Writing Plugins

In this session

- Motivations for creating Grails plugins
- How to create and install plugins
- Capabilities available to plugin developers

Public and private plugins

There are over 1,100 plugins on grails.org but not all plugins are published and shared there

You can create private plugins for personal use or to be used in your organization

Why make a plugin?

Wrap an existing library or tool

- Spring Security, Quartz, DB Migrations (Liquibase)

Provide a new feature

- Asset Pipeline, Build Test Data

Plugins for modular applications

- Use your imagination - could be anything

Plugins for modular applications

Share Grails artefacts, Java and Groovy source, and static resources across your Grails projects

- domain, service, controller, taglib, gsp, script
- src/groovy, src/java
- css, js, images

Plugin project structure

A newly created plugin project is almost identical to an application project

The primary and most interesting difference is the **plugin descriptor** in the root directory of the plugin project named ***GrailsPlugin.groovy**

Create a plugin project

Where you would use create-app for an application project, instead use **create-plugin**

```
> grails create-plugin event-log
```

Plugin descriptor, part 1 - metadata

General information about the plugin - title, author, description, links to plugin source, documentation, issue tracking, license

Version your plugin here and indicate which versions of Grails are compatible

Exclude resources from plugin packaging

Install a local plugin

A plugin can be installed locally for integration into application(s)

```
> grails maven-install
```

Add installed plugin to an app

Add the installed plugin to an application the same way you would any other plugin, in the plugins section of BuildConfig.groovy

```
plugins {
    compile ":event-log:0.1"
}
```

Inline plugins

Test your plugin in an application without packaging and installing - BuildConfig.groovy:

```
grails.plugin.location.'event-log'=
                          '/Users/me/projects/event-log'
```

Or use relative path:

```
grails.plugin.location.'event-log'='../event-log'
```

Workshop

Create a plugin project called event-log in the same directory as the questionApp project

```
> grails create-plugin event-log
```

Workshop

Modify the plugin descriptor to provide standard information in the top section (metadata). In the plugin project's root directory:

EventLogGrailsPlugin.groovy

Workshop

Create a domain, com.opi.Event

```
class Event {
    Date eventTime
    String eventSource

    static hasMany = [details: EventDetail]
    static constraints = {
        eventSource size: 4..100, blank: false
        details size: 1..50
    }
}
```

Workshop

Create a domain, com.opi.EventDetail

```
class EventDetail {
    String name
    String value

    static constraints = {
        name size: 4..200, blank: false
        value size: 1..200, blank: false
    }
}
```

Workshop

Create a service, com.opi.EventService and add a method with this signature to save and return an Event domain:

```
Event createEvent(Date eventTime, String eventSource,
                  Map<String, String> eventDetails){

//create, save and return an instance of Event domain
}
```

Workshop

1. Install your plugin in the local maven cache
2. Integrate the plugin into the questionApp via BuildConfig.groovy
3. In the VotingService when a vote is added to a question or answer use the EventService method provided by the plugin to create and save an Event (any key/value pairs are OK for EventDetails)

Workshop

1. Run the questionApp and use the dbconsole to see the additional tables for the plugin's domains

2. Vote on questions or answers and inspect the new tables via dbconsole

Workshop - extra

1. Change the questionApp to inline the event-log plugin

```
grails.plugin.location.'event-log'='../event-log'
```

2. Make changes in the plugin source and watch them affect the questionApp without re-installing the plugin to the maven cache

Plugin descriptor, part 2 - life cycle callbacks

doWithWebDescriptor

doWithSpring

doWithDynamicMethods

doWithApplicationContext

onChange

onConfigChange

onShutdown

Evaluating conventions

Every plugin has an implicit `application` variable of type GrailsApplication

GrailsApplication stores information about all artefact classes in your application

Artefact classes implement the GrailsClass interface

Evaluating conventions

Examples:

All artefact classes

```
for (grailsClass in application.allClasses) {
    println grailsClass.name
}
```

A specific Service class

```
def mailer = application.getServiceClass("MailService")
if(mailer.hasProperty("foo")){ println "Mailer has foo!" }
```

doWithWebDescriptor

Add to the generated /WEB-INF/web.xml

- add servlets and servlet mappings
- add filters and filter mappings

```
def doWithWebDescriptor = { xml ->
    //add servlets and filters to web.xml
}
```

`xml` is an XmlSlurper GPathResult - see docs for examples

doWithSpring

Add beans to the Spring application context using the Groovy bean builder DSL (like resources.groovy)

```
def doWithSpring = {
    myBean(MyBeanClass){
        fooProperty = "bar"
    }
}
```

doWithDynamicMethods

Register dynamic methods with Grails-managed classes, or any other class

```
def doWithDynamicMethods = { ctx ->
   for(domainClass in application.domainClasses{
         domainClass.metaClass.hello = {-> println "hello!" }
   }
   String.metaClass.tokenizeAndReverse = {
      delegate.tokenize().reverse()
   }
}
```

doWithApplicationContext

Interact with the initialized Spring app context

```
def doWithApplicationContext = { ctx ->
    def someInterestingBean = ctx.someInterestingBean
    //do stuff with someInterestingBean
}
```

Participate in auto reload events (1)

Be notified at runtime if resources of interest are modified

```
def watchedResources = "file:./grails-app/services/*Service.groovy "


def onChange = { event ->
    //respond to the modification event by replacing the modified
service's
    //bean in the application context
    //(see Grails docs for details)
}
```

Participate in auto reload events (2)

A plugin can influence another plugin

```
def influences = ['controllers']
```

A plugin can observe another plugin

```
def observe = ['controllers']
```

Same onChange callback as watched resources

```
def onChange = { event -> }
```

Respond to config changes

- No registration required
- Same event as onChange

```
onConfigChange = { event ->
    //respond to a change in configuration
}
```

Load order, environments, scopes

Plugin load order can be managed

A plugin can require other plugins to be loaded prior to loading itself (and not load if dependencies aren’t met)

A plugin can specify the environments and build scopes in which it should load

Add new Grails artefact types

A plugin can add new artefact types

Common example is the Quartz plugin which adds the Job artefact type

To make a place for these artefacts, the Quartz plugin add a grails-app/job directory in the _Install.groovy script

Binary plugins

A plugin can be packaged in binary format

- packaged as a standard jar instead of a zip
- no source included
- to depend on a binary plugin use standard dependency instead of plugin dependency

```
> grails package-plugin --binary
```

Or, in plugin descriptor:

```
def packaging = binary
```

Other resources

“Grails Plugin Best Practices”, Burt Beckwith - Jan, 2014 https://www.youtube.com/watch?v=U2ZFPVSq2jI&list=UU-Vs_q9uWBlSx5NsDkxoGAQ

“Cut Your Grails App to Pieces - Build Feature Plugins”, Goran Ehrsson, June 2014

https://www.youtube.com/watch?v=LZQ-1f9RGqg