set of events that can be used to coordinate their initialization and even communicate one addon to another, whether they have a strict dependency on one another or not.

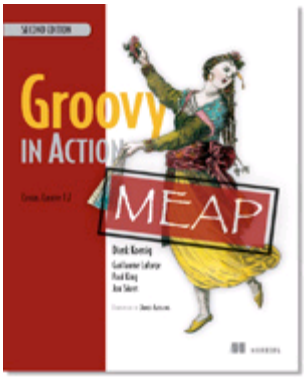
[3] http://svn.codehaus.org/griffon/plugins

Here are some other Manning titles you might be interested in:



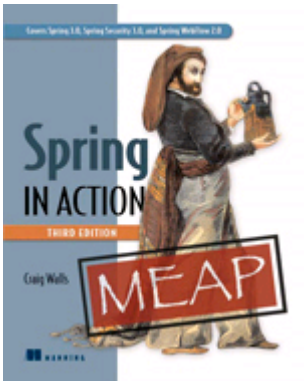
[Grails in Action](#)
IN PRINT

Glen Smith and Peter Ledbrook
Softbound print: May 2009 | 360 pages
ISBN: 1933988932



[Groovy in Action, Second Edition](#)
EARLY ACCESS EDITION

Dierk König, Paul King, Guillaume Laforge, Jon Skeet
MEAP Began: June 2009
Softbound print: Early 2011 | 700 pages
ISBN: 9781935182443



[Spring in Action, Third Edition](#)
EARLY ACCESS EDITION

Craig Walls
MEAP Began: June 2009
Softbound print: Fall 2010 | 700 pages
ISBN: 9781935182351