Developing Good Plugins

Presented by David Estes (@davydotcom)

About David Estes (@davydotcom)

- Occupation: VP of Engineering at Morpheus Data
- Grails Plugin Developer
- Creator of the Asset-Pipeline Plugin for the JVM
- Karman, Selfie, ForceSSL, SecurityBridge, Spud, RetinaTag, Seed-Me, and more.

MORPHEUS

What are we going to talk about?

- The Joys of plugin development
- Tips
- What makes a plugin great
- How to get started

Why develop Plugins?

- FUN
- INSIGHTFUL
- EDUCATIONAL
- EFFICIENT
- SUPPORTIVE
- COMMUNITY
- SELF-IMPROVEMENT

Getting Started

— Start by contributing to other plugins

— Think of a Need

— Engage the Community (ignore the naysayers)

— `grails create-plugin [name]`

The Key to getting started with plugin development is to minimize the boring stuff so you can have the small wins! The rest will fall in line as you move forward.

Document

- Be sure to document your plugin as you go.
- If you are just experimenting , save it for later.
- JavaDoc / GroovyDoc are a good place to start but go beyond
- AsciiDoc
- README
- LICENSE

Number 1 Rule about Plugin Development

TALK ABOUT PLUGIN DEVELOPMENT!

Number 2 Rule about Plugin Development

ALWAYS MERGE PULL REQUESTS FROM BURT!

Seriously, its usually worth it!

Number 3 Rule about Plugin Development

Be Responsive

Limit Scope

- Minimize transitive dependencies
- KISS! (Keep it Simple)
- Don't be afraid to say no to feature requests!
- Minimize Coupling to Framework Classes (don't forget about your /src folder)
- Not Everything has to be a Bean
- Stay Organized
- Stay Extensible (protected instead of private)

TDD

- Useful for keeping code testable and covered
- Overly tedious for expirementation (NOT A REQUIREMENT TO USE TDD)
- You can totally add tests at the end.
- Asset-Pipeline was Function First not Test First, but was quickly >90% covered.
- Use Unit-Tests if possible instead of Integration Tests

Spaces Vs. Tabs

Let the Holy War Ensue

Try to follow the frameworks preferences for broader community contributions.

TABS!

;-D

Spaces Vs. Tabs

- 4-Space width indentation for Grails
- or just go your own way

Be Mindful of Extensions your Plugin Provides

- Data Layer
- View Layer
- Assets Layer
- Service Layer
- Build Layer (trickier, learn Gradle)

Minimize End-User Requirements

Some plugins require steps to installing, configuring, and using the plugin. Keep these steps as simple as possible to foster quicker adoption.

Be Nice!

- Its Ok to be confident and opinionated
- Be supportive of suggestions and ideas
- If a PR doesnt meet your standards, explain it to the contributor and make suggestions for improvement

Tips for Contributors

- BE NICE!
- Keep Pull Requests small and neat (noone wants a full refactor PR)
- Narrow Scope, send multiple PRs if you must
- Follow plugin owner conventions (its not all about you)
- BE NICE!
- Provide Feedback (Good and Bad)

Grails Plugin Class

Informs Grails about various metadata information your plugin provides, like author information, load order, minimum framework version requirement, exclusions, start/shutdown events, bean definitions, etc.

Grails Plugin Class - `doWithSpring()`

```
Closure doWithSpring() { {->
        // TODO Implement runtime spring config (optional)
        //example:
        rabbitRetryHandler(StatefulRetryOperationsInterceptorFactoryBean) {
            autowire true
        }
    }
}
```

NOTE: Use this similar to an apps `resources.groovy` file for specifying non artefact beans.

Grails Plugin Class - `doWithDynamicMethods()`

Useful for dynamically registering methods using Groovy metaClass. (Try to use @Enhances before going this route.

```
void doWithDynamicMethods() {
    // TODO Implement registering dynamic methods to classes (optional)
    // example
    String.metaClass.humanize = {
        def output = delegate.replaceAll(/[\_\-]+/," ")
    }
}
```

Grails Plugin Class - `doWithApplicationContext()`

Useful for doing any post startup calls where a bean may need additional configuration. Some of this can be handled via Spring Bean Annotations and done in place instead.

```
void doWithApplicationContext() {
    // TODO Implement post initialization spring config (optional)
    // example
    applicationContext.adminApplicationService.initialize()
}
```

Grails Plugin Class - onStartup() and onShutdown()

Useful for any startup operations or shutdown operations. This is not executed until all other plugins are initialized and executes before Bootstrap classes

```
void onStartup(Map<String, Object> event) {
    def autoStart = grailsApplication.config.getProperty('quartz.autoStartup')?.toBoolean()
    if(autoStart) {
        applicationContext.quartzScheduler.start()
        log.info("Quartz Scheduler - Started")
    }
}

void onShutdown(Map<String, Object> event) {
    def waitForJobsToCompleteOnShutdown = grailsApplication.config.getProperty('quartz.waitForJobsToCompleteOnShutdown')?.toBoolean() ?: true
    applicationContext.quartzScheduler.shutdown(waitForJobsToCompleteOnShutdown)
}
```

Grails Plugin Class - onChange()

This method is fired when the file watcher in Grails detects a change to the source file. Some plugins need to do more than just swapping out a class via springloaded in development. Example: `Quartz`

```
void onChange(Map<String, Object> event) {
    // TODO Implement code that is executed when any artefact that this plugin is
    // watching is modified and reloaded. The event contains: event.source,
    // event.application, event.manager, event.ctx, and event.plugin.
}

void onConfigChange(Map<String, Object> event) {
    // TODO Implement code that is executed when the project configuration changes.
    // The event is the same as for 'onChange'.
}
```

Grails Plugin Config

Can use `application.yml` for test runs within plugin but these properties will not be propagated to the app using the plugin. Use `plugin.yml` if default properties are desired to be passed along to an application

Grails Plugin Tricks - Injecting Artefact Methods

The @Enhances annotation can be used to add methods onto existing artefact types at compile time:

Define a Trait in `src/main/groovy`:

```
import grails.artefact.Enhances
import groovy.transform.CompileStatic
import org.grails.core.DefaultGrailsControllerClass
import org.grails.core.DefaultGrailsServiceClass

@Enhances([DefaultGrailsServiceClass.SERVICE, DefaultGrailsControllerClass.CONTROLLER])
@CompileStatic
trait CoolMethod {
    public coolMethod() {
        println "Do Something Cool!"
    }
}
```

Grails Plugin Tricks - Artefact API

```
class MyGrailsPlugin {
    def artefacts = [ org.somewhere.MyArtefactHandler ]
    ...
}
```

The Quartz plugin is a good source of reference as it creates the `Job` artefact using 3 classes: `GrailsJobClass`, `DefaultGrailsJobClass`, and `JobArtefactHandler`

Gradle Plugins!

— Gradle extensions can be added into grails plugins for optional build time operations or as seperate plugins

— The consumer of the plugin has to be sure to add the same plugin to the build classpath and apply the plugin

— compileOnly is useful here!

```
dependencies {
    compileOnly gradleApi()
}
```

Gradle Plugins!

- Different DSL https://docs.gradle.org/current/userguide/custom_plugins.html
- Similar Concept as Grails Plugin Class: Implement `org.gradle.api.Plugin`

Gradle Plugins!

Example:

```
class SeedMePlugin implements Plugin<Project> {

    void apply(Project project) {
        def defaultConfiguration = project.extensions.create('seedme', SeedMeExtension)

        defaultConfiguration.seedPath = "${project.projectDir}/src/seed"
        project.tasks.create('seedPackage', SeedMePackage)
        def seedMePackageTask = project.tasks.getByName('seedPackage')

        project.afterEvaluate { }
}
```

Gradle Plugins! - Tasks

- Tasks can define inputs and output files
- Files and inputs can be checked for differences to determine if the task needs rerun or not.

Example use can be found in the grails seed-me plugin:

github.com/bertramdev/seed-me/tree/master/src/main/groovy/seedme/gradle

Lets Build a Plugin!

- Grails Plugin Example!
- Lets build a Profiler Plugin!
- https://github.com/davydotcom/tuneup

What should our Plugin Do?

- Start Simple
- Track web request performance
- Present a simple web view for tracking performance
- Add more granular metrics

Creating a Plugin

First we need to create the plugin skeleton.

```
grails create-plugin com.bertramlabs.plugins.tuneup.tuneup
```

TIP: Including the package name in the name automatically sets the projects default package.

Setup Version Control

Setting up a repository is fairly simple. Simply go to github.com (yes you can use other options) and create a new repository. Follow the instructions provided and push your first commit

```
git init
git add .
git commit -am "first commit"
git remote add git@blahblahblah
git push -u origin master
```

Add a README

Creating a `README.md` file at the base of your project will automatically display a nicely formatted document at the top of your github repository. Markdown is a very simple plain text formatting engine.

— Give a brief summary of your plugin

— Provide basic Install Instructions

— Add a Usage Section

— Discuss work to be done or give gratitude to contributors

Setup an Inline Plugin for Quick Testing

```
grails create-app tuneuptestapp
echo "include 'tuneup', 'tuneuptestapp'" > settings.gradle
```

Edit `build.gradle` in tuneupttestapp directory:

```
grails {
    plugins {
        compile project(':tuneup')
    }
}
```

NOTE: While this can go in the `dependencies {}` block, you would lose live reloading while the app is running

Create Some Artefacts

The grails command provides all of the same generators you get with an app so development feels fluid.

```
grails create-interceptor TuneUpMetric
grails create-controller SlowTransactions
```

Namespaces are your friend

When it comes to grails controllers, take advantage of the namespace concept for better plugin encapsulation and extensibility. Using these instead of using plugin scope on mappings allow for third party overrides or extensibility

```
class SlowTransactionsController {
    static namespace = 'tuneup'
    def index() { }
}
```

NOTE: When you do this, views need to be moved to a subfolder matching the name of the namespace.

Be Explicit about Url Mappings

The default `:controller/:action` url mappings should be removed as to minimize clashing with the plugin consumers routing tables.

```
class UrlMappings {
    static mappings = {
        "/tuneup"(resources: 'slowTransactions', namespace: 'tuneup')
    }
}
```

URL Mappings in Plugins

- Mappings can be scoped via the `plugin:` property or better yet via the `namespace:` property
- URL Mappings can be changed by configuration (the URL Mapping Factory implements `GrailsApplicationAware`

URL Mappings in Plugins

```
import grails.util.Environment

class UrlMappings {
    static mappings = { it ->
        String baseUrl = getGrailsApplication().config.getProperty('grails.plugins.tuneup.baseUrl',String,'tuneup')
        Boolean enabled = getGrailsApplication().config.getProperty(
            'grails.plugins.tuneup.enabled',
            Boolean,
            Environment.currentEnvironment == Environment.DEVELOPMENT
        )
        if(enabled) {
            group("/${baseUrl}") {
                "/"(resources: 'slowTransactions', namespace: 'tuneup')
                "/clear"(controller: 'slowTransactions', namespace: 'tuneup', action: 'clear', method: 'POST')
            }
        }
    }
}
```

Clean Up Plugin Views

Create tuneup specific layout and structure like so:

```
views/
  layouts/
    tuneup.gsp
  tuneup/
    slowTransactions/
      index.gsp
```

NOTE: Some of the auto generated views are not needed and should be removed

Static Assets in a Plugin

By default assets in a plugin are unprocessed and not processed until the plugin is used in an application.

This Allows for consumer extensibility and overrides.

This is done in the plugins `build.gradle`:

```
assets {
    packagePlugin = true
}
```

Static Assets in a Plugin

- Use a plugin named isolated path for scope isolation
- `//=require_full_tree .` is fantastic for allowing extensions

GrailsCompileStatic is your friend

Use Static Compilation where it makes sense to reduce overhead of your plugin on the consumer. Performance is important!

Testing

Grails provides all the same testing frameworks you get in an application for plugins!

```
grails test-app
```

Add Travis-CI for build automation!

Continuous Build - Travis

- Go to travis-ci.org and sign-in with github.
- Click on name on upper right and find your repository and activate it.
- setup gradle wrapper: `gradle wrapper`
- Add a `.travis.yml`:

```
language: groovy
sudo: false
jdk:
  - oraclejdk8
script: ./gradlew check
```

Continuous Build - Travis

- Add a badge to your `README.md` to show the current build status in your project
- Pull Requests are automatically tested before merge
- Receive notifications of build failures
- Test against multiple environments

Documentation

There are several options to choose from when it comes to providing documentation for your plugin:

- Javadoc/Groovydoc are great but usage guides are important too
- AsciiDoc via Gradle is very popular amongst the community
- Github Wiki is also a viable location but harder to version.

Documentation - AsciiDoctor

Setup a new subproject in your repository called `tuneup-docs`. (arent gradle multiproject builds great!)

```
mkdir tuneup-docs
echo "include 'tuneup', 'tuneuptestapp', 'tuneup-docs'" > settings.gradle
```

Documentation - AsciiDoctor

Create a `tuneup-docs/build.gradle`:

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'org.asciidoctor:asciidoctor-gradle-plugin:1.5.0'
    }
}
apply plugin: 'org.asciidoctor.gradle.asciidoctor'
asciidoctor {
    outputDir = new File("$buildDir/docs")
    options = [
        doctype: 'book',
        attributes: [
            'source-highlighter': 'coderay',
            toc                 : 'left',
            'toc-title': 'Table of Contents',
            idprefix            : '',
            idseparator         : '-'
        ]
    ]
}
```

Documentation - AsciiDoctor

Documentation goes in `src/asciidoc` and always starts with `index.adoc`

Example `index.adoc`:

```
= Asset Pipeline Users Guide
David Estes <davydotcom@gmail.com>
{project-version}

:homepage: http://asset-pipeline.com
:apidocs: http://asset-pipeline.com/apidoc/index.html


include::introduction.adoc[]
```

Documentation - AsciiDoctor

Learn More about AsciiDoctor gradle:
http://asciidoctor.org/docs/asciidoctor-gradle-plugin/

Publishing

- Create a Bintray Account
- Add a Maven Repository to Bintray
- Configure `build.gradle`

Publishing

Update the `build.gradle`:

```
grailsPublish {
    user = 'user'
    key = 'key'
    githubSlug = 'davydotcom/tuneup'
    license {
        name = 'Apache-2.0'
    }
    title = "TuneUp"
    desc = "Basic TuneUp Plugin providing web transaction metrics"
    developers = [davydotcom:"David Estes"]
}
```

NOTE: Do NOT Embed your bintray credentials in `build.gradle`: Better to store in environment and reference

Publishing - Configuration

```
grailsPublish {
    if(project.hasProperty('bintrayUser')) {
        user = bintrayUser
        key = bintrayKey
    }

    githubSlug = 'davydotcom/tuneup'
    userOrg = 'bertramlabs' //if using an organization on bintray
    repo = 'grails3-plugins'
    license {
        name = 'Apache-2.0'
    }
    title = "TuneUp"
    desc = "Basic TuneUp Plugin providing web transaction metrics"
    developers = [davydotcom:"David Estes"]
}
```

Advanced Publishing

- Adding to jcenter()
- Adding to Maven Central
- Adding to Travis-CI: travis-build.sh example

Adding to Grails Plugins Page

Adding your plugin to the grails plugin catalog can be done directly from bintray.

- Go to the `grails/plugins` repository https://bintray.com/grails/plugins
- Click "Include My Package" and await approval

http://plugins.grails.org

Getting the Word Out!

- Use Social Media (Twitter)
- Blog about Grails and use cases
- Present at a Groovy Users Group
- Slack Community
- Screencasts

Evolving the Plugin

- Think of optimizations that can be done later
- Features that could be added
- Add Test Coverage and Documentation
- Reduce Dependencies

Questions?