SCOTTSDALE GROOVY BRIGADE

David Kuster
dave@choicetrax.com

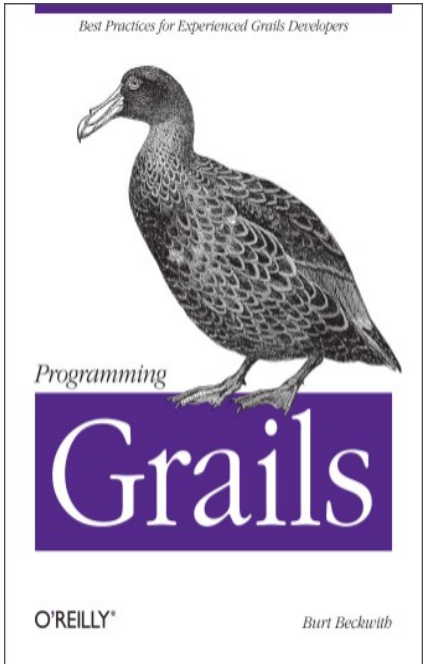
Grails Best Practices

www.scottsdale-groovy.org
twitter: @groovyandgrails

A Semi-Opinionated Look at Making Grails Apps Better

First, Some References

- http://groovy.dzone.com/articles/grails-best-practices
- http://www.infoq.com/articles/grails-best-practices
- Burt Beckwith – Programming Grails



Starting at the start

Domain Driven Design

- Rich Domain Model
  - Business logic specific to the domain
- Not an Anemic Domain Model
  - Bag of getters and setters

```
class Domain {

  def retrieveSpecialInstances() {}

  def updateDetails( String name, etc ) {}

  def determineCountOfWhatever() {}

}
```

Getting ahead a bit

- Generally push everything down to the lowest level possible – this simplifies all the levels above
  - Don't do in a view what you can do in a controller
  - Don't do in a controller what you can do in a service
  - Don't do in a service what you can do in a domain class

```
// antipatterns

// views
domainObj.retrieveSpecialInstances()?.sort {
  it.name }

// controllers
def update() {
  def obj = Domain.get(params.id)
  bindData( obj, params )
  obj.save()
}

// services
def determineCountOfWhatever() {}
```

Keep Domains Atomic

- Don't do cross-domain work
  - Except for some simple managing of existing relationships
- Don't inject services into domains
- Don't expose internal variables
  - “T” or “F” for enabled/activated

```
class Domain {
  String enabled = “F”

  boolean isEnabled() {
    enabled == “T”
  }
}

// calling code
if ( domain.isEnabled() ) { .. }

// instead of
if ( domain.enabled == “T” ) { .. }
```

Other Domain Best Practices

- Complicated query logic
  - Name these methods retrieve*()
  - Distinguishes them from dynamic find*() methods
- Util methods to navigate complicated relationships
  - Loose coupling of calling code
- Named Queries
  - Implementation detail
  - I'm hesitant to expose those outside the domain

```
def retrieveSubObjsForType( type ) {
  Domain.withCriteria {
    eq( "domain.id", this.id )
    collectionOfSubObjs {
      eq( "subObj.id", type.id )
    }
  }
}

boolean hasCertainAssociations() {
  this.collection?.any {
    it.otherCollection?.size() > 0 }
}

def getAssociationCount() {
  collection*.subObjs?.sum { it.size() }
}
```

Controllers

- Actions should be methods, not closures
  - Better performance
- Keep 'em thin
  - No logic
  - Limited to no DB operations
  - No transactions
- Should primarily deal with navigation flow
- Define and (usually) type input variables in method signature
- Use command objects for more complex inputs and/or large form submission

```
// this is a DB operation, but oh well
def edit( Long id ) {
  [ obj:Domain.get(id) ]
}

// define and type input params
def update( Long id, String name, String type,
Long statusCode ) { }

// command obj input
def update( MyCommandObj cmd ) { }
```

Controller Exception Handling

- Just because you don't HAVE to catch exceptions, doesn't mean you SHOULDN'T
  - Common example is to redirect everything to a common error page
- try/catch block around probably all service calls – especially ones that update data
- Let exceptions influence navigation
- Info and error messages in flash

```
// more about navigation flow
def update( Long id, String name, etc ) {
  if ( request.method == 'POST' ) {
    try {
      myService.updateData( id, name, etc )
      flash.message = “Data updated”
      redirect( action:'show', id:id )
    }
    catch ( e ) {
      handleException( “Could not update”, e )
      // could redirect somewhere else here
      // or can let line below return to update page
    }
  }

  // get data for editing/updating, instead of separate action
  [ obj:Domain.get( id ) ]
}
```

Handling Those Exceptions

- Shared handleException() method
- Deal with error handling, logging consistently
- Base controller or metaclass/AST transform if you want to get fancy

```
def handleException( String msg, Exception e ) {
  if ( e instanceof DomainValidationException ) {
    flash.errors =
      e.domain.errors?.fieldErrors?.toList()?.unique {
        it.defaultMessage }
    log.info( msg, e )
  }
  else if ( e instanceof ... ) {
    // deal with other types
    log.warn( msg, e )
  }
  else {
    flash.errorMsg =
      “Something unexpected happened: $msg”
    log.error( msg, e )
  }
}
```

Services

- Complex, multi-business domain logic
  - Updates
  - Data retrieval
- Shouldn't know where they're being invoked from – web request, AJAX, API, etc
  - Don't pass params obj
  - Define service API more strongly
- Input validation
  - Smells like business logic

```
def activateUser( UserCmd cmdObj ) {
  // validate input data
  // set user to active
  // initialize/config associated objects
  // save data
  // send notification email
  // etc
}
```

Services and Transactions

- All DB writes should happen in a transactional service
- Complex reads can be in a non-transactional service
  - Can also specify @Transactional(readOnly=true) if concerned about dirty reads
    - Note that one @Transactional is the same as:
      - static transactional = false
- failOnError=true
  - Silent save failures can be difficult to debug

Doing Those Saves

- Shared doSave() method
- Deal with validation checks consistently
  - Note validate:false
  - Don't validate twice, especially with custom validators doing queries
- Ensure DB rollbacks
- Base service, metaclass/AST, or separate SaveService

```
class SaveService {
  static transactional = true

  void doSave( domainObj, flush=false ) {
    if ( domainObj.validate() )
      domainObj.save( flush:flush,
                      validate:false )
    else
      throw new DomainValidationException(
        "${domainObj.getClass()} obj failed
validation",
        domainObj )
  }

  // or params as map ("flush:true")
  // could make validation optional
  // should prob check Hibernate version

}
```

Rollback Gotcha

- Throwing RuntimeException to rollback transaction will detach all current objects from Hibernate session
- Will need to redirect in controller or .get()/.read() any objs before passing to view
- Probably avoid .merge()/.attach()
  - If exception was thrown due to bad data, don't want to try and put it back in session

```
def update( MyCmdObj cmd ) {
  try {
    myService.doUpdate( cmd )
    flash.message = 'Update successful'
    redirect( action:'show', id:cmd.id )
  }
  catch ( e ) {
    handleException( “Update failed”, e )
    // if we want to return to update view
    [ domainObj:Domain.get(cmd.id) ]
  }
}
```

Views

- Use templates and taglibs for repeated content
- Flash message/error display in layout (main.gsp)
- Avoid scriplets – we're not writing JSPs

```
<!-- main.gsp -->
<g:if test="${flash.errorMsg ||
flash.errorObj}">

  <div id=”errors”>
    ${flash.errorMsg}

    <g:hasErrors bean="${flash.errorObj}">
      <g:renderErrors
        bean="${flash.errorObj}" as="list"/>
    </g:hasErrors>
  </div>

  <!-- clear flash data, once shown -->
  <% flash.errorMsg = null %>
  <% flash.errorObj = null %>
  <!-- but don't use scriptlets :) -->
</g:if>
```

No Logic In Those Views

- No logic or DB queries
- Multiple if/else statements – use a taglib
- Be wary:
  - <%@page import=”com.package.Class” %>
  - collect, grep, join, set
  - Any time you feel like you're writing Groovy code
- Probably can't get away from navigating domain relationships, but view shouldn't know too much

```
<!-- get these out of your GSPs -->

<g:each var=”obj” from=”Domain.list()?.sort
{ it.name }”>

<g:each var=”obj”
from=”Domain.findByActivated(“T”)?.sort
{ rand() }”>

<g:if test=”${obj instanceof X}”>
<g:elseif test=”${obj instanceof Y}”>
<g:elseif test=”${obj instanceof Z}”>
```

Command Objects

- Automatic data binding is great
- Theoretically do validation, but I've encountered flakiness
- Can reuse as DTOs for complex view generation
- src/groovy instead of defined inline in controller
- Can nest command objs
  - Use Apache Commons Collections Utils classes

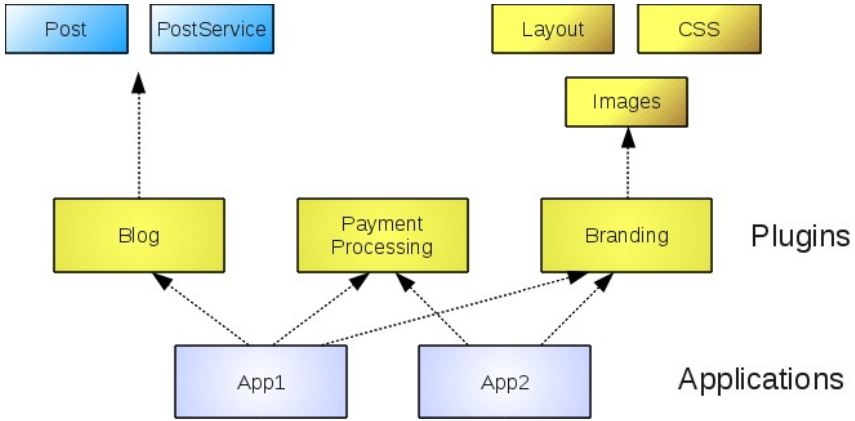
```
@Validateable
class MyCommandObj {
  Long myNum
  List nestedList = ListUtils.lazyList(
    [] as LinkedList,
    FactoryUtils.instantiateFactory(
      MyOtherCmdObj))
  Map nestedMap = MapUtils.lazyMap([:],
    FactoryUtils.instantiateFactory(
      MyOtherOtherCmdObj))
}

<g:textField name=”myNum” value=”1” />
<g:textField name=”nestedList[0].varName” />
<g:textField
  name=”obj['x'].subObj['y'].otherVar” />
```

grails-app/util vs src/groovy vs src/java

- grails-app/util should really only be used for codecs
  - .encodeAsXYZ()
  - .decodeAsXYZ()
- Util classes should go in src/groovy
  - Custom exceptions, enums, DTOs, cmd objs, rules classes, etc
- src/java = ?
  - Perhaps good if you need to interface with Java code (Java DTOs for GWT)

Plugins

- Think about breaking out larger logical app components
- Can provide cleaner separation/looser coupling between different parts of the app
- Can go so far as to embrace a Plugin Oriented Architecture
  - Share among multiple apps
  - Decompose one large app
    - Incremental deploy/updates
    - Scalability
    - Reliability
  - Complexity vs. needs



(Peter Ledbrook - http://blog.springsource.org/2010/06/01/whats-a-plugin-oriented-architecture/)

Groovy Tips

- Learn the goodness
- Collections methods
- Don't get too Groovy
  - Readable
  - Maintainable
- Don't get too lazy with def
  - good to specify type on one side of an assignment

```
*.  <=>  ?.  ?:  <<

.each { }, .eachWithIndex { i -> }, .collect
{ }, .find { }, .findAll { }, .sort { }, .any
{ }, .every { }

// you could, but don't (outside of Twitter)
m={s,n,c->[0:s].withDefault{''}[n%c]};
(1..100).findResults{n->
m('fizzbuzz',n,15)?:m('fizz',n,3)?:m('buzz',n,5)?:n
}
// @tim_yates
(1..100).collect{n->[0:'fizzbuzz'].get(n%15,
[0:'fizz'].get(n%3,[0:'buzz'].get(n%5,n)))}

// type is obvious
def x = new MyObject()
// this creates what?
def x = createObj()
// better
MyObject x = createObj()
```

Database Performance Tips

- Navigating the domain structure = GORM/Hibernate queries
  - logSql = true
    - Prepare to be horrified
- Don't do individual queries in a loop
- Use .read() instead of .get() for objs that won't be modified
  - Better memory performance
  - Not really read only, can still change values and call .save()

```
// loads full program and status objects
Program.findByStatus(Status.findByName("Completed")).department*.name
// loads just the data needed, in one query
Program.createCriteria.list() {
  projections {
    department {
      property( 'name' )
    }
  }
  status {
    eq( 'name', 'Completed' )
  }
}

// multiple queries
[1,2,3].each { list << Domain.get(it) }
// one query
list << Domain.getAll([1,2,3])
```

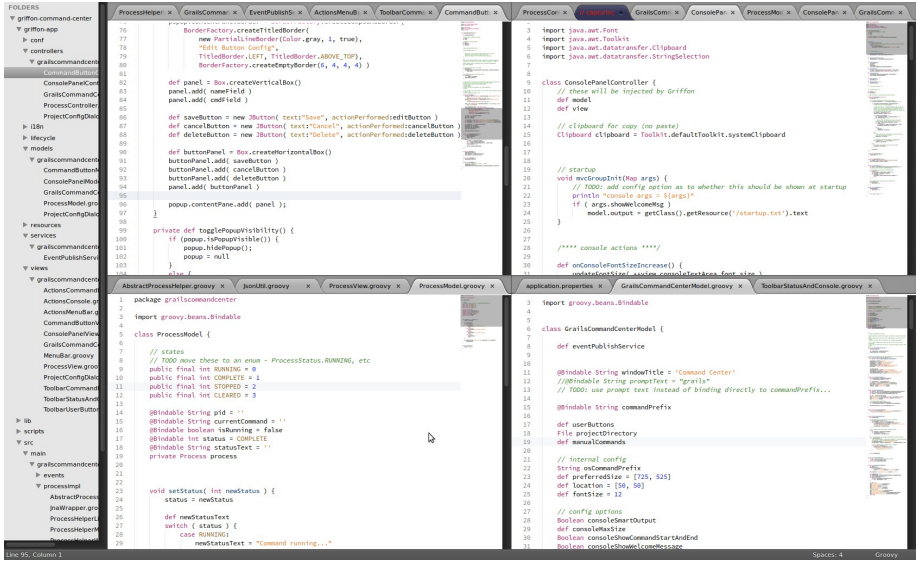
Additional Domain/DB Tips

- Use withNewSession in custom validators that need to do DB checks
  - Hibernate flushes queued updates before executing queries
  - withNewSession avoids unwanted/unexpected flush
- Use @EqualsAndHashCode on domains
  - Can have problems comparing proxy objs to actual domain instances
  - Or implement Comparable interface
- Use .read() instead of .get() for objs that won't be modified
  - Better memory performance
  - Not really read only, can still change values and call .save()

```
// access db to check for duplicate obj
Domain.withNewSession { session →
  Domain.createCriteria().list() {
    ilike( 'name', name )
    if ( id ) {
      ne( 'id', id )
    }
    maxResults( 1 )
  }
}

// comparable, maybe be as simple as
int compareTo(that) {
  id <=> that?.id
}
```

General Development Tips

- Web Console Plugin
  - Write code here, verify it works
  - Especially good for domain logic to avoid app restarts/reloading flakiness
  - Great way to experiment with Groovy
- Sublime Text
  - Grid layout – view 4 artefacts simultaneously
  - Work in smaller windows = smaller methods
- Code Coverage / Cobertura Plugin
  - Code coverage reports when running tests



Resources For Ongoing Tips

- This Week In Grails
  - http://burtbeckwith.com/blog
- Groovy Blogs
  - http://groovyblogs.org
- Tomas Lin
  - http://fbflex.wordpress.com

Future Presentation Suggestions

- Plugin Oriented Architectures
- Plugin development in general
- Command objs as primary app artefact
- Other application architectures
- Integrations with other frameworks/technologies
  - Vert.x / web sockets / JMS / etc
- Same presentation as today, except showing how this is all wrong

Next Meeting

July 2 @ Noon

Red Jasper room

Topic = ?