vol.1

Magazine

Japan Grails/Groovy User Group

G* Magazine 創刊のごあいさつ

日本Grails/Groovyユーザーグループ (Japan Grails/Groovy User Group = JGGUG) は、2009年2月。デベロッパ・サミットへの参加に合わせて発足しました。

その前身は2007年7月25日に始まったGrailsコード・リーディングであり、2006年8月からおもにGrailsのドキュメント翻訳を目的に立ち上がったgrails-ja (Google Groups) です。Grailsはまだバージョン1.0にも満たない、いわば胎児ともいえる時期でした。現在JGGUGには200名を超えるメンバが参加し、首都圏、関西、名古屋を拠点として、ほぼ毎月ワークショップを開催、2011年2月からは電子雑誌としてG* Magazineを創刊します。

Groovyを中心として、Grails、Griffon、Gaelyk、Gradleなどは、Javaを補完する強力なG*エコ・システムを構築しつつあります。これからのソフトウェア開発に欠かせないG*を学び、使いこなすためにJGGUGへの参加をお待ちしています。 http://www.jggug.org/ をご覧下さい。

日本 Grails/Groovy ユーザーグループ

代表　山田 正樹

Contents

series 01

Grailsをコントロールせよ！ Part 1

山本　剛（株式会社ニューキャスト）
出版・印刷関連のシステム設計開発等に従事するテクニカル DTP アーキテクト。
日本 Grails/Groovy ユーザーグループ名古屋支部長。2006 年より Grails のドキュメント翻訳、
その後、Grails 公式の Acegi プラグインを開発。書籍『Grails 徹底入門』(翔泳社発行) の 9、10、11 章を執筆。

本稿から2回に分けて、「Grailsをコントロールせよ！」と題して、コントローラからプレゼンテーション層の連携と、Webアプリケーションの UI 設計との連携に必要な URL の定義方法などを解説します。先ずは、どのように要求を受けてどのように結果を返すか、そこでどのような受け返しができるか、その内容を内外共に解りやすく定義できるかを理解することで、Grailsをきれいにコントロールできるようになれると良いでしょう。

コントローラ

コントローラは、クライアントからのリクエストを受け取り、リクエストの内容に応じて処理を行い、ビューを返すクラスです。（※ここではMVC2の細かな話は割愛させていただきます。）アプリケーションのリクエスト・処理などの動作をコントロールする部分になります、と、簡単に言うと怒られるかもしれませんが、初心者の方はその解釈で充分だと思います。

Grailsでコントローラクラスを作成するには、create-controllerコマンドを実行することで、コントローラのスケルトンを生成してくれます。

■Grailsでのコントローラの基本

まずはじめにコントローラクラスを生成します。プロジェクトの階層で以下のコマンドを実行してHomeコントローラを作成。
※ Grails での開発経験がある方はここは読み飛ばしてもかまいません。

```
grails create-controller home
#クラス生成時にパッケージを指定する事もできます。
grails create-controller jp.company.Home
```

grails-app/controllers階層のパッケージ階層以下にコントローラクラスが生成されます。同時にビュー用のgspを配置するための階層 grails-app/views/home が作成されます。今回のコマンドで生成されたコントローラクラスの内容は、次のような内容になります。

▼HomeController.groovy

```
package gmag01
class HomeController {
    def index = { } //アクション
}
```

この例では、生成時にパッケージを指定していないため、アプリケーション名がパッケージ名として自動的に付加されています。（grails-app/controllers/gmag01/HomeController.groovy）

コントローラクラスは、grails-app/controllers階層内に配置され、必ずクラス名の最後がControllerで終わる必要があります。生成コマンドを使わないでコントローラクラスを追加するときは、クラス名の最後がControllerにするのを忘れないようにしましょう！

では生成されたコントローラクラスの内容を見ていきます。単純なgroovyのクラスとして生成され、コードとしてはindexというクロージャがあらかじめ記述されています。これを処理単位を記述する場所 "アクション" と呼びます。アクションはコントローラクラス内に複数記述することが可能です。

このアクションにドメインモデル操作などの処理を記述して、アクションからビューにモデルを渡す、またはアクションから、そのまま文字列を返す等の処理を記述できます。

コントローラクラスには、ビューテンプレート指定や、リダイレクト処理、他にも様々なコントローラ・アクション内で使用できるメソッドや、request,response,sesson等の変数が動的に追加されています。これは、Grailsで初めて開発する人には不思議な状態に見えます。そして何が活用できるかは、実際にドキュメント無しには解りにくくなっているのもGrailsの特徴!?の一つとなります。（公式ドキュメント http://grails.org/doc/latest）

解りにくいと言い切ってしまうのも・・・なので、最低限のコントローラクラスの説明は終わりとして、ここから実際に動作できるコードでしていきます。

■ 簡単に動作させてみよう

最初に (※よく質問受けるので) 前提としてコントローラは、プロトタイプです。リクエスト毎にコントローラは生成されます。コントローラからビューの連携には様々な方法があります。まずは、簡単なハローワールドを記述して動かしてみましょう。

```
class HomeController {
    def index = {
        render 'Hello World!'
    }
}
```

render 'Hello World!' と記述して起動します（grails run-app）、そして、http://localhost:8080/gmag01/home/index にアクセスします。この例では、単純に'Hello World!'と文字列を返しブラウザー画面に表示します。この "render" は単純なテキストのレスポンス処理から、ビューへの連携等、多彩に幅広く活用できるメソッドです。またこの例では、省略していますが、render 'Hello World!'の部分を、次のように、text: と指定して記述することもできます。

```
render text:'Hello World!'
```

ハローワールド画面

次のサンプルはリクエストクエリー付きになります。

```
class HomeController {
    def index = {
        render "Hello ${params.who}!"
    }
}
```

render 'Hello World!' を"Hello $\{params.who\}!" に変更してみます。そして、先ほどのURLの後ろに ?who=Worldを追加した http://localhost:8080/gmag01/home/index?who=World にアクセスします。今回は、params変数をプロパティ参照した、params.whoをレスポンス結果を返すコードに追加しました。リクエストクエリーのパラメータは、コントローラの機能で自動的に収集され、params（詳細は後述）変数に収納されます。パラメータを参照するには、params.whoのように、params変数に対して、"."に次いでパラメータキーを指定するだけです。

では、次に簡単にGSPと連携をしてみましょう。GSPは、何も定義しない場合、"grails-app/views/コントローラ名/アクション名.gsp"ファイルに関連づけられます。例えば次のコード例では、アクションに特に何も指定しないと自動的に、"grails-app/views/home/index.gsp"をGSPとしてビューを表示します。そして該当ファイルが無い場合は、次に"grails-app/views/home/index.jsp"ファイルを探しにいき、JSPファイルも存在しなければ、エラーコード404を返す流れとなります。

```
class HomeController {
    def index = {
        //
    }
}
```

前の例では返値を何も返してません。アクションの最後に返値のMapを返すことでGSPへモデルを渡すことができます。

```
class HomeController {
    def index = {
        [username:"tyama",score:2500]
    }
}
```

GSPでのモデルの参照は、このようになります。

▼index.gsp

```
<html>
<head>
    <title>indexページ</title>
    <meta name="layout" content="main" />
</head>
<body>
    username: ${username} score is ${score}
</body>
</html>
```

ビュー用のGSPファイルを指定することもできます。次の例では"grails-app/views/home/foo.gsp"を指定しています。

```
class HomeController {
    def index = {
        render view:"foo", model:
                [username:"tyama",score:2500]
    }
}
```

ここまでの内容で、簡単にリクエストを行いパラメータを取得してアクションで処理を行いレスポンスするという一連の動作はわかったと思います。

そして、もう気付いてると思いますが、ブラウザーでアクセスしたURLをみると、"/gmag01/home/index" これを置き換えて説明すると、"/アプリケーション名/コントローラ名/アクション名" となっていることがわかります。これはGrailsでのURLの基本となります。URLマッピング（詳細は次号）を定義して変更する事も可能です。

URLとコントローラの関係

■render、redirect、chain、forward

簡単にコントローラを動作させることができたところで、アクションからのコントロールをするために大事な、ここまでに一部例としても出てきた render、redirect、chain、forwardという４つのメソッドを紹介します。ほとんどがそれぞれ読んで字のごとくですが、多彩な機能を持つrenderをはじめ、それぞれ使い方がシンプルのようで、それなりに複雑さも持ち合わせています。これらを上手く活用することでGrailsのリクエスト処理を快適にコントロールできます。

●render

renderは様々なビューレスポンスを返すためのメソッドです。renderを使用して単純なテキスト、ビューやテンプレートの指定、XML、JSON、マークアップを使ったXMLのレスポンス、エラーコード等、多彩にレンダリング操作を行えます。
単純なテキストレスポンス例。

```
render "Hello!"
//又は
render text:"Hello!"
{code}
コンテントタイプ・エンコーディングも指定できる。
{code}
render text:"<message>こんにちわ</message>",
contentType:"text/xml", encoding:"UTF-8"
```

ビューを指定する例。前述しましたが、viewにビューファイル名、modelに返すモデルを指定します。

```
render view:"foo", model:
[username:"tyama",score:2500]
//上記の例では、アクションを記述したコントローラ用のビュー階層を参照します。
//別の階層を参照するには、パスで指定します。(.gspは省略)
render view:"/common/foo", model:
[username:"tyama",score:2500]
//プラグイン内にあるビューを指定するには。
render plugin:"myplugin", view:"foo", model:
[username:"tyama",score:2500]
```

※コード例中に説明があるように、ビューファイルの指定は、grails-app/viewsをルートとして、拡張子なしの絶対パスで指定します。

テンプレートを指定する例。テンプレート用ビューファイルは、ファイル名の先頭に"_"を付加します。（例）_mytemplate.gsp

```
render template:"mytemplate", model:
[username:"tyama",score:2500]
```

テンプレートにビーンを指定してビューに渡す。

```
render template:"list",bean:new
Person(name:'tyama', age:27)
//gspの内容
<user>${it.name} (${it.age})</user>
```

テンプレートでコレクションを扱う。テンプレート _list.gsp を指定して、コレクション [1,2,3,4] を表示させます。

▼_list.gsp

```
<item>${num}</item>
```

collection にコレクションを指定して、テンプレートで参照する変数を var に指定します。var が省略された場合は、テンプレート内では "it" で参照できます。

```
render template:"list",collection:[1,2,3,4],
var:"num"
```

マークアップビルダーを使用する。

```
def users = ['user1','user2']
render(contentType:"text/xml") {
    followers {
        users.each{user->
            username(user)
        }
    }
}
```

例では若干中身を変えないと動きませんが、JSONの場合は、単純にコンテントタイプを"text/json"に変更。

```
def users = ['user1','user2']
render(contentType:"text/json") {
    followers:users
}
```

次回のRESTでも紹介しますが、JSON・XMLを返す場合は、コンバーターを使用できます。
コンバーターをimportします。

```
import grails.converters.*
```

そしてrenderでは、それぞれ次のように記述します。

```
render Book.list(params) as JSON
render Book.get(params.id) as XML
```

HTTPのステータスを返す事もできます。

```
render status: 503, text: 'Failed..'
```

以上がrenderの使用方法になります。レンダリングの指定方法が多数有りますが、AJAXを使用したり、ポータル的にパーツを返したり等の用途によって、使いやすい方法を選択して活用しましょう。

●redirect

リダイレクトを行うメソッドです。リクエストを受けたアクションから、指定した場所へHTTPのリダイレクトを行います。
コントローラ・アクション・IDを指定すると、内容に従ってURIを形成した対象へリダイレクトします。次の例では、"/user/show/1"へリダイレクトされます。

```
redirect controller:"user", action:"show", id:1
```

このサンプルにparamsを指定する事もできます。次の例では、"/user/show/1?status=good"へリダイレクトされます。

```
redirect controller:"user", action:"show", id:1,
params:[status:'good']
```

フラグメント指定。先ほどの例に"/user/show/1#comment"のように"#comment"と最後に追加します。

```
redirect controller:"user", action:"show", id:1,
fragment:"comment"
```

通常のURL、URIの指定も可能です。

```
redirect uri:"/user/list"
redirect url:"http://www.yahoo.co.jp"
```

●forward

HTTPリダイレクトとは違いサーバサイド側でフォワードするメソッドです。redirectとほとんど同じですが、forwardでは、コントローラ・アクション・ID・パラーメータのみの扱いになります。

```
forward controller:"user", action:"show", id:1,
params:[status:'good']
```

●chain

別のアクションへHTTPリダイレクトを行いながら、アクションを連携する場合に使用するメソッドです。
公式ドキュメントから例を拝借。

```
class HomeController {
    def first = {
        chain(action:second,model:[one:1])
    }
    def second  = {
        println chainModel.one
        chain(action:third,model:[two:2])
    }
    def third = {
        [three:3]
    }
}
```

この例で/home/firstにアクセスすると、最終的に/home/thirdに到達して、アクションthirdから返るモデルは、[one:1, two:2, three:3]になります。chainで渡されたモデルは、"chainModel"から参照できます。

■コントローラの変数

コントローラには、さまざまなプロパティが動的に設定されています。「簡単に動かしてみよう」で試したように、リクエストのパラメータを参照するためのparamsをはじめとして、GrailsはJavaのフレームワークなので、もちろん、session、request、response、servletContextも参照可能です。他に次のリクエストまで内容を保持するテンポラリ収納用マップの変数flash等があります。

●actionName、controllerName

先ずはじめに簡単な物から紹介。意外と使わないようでお世話になる変数、actionName、controllerName。これももちろん読んで字のごとく、それぞれ参照した場所のアクション名、コントローラ名を取得できます。

```
render "コントローラ名は${controllerName}でアクション名は${actionName}です。"
```

●session

sessionは、HttpSessionのインスタンスです。HttpSessionの実装であるorg.codehaus.groovy.grails.web.servlet.mvc.GrailsHttpSessionが参照できます。HttpSessionのメソッド実行のほか、セッション内容に対してGroovyマップのようにアクセスすることができます。

```
def username = session["username"]
session.username
```

●request

requestは、HttpServletRequestのインスタンスです。HttpServletRequestの実装であるorg.apache.catalina.core.ApplicationHttpRequestが参照できます。HttpServletRequestにある通常の機能に加えて、Grailsでは便利なメソッド・フィールドが追加されています。
※（注意）機能によっては開発時に使う必要は無く、内部的に利用するための機能でもあるので、乱用注意！

RESTで活用できる機能。それぞれXML、JSONを受け取った場合に以下のように参照できます。

```
def xml = request.XML // XmlSlurperのGPathResult
def json = request.JSON // GrailsのJSONObject
```

リクエストがgetまたはpostかを判別する。

```
if(request.get){
     //getの場合
}else if(request.post){
     //postの場合
}
```

HttpServletRequestのgetAttributeも簡単にアクセスできるようになっています。アトリビュートの参照は、例のように簡単にアクセスできるほか、requestに対して、each、find、findAllも使用できます。

```
def username = request['username']
def score = request.score
```

フォーム送信時に<form>にenctype="multipart/form-data"が存在した場合、言い換えるとマルチパートデータのフォーム送信時は、自動的に、MultipartHttpServletRequestになります。ファイルアップロードを行う場合は、request.getFile()として、次のコード例のように取得できます。request.getFile()で取得したイ

ンスタンスはSpringのMultipartFile（org.springframework.web.multipart.MultipartFile）です。

```
def uploadFile = request.getFile('upload')
uploadFile.transferTo( new File('/dir/to/save/file.jpg') )
```

●response

responseは、HttpServletResponseのインスタンスです。HttpServletResponseの実装であるorg.codehaus.groovy.grails.web.sitemesh.GrailsContentBufferingResponseが参照できます。

```
def downloadFile = {
    byte[] bytes = // バイト配列
    response.outputStream << bytes
}
```

●servletContext

servletContextは、ServletContextのインスタンスです。ServletContextの実装であるorg.apache.catalina.core.ApplicationContextFacadeが参照できます。
servletContextも他のと同様に、getAttributeが簡単にアクセスできるようになっています。

```
def setting = servletContext["setting"]
servletContext["setting"] = "hoge"
```

●flash

flashは、次のリクエストまで使用できる一時保存マップです。次のリクエストが完了すると動的に内容がクリアされます。
実態は、org.codehaus.groovy.grails.web.servlet.GrailsFlashScopeです。

次の例では、indexアクションにリクエストを受けるとflash.messageにメッセージを代入して、homeアクションにリダイレクト、homeアクションで処理されるビューでflash.messageが参照できます。そして処理が完了すると、flashはクリアされます。

```
def index = {
    flash.message = "Hello!"
    redirect(action:home)
}
def home = {
     //
}
```

■params

前半でも簡単に紹介した、リクエストパラメータを収納している変数です。公式ドキュメントでは、ミュータブル多次元ハッシュマップと説明されています。クラスの実態は、org.codehaus.groovy.grails.web.servlet.mvc.GrailsParameterMapです。

参照方法の簡単な例です。

```
println params.username
def user = User.get(params.id)
```

paramsは、ただ単にリクエストパラメータを収納して参照するだけではなく、コントローラの開発を柔軟でスムーズに進めるための機能を多く持っています。その説明は次の「リクエストパラメータ型変換、バインディング、コマンドオブジェクト」で細かく説明していきます。

■リクエストパラメータ型変換、バインディング、コマンドオブジェクト

Grailsでは、リクエストパラメータをドメインクラス・モデルと簡単にバインドできる仕組みが提供されています。バインディングの説明をするには、ドメインクラスが関係してきますが、今回はドメインクラスの深い内容は割愛します。

●型変換

paramsにはリクエストで受けた内容を正確に受け取るために便利な型変換機能が備えられています。

例えば、次のようなコードがあるとします。

```
def num = params.num
if(num){
     println num+10
}
```

この例では、params.numに文字列がセットされていた場合は、正常な処理結果が得られません。これを解決するのにparams型変換を使用します。

```
def num = params.int('num')
if(num){
     println num+10
}
```

params.numであれば文字列の場合でもif文を通ることができますが、params.int('num')と指定する事で、数値以外であれば確実にnullを返すようになるので、条件を通ることができません。このように、params型変換（nullセーフコンバータとも呼ぶ）を使用することで処理を無駄なく確実にします。

通常Webリクエストは主に文字列でパラメータから受け取るので、型検証や変換を予め行う必要があります。その処理をわかりやすくシンプルに記述できます。intの他にboolean、long、char、shortが使用できます。

この型変換には同じ名称のパラメータを配列に変換するリスト変換も存在します。リクエストの内容が、/home/index?key=apple&key=lemon の用に同じ名称が複数存在する物を処理するには次のように記述します。

```
def keys = params.list('key')
keys.each{
     println it
}
```

params.keyと取得しても確実にリクエスト内容に "key" が複数

存在するのであれば、リストとしても取得できるのですが、もし"key"が1個のみの場合はリストにならないので、その場合の処理を正確に記述するために使えます。

```
// リクエストが/index?key=appleの場合は。
def keys = params.key  //keysはStringになる。
// リクエストが/index?key=apple&key=lemonの場合は。
def keys = params.key //keysは配列になる。
def keys = params.list('key') // リクエスト内容の数に関係無く確実に配列になる。
```

●paramsの基本とバインディングの基本

先ずはじめに単純な使用方法の例をおさらいします。リクエスト内容が、/home/index?username=tyama&age=27&email=tyama@xmldo.jp とすると、アクションの内部で、paramsから簡単に取得できました。

```
def username = params.username
def email = params.email
```

ドメインクラスUserを例として、

▼User.groovy

```
class User {
     String username
     String password
     String email
     int age
}
```

先ほどのパラメータ内容をドメインクラスにバインドする場合は、次のようにするのではなく。

```
def user = new User()
user.username = params.username
user.email = params.email
```

ドメインにリクエストパラメータをバインドさせるには、単純にparamsをconstructorに与えるか、propertiesに代入するだけです。

```
def user = new User(params)
//または
def user = new User()
user.properties = params
```

bindDataメソッドでバインディングを行うこともできます。

```
def user = new User()
bindData(user, params)
```

※但しデータバインドは、内部で自動でバインドを行うために速度的な懸念もあります。従って先に挙げた例も間違いではありませんので、速度を検証してみて使いやすい方を選ぶのも良いと思います。個人的意見では、間違った代入や、コードの見やすさを考えるとGrailsが提供するデータバインドを使用した方が良いと思います。

●パラメータ指定でセキュリティ向上

データバインディングの問題点として、リクエストに更新を行うべきでない不正なパラメータも付加されてしまった場合でも、自動でセットしてしまうという懸念があります。それを制御するために、不用なパラメータのバインディングを行わないようにするために、propertiesにバインディングを行いたいフィールドを指定する方法があります。この例ではリクエストとして、username、emailの他に不正に付加してpasswordが含まれているとします。user.properties['username','email'] と指定する事で、不用なパラメータpasswordは、paramsからバインドされることはありません。

```
def user = User.get(1)
user.properties['username','email'] = params
```

bindDataを使用した例。

```
def user = new User()
bindData(user, params, [ include:
                ['username','email'] ])
//除外する場合は。
bindData(user, params, [exclude:'password'])
```

自動生成されるスカッフォルドのコードについては、このような対策がされていません。スカッフォルドで生成したコードを重要箇所にそのまま使用することは無いとは思いますが、それも含めて、セキュリティを重要視するモデルに対してのバインディングには必ずこの方法を使用しましょう。

今回はここまでです。今回はコントローラの基礎から、リクエストの処理から、簡単なバインディングまで解説しました。いよいよ、次回はリレーショナルのあるドメインのバインディング、コマンドオブジェクト、REST、コントローラにまつわる制御の仕組み、GSP、URLマッピングを解説します。ではまた次号でお会いしましょう！

series 02

Griffon 不定期便 ～第 2 回 バインディング編～

奥 清隆 (おく きよたか)

仕事でもときどき Groovy と戯れるプログラマ。
日本 Grails/Groovy ユーザーグループ関西支部長。
著書：『Seasar2 による Web アプリケーションスーパーサンプル』

今回はGriffonでのバインディングについて紹介します。バインディングについては前回少し紹介しましたが、今回はもう少し深く紹介します。

Griffonのバインディングといっても基本的な機能はGroovyのSwingBuilderで提供されています。今回はGriffonというよりGroovyのSwingBuilderの話が中心になってしまいますが、SwingBuilderはGriffonでも重要な機能ですので、SwingBuilderを使った様々なバインディングを紹介したいと思います。

■ビューからモデルへのバインディング

テキストフィールドに入力された値をモデルにバインディングさせてみましょう。次のようなサンプルコードでビューからモデルへのバインディングを確認します。SwingBuilderの機能しか使っていませんのでGroovyConsoleに貼付けて実行する事ができます。

```
import groovy.swing.SwingBuilder
import groovy.beans.Bindable

class Model {
    @Bindable name
}

def model = new Model()

new SwingBuilder().edt {
    frame(show:true, pack:true) {
        gridLayout(cols:1, rows:2)
        textField(text:bind(target:model,
                targetProperty:'name'))
        button('Hello', actionPerformed: {
            optionPane(message:"Hello,
                ${model.name}!").
                    createDialog('Message').show()
        })
    }
}
```

このサンプルを実行すると次のようなアプリケーションが起動します。

このテキストフィールドに名前を入れてボタンをクリックするとダイアログが表示されます。

シンプルな例ですが、ポイントはModelクラスのプロパティで宣言している@Bindableアノテーションと、テキストフィールドのtextプロパティでbindメソッドを使用している2点です。この2点でテキストフィールドの値が変更されるとモデルに反映されるようになります。

このバインディングは次のように少し省略して書く事もできます。

```
textField(text:bind(target:model, 'name'))
```

■モデルからビューへのバインディング

今度は逆にモデルからビューへのバインディングを見てみましょう。

ここではボタンをクリックすると、運勢を占ってくれるおみくじアプリを作ってみましょう。

このサンプルもGroovyConsole上で実行できます。

```
import groovy.swing.SwingBuilder
import groovy.beans.Bindable

def FORTUNE_TEXTS = ['大吉', '吉', '中吉', '小吉
', '末吉', '凶']

class Model {
    @Bindable fortune
}

def model = new Model()

new SwingBuilder().edt {
    frame(title:'おみくじ', show:true, pack:true)
{
        gridLayout(cols:1, rows:2)
        label(text:bind(source:model,
                sourceProperty:'fortune'))
        button('click', actionPerformed: {
            model.fortune = "今日の運勢 :
                ${FORTUNE_TEXTS[(int) (FORTUNE_
                  TEXTS.size() * Math.random())]}"
        })
    }
}
```

実行結果は次のようになります。ボタンをクリックする事で運勢が表示されます。

最初のビューからモデルへのバインディングと似ていますが、今回はbindメソッドの引数にsourceとsourcePropertyを指定することでビューへのバインディングになっています。

```
label(text:bind(source:model,
sourceProperty:'fortune'))
```

このコードも次のように省略形で記述する事ができます。

```
label(text:bind(source:model, 'fortune'))
```

bindメソッドでsourceを指定する場合は、クロージャを使用してさらに省略する事ができます。

```
label(text:bind { model.fortune })
```

■双方向のバインディング

テキストフィールドなどの入力コンポーネントを使用していると同じプロパティを使ってモデルからビューへ,そしてビューからモデルへと双方向のバインディングが必要になる場合もあります。

これまでのサンプルでsourceとtargetプロパティを使用したので次のように記述すれば双方向のバインディングができるように思うかもしれません。

```
textField(text:bind(sourceProperty:'val
ue', targetProperty:'value', target:model,
source:model))
```

しかし、これでは期待した結果は得られません。モデルからビューへのバインディングとビューからモデルへのバインディングがループしてしまいStackOverflowErrorが発生してしまいます。

次のように、mutualプロパティを指定することで双方向のバインディングが可能になります。

```
textField(text:bind(targetProperty:'value',
target:model, mutual:true))
```

■ビューからビューへのバインディング

すべてのバインディングをモデルとビューの間で簡単にできるのは便利ですが、わざわざモデルにプロパティを持たせるまでもない（あるいは持たせたくない）ようなバインディングもあります。

たとえば、ラジオボタンの選択によって他の入力コンポーネントの状態を変える場合を考えてみます。

```
import groovy.swing.SwingBuilder
import groovy.beans.Bindable

class Model {
    @Bindable String text
}

def model = new Model()

new SwingBuilder().edt {
    frame(title:'アンケート',show:true,pack:true)
{
        gridLayout(cols:1, rows:6)
        label '今回のG*Magazineは？'
        buttonGroup(id:'bg')
        radioButton 'とても良かった', buttonGroup:bg
        radioButton '良かった', buttonGroup:bg
        radioButton id:'other', 'その他', buttonGroup:bg
        textField(text:bind('text', target:model),
        editable:bind(source:other,
                sourceEvent:'itemStateChanged',
                sourceValue: {other.selected}))
    }
}
```

このサンプルを実行すると次のようになります。

よくあるアンケートの入力フォームですが、ラジオボタンで「その他」を選んだ場合だけテキストを自由に入力することができるようになります。

モデルにテキストフィールドの編集可能かどうかを判定するプロパティを持たせる事もできますが、モデルで持つほどのものでもないのでラジオボタンの選択によってテキストフィールドの状態を変えています。

sourceEvent、sourceValueという新しいプロパティをbindメソッドに指定しました。また、sourcePropertyは指定していません。

sourceEventを指定する事でsourceオブジェクトの任意のイベントでバインディングさせる事が可能になります。また、sourceValueを指定するとsourceとは異なるオブジェクトからバインディングさせる事が可能になります。

■変換処理

バインディング時に変換処理を間に入れることも可能です。例えば、日付の年月日をそれぞれ別のコンポーネントに表示させるような場合、年月日の3つのプロパティをモデルに持たせるのは冗長で、コントローラなどからも扱いづらくなってしまいます。そんな場合はモデルに1つだけDate型のプロパティを定義しておきビューの各コンポーネントではconverterプロパティを指定することで変換した値を表示させる事ができます。

```
import groovy.swing.SwingBuilder
import groovy.beans.Bindable

class Model {
    @Bindable Date now
}

def model = new Model()
model.now = new Date()

new SwingBuilder().edt {
    frame(id:'frame', show:true, pack:true) {
        gridLayout(cols:6, rows:1)
        label(text:bind('now', source:model,
                converter: {it.format('yyyy')}))
        label('年')
        label(text:bind('now', source:model,
                converter: {it.format('MM')}))
        label('月')
        label(text:bind('now', source:model,
                converter: {it.format('dd')}))
        label('日')
    }
}
```

実行結果は次のとおりです。

converterはsourceValueと似ていますが、converterで指定するクロージャにはsourcePropertyの値が引数で渡されます。逆にsourceValueのクロージャには引数が渡されませんので任意の値を使用する事ができます。

■バリデーション

変換処理が必要になってくると、ビューからモデルへのバインディング時に入力値のバリデーションも欲しくなってきます。モデルのすべてのプロパティをString型にしてしまう方法もありますが、とても扱いづらくなってしまうでしょう。バリデーションの実装も簡単にできます。

```
import groovy.swing.SwingBuilder
import groovy.beans.Bindable

class Model {
    @Bindable String postCode
}

def model = new Model()

new SwingBuilder().edt {
    frame(id:'frame', show:true, pack:true) {
        gridLayout(cols:1, rows:2)
        textField(text:bind('postCode',
                target:model, validator:{
            it ==~ /\d{3}-\d{4}/
        }))
        button('click', actionPerformed: {
            optionPane(message:model.postCode).
                createDialog('Message').show()
        })
    }
}
```

validatorプロパティで入力値を検証するクロージャを定義するだけです。このクロージャがfalseを返した場合、ビューの表示はそのままですがモデルには値が反映されません。入力値がエラーになる場合は何らかの通知をユーザに対して行う必要があります。

サンプルのバリデーションは簡単なものですが、実際にアプリケーションを作成するとGrailsのドメインクラスの様なバリデーションが欲しくなってくると思います。SwingBuilderではGrailsの様なバリデーションは提供されていませんが、GriffonのプラグインとしてValidationプラグインが提供されています。Validationプラグインについては後述します。

■PropertyChangeListenerを追加する

ここまではSwingBuilderのバインディング機能を紹介してきましたが、Griffonではバージョン0.9からPropertyChangeListenerを追加するListenerアノテーションが提供されています。

Listenerアノテーションとリスナー用のクロージャを定義する事でPropertyChangeListenerを簡単に追加する事ができます。

```
package sample

import groovy.beans.Bindable
import griffon.beans.Listener

@Listener(loggingListener)
class ListenerSampleModel {

    @Bindable String value1
    @Bindable String value2

    def loggingListener = {
        println "${it.propertyName}:
                ${it.oldValue} -> ${it.newValue}"
    }
}
```

Validationプラグイン

ここではGriffonのValidationプラグインを紹介します。バリデーションについてはSwingBuilderの機能でもできない事はありませんが、Grailsの様な豊富なバリデーションと入力エラーを通知するのにこのプラグインが利用できます。

■アプリケーションの作成とプラグインのインストール

今のところValidationプラグインはGriffonのバージョン0.9にしか対応していませんので、Griffon 0.9でサンプルアプリケーションを作成していきます。

```
$ griffon create-app validationsample
Picked up _JAVA_OPTIONS: -Dfile.encoding=UTF-8
Welcome to Griffon 0.9 - http://griffon.codehaus.
org/
Licensed under Apache Standard License 2.0
Griffon home is set to: /usr/local/griffon/
current
...
```

プラグインをインストールします。

```
$ cd validationsample
$ griffon install-plugin validation
...
```

■モデルの編集

griffon-app/models/validationsample/ValidationsampleModel.groovyを編集していくつかのプロパティと制約を追加します。

```
package validationsample

import groovy.beans.Bindable

class ValidationsampleModel {
    @Bindable String requiredText
    @Bindable String url
    @Bindable String email
    @Bindable String host

    static constraints = {
        requiredText(blank:false)
        url(url:true)
        email(email:true)
        host(inetAddress:true)
    }
}
```

制約の指定方法はGrailsのドメインクラスと同じです。

■ビューの編集

作成したモデルをテストするためにgriffon-app/views/validationsample/ValidationsampleView.groovyを変更しましょう。

```
package validationsample

import net.sourceforge.gvalidation.swing.
ErrorMessagePanel

application(title: 'Validation Sample',
  size: [600,300],
  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]) {
    tableLayout {
      tr {
        td(colspan:2) {
          widget(new ErrorMessagePanel(messageSource),
           id: 'errorMessagePanel', errors: bind(source: model, 'errors'))
        }
      }
      tr {
        td(align:'right') { label 'Not blank' }
        td{textField(text:bind('requiredText',target:model),columns:30)}
      }
      tr {
        td(align:'right') { label 'URL' }
        td{textField(text:bind('url', target:model), columns:30) }
      }
      tr {
        td(align:'right') { label 'Email' }
        td{textField(text:bind('email', target:model), columns:30) }
      }
      tr {
        td(align:'right') { label 'Host' }
        td{textField(text:bind('host', target:model), columns:30) }
      }
      tr {
        td(colspan:2){button('validate',actionPerformed:{model.validate()})}
      }
    }
}
```

ErrorMessagePanelクラスはValidationプラグインで提供されているエラーを通知するための簡単なコンポーネントです。ボタンをクリックしたときにモデルのvalidateメソッドを呼び出してバリデーションを行います。このvalidateメソッドはValidationプラグインによって拡張されたもので、モデルにエラーがあった場合、errorsというプロパティにエラー情報が格納されます。

ErrorMessagePanelにモデルのerrorsをバインディングする事でエラー情報をビューで通知する事が可能になります。

■エラーメッセージの追加

griffon-app/i18n/messages.propertiesにエラーメッセージを追加します。今回はValidationプラグインのドキュメントページのエラーメッセージをそのまま追加します。

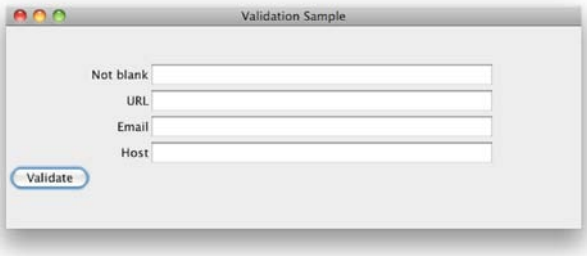
```
default.matches.message=Property [{0}] of class [{1}] with value [{2}] does not match the required pattern [{3}]
default.url.message=Property [{0}] of class [{1}] with value [{2}] is not a valid URL
default.creditCard.message=Property [{0}] of class [{1}] with value [{2}] is not a valid credit card number
default.email.message=Property [{0}] of class [{1}] with value [{2}] is not a valid e-mail address
default.range.message=Property [{0}] of class [{1}] with value [{2}] does not fall within the valid range
default.size.message=Property [{0}] of class [{1}] with value [{2}] does not fall within the valid size
default.max.message=Property [{0}] of class [{1}] with value [{2}] exceeds maximum value [{3}]
default.min.message=Property [{0}] of class [{1}] with value [{2}] is less than minimum value [{3}]
default.maxSize.message=Property [{0}] of class [{1}] with value [{2}] exceeds the maximum size of [{3}]
default.minSize.message=Property [{0}] of class [{1}] with value [{2}] is less than the minimum size of [{3}]
default.validator.message=Property [{0}] of class [{1}] with value [{2}] does not pass custom validation
default.inList.message=Property [{0}] of class [{1}] with value [{2}] is not contained within the list [{3}]
default.blank.message=Property [{0}] of class [{1}] cannot be blank
default.notEqual.message=Property [{0}] of class [{1}] with value [{2}] cannot equal [{3}]
default.nullable.message=Property [{0}] of class [{1}] cannot be null
```

■アプリケーションの実行

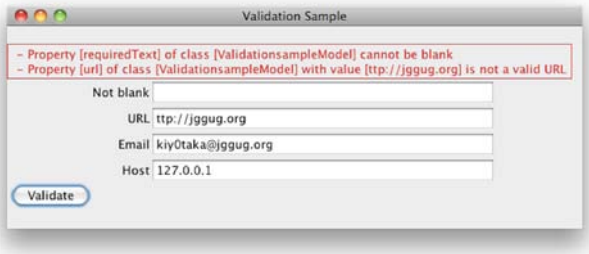
それではアプリケーションを実行してみましょう。

```
griffon run-app
```

実行すると以下のようになります。



ボタンをクリックしてエラーがあると以下のようにエラーメッセージが表示されます。



まとめ

今回はGriffonでよく使うバインディングを紹介しました。バインディングを理解する事でMVC間での情報のやりとりも、よりスマートな形で実装する事ができるようになります。次回はスレッドについて紹介したいと思います。それではまた、お会いしましょう。

series 03

CodeNarcを利用してGROOVYのコード品質を上げる ～第１回 CodeNarcとは～

荒井健太郎

某 IT ベンダーのセキュリティコンサルタント、ソフトウエアをよりセキュアにするために日々精進しています。
テスト系スクリプトは、G* を利用して書いています。

静的コード解析とは

Codenarkは、Groovyで作成された、Groovy向けの静的コード解析ツールである。

静的コード解析は、プログラムを実行せずにソースコードを構造的に解析し、一般的に知られているコードの問題が解析対象のソースコードに存在していないかをチェックすることである。ここでいう一般的に知られているコードの問題は、拡張性低下の原因となるコード、性能劣化の原因となるコード、セキュリティの脆弱性を持つコードやコーディング規約に準拠していないコードを指す。オープンソースの静的コード解析ツールとして、Java向けに、「PMD（http://pmd.sourceforge.net/）」、「CheckStyle（http://checkstyle.sourceforge.net/）」、「FindBugs（http://findbugs.sourceforge.net/）」 や「Jlint（http://jlint.sourceforge.net/）」 などがある。これらのツールの違いは、「どのようなコードの問題を検証するのが得意か」という点で異なっている。FindBugsやJlintは、潜在的なバグを検証することを主眼としている。一方、CheckStyleはコーディング規約（コメントの書き方等）の問題を検証することを主な目標としている。PMDは汎用的な（幅広い）問題を検証することを目標としている。Groovy向けの静的コード解析ツールとして、IntelliJ IDEA（http://www.jetbrains.com/idea/）にバンドルされた静的コード解析ツールがある。 このツールは、IDEのIntelliJ上で静的コード解析を実行することを目標としている。

PMD、CheckStyleやIntteliJ IDEAにバンドルされた静的コード解析ツール の影響を受けて開発されていることを考慮すると、CodeNarkは、汎用的な問題を検証し、汎用的な用途で利用できる静的コード解析ツールを目標としていると考えることができる。

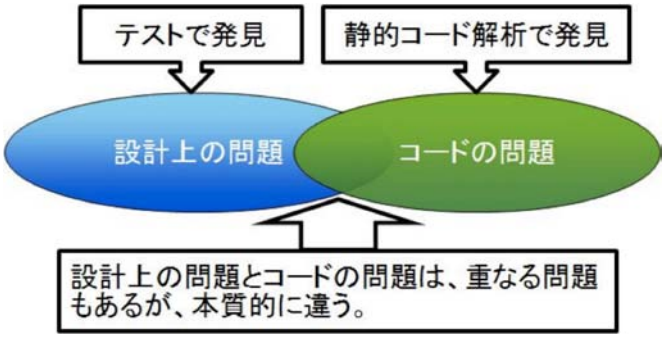
また、商用の静的コード解析ツールとしては、「Fortify SCA（https://www.fortify.com/）」、「Coverity（http://www.coverity.com/html_ja/）」などがある。オープンソースと商用の差は、結果の表示手法の違い、コード解析の深さの違い（ファイルやクラス間にまたがるデータフローを解析できるか等）やルールの違い（*ルールについては後述）などがある。Groovyのコードを解析することが可能な商用の静的コード解析ツールは私が調査した限りでは存在しない。

設計上の問題とコードの問題

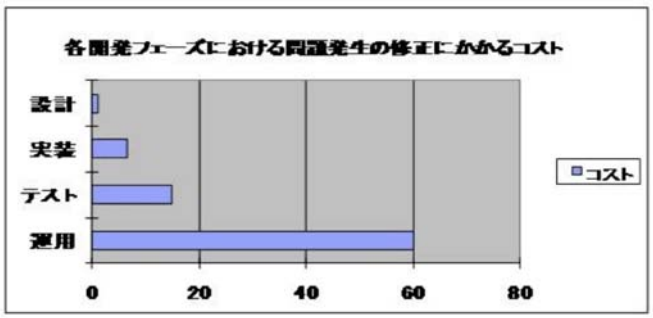
前述した一般的に知られているコードの問題は、単体テストや結合テストと呼ばれるテストでみつけるものだと考える方が多いかもしれない。

しかし、テストのみで拡張性の低下となるコードや性能劣化の原因となるコードが発見できるだろうか。たしかに性能劣化の原因となるコードの一部分はパフォーマンステスト等で発見できるかもしれないが、それは問題の一部分である。

テストと静的コード解析は目標としていることが本質的に違う。テストは、バグすなわち設計上の問題点を検出することを目標としている。すなわち、ユーザの要件通りにソフトウエアが実行されることを検証している。一方、静的コード解析は、コードの品質が高いかの検証、すなわち、非機能要件について検証をしている。設計上の問題とコードの問題のイメージは下図のようになる。



また、静的コード解析はプログラムを実行することなく実行することができるので、単体テスト、結合テストなどのテストより前、コードを1行書き始めた時から実行することが可能である。したがって発見された問題点による手戻りが少なくなるというメリットがある。参考までに、ソフトウエア開発の工程で手戻りが発生した場合のコストの例を以下に示す。出典は、「Soo Hoo, Kevin, Andrew W. Sudbury, and Andrew R. Jaquith. "Tangible ROI through Secure Software Engineering." Secure Business Quarterly. 1-2. 2001.」である。



静的コード解析ツールのまとめ

以上をまとめると、静的コード解析はテストで検証することが難しいコードの問題を検証し、開発の早い段階で問題点を検証できるということである。

CodeNarcのルール

CodeNarcは、一般的な静的コード解析ツールと同様に静的コード解析をルールに従って実施する。解析に使用するルールはCodeNarcに標準で付属しているルール及びCodeNarcの仕様に従って作成したルールを使用することができる。静的解析ごとに使用するルールを切り替えることも可能である。

CodeNarcに標準で付属しているルールは、http://codenarc.sourceforge.net/codenarc-rules-basic.htmlの左メニューのRulesから参照することが可能である。

このコラムの内容

第1回は、CodeNarcの基本的な実行方法を紹介し、第2回以降でルールのカスタマイズ方法や各種プラグインの利用方法を紹介する。

■基本的な実行方法

CodeNarcを実行するためには、以下のものを用意しておく必要がある。Log4Jのバイナリが CodeNarcに付属していることを考慮すると、Groovyの実行環境を構築している方であれば、CodeNarcをダウンロードすれば他に用意するものはない。

- Groovy 1.7 以上
- Java1.5以上
- Log4J1.2.13以降（CodeNarcのtarballに付属）

CodeNarcは、コマンドラインからの実行以外にも様々な実行方法がある。今回は、CodeNarcの基本的な動作を理解するという観点から、CodeNardに付属しているサンプルを利用して、「コマンドラインからの実行」と「Antタスクとしての実行」を紹介する。ここでは、windows環境にgroovyの環境設定が終了している、CodeNarc-0.12を利用している前提で話を進める。

CodeNarcのダウンロード

http://sourceforge.net/projects/codenarc/files/ から CodeNarc-（バージョン番号）-bin.zipのファイルをダウンロードし任意のディレクトリに解凍する。

コマンドランインからの実行

コマンドラインからの実行は、CodeNarcをJavaの実行ファイルとして実行する。その際に、以下を設定する必要がある。

- GroovyのJarファイル
- CodeNarcのjarファイル
- Log4Jのjarファイル

CodeNarkに付属しているsampleに対して静的コード解析を実行する例は次の通りである。

（展開したディレクトリ）\samples\に移動し、以下のようにRunCodeNarc.batを編集する。

```
@set GROOVY_JAR="%GROOVY_HOME%/embeddable/groovy-all-1.7.6.jar"
java -classpath %GROOVY_JAR%;../CodeNarc-0.12.jar;../lib/log4j-1.2.13.jar org.codenarc.CodeNarc -title=HelloCodeNarc %*
```

1行目は、groovyのjarファイルの場所を指定している。2行目は、CodeNarcのjarファイルの位置、Log4Jのjarファイルの位置及びCodeNarcのオプションを指定している。指定可能なCodeNarcのオプションは以下の表の通りである。オプションを指定しなかった場合は、表に記述してあるデフォルトの値で動作する。

パラメータ	説明	例	デフォルト
-basedir=DIR	解析対象のベースディレクトリを指定する。	-basedir=src/main/groovy	カレントディレクトリ(.)
-includes=PATTERNS	解析対象ファイルのパターンをAnt形式で指定する。	-includes=**/*.gr	.groovyファイルすべて（**/*.groovy）
-excludes=PATTERNS	解析対象からはずすファイルのパターンをAnt形式で指定する。	-excludes=**/templates/**, **/*Test.*	指定なし
-rulesetfiles=FILENAMES	解析するルールを指定する。このオプションでルールのON-OFFをルールセット単位で指定できる。	-rulesetfiles=rulesets/imports.xml, rulesets/naming.xml	rulesets/basic.xml
-report=REPORT-TYPE[:FILENAME]	結果の出力形式をTYPE[:FILENAME]で指定する。 TYPEは、"html", "xml", "text", "console"のいずれかで設定する。出力タイプを独自に定義した場合は、定義したクラスを指定することが可能である。（例：org.codenarc.report.ReportWriter） FILENAMEは出力ファイルのファイル名を指定する。	-report=html -report=html:MyProject.html -report=xml -report=xml:MyXmlReport.xml -report=org.codenarc.report.HtmlReportWriter	html
-title=REPORT TITLE	出力するプロジェクト名を指定する。	-title="My Project"	指定なし
-help	この表のヘルプを表示する。	-help	指定なし

編集したbatファイルを実行すると、実行したディレクトリ直下に、「CodeNarcReport.html」が作成される。このファイルをブラウザ等で開くことによって、以下のような分析結果を参照することが可能である。

Antタスクとしての実行

AntのタスクとしてCodeNarcを実行することが可能である。Antタスクは、build.xmlにCodeNarcエレメント、reportエレメント及びfilesetエレメントを記述することによって定義する。各エレメントの詳細は、http://codenarc.sourceforge.net/codenarc-ant-task.html　を参照のこと。定義したエレメントを実行する際は、コマンドラインから実行するときと同様にant –libオプションで、GroovyのJarファイル、CodeNarcのjarファイル及びLog4Jのjarファイルを指定する必要がある。

CodeNarkに付属しているsampleに対して、Antタスクを利用して静的コード解析を実行する例は次の通りである。

（展開したディレクトリ）\samples\に移動し、以下のように、build.xml及びRunAntTask.batを編集する。

▼build.xml

```
<project name="HelloCodeNarc" >
 <taskdef name="codenarc" classname=
              "org.codenarc.ant.CodeNarcTask"/>
 <target name="runCodeNarc">
  <mkdir dir="reports"/>
  <codenarc
    ruleSetFiles="rulesets/basic.xml,rulesets/
    exceptions.xml,rulesets/imports.xml,rulesets/
    braces.xml">
   <report type="html"
     toFile="reports/CodeNarcAntReport.html"
     title="My Sample Code"/>
   <fileset dir="src">
    <include name="**/*.groovy"/>
   </fileset>
  </codenarc>
  <echo message="Done."/>
 </target>
</project>
```

codenarcエレメントで解析に使用するルールを指定、reportエレメントで出力形式を指定、filesetエレメントで解析対象ファイルを設定している。

▼RunAntTask.bat

```
set GROOVY="%GROOVY_HOME%\embeddable\groovy-all-
1.7.6.jar"
set CODENARC_JAR=../CodeNarc-0.12.jar
set LOG4J_JAR=../lib/log4j-1.2.13.jar
ant -lib %CODENARC_JAR% -lib %GROOVY% -lib
%LOG4J_JAR% -lib lib runCodeNarc
```

ant –libオプションで、GroovyのJarファイル、CodeNarcのjarファイル及びLog4Jのjarファイルを指定している。

編集したbatファイルを実行すると、実行したディレクトリ直下に、「reports」ディレクトリが作成され、「CodeNarcReport.html」が作成される。このファイルをブラウザ等で開くことによって、コマンドラインからの実行と同様に分析結果を参照することが可能である。

まとめ

今回は、静的コード解析ツールのメリットと、Groovy向けの静的コード解析ツールCodeNarcの基本的な操作方法を解説した。CodeNarkの操作方法はそれほど難しくないと感じて頂けたと考えている。

CodeNarkは、静的解析に利用するルールのカスタマイズ機能や各種開発管理ツールと連携する機能を持っている。これらの機能を利用することによってより現実の開発プロセスに静的コード解析を取込みやすくなる。次回以降は、これらの機能を紹介させて頂ければと考えている。

おまけ

jggug合宿で制作したwiki(https://github.com/jggug/simple-wiki）に対して、CodeNarkのbasicルールセットを利用して静的コード分析を行った結果は、以下の通りである。

CodeNarcで分析した結果は、問題はみつからなかった。ルールセットを変えると問題がみつかる可能性がある。

series 04

Gebではじめる Webテスト ～第１回 導入編～

綿引 琢磨（わたびき　たくま）
Groovy/Grails/Griffon/Java 界隈のやさぐれエンジニア。
日本 Grails/Groovy ユーザーグループ運営委員。

はじめに

読者の皆さんは、普段Webアプリケーションでの機能テストはどのように行っていますか？今日では商用・OSSを問わず、様々なWebアプリケーション向けのテストツールが存在していますので、既にWeb機能テストを自動化している方も多いと思いますが、まだまだ手動で実施している方もいるのではないでしょうか。また、テストツールを導入したものの、プロジェクトの開発手法に合わなかったり、テストスクリプトを修正するコストが高く、保守されなくなってしまったケースもあるかもしれません。

本稿では、Groovyを用いたWebアプリケーション向けの機能テストライブラリであるGeb（「ジェブ」と発音します）を用いたWebテストの実施方法を紹介していきます。Gebはアジャイル開発と相性が良く、簡潔なDSLでテストスクリプトを記述することのできるテストライブラリですので、これからWebテストの自動化を検討している方や、現在利用しているテストツールに不満のある方にもお勧めのライブラリです。

初回は導入編として、Gebの概要とインストールから実行までを解説していきます。

Gebとは

Gebは、Luke Daley氏を中心に開発されているWebアプリケーション向けの機能テストを自動化するためライブラリです。Internet Explorer, FireFox, Chrome などのブラウザを操作することが可能なSeleniumのWebDriverとJQueryライクな記述でコンテンツの操作や検査を可能にするAPIを組み合わせ、Groovyの豊かな表現力により簡潔なDSLでテストスクリプトを記述することができます。

また、JUnitやSpockなどのテストフレームワークと統合することも可能なため、TDDやBDDなどの開発プロセスに取り入れやすいですし、使用するブラウザによってはテスト中の画面をキャプチャして出力する機能も提供されていますので、開発者が機能テストやシナリオテストを効率良く行うことができます。

■Gebのコンセプト

Gebは冗長な記述を排除し、より簡単にWebテストを書くことを実現するために、次の２つの概念を取り入れています。

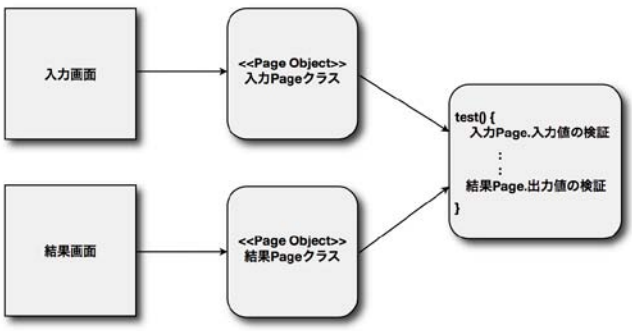
●jQueryライクなコンテンツ操作・検査API

jQueryはDOMの階層構造を意識せずにDOM操作を容易に扱うことができ、今日ではJavaScriptライブラリのデファクトスタンダードとなっているライブラリです。GebはWeb開発者に馴染みのあるjQueryライクなAPIを提供し、テストスクリプトを記述し易くしています。

```
// ページ上の全ての'div'要素を取得
$("div")

// ページ上の最初の'div'要素を取得
$("div", 0)

// ページ上のタイトル属性の値が'section'である全ての
'div'要素を取得
$("div", title: "section")

// ページ上のclass属性が'main'である全ての'div'要
素を取得
$("div.main")
```

●Page Objectパターン

Page ObjectパターンではWebアプリケーションの画面を一連のオブジェクトとして表現し、テスト対象となる画面上のUIコンポーネントをモデル化します。これにより重複したコードを減らすことができますし、UIが変更された場合にもコードの修正が少なくなりますので、変更に強く再利用性の高い実装をすることができます。



■Gebを使用するメリット

OSSだけでもWebアプリケーション向けのテストツールは数多くあります。その内、現在主流と思われるSelenium、Canoo WebTestと比較しながらGebを使用するメリットを示します。

●Seleniumとの比較

Seleniumはキャプチャ／プレイバックツールにカテゴリされるテストツール※です。Seleniumでは実際のブラウザ上でアプリケーションを操作し、その操作記録にassertなどの検証コマンドを追加してあとで再生してテストを行います。このキャプチャ／プレイバックのアプローチは、テストスクリプトが自動生成されるため一見便利なように思うかもしれませんが、問題点もあります。それは一度は手動でアプリケーションを操作しなければな

らないことです。つまり、事前にある程度正常に動作する状態になっていなければ、テストの実施どころか、テストのための記録も取ることができません。

一方、Gebは想定される動作をスクリプトに記述してテストを行うアプローチですので、予めUIコンポーネントのタグ名やidなどがわかっていればテストスクリプトを先に書くことができます。TDDのようにプレゼンテーション層でもテストファーストで開発することができますし、開発チームとテストチームが分かれている場合では並行して作業することもできます。

また、GebではSeleniumのWebDriverをエンジンとして使用していることから、Selenium同様に実際のブラウザを使用したテストを行うことができます。

※Seleniumでもキャプチャ／プレイバック機能を使用せず、JavaやGroovyなどでスクリプトを記述することも可能です。

●Canoo WebTestとの比較

Canoo WebTest（以降、WebTest）はスイスのCanoo Engineering AG社により2002年から開発されているテストツールです。Geb同様スクリプトを記述してテストを行うアプローチを採用しており、通常はテストスクリプトはAntのビルドファイル形式で記述します。スクリプトの記述にはGroovyもサポートされているため、GroovyのAntBuilderを使用して記述することもできますが、簡潔な記述とまでは言い切れません。

これに対してGebでは、前述している通りDSLにより簡潔な記述ができますし、Page Objectパターンを採用することで冗長な記述を排除することができます。簡潔で感覚的にわかりやすいスクリプトを記述できる点は、生産性や保守性を考えると大きなメリットとなります。

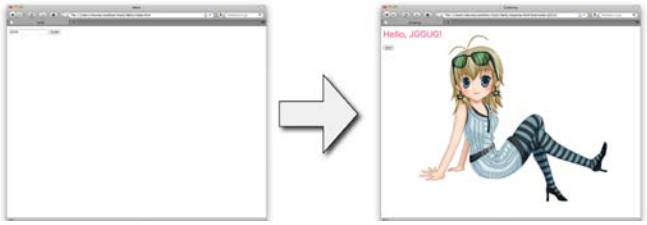
また、WebTestでは実際のブラウザを起動するのではなく、HtmlUnitを使用して内部的にブラウザ上の動作をシミュレートしてテストを行います。ブラウザのシミュレートは実際のブラウザが動作する環境とは異なりますが、Gebではブラウザや動作環境に起因する問題について憂慮する必要はありません。

このように、Gebには他のテストツールにはないメリットがあります。では、実際にGebでのテストがどのようなものか見ていきましょう。

Gebを使用したWebアプリケーションの機能テスト

今回は、Eclipse上で開発するJavaのWebアプリケーションに対して、Gebを使用して機能テストを行います。

対象のアプリケーションはいわゆるHello Worldです。テキストボックスにテキストを入力し【Greet】ボタンを押下すると、次画面に入力されたテキストを含めたメッセージが<h1>タグで表示される単純なものです。



Hello画面　　Greeting画面

■環境設定

Gebを使用するためには、設定をいくつか行う必要があります。

●Eclipseの設定

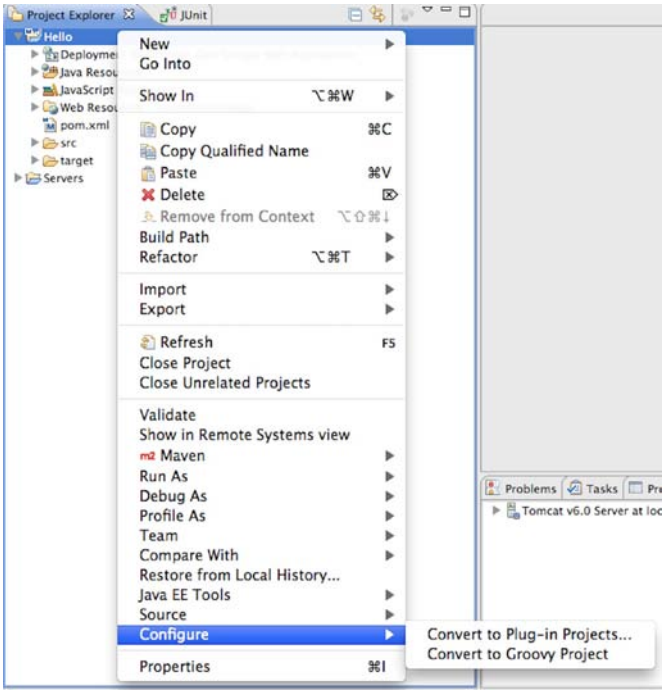
Groovy-Eclipseプラグインのインストール

まずはじめに、Groovy-Eclipseプラグインをインストールします。使用しているEclipseのバージョンに応じたアップデートサイトを指定し、プラグインをインストールをしてください。

- Eclipse 3.6の場合
  http://dist.springsource.org/release/GRECLIPSE/e3.6/
- Eclipse 3.5の場合
  http://dist.springsource.org/release/GRECLIPSE/e3.5/

プロジェクトの変換

次に、当該プロジェクトでGroovyのクラスファイルを扱えるようにするためにプロジェクトの変換を行います。プロジェクトで右クリックし、ポップアップメニュー ＞ Configure※ ＞ Convert to Groovy Project を選択します。



※日本語環境では「構成」です。

●Gebの設定

Gebを使用するにはgeb-core.jarとWebDriverの実装が必要になります。通常はMavenまたはGradleなどのビルドツールまたは@Grab経由で取得します。Gebの現時点での最新バージョンは0.5.1です。ここではpom.xmlを使用して取得します。

▼xml

```
<dependency>
    <groupId>org.codehaus.geb</groupId>
    <artifactId>geb-core</artifactId>
    <version>RELEASE</version>
    <type>jar</type>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-firefox-driver</artifactId>
    <version>RELEASE</version>
    <type>jar</type>
    <scope>test</scope>
</dependency>
```

●テストフレームワークの設定

Gebのテストを Eclipse から実行するために、ここでは JUnit を使用します。

▼xml

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>RELEASE</version>
    <type>jar</type>
    <scope>test</scope>
</dependency>
```

テストケース

このアプリケーションに対して、次のテストを行います。

テスト条件	入力値	テスト手順	期待する結果
Hello 画面が表示されていること	JGGUG	1.テキストボックスに入力 2.Greet ボタンを押下	Greeting 画面に遷移すること Hello, JGGUG! とメッセージが表示されること

テストスクリプトの実装

早速、前述のテストケースを実装してみます。ここでは Page Object パターンを適用する実装と適用しない実装の 2 例を紹介します。

■Page Object パターンを適用しない実装

▼WebTest.groovy

```
package hello;

import org.junit.Test
import geb.Browser

class WebTest {

    @Test
    void helloJGGUG() {
        Browser.drive('http://localhost:8080/Hello') {

            // Hello画面が表示されていること
            assert title == 'Hello'

            // テキストボックスに「JGGUG」と入力
            // Greetボタンを押下
            $('form').username = 'JGGUG'
            def greetButton = $('input', name:'greet')
            greetButton.click()

            // Greeting画面に遷移すること
            assert title == 'Greeting'

            //「Hello, JGGUG!」とメッセージが表示されること
            assert $('h1').text() == 'Hello, JGGUG!'
        }
    }
}
```

■Page Object パターンを適用する実装

Page Object パターンでは画面を 1 つのオブジェクトとして表現します。Hello 画面に対する Page クラスは次のようになります。

▼HelloPage.groovy

```
package pages

import geb.*

class HelloPage extends Page {
    static url = 'http://localhost:8080/Hello'
    static at = { title == 'Hello' }
    static content = {
        username { $('form').username }
        greetButton(to:GreetingPage) {
            $('input', name:'greet')
        }
    }
}
```

Greet 画面に対する Page クラスも同様です。

▼GreetingPage.groovy

```
package pages

import geb.*

class GreetingPage extends Page {
    static at = { title == 'Greeting' }
    static content = {
        h1Text { $('h1').text() }
    }
}
```

これらのPageクラスを適用したテストの実装は次のようになります。

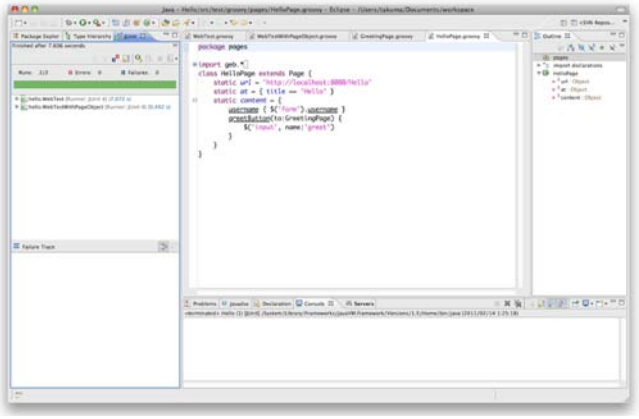
▼WebTestWithPageObject.groovy

```
package hello

import org.junit.Test
import geb.Browser
import pages.*

class WebTestWithPageObject {

    @Test
    void helloJGGUG() {
        Browser.drive(HelloPage) {

            // Hello画面が表示されていること
            assert at(HelloPage)

            // テキストボックスに「JGGUG」と入力
            // Greetボタンを押下
            username = 'JGGUG'
            greetButton.click()

            // Greeting画面に遷移すること
            assert at(GreetingPage)

            //「Hello, JGGUG!」とメッセージが表示されること
            assert h1Text == 'Hello, JGGUG!'
        }
    }
}
```

Page Objectパターンを適用すると、画面内のUIコンポーネントに関するコードをPageクラス内に隠蔽することができるため、テストスクリプトが簡潔になり感覚的にもわかりやすくなります。

テストの実行

テストの実行は、Eclipseに統合したServerを起動しておきJUnitテストを実行します。結果はJUnitビューに表示されます。



テストフレームワークとの統合

ここまでは、JUnitを使用してテストを実装してきましたが、GebにはテストフレームワークとGebの統合するためのjarが別途用意されていますので、こちらも試してみましょう。

pom.xmlに次の設定を追加してください。

▼xml

```
<dependency>
    <groupId>org.codehaus.geb</groupId>
    <artifactId>geb-junit4</artifactId>
    <version>RELEASE</version>
    <type>jar</type>
    <scope>test</scope>
</dependency>
```

このgeb-junit4.jarにはGroobyTestCaseクラスを継承したGebTestクラスと、GebTestクラスにレポート出力機能を追加したGebReportingTestクラスがあります。GebTestクラスを使用するとBrowserクラスが隠蔽されるため、よりシンプルな実装となります。

```
@RunWith(JUnit4)
class WebTestWithJUnitIntegration extends GebTest {
    @Test
    void helloJGGUG() {
        to HelloPage
        assert at(HelloPage)

        username = 'JGGUG'
        greetButton.click()
        assert at(GreetingPage)

        assert h1Text == 'Hello, JGGUG!'
    }
}
```

■レポートの出力

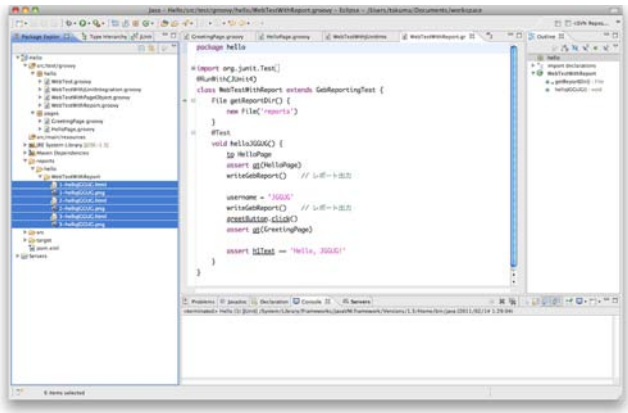
FirefoxのWebDriver実装を使用する場合は、GebReportingTestクラスを継承することでテスト実行時の画面をキャプチャして出力することができます。その際にはレポート出力先のディレクトリを指定するgetReportDir()メソッドをオーバーライドする必要があります。

```
@RunWith(JUnit4)
class WebTestWithReport extends GebReportingTest
{
    File getReportDir() {
        new File('reports')
    }
            :
}
```

通常、レポート出力はJUnitの@After時に出力されますが、明示的にwriteGebReport()を呼び出すことにより任意のタイミングでレポート出力をすることもできます。テストのエビデンスとしてフォーム入力時の画面を出力したい場合などに使用すると良いでしょう。

```
@Test
void helloJGGUG() {
    to HelloPage
    assert at(HelloPage)
    writeGebReport()    // 初期表示の状態で出力

    username = 'JGGUG'
    writeGebReport()    // テキスト入力後の状態で出力
    greetButton.click()
    assert at(GreetingPage)

    assert h1Text == 'Hello, JGGUG!'
}
```

このテストクラスを実行するとレポート出力先に指定したディレクトリ配下にテストクラスのパッケージ名、テストクラス名のディレクトリが生成されてhtmlとpngファイルが出力されます。



レポート出力結果の画像は以下のようになります。

まとめ

今回はGebの概要とJavaプロジェクトへの導入までを行いました。GebはSpockやEasybなどのBDDテストフレームワークと統合することでシナリオテストを自動化することができますし、Grailsアプリケーションにプラグインとして導入することもできます。残念ながら今回は導入部分しか紹介できませんでしたが、Gebは様々なWebアプリケーションのテストに利用可能なライブラリですので、読者の皆さんにも実際に試していただいてGebの魅力を感じていただけたらと思います。

次回はGebが提供する主要なクラスの詳細と各種APIについて紹介する予定です。ご期待ください。

Grails Plugin 探訪

第 2 回

~ Spock プラグイン ~

URL: http://www.grails.org/plugin/spock
プラグインのバージョン: 0.5
対応する Grails のバージョン: 1.2 以上

杉浦孝博

最近は Grails を使用したシステムの保守をしている自称プログラマ。
日本 Grails/Groovy ユーザーグループ事務局長。
共著『Grails 徹底入門』、共訳『Groovy イン・アクション』

はじめに

今回紹介する Grails プラグインは、Spock プラグインです。
このプラグインを組み込むことにより、Spock を使ってテストケースを記述することができます。

Spock とは

Spock は、Java や Groovy 用のテストと仕様のためのフレームワークです。
Spock については、次の URL を参照してください。
http://spockframework.org/

プラグインのインストール

プラグインのインストール手順は、Grails のバージョンによって異なります。

■Grails 1.2.x の場合

Grails 1.2.x の場合、次のとおりです。

```
$ grails install-plugin spock 0.5-groovy-1.6
```

■Grails 1.3.x の場合

Grails 1.3.x の場合、次のとおりです。

```
$ grails install-plugin spock 0.5-groovy-1.7
```

または

```
$ grails install-plugin spock
```

これ以降、Grails 1.3.6 を前提として説明します。

テストのケースの作成

本をあらわす次のドメインクラスがあります。

```
package pluginspock

class Book {

    String title
    Integer price

    static constraints = {
        title()
        price(min: 0)
    }
}
```

このドメインクラスのテストケースを作成します。
ファイル自体は、test/unit ディレクトリに作成します。
クラス/ファイル名は、"Spec" あるいは "Specification" で終わる必要があります。
ここでは、test/unit/BookSpec.goorvy とします。

```
package pluginspock

import grails.plugin.spock.UnitSpec

class BookSpec extends UnitSpec {

    def '本のタイトルで検索できる'() {
        setup:
            mockDomain(Book)

        when: '本の保存'
            new Book(title: title, Price: price).save()

        then: 'mockDomainを使ってタイトルで検索'
            def book = Book.findByTitle(title)
            book != null
            book.title == title
            book.price == price

        where:
            title = 'Groovyイン・アクション'
            price = 3500
    }
}
```

Spockプラグインでは、SpockのSpecificationクラスを継承したテストクラスを提供しています。

ユニットテスト用にUnitSpecというクラスがありますので、このクラスを継承してドメインクラスのテストケースを作成します。

テストケースの実行

テストケースの実行は、test-appコマンドで実行します。

```
$ grails test-app
```

Spockのテストケースだけ実行することもできます。

```
$ grails test-app :spock
```

いろいろなテストケースクラス

Spockプラグインは、UnitSpec以外にも、複数のテストクラスを提供しています。

どのようなテストクラスがあるのか、簡単に説明します。

■UnitSpec

ユニットテスト用のテストクラスです。UnitSpecを継承するテストケースは、test/unitディレクトリに作成します。

```
package pluginspock

import grails.plugin.spock.UnitSpec

class BookSpec extends UnitSpec {

    def '本のタイトルで検索できる'() {
        setup:
            mockDomain(Book)
        when: '本の保存'
            new Book(title: title, Price: price).save()
        then: 'mockDomainを使ってタイトルで検索'
            def book = Book.findByTitle(title)
            book != null
            book.title == title
            book.price == price
        where:
            title = 'Groovyイン・アクション'
            price = 3500
    }
    def 'titleのバリデーション'() {
        setup:
            mockForConstraintsTests(Book)
        when:
            def book = new Book(title: title, price: 3500)
        then:
            book.validate(['title']) == validationResult

        where:
            title << [null, 'Groovyイン・アクション']
            validationResult << [false, true]
    }
    def 'priceのバリデーション'() {
        setup:
            mockForConstraintsTests(Book)
        when:
            def book = new Book(title: 'Groovyイン・アクション', price: price)
        then:
            book.validate(['price']) ==
                validationResult
        where:
            price << [-1, 0, 1]
            validationResult << [false, true, true]
    }
}
```

■ControllerSpec

コントローラ用のテストクラスです。
ドメインクラスBookのコントローラクラス

```
package pluginspock

class BookController {

    static allowedMethods = [save: "POST",
        update: "POST", delete: "POST"]

    def index = {
        redirect(action: "list", params: params)
    }

    def list = {
        params.max = Math.min(params.max ?
                params.int('max') : 10, 100)
        [bookInstanceList: Book.list(params),
                bookInstanceTotal: Book.count()]
    }

    // 以下省略
}
```

に対するテストクラスは次のとおりです。ControllerSpecを継承するテストケースは、test/unitディレクトリに作成します。

```
package pluginspock

import grails.plugin.spock.ControllerSpec

class BookControllerSpec extends ControllerSpec {

    def 'indexアクションのテスト'() {
        when:
            controller.index()
        then:
            redirectArgs.action == 'list'
            redirectArgs.params.size() == 0
    }

    def 'listアクションのテスト'() {
        when:
         Book.metaClass.static.list = { params ->
                books
            }
            Book.metaClass.static.count = {
                books.size()
            }
        then:
            controller.list() == [bookInstanceList:
                books, bookInstanceTotal: books.size()]
        where:
            books = [
           new Book(title: 'Groovyイン・アクション', price: 3500),
           new Book(title: 'Grailsイン・アクション', price: 3600),
            ]
    }
}
```

■TagLibSpec

タグリブ用のテストクラスです。
ドメインクラスBookを引数にとるタグリブクラス

```
package pluginspock

class BookTagLib {

    def print = { attrs, body ->
        def book = attrs.book
        out << "<book title=\"${book.title}\"
                        price=\"${book.price}\">"
        out << body()
        out << '</book>'
    }
}
```

に対するテストクラスは次のとおりです。TagLibSpecを継承するテストケースは、test/unitディレクトリに作成します。

```
package pluginspock

import grails.plugin.spock.TagLibSpec

class BookTagLibSpec extends TagLibSpec {

    def 'bookタグのテスト'() {
        expect:
            print(book: book) { 'オススメです。' } == '<book title="Groovyイン・アクション" price="3500">オススメです。</book>'

        where:
            book = new Book(title: 'Groovyイン・アクション', price: 3500)
    }
}
```

■IntegrationSpec

インテグレーションテスト用のテストクラスです。
サービスクラス

```
package pluginspock

class BookService {

    static transactional = true

    def save(title, price) {
        new Book(title: title, price: price).
                                save(flush: true)
    }

    def update(book, title, price) {
        book.title = title
        book.price = price
        book.save(flush: true)
    }
}
```

に対するテストクラスは次のとおりです。IntegrationSpecを継承するテストケースは、test/integrationディレクトリに作成します。

```
package pluginspock

import grails.plugin.spock.IntegrationSpec

class BookServiceSpec extends IntegrationSpec {

    def bookService

    def 'saveメソッドのテスト'() {
        when:
            bookService.save(title, price)

        then:
            Book.findByTitleAndPrice(title, price) != null
            Book.count() == 1

        where:
            title = 'Groovyイン・アクション'
            price = 3500
    }

    def 'updateメソッドのテスト'() {
        when:
            def book = bookService.save(oldTitle, oldPrice)
            bookService.update(book, newTitle, newPrice)

        then:
            Book.findByTitleAndPrice(newTitle, newPrice) != null
            Book.count() == 1

        where:
            oldTitle = 'Groovyイン・アクション'
            oldPrice = 3500
            newTitle = '改訂版Groovyイン・アクション'
            newPrice = 4000
    }
}
```

■GroovyPagesSpec

GSP用のテストクラスです。

先述のBookTagLib#printを使用したGSPコードのテストクラスは次のとおりです。GroovyPagesSpecを継承するテストケースは、test/integrationディレクトリに作成します。

```
package pluginspock

import grails.plugin.spock.GroovyPagesSpec

class BookTagSpec extends GroovyPagesSpec {

    def 'printタグのテスト'() {
        when:
            template = '<g:print book="${book}">
これは良い本です。</g:print>'
            params = [book: book]

        then:
            output == '<book title="Groovyイン・
アクション" price="3500">これは良い本です。</book>'

        where:
            book = new Book(title: 'Groovyイン・
アクション', price: 3500)
    }

}
```

機能テストについて

Spockプラグインでは、機能テストは提供していません。機能テスト用のプラグインとして、Gebプラグインなどがあります。

最後に

Spockはとても強力なテストフレームワークですので、Grailsアプリケーションのテストケースを Spockで書いてみてはいかがでしょうか？

リリース情報 2011.02.11

Grails

Grailsは、GroovyやHibernateなどをベースとしたフルスタックのWebアプリケーションフレームワークです。
URL: http://grails.org/
バージョン: 1.2.5, 1.3.6
■更新情報

- 1.3.6では、beforeValidateのサポートやSpringライブラリの更新など、いくつか新機能の追加と改良が行われています。
- 1.3.6リリースノート: http://www.grails.org/1.3.6+Release+Notes

Groovy

Groovyは、JavaVM上で動作する動的言語です。
URL: http://groovy.codehaus.org/
バージョン: 1.7.7, 1.8-beta-4
■更新情報

- 1.7.7では、スタブ・ジェネレータや@Delegateなどのバグフィックスや改良がいくつか行われています。
- 1.7.7リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=10242&version=17020
- 1.8-beta-4では、JSONのサポートやGParsライブラリ同梱など、バグフィックスと改良がいくつか行われています。
- 1.8-beta-4リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=10242&version=17015

Griffon

Griffonは、デスクトップアプリケーションを開発するためのアプリケーションフレームワークです。
URL: http://griffon.codehaus.org/
バージョン: 0.9.1a, 0.9.2-rc1
■更新情報

- 0.9.2-rc1では、AddonManagerやコントローラでの自動スレッド管理など、いくつかのバグフィックスと改良が行われています。
- 0.9.2-rc1リリースノート: http://docs.codehaus.org/display/GRIFFON/Griffon+0.9.2-rc1

Gant

Gantは、XMLの代わりにGroovyでAntタスクを記述し実行するビルド管理ツールです。
URL: http://gant.codehaus.org/
バージョン: 1.9.3

Gradle

Gradleは、Groovyでビルドスクリプトを記述し実行するビルド管理ツールです。
URL: http://www.gradle.org/
バージョン: 0.9.2
■更新情報

- 0.9.2では、DSLの改良など、いくつかのバグフィックスと改良が行われています。
- 0.9.2リリースノート: http://docs.codehaus.org/display/GRADLE/Gradle+0.9.2+Release+Notes

Gaelyk

Gaelykは、Groovyで記述するGoogle App Engine for Java用のライトウェイトなフレームワークです。
URL: http://gaelyk.appspot.com/
バージョン: 0.6.1
■更新情報

- 0.6.1では、プラグインのリロードに関するバグフィックスなどが行われています。
- 0.6.1リリースノート: http://gaelyk.appspot.com/download

Google App Engine SDK for Java

Google App Engine SDK for Javaは、JavaでGoogle App Engine用のWebアプリケーションを開発するためのSDKです。
URL: http://code.google.com/intl/ja/appengine/
バージョン: 1.4.2
■更新情報

- 1.4.2では、JDKでデータベース・インデックスのバキュームが行えるようになったり、XMPP APIが更新されたりと、い

くつかのバグフィックスと改良が行われています。
- 1.4.2 リリースノート: http://code.google.com/p/googleappengine/wiki/SdkForJavaReleaseNotes

GPars

GParsは、Groovyに直感的で安全な並行処理を提供するシステムです。
URL: http://gpars.codehaus.org/
バージョン: 0.11GA
■更新情報
- 0.11GAでは、Actorのスピードアップや Memoize(メモ化)など、いくつかのバグフィックスと改良および新機能の追加が行われています。
- 0.11GA リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=12030&version=16391

Groovy++

Groovy++は、Groovy言語に対して静的な機能を拡張します。
URL: http://code.google.com/p/groovypptest/
バージョン: 0.4.155
■更新情報
- 0.4.155では、いくつかの重要なバグフィックスが行われています。
- 0.4.155 リリースコメント: http://groups.google.com/group/groovyplusplus/browse_thread/thread/9ac29082ab33b0d9?pli=1

Spock

Spockは、JavaやGroovy用のテストと仕様のためのフレームワークです。
URL: http://code.google.com/p/spock/
バージョン: 0.5
■更新情報
- 0.5では、Hamcrestライブラリのサポートやビルトイン拡張の追加など、いくつかのバグフィックスと改良および新機能の追加が行われています。
- 0.5 リリースコメント: http://groups.google.com/group/spockframework/browse_thread/thread/6f84881e99d83531

GroovyServ

GroovyServは、Groovy処理系をサーバとして動作させることでgroovyコマンドの起動を見た目上高速化するものです。
URL: http://kobo.github.com/groovyserv/
バージョン: 0.5
■更新情報
- 0.5では、起動オプションの追加など、いくつかのバグフィックスと改良および新機能の追加が行われています。
- 0.5 チェンジログ: http://kobo.github.com/groovyserv/changelog.html#0.5

Geb

Gebは、Groovyを使用したWebブラウザを自動化する仕組みです。
URL: http://geb.codehaus.org/
バージョン: 0.5.1
■更新情報
- 0.5.1では、バグフィックスが行われています。
- 0.5.1 リリースノート: http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=12101&version=17012

G* Magazine vol.1 2011.02
http://www.jggug.org

発行人：日本 Grails/Groovy ユーザーグループ
編集長：川原正隆
編集：G* Magazine 編集委員（杉浦孝博、奥清隆）
デザイン：㈱ニューキャスト
表紙：川原正隆
編集協力：JGGUG 運営委員会
Mail：info@jggug.org

Publisher：Japan Grails/Groovy User Group
Editor in Chief：Masataka Kawahara
Editors：G* Magazine Editors Team
(Takahiro Sugiura, Kiyotaka Oku)
Design：NEWCAST inc.
CoverDesign：Masataka Kawahara
Cooperation：JGGUG Steering Committee
Mail：info@jggug.org