Grails Goodness

Experience Grails through code snippets

NOTE
BOOK

Hubert A. Klein Ikkink

Grails Goodness Notebook

Experience the Grails platform through code snippets

Hubert A. Klein Ikkink (mrhaki)

This book is for sale at http://leanpub.com/grails-goodness-notebook

This version was published on 2023-04-19

Tweet This Book!

Please help Hubert A. Klein Ikkink (mrhaki) by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Grails Goodness Notebook with Grails Goodness blog posts bundled into one book. #grails #grailsfw @mrhaki

The suggested hashtag for this book is #grailsnotebook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#grailsnotebook

This book is dedicated to my lovely family. I love you.

Contents

Configuration

Using Bintray JCenter as Repository

Bintray JCenter is the next generation (Maven) repository. The repository is already a superset of Maven Central, so it can be used as a drop-in replacement for Maven Central. To use Bintray JCenter we only use `jcenter()` in our `repositories` configuration block in `BuildConfig.groovy`. This is the same as we would use in a Gradle build.

```
// File: grails-app/conf/BuildConfig.groovy
...
repositories {
    ...
    // Configure Bintray JCenter as repo.
    jcenter()
    ...
}
...
```

We can also set the update and checksum policies by applying a configuration closure to the `jcenter` method:

```
// File: grails-app/conf/BuildConfig.groovy
...
repositories {
    ...
    // Configure Bintray JCenter as repo.
    jcenter {
        // Allowed values: never, always, daily (default), interval:seconds.
        updatePolicy 'always'
        // Allowed values: fail, warn, ignore
        checksumPolicy 'fail'
    }
    ...
}
...
```

Code written with Grails 2.4.2.

Original blog post written on August 05, 2014.

Add Banner to Grails Application

Grails 3 is based on Spring Boot. This means we get a lot of the functionality of Spring Boot into our Grails applications. A Spring Boot application has by default a banner that is shown when the application starts. The default Grails application overrides Spring Boot's behavior and disables the display of a banner. To add a banner again to our Grails application we have different options.

First we can add a file `banner.txt` to our classpath. If Grails finds the file it will display the contents when we start the application. Let's add a simple banner with Grails3 in Ascii art in the file `src/main/resources/banner.txt`. By placing the file in `src/main/resources` we can assure it is in the classpath as `classpath:/banner.txt`:

```
  ________             .__.__         ________
 /  _____/___________  |__|  |   _____\_____  \
/   \  __\_  __ \__  \ |  |  |  /  ___/ _(__  <
\    \_\  \  | \// __ \|  |  |__\___ \ /       \
 \______  /__|  (____  /__|____/____  >______  /
        \/           \/             \/       \/
```

Let's run our application with the `bootRun` task:

```
$ gradle bootRun
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources
:classes
:findMainClass
:bootRun

  ________             .__.__         ________
 /  _____/___________  |__|  |   _____\_____  \
/   \  __\_  __ \__  \ |  |  |  /  ___/ _(__  <
\    \_\  \  | \// __ \|  |  |__\___ \ /       \
 \______  /__|  (____  /__|____/____  >______  /
        \/           \/             \/       \/

Grails application running at http://localhost:8080
...
```

To have more information in the banner we can implement the `org.springframework.boot.Banner` interface. This interface has a `printBanner` method in which we can write the implementation for the banner. To use it we must create an instance of the `GrailsApp` class and set the `banner` property:

```
// File: grails-app/init/banner/Application.groovy
package banner

import grails.boot.GrailsApp
import grails.boot.config.GrailsAutoConfiguration
import grails.util.Environment
import org.springframework.boot.Banner

import static grails.util.Metadata.current as metaInfo

class Application extends GrailsAutoConfiguration {
    static void main(String[] args) {
        final GrailsApp app = new GrailsApp(Application)
        app.banner = new GrailsBanner()
        app.run(args)
    }
}

/**
 * Class that implements Spring Boot Banner
 * interface to show information on application startup.
 */
class GrailsBanner implements Banner {

    private static final String BANNER = '''
  ________             .__.__         ________
 /  _____/___________  |__|  |   _____\\_____  \\
/   \\  __\\_  __ \\__  \\ |  |  |  /  ___/ _(__  <
\\    \\_\\  \\  | \\// __ \\|  |  |__\\___ \\ /       \\
 \\______  /__|  (____  /__|____/____  >______  /
        \\/           \\/             \\/       \\/'''

    @Override
    void printBanner(
            org.springframework.core.env.Environment environment,
            Class sourceClass,
            PrintStream out) {

        out.println BANNER

        row 'App version', metaInfo.getApplicationVersion(), out
        row 'App name', metaInfo.getApplicationName(), out
        row 'Grails version', metaInfo.getGrailsVersion(), out
        row 'Groovy version', GroovySystem.version, out
        row 'JVM version', System.getProperty('java.version'), out
        row 'Reloading active', Environment.reloadingAgentEnabled, out
        row 'Environment', Environment.current.name, out

        out.println()
    }

    private void row(final String description, final value, final PrintStream out) {
        out.print ':: '
        out.print description.padRight(16)
        out.print ' :: '
        out.println value
    }

}
```

Now we run the `bootRun` task again:

```
$ gradle bootRun
:compileJava UP-TO-DATE
:compileGroovy
:processResources
:classes
:findMainClass
:bootRun

  ________             .__.__         ________
 /  _____/___________  |__| |   _____\_____  \
/   \  __\_  __ \__  \ |  |  |  /  ___/ _(__  <
\    \_\  \  | \// __ \|  |  |__\___ \ /       \
 \______  /__|  (____  /__|____/____  >______  /
        \/           \/             \/       \/
:: App version      :: 0.1
:: App name         :: grails-banner-sample
:: Grails version   :: 3.0.1
:: Groovy version   :: 2.4.3
:: JVM version      :: 1.8.0_45
:: Reloading active :: true
:: Environment      :: development

Grails application running at http://localhost:8080
...
```

Written with Grails 3.0.1.

Ascii art is generated with this website.

Original blog post written on April 15, 2015.

Add Banner To Grails 3.1 Application

In a previous post we learned how to add a banner to a Grails 3.0 application. We used the Spring Boot support in Grails to show a banner on startup. The solution we used doesn't work for a Grails 3.1 application. We need to implement a different solution to show a banner on startup.

First of all we create a new class that implements the `org.springframework.boot.Banner` interface. We implement the single method `printBanner` and logic to display a banner, including colors:

```
// File: src/main/groovy/mrhaki/grails/GrailsBanner.groovy
package mrhaki.grails

import org.springframework.boot.Banner
import grails.util.Environment
import org.springframework.boot.ansi.AnsiColor
import org.springframework.boot.ansi.AnsiOutput
import org.springframework.boot.ansi.AnsiStyle

import static grails.util.Metadata.current as metaInfo
```

```
/**
 * Class that implements Spring Boot Banner
 * interface to show information on application startup.
 */
class GrailsBanner implements Banner {

    /**
     * ASCCI art Grails 3.1 logo built on
     * http://patorjk.com/software/taag/#p=display&f=Graffiti&t=Type%20Something%20
     */
    private static final String BANNER = $/
  _________________    _____  .___.____       _________ ________       ____
 /  _____|______   \  /  _  \ |   |    |     /   _____/ \_____  \    /_   |
/   \  ___|       _/ /  /_\  \|   |    |     \_____  \    _(__  <     |   |
\    \_\  \    |   \/    |    \   |    |___  /        \  /       \    |   |
 \______  /____|_  /\____|__  /___|_______ \/_______  / /______  / /\ |___|
        \/       \/          \/            \/        \/          \/  \/
    /$

    @Override
    void printBanner(
            org.springframework.core.env.Environment environment,
            Class<?> sourceClass,
            PrintStream out) {

        // Print ASCII art banner with color yellow.
        out.println AnsiOutput.toString(AnsiColor.BRIGHT_YELLOW, BANNER)

        // Display extran infomratio about the application.
        row 'App version', metaInfo.getApplicationVersion(), out
        row 'App name', metaInfo.getApplicationName(), out
        row 'Grails version', metaInfo.getGrailsVersion(), out
        row 'Groovy version', GroovySystem.version, out
        row 'JVM version', System.getProperty('java.version'), out
        row 'Reloading active', Environment.reloadingAgentEnabled, out
        row 'Environment', Environment.current.name, out

        out.println()
    }

    private void row(final String description, final value, final PrintStream out) {
        out.print AnsiOutput.toString(AnsiColor.DEFAULT, ':: ')
        out.print AnsiOutput.toString(AnsiColor.GREEN, description.padRight(16))
        out.print AnsiOutput.toString(AnsiColor.DEFAULT, ' :: ')
        out.println AnsiOutput.toString(AnsiColor.BRIGHT_CYAN, AnsiStyle.FAINT, value)
    }

}
```

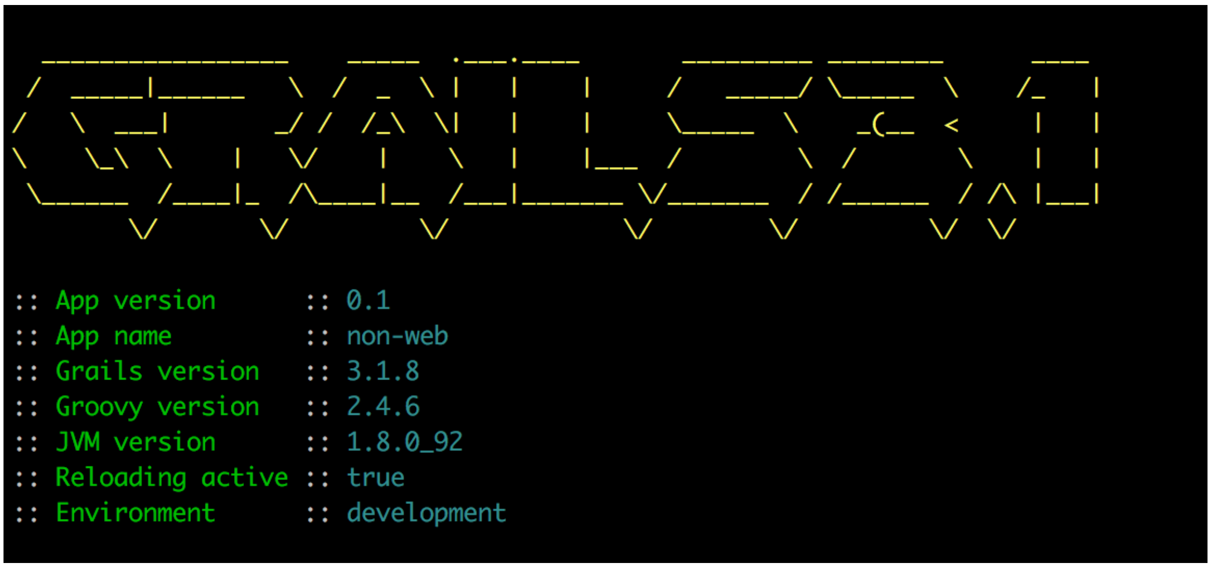
Next we must override the `GrailsApp` class. We override the `printBanner` method, which has no implementation in the `GrailsApp` class. In our `printBanner` method we use `GrailsBanner`:

```
// File: src/main/groovy/mrhaki/grails/BannerGrailsApp.groovy
package mrhaki.grails

import grails.boot.GrailsApp
import groovy.transform.InheritConstructors
import org.springframework.core.env.Environment

@InheritConstructors
class BannerGrailsApp extends GrailsApp {

    @Override
    protected void printBanner(final Environment environment) {
        // Create GrailsBanner instance.
        final GrailsBanner banner = new GrailsBanner()

        banner.printBanner(environment, Application, System.out)
    }

}
```

Finally in the `Application` class we use `BannerGrailsApp` instead of the default `GrailsApp` object:

```
// File: grails-app/init/mrhaki/grails/Application.groovy
package mrhaki.grails

import grails.boot.config.GrailsAutoConfiguration

class Application extends GrailsAutoConfiguration {
    static void main(String[] args) {
        final BannerGrailsApp app = new BannerGrailsApp(Application)
        app.run(args)
    }
}
```

When we start our Grails application on a console with color support we see the following banner:



```
  ________________    _____   .___.____      _________ ________     ____
 /  _____|______   \  /  _  \ |   |    |    /   _____/ \_____  \   /_   |
/   \  ___|       _/ /  /_\  \|   |    |    \_____  \    _(__  <    |   |
\    \_\  \    |   \/    |    \   |    |___ /        \  /       \   |   |
 \______  /____|_  /\____|__  /___|_______ \/_______  / /______  / /\ |___|
        \/        \/         \/            \/        \/         \/  \/

:: App version       :: 0.1
:: App name          :: non-web
:: Grails version    :: 3.1.8
:: Groovy version    :: 2.4.6
:: JVM version       :: 1.8.0_92
:: Reloading active  :: true
:: Environment       :: development
```

Written with Grails 3.1.8.

Original blog post written on June 20, 2016.

Set Log Level for Grails Artifacts

A good thing in Grails is that in Grails artifacts like controllers and services we have a `log` property to add log statements in our code. If we want to have the output of these log statements we must use a special naming convention for the log names. Each logger is prefixed with `grails.app` followed by the Grails artifact. Valid artifact values are `controllers`, `services`, `domain`, `filters`, `conf` and `taglib`. This is followed by the actual class name. So for example we have a controller `SampleController` in the package `mrhaki.grails` then the complete logger name is `grails.app.controllers.mrhaki.grails.SampleContoller`.

The following sample configuration is for pre-Grails 3:

```
// File: grails-app/conf/Config.groovy
...
log4j = {
    ...
    info 'grails.app.controllers'
    debug 'grails.app.controllers.mrhaki.grails.SampleController'
    info 'grails.app.services'
    ...
}
...
```

In Grails 3 we can use a common Logback configuration file. In the following part of the configuration we set the log levels:

```
// File: grails-app/conf/logback.groovy
...
logger 'grails.app.controllers', INFO, ['STDOUT']
logger 'grails.app.controllers.mrhaki.grails.SampleController', DEBUG, ['STDOUT']
logger 'grails.app.services', INFO, ['STDOUT']
...
```

Written with Grails 2.5.0 and 3.0.1.

Original blog post written on April 15, 2015.

Add Some Color to Our Logging

Grails 3 is based on Spring Boot. This means we can use a lot of the stuff that is available in Spring Boot now in our Grails application. If we look at the logging of a plain Spring Boot application we notice the logging has colors by default if our console supports ANSI. We can also configure our Grails logging so that we get colors.

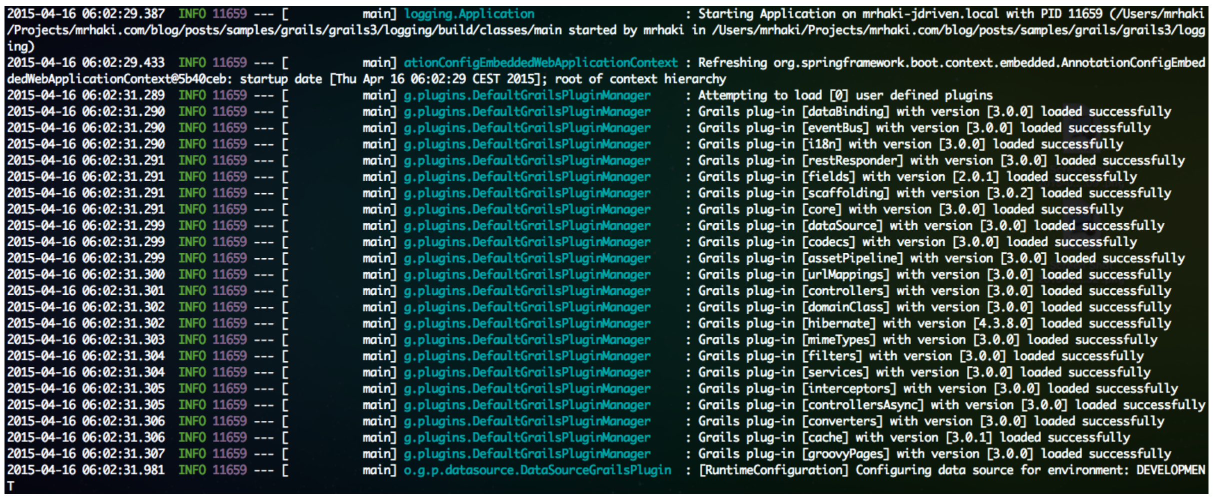
First we need to change our logging configuration in the file `grails-app/conf/logback.groovy`:

```
// File: grails-app/conf/logback.groovy
import grails.util.BuildSettings
import grails.util.Environment
import org.springframework.boot.ApplicationPid

import java.nio.charset.Charset

// Get PID for Grails application.
// We use it in the logging output.
if (!System.getProperty("PID")) {
    System.setProperty("PID", (new ApplicationPid()).toString())
}

// Mimic Spring Boot logging configuration.
conversionRule 'clr', org.springframework.boot.logging.logback.ColorConverter
conversionRule 'wex', org.springframework.boot.logging.logback.WhitespaceThrowableProxyConverter

appender('STDOUT', ConsoleAppender) {
    encoder(PatternLayoutEncoder) {
        charset = Charset.forName('UTF-8')

        // Define pattern with clr converter to get colors.
        pattern =
                '%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} ' + // Date
                '%clr(%5p) ' + // Log level
                '%clr(%property{PID}){magenta} ' + // PID
                '%clr(---){faint} %clr([%15.15t]){faint} ' + // Thread
                '%clr(%-40.40logger{39}){cyan} %clr(:){faint} ' + // Logger
                '%m%n%wex' // Message
    }
}

// Change root log level to INFO,
// so we get some more logging.
root(INFO, ['STDOUT'])
...
```

Normally when we would run our application Grails should check if the console support ANSI colors. If the console supports it the color logging is enabled, otherwise we still get non-colored logging. On my Mac OSX the check doesn't work correctly, but we can set an environment property `spring.output.ansi.enabled` to the value `always` to force colors in our logging output. The default value is `detect` to auto detect the support for colors. We can set this property in different ways. For example we could add it to our application configuration or we could add it as a Java system property to the JVM arguments of the `bootRun` task. In the following build file we use the JVM arguments for the `bootRun` task:

```
// File: build.gradle
...
bootRun {
    // If System.console() return non null instance,
    // we force ANSI color support with 'always',
    // otherwise use default 'detect'.
    jvmArgs = ['-Dspring.output.ansi.enabled=' + (System.console() ? 'always' : 'detect')]
}
...
```

When we run the Grails application using `bootRun` we get for example the following output:



```
2015-04-16 06:02:29.387  INFO 11659 --- [           main] logging.Application                      : Starting Application on mrhaki-jdriven.local with PID 11659 (/Users/mrhaki
/Projects/mrhaki.com/blog/posts/samples/grails/grails3/logging/build/classes/main started by mrhaki in /Users/mrhaki/Projects/mrhaki.com/blog/posts/samples/grails/grails3/logg
ing)
2015-04-16 06:02:29.433  INFO 11659 --- [           main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbed
dedWebApplicationContext@5b40ceb: startup date [Thu Apr 16 06:02:29 CEST 2015]; root of context hierarchy
2015-04-16 06:02:31.289  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Attempting to load [0] user defined plugins
2015-04-16 06:02:31.290  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [dataBinding] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.290  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [eventBus] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.290  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [i18n] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.291  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [restResponder] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.291  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [fields] with version [2.0.1] loaded successfully
2015-04-16 06:02:31.291  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [scaffolding] with version [3.0.2] loaded successfully
2015-04-16 06:02:31.291  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [core] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.299  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [dataSource] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.299  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [codecs] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.299  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [assetPipeline] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.300  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [urlMappings] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.301  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [controllers] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.302  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [domainClass] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.302  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [hibernate] with version [4.3.8.0] loaded successfully
2015-04-16 06:02:31.303  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [mimeTypes] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.304  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [filters] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.304  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [services] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.305  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [interceptors] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.305  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [controllersAsync] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.306  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [converters] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.306  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [cache] with version [3.0.1] loaded successfully
2015-04-16 06:02:31.307  INFO 11659 --- [           main] g.plugins.DefaultGrailsPluginManager     : Grails plug-in [groovyPages] with version [3.0.0] loaded successfully
2015-04-16 06:02:31.981  INFO 11659 --- [           main] o.g.p.datasource.DataSourceGrailsPlugin  : [RuntimeConfiguration] Configuring data source for environment: DEVELOPMEN
T
```

Written with Grails 3.0.1.

Original blog post written on April 16, 2015.

Save Application PID in File

Since Grails 3 we can borrow a lot of the Spring Boot features in our applications. If we look in our `Application.groovy` file that is created when we create a new Grails application we see the class `GrailsApp`. This class extends `SpringApplication` so we can use all the methods and properties of `SpringApplication` in our Grails application. Spring Boot and Grails

comes with the class `ApplicationPidFileWriter` in the package `org.springframework.boot.actuate.system.`
This class saves the application PID (Process ID) in a file `application.pid` when the application starts.

In the following example `Application.groovy` we create an instance of `ApplicationPid-FileWriter` and register it with the `GrailsApp`:

```
package mrhaki.grails.sample

import grails.boot.GrailsApp
import grails.boot.config.GrailsAutoConfiguration
import org.springframework.boot.actuate.system.ApplicationPidFileWriter

class Application extends GrailsAutoConfiguration {

    static void main(String[] args) {
        final GrailsApp app = new GrailsApp(Application)

        // Register PID file writer.
        app.addListeners(new ApplicationPidFileWriter())

        app.run(args)
    }

}
```

So when we run our application a new file `application.pid` is created in the current directory and contains the PID:

```
$ grails run-app
```

From another console we read the contents of the file with the PID:

```
$ cat application.pid
20634
$
```

The default file name is `application.pid`, but we can use another name if we want to. We can use another constructor for the `ApplicationPidFileWriter` where we specify the file name. Or we can use a system property or environment variable with the name `PIDFILE`. But we can also set it with the configuration property `spring.pidfile`. We use the latest option in our Grails application. In the next example `application.yml` we set this property:

```
...
spring:
    pidfile: sample-app.pid
...
```

When we start our Grails application we get the file `sample-app.pid` with the application PID as contents.

Written with Grails 3.0.1.

Original blog post written on April 22, 2015.

Saving Server Port In A File

In a previous post we learned how to save the application PID in a file when we start our Grails application. We can also save the port number the application uses in a file when we run our Grails application. We must use the class `EmbeddedServerPortFileWriter` and add it as a listener to the `GrailsApp` instance. By default the server port is saved in a file `application.port`. But we can pass a custom file name or `File` object to the constructor of the `EmbeddedServerPortFileWriter` class.

In the following example we use the file name `application.server.port` to store the port number:

```
// File: grails-app/init/sample/Application.groovy
package sample

import grails.boot.GrailsApp
import grails.boot.config.GrailsAutoConfiguration
import groovy.transform.CompileStatic
import org.springframework.boot.actuate.system.EmbeddedServerPortFileWriter

@CompileStatic
class Application extends GrailsAutoConfiguration {

    static void main(String[] args) {
        final GrailsApp app = new GrailsApp(Application)

        // Save port number in file name application.server.port
        final EmbeddedServerPortFileWriter serverPortFileWriter =
                new EmbeddedServerPortFileWriter('application.server.port')

        // Add serverPortFileWriter as application listener
        // so the port number is saved when we start
        // our Grails application.
        app.addListeners(serverPortFileWriter)

        app.run(args)
    }

}
```

When the application is started we can find the file `application.server.port` in the directory from where the application is started. When we open it we see the port number:

```
$ cat application.server.port
8080
$
```

Written with Grails 3.1.

Original blog post written on February 05, 2016.

Log Startup Info

We can let Grails log some extra information when the application starts. Like the process ID (PID) of the application and on which machine the application starts. And the time needed to start the application. The `GrailsApp` class has a property `logStartupInfo` which is `true` by default. If the property is true than some extra lines are logged at INFO and DEBUG level of the logger of our `Application` class.

So in order to see this information we must configure our logging in the `logback.groovy` file. Suppose our `Application` class is in the package `mrhaki.grails.sample.Application` then we add the following line to see the output of the startup logging on the console:

```
...
logger 'mrhaki.grails.sample.Application', DEBUG, ['STDOUT'], false
...
```

When we run our Grails application we see the following in our console:

```
...
INFO mrhaki.grails.sample.Application - Starting Application on mrhaki-jdriven.local with PID 20948 (/\
Users/mrhaki/Projects/blog/posts/sample/build/classes/main started by mrhaki in /Users/mrhaki/Projects\
/mrhaki.com/blog/posts/sample/)
DEBUG mrhaki.grails.sample.Application - Running with Spring Boot v1.2.3.RELEASE, Spring v4.1.6.RELEASE
INFO mrhaki.grails.sample.Application - Started Application in 8.29 seconds (JVM running for 9.906)
Grails application running at http://localhost:8080
...
```

If we want to add some extra logging we can override the `logStartupInfo` method:

```
package mrhaki.grails.sample

import grails.boot.GrailsApp
import grails.boot.config.GrailsAutoConfiguration
import grails.util.*
import groovy.transform.InheritConstructors

class Application extends GrailsAutoConfiguration {

    static void main(String[] args) {
        // Use extended GrailsApp to run.
        new StartupGrailsApp(Application).run(args)
    }

}

@InheritConstructors
class StartupGrailsApp extends GrailsApp {
    @Override
    protected void logStartupInfo(boolean isRoot) {
        // Show default info.
        super.logStartupInfo(isRoot)
        // And add some extra logging information.
        // We use the same logger if we get the
        // applicationLog property.
        if (applicationLog.debugEnabled) {
            final metaInfo = Metadata.getCurrent()
            final String grailsVersion = GrailsUtil.grailsVersion
            applicationLog.debug "Running with Grails v${grailsVersion}"

            final sysprops = System.properties
            applicationLog.debug "Running on ${sysprops.'os.name'} v${sysprops.'os.version'}"
        }
    }
}
```

If we run the application we see in the console:

```
...
DEBUG mrhaki.grails.sample.Application - Running with Spring Boot v1.2.3.RELEASE, Spring v4.1.6.RELEASE
DEBUG mrhaki.grails.sample.Application - Running with Grails v3.0.0
DEBUG mrhaki.grails.sample.Application - Running on Mac OS X v10.10.3
...
```

Written with Grails 3.0.1.

Original blog post written on April 23, 2015.

Changing Gradle Version

Since Grails 3 Gradle is used as the build tool. The Grails shell and commands use Gradle to execute tasks. When we create a new Grails 3 application a Gradle wrapper is added to our project. The Gradle wrapper is used to download and use a specific Gradle version for a project. This version is also used by the Grails shell and commands. The default version (for Grails 3.0.12) is Gradle 2.3, which is also part of the Grails distribution. At the time of writing this blog post the latest Gradle version is 2.10. Sometimes we use Gradle plugins in our project that need a higher Gradle version, or we just want to use the latest version because of improvements in Gradle. We can change the Gradle version that needs to be used by Grails in different ways.

Grails will first look for an environment variable `GRAILS_GRADLE_HOME`. It must be set to the location of a Gradle installation. If it is present is used as the Gradle version by Grails. In the following example we use this environment variable to force Grails to use Gradle 2.10:

```
$ GRAILS_GRADLE_HOME=~/.sdkman/gradle/2.10 grails

BUILD SUCCESSFUL

| Enter a command name to run. Use TAB for completion:
grails> gradle help
:help

Welcome to Gradle 2.10.

To run a build, run gradle <task> ...

To see a list of available tasks, run gradle tasks

To see a list of command-line options, run gradle --help

To see more detail about a task, run gradle help --task <task>

BUILD SUCCESSFUL

Total time: 0.861 secs
grails>
```

Another way to set the Gradle version is by change the Gradle wrapper version. In our `build.gradle` file there is a task `wrapper`. This creates a Gradle wrapper for our project with the version that is specified in the file `gradle.properties` with the property `gradleWrapperVersion`. Let's change the value of `gradleWrapperVersion` to `2.10` and execute the `wrapper` task. We can change the value in the `grade.properties` file, the `build.gradle` file or pass it via the command line:

```
$ ./gradlew wrapper -PgradleWrapperVersion=2.10
:wrapper

BUILD SUCCESSFUL

Total time: 2.67 secs

$ grails

BUILD SUCCESSFUL

| Enter a command name to run. Use TAB for completion:
grails> gradle help
:help

Welcome to Gradle 2.10.

To run a build, run gradle <task> ...

To see a list of available tasks, run gradle tasks

To see a list of command-line options, run gradle --help

To see more detail about a task, run gradle help --task <task>
```

```
BUILD SUCCESSFUL

Total time: 0.861 secs
grails>
```

It could be that we get an `org/gradle/mvn3/org/apache/maven/model/building/ModelBuildingException` exception after upgrading to a newer version. This is because the `io.spring.dependency-management` plugin is set to a version not supported by the newer Gradle version. If we change the version of the plugin to the latest version (`0.5.4.RELEASE` at the time of writing this blog post) the error is solved.

It also important to notice that Grails will look for Gradle wrapper defined for the base project if we use our Grails project in a multi-module project. So the directory that contains the `settings.gradle` file is then used to look for a Gradle wrapper. If it is not found the default Gradle version that is defined by the Grails distribution is used.

Written with Grails 3.0.12.

Original blog post written on January 27, 2016.

Adding Health Check Indicators

With Grails 3 we also get Spring Boot Actuator. We can use Spring Boot Actuator to add some production-ready features for monitoring and managing our Grails application. One of the features is the addition of some endpoints with information about our application. By default we already have a `/health` endpoint when we start a Grails (3+) application. It gives back a JSON response with status UP. Let's expand this endpoint and add a disk space, database and url health check indicator.

We can set the application property `endpoints.health.sensitive` to `false` (securing these endpoints will be another blog post) and we automatically get a disk space health indicator. The default threshold is set to 10MB, so when the disk space is lower than 10MB the status is set to DOWN. The following snippet shows the change in the `grails-app/conf/application.yml` to set the property:

```
...
---
endpoints:
    health:
        sensitive: false
...
```

If we invoke the `/health` endpoint we get the following output:

```
{
    "status": "UP",
    "diskSpace": {
        "status": "UP",
        "free": 97169154048,
        "threshold": 10485760
    }
}
```

If we want to change the `threshold` we can create a Spring bean of type `DiskSpaceHealthIndicatorProperties` and name `diskSpaceHealthIndicatorProperties` to override the default bean. Since Grails 3 we can override `doWithSpring` method in the `Application` class to define Spring beans:

```
package healthcheck

import grails.boot.GrailsApp
import grails.boot.config.GrailsAutoConfiguration
import org.springframework.boot.actuate.health.DiskSpaceHealthIndicatorProperties

class Application extends GrailsAutoConfiguration {

    static void main(String[] args) {
        GrailsApp.run(Application)
    }

    @Override
    Closure doWithSpring() {
        { ->
            diskSpaceHealthIndicatorProperties(DiskSpaceHealthIndicatorProperties) {
                // Set threshold to 250MB.
                threshold = 250 * 1024 * 1024
            }
        }
    }
}
```

Spring Boot Actuator already contains implementations for checking SQL databases, Mongo, Redis, Solr and RabbitMQ. We can activate those when we add them as Spring beans to our application context. Then they are automatically picked up and added to the results of the `/health` endpoint. In the following example we create a Spring bean `databaseHealth` of type `DataSourceHealthIndicator`:

```
package healthcheck

import grails.boot.GrailsApp
import grails.boot.config.GrailsAutoConfiguration
import org.springframework.boot.actuate.health.DataSourceHealthIndicator
import org.springframework.boot.actuate.health.DiskSpaceHealthIndicatorProperties

class Application extends GrailsAutoConfiguration {

    static void main(String[] args) {
        GrailsApp.run(Application)
    }

    @Override
    Closure doWithSpring() {
        { ->
            // Configure data source health indicator based
            // on the dataSource in the application context.
            databaseHealthCheck(DataSourceHealthIndicator, dataSource)

            diskSpaceHealthIndicatorProperties(DiskSpaceHealthIndicatorProperties) {
                threshold = 250 * 1024 * 1024
            }
        }
    }
}
```

To create our own health indicator class we must implement the `HealthIndicator` interface. The easiest way is to extend the `AbstractHealthIndicator` class and override the method `doHealthCheck`. It might be nice to have a health indicator that can check if a URL is reachable. For example if we need to access a REST API reachable through HTTP in our application we can check if it is available.

```
package healthcheck

import org.springframework.boot.actuate.health.AbstractHealthIndicator
import org.springframework.boot.actuate.health.Health

class UrlHealthIndicator extends AbstractHealthIndicator {

    private final String url

    private final int timeout

    UrlHealthIndicator(final String url, final int timeout = 10 * 1000) {
        this.url = url
        this.timeout = timeout
    }

    @Override
    protected void doHealthCheck(Health.Builder builder) throws Exception {
        final HttpURLConnection urlConnection =
                (HttpURLConnection) url.toURL().openConnection()

        final int responseCode =
                urlConnection.with {
                    requestMethod = 'HEAD'
                    readTimeout = timeout
                    connectTimeout = timeout
                    connect()
                    responseCode
                }

        // If code in 200 to 399 range everything is fine.
        responseCode in (200..399) ?
```

```
                builder.up() :
                builder.down(
                        new Exception(
                                "Invalid responseCode '${responseCode}' checking '${url}'."))
    }
}
```

In our `Application` class we create a Spring bean for this health indicator so it is picked up by the Spring Boot Actuator code:

```
package healthcheck

import grails.boot.GrailsApp
import grails.boot.config.GrailsAutoConfiguration
import org.springframework.boot.actuate.health.DataSourceHealthIndicator
import org.springframework.boot.actuate.health.DiskSpaceHealthIndicatorProperties

class Application extends GrailsAutoConfiguration {

    static void main(String[] args) {
        GrailsApp.run(Application)
    }

    @Override
    Closure doWithSpring() {
        { ->
            // Create instance for URL health indicator.
            urlHealthCheck(UrlHealthIndicator, 'http://intranet', 2000)

            databaseHealthCheck(DataSourceHealthIndicator, dataSource)

            diskSpaceHealthIndicatorProperties(DiskSpaceHealthIndicatorProperties) {
                threshold = 250 * 1024 * 1024
            }
        }
    }
}
```

Now when we run our Grails application and access the `/health` endpoint we get the following JSON:

```
{
    "status": "DOWN",
    "urlHealthCheck": {
        "status": "DOWN"
        "error": "java.net.UnknownHostException: intranet",
    },
    "databaseHealthCheck": {
        "status": "UP"
        "database": "H2",
        "hello": 1,
    },
    "diskSpace": {
        "status": "UP",
        "free": 96622411776,
        "threshold": 262144000
    },
}
```

Notice that the URL health check fails so the complete status is set to DOWN.

Written with Grails 3.0.1.

Original blog post written on April 24, 2015.

Add Git Commit Information To Info Endpoint

We know Grails 3 is based on Spring Boot. This means we can use Spring Boot features in our Grails application. For example a default Grails application has a dependency on Spring Boot Actuator, which means we have a `/info` endpoint when we start the application. We add the Git commit id and branch to the `/info` endpoint so we can see which Git commit was used to create the running application.

First we must add the Gradle Git properties plugin to our `build.gradle` file. This plugin create a `git.properties` file that is picked up by Spring Boot Actuator so it can be shown to the user:

```
buildscript {
    ext {
        grailsVersion = project.grailsVersion
    }
    repositories {
        mavenLocal()
        maven { url "https://repo.grails.org/grails/core" }
        maven { url "https://plugins.gradle.org/m2/" }
    }
    dependencies {
        classpath "org.grails:grails-gradle-plugin:$grailsVersion"
        classpath "com.bertramlabs.plugins:asset-pipeline-gradle:2.8.2"
        classpath "org.grails.plugins:hibernate4:5.0.6"

        // Add Gradle Git properties plugin
        classpath "gradle.plugin.com.gorylenko.gradle-git-properties:gradle-git-properties:1.4.16"
    }
}

version "1.0.0.DEVELOPMENT"
group "mrhaki.grails.gitinfo"

apply plugin:"eclipse"
apply plugin:"idea"
apply plugin:"war"
apply plugin:"org.grails.grails-web"
apply plugin:"org.grails.grails-gsp"
apply plugin:"asset-pipeline"

// Add Gradle Git properties plugin
```

```
apply plugin: "com.gorylenko.gradle-git-properties"
...
```

And that is everything we need to do. We can start our application and open the `/info` endpoint and we see our Git commit information:

```
$ http --body localhost:8080/info
{
    "app": {
        "grailsVersion": "3.1.8",
        "name": "grails-gitinfo",
        "version": "1.0.0.DEVELOPMENT"
    },
    "git": {
        "branch": "master",
        "commit": {
            "id": "481efde",
            "time": "1466156037"
        }
    }
}
$
```

Written with Grails 3.1.8.

Original blog post written on June 17, 2016.

Adding Custom Info To Info Endpoint

In a previous post we learned how to add Git commit information to the `/info` endpoint in our Grails application. We can add our own custom information to this endpoint by defining application properties that start with `info.`.

Let's add the Grails environment the application runs in to the `/info` endpoint. We create the file `grails-app/conf/application.groovy`. To get the value we must have a piece of code that is executed so using the `application.groovy` makes this possible instead of a static configuration file like `application.yml`:

```
// File: grails-app/conf/application.groovy
import grails.util.Environment

// Property info.app.grailsEnv.
// Because it starts with info. it ends
// up in the /info endpoint.
info {
    app {
        grailsEnv = Environment.isSystemSet() ? Environment.current.name : Environment.PRODUCTION.name
    }
}
```

We also want to have information available at build time to be included. Therefore we write a new Gradle task in our `build.gradle` that create an `application.properties` file in the build directory. The contents is created when we run or build our Grails application. We just have to make sure the properties stored in `application.properties` start with `info.`:

```
// File: build.gradle
...
task buildInfoProperties() {
    ext {
        buildInfoPropertiesFile =
            file("$buildDir/resources/main/application.properties")

        info = [
            // Look for System environment variable BUILD_TAG.
            tag: System.getenv('BUILD_TAG') ?: 'N/A',
            // Use current date.
            time: new Date().time,
            // Get username from System properties.
            by: System.properties['user.name']]
    }

    inputs.properties info
    outputs.file buildInfoPropertiesFile

    doFirst {
        buildInfoPropertiesFile.parentFile.mkdirs()

        ant.propertyfile(file: ext.buildInfoPropertiesFile) {
            for(me in info) {
                entry key: "info.buildInfo.${me.key}", value: me.value
            }
        }
    }
}
processResources.dependsOn(buildInfoProperties)

// Add extra information to be saved in application.properties.
buildInfoProperties.info.machine = "${InetAddress.localHost.hostName}"
...
```

Let’s run our Grails application:

```
$ export BUILD_TAG=jenkins-grails_app-42
$ grails run-app
...
| Running application...
Grails application running at http://localhost:8080 in environment: development
```

And we look at the output of the `/info` endpoint:

```
$ http --body http://localhost:8080/info
{
    "app": {
        "grailsEnv": "development",
        "grailsVersion": "3.1.8",
        "name": "grails-gitinfo",
        "version": "1.0.0.DEVELOPMENT"
    },
    "buildInfo": {
        "by": "mrhaki",
        "machine": "mrhaki-laptop-2015.local",
        "time": "1466173858064",
        "tag": "jenkins-grails_app-42"
    }
}
$
```

Written with Grails 3.1.8.

Original blog post written on June 17, 2016.

Passing System Properties With Gradle

In a other post we learned how to pass Java system properties from the command-line to a Java process defined in a Gradle build file. Because Grails 3 uses Gradle as the build tool we can apply the same mechanism in our Grails application. We need to reconfigure the `run` task. This task is of type `JavaExec` and we can use the method `systemProperties` to assign the system properties we define on the command-line when we invoke the `run` task.

We have a simple Grails 3 application with the following controller that tries to access the Java system property `sample.message`:

```
// File: grails-app/controllers/com/mrhaki/grails/SampleController.groovy
package com.mrhaki.grails

class SampleController {

    def index() {
        final String message =
            System.properties['sample.message'] ?: 'gr8'
        render "Grails is ${message}!"
    }
}
```

Next we configure the `run` and `bootRun` tasks and use `System.properties` with the Java system properties from the command-line as argument for the `systemProperties` method:

```
// File: build.gradle
...

[run, bootRun].each { runTask ->
    configure(runTask) {
        systemProperties System.properties
    }
}

...
```

Now we can invoke the `run` or `bootRun` tasks with Gradle:

```
$ gradle -Dsample.message=cool run
```

Or we can execute the `run-app` command with the `grails` command:

```
grails> run-app -Dsample.message=cool
```

Written with Grails 3.0.7.

Original blog post written on September 22, 2015.

Update Application With Newer Grails Version

In this blog post we see how to update the Grails version of our application for a Grails 3 application. In previous Grails versions there was a special command to upgrade, but with Grails 3 it is much simpler. To update an application to a newer version in the Grails 3.0.x range we only have to change the value of the property `grailsVersion` in the file `gradle.properties`.

```
# gradle.properties
grailsVersion=3.0.8
gradleWrapperVersion=2.3
```

After we have changed the value we run the `clean` and `compile` tasks so all dependencies are up-to-date.

Written with Grails 3.0.8.

Original blog post written on September 28, 2015.

Access Grails Application in BootStrap

Accessing the Grails application object in BootStrap.groovy is easy. We only have to define a variable named grailsApplication and Spring's name based injection takes care of everything.

```
class BootStrap {
    // Reference to Grails application.
    def grailsApplication

    def init = { servletContext ->
        // Access Grails application properties.
        grailsApplication.serviceClasses.each {
            println it
        }
    }

    def destroy = {}
}
```

Code written with Grails 1.1.2.

Original blog post written on January 14, 2010.

Using Spring Cloud Config Server

The Spring Cloud project has several sub projects. One of them is the Spring Cloud Config Server. With the Config Server we have a central place to manage external properties for applications with support for different environments. Configuration files in several formats like YAML or properties are added to a Git repository. The server provides an REST API to get configuration values. But there is also a good integration for client applications written with Spring Boot. And because Grails (3) depends on Spring Boot we can leverage the same integration support. Because of the Spring Boot auto configuration we only have to add a dependency to our build file and add some configuration.

Before we look at how to use a Spring Cloud Config server in our Grails application we start our own server for testing. We use a local Git repository as backend for the configuration. And we use the Spring Boot CLI to start the server. We have the following Groovy source file to enable the configuration server:

```
// File: server.groovy
@DependencyManagementBom('org.springframework.cloud:spring-cloud-starter-parent:Brixton.M4')
@Grab('spring-cloud-config-server')
import org.springframework.cloud.config.server.EnableConfigServer

@EnableConfigServer
class ConfigServer {}
```

Next we create a new local Git repository with `$ git init /Users/mrhaki/config-repo`. We use the Spring Boot CLI and our Groovy script to start a sample Spring Cloud Config Server:

```
$ spring run server.groovy -- --server.port=9000 --spring.cloud.config.server.git.uri=file:///Users/mr\
haki/config-repo --spring.config.name=configserver
...
2016-01-13 14:35:19.679  INFO 69933 --- [       runner-0] s.b.c.e.t.TomcatEmbeddedServletContainer : T\
omcat started on port(s): 9000 (http)
2016-01-13 14:35:19.682  INFO 69933 --- [       runner-0] o.s.boot.SpringApplication               : S\
tarted application in 4.393 seconds (JVM running for 7.722)
```

Next we create a YAML configuration file with a configuration property `app.message`. The name of the configuration file must start with the application name that want to use the configuration. It is best to not use hyphens in the name. Optionally we can use a Spring profile name to override configuration properties for a specific profile. The profile maps to the Grails environment names so we can use the pattern also for our Grails configuration. To learn about even more possibilities we must read the Spring Cloud Config documentation.

Here are two configuration files with a default value and a override property for the development environment:

```
# grails_cloud_config.yml
app:
    message: Default message
```

```
# grails_cloud_config-development.yml
app:
    message: Running in development mode
```

We add and commit both files in our local Git repository.

Let's configure our Grails application so it uses the configuration from our Spring Cloud Config Server. First we change `build.gradle` and add a dependency on `spring-cloud-starter-config`. We also add an extra BOM for the Spring Cloud dependencies so the correct version is automatically used.

```
// File: build.gradle
...
dependencyManagement {
    imports {
        mavenBom "org.grails:grails-bom:$grailsVersion"
        mavenBom 'org.springframework.cloud:spring-cloud-starter-parent:Angel.SR4'
    }
    applyMavenExclusions false
}
...
dependencies {
    // Adding this dependency is enough to use
    // Spring Cloud Server.
    compile "org.springframework.cloud:spring-cloud-starter-config"
}
```

Next we need to define the URL for our configuration server. We can set the system property `spring.cloud.config.uri` when we start our Grails application or we can add the file `grails-app/conf/bootstrap.yml` with the following contents:

```
# File: grails-app/conf/bootstrap.yml
spring:
    cloud:
        config:
            uri: http://localhost:9000
```

Finally we set our application name to `grails_cloud_config` which is used to get the configuration properties from the Config server:

```
# File: grails-app/conf/application.yml
...
spring:
    application:
        name: grails_cloud_config
...
```

That is it, we can now use properties defined in the configuration server in our Grails application. Let's add a controller which reads the configuration property `app.message`:

```
// File: grails-app/controllers/sample/MessageController.groovy
package sample

import org.springframework.beans.factory.annotation.Value

class MessageController {

    @Value('${app.message}')
    private String message

    def index() {
        render message
    }
}
```

When we start our application with the development environment we get the following message:

```
$ http localhost:8080/message
HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Date: Wed, 13 Jan 2016 14:08:37 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked
X-Application-Context: grails_cloud_config:development

Running in development mode

$
```

And when in production mode we get:

```
$ http localhost:8080/message
HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Date: Wed, 13 Jan 2016 14:08:37 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked
X-Application-Context: grails_cloud_config:production

Default message

$
```

Written with Grails 3.0.11

Original blog post written on January 13, 2016.

Multiple BootStraps

In Grails we can execute code when the application starts and stops. We just have to write our code in `grails-app/conf/BootStrap.groovy`. Code that needs to be executed at startup must be written in the closure `init`. In the `destroy` closure we can write code that needs be executed when the application stops. But we are not limited to one `BootStrap` class. We can create multiple BootStrap classes as long as it is placed in the `grails-app/conf` directory and the name ends with `BootStrap`.

```
// File: grails-app/conf/BootStrap.groovy
class BootStrap {
    def init = { servletContext ->
        log.debug("Running init BootStrap")
    }
    def destroy = {
        log.debug("Running destroy BootStrap")
    }
}
```

And we can create another bootstrap class:

```
// File: grails-app/conf/SampleBootStrap.groovy
class SampleBootStrap {
    def init = { servletContext ->
        log.debug("Running init SampleBootStrap")
    }
    def destroy = {
        log.debug("Running destroy SampleBootStrap")
    }
}
```

Code written with Grails 2.3.7.

Original blog post written on March 26, 2014.

Cleaning Up Before WAR Creation

Grails provides a mechanism where we can execute a closure to *do stuff* before we create a WAR file. Technically speaking we can change the contents of the staging directory. So when we run the application as an exploded WAR file or we create a WAR file in both cases the closure is executed.

The closure is very useful to delete files we don't want to be in the final WAR file, but are copied by default. We define the closure in `conf/BuildConfig.groovy` and it must be named `grails.war.resources`. The closure has a parameter which is the staging directory. The context of the closure is `AntBuilder`, so all methods we define in the closure are executed for an `AntBuilder` object. For example if we normally would use the following statement: `ant.echo(message: 'Hello')`, we must now use `echo(message: 'Hello')`. The `ant` object is implicit for the context of the closure.

In the following sample we want to delete the `Thumbs.db` files Windows generates from the application:

```
// File: conf/BuildConfig.groovy
grails.war.resources = { stagingDir ->
    echo message: "StagingDir: $stagingDir"
    delete(verbose: true) {
        fileset(dir: stagingDir) {
            include name: '**/Thumbs.db'
        }
    }
}
```

Code written with Grails 1.1.2.

Original blog post written on December 16, 2009.

Logging Service Method Calls with Dynamic Groovy

Because Grails is a Groovy web application framework we can use all the nice features of Groovy, like dynamic programming. Suppose we want to log all method invocation of Grails services. We have to look up the Grails services and override the `invokeMethod()` for the classes. Here we invoke the original method, but also add logging code so we can log when we enter and exit the method.

The best place to put our code is in `grails-app/conf/BootStrap.groovy` of our Grails application. Here we use the `init` closure to first look up the Grails service classes. Next we override the `invokeMethod()`.

```
// File: grails-app/conf/BootStrap.groovy
class BootStrap {
    def grailsApplication

    def init = { ctx ->
        setupServiceLogging()
    }

    def destroy = { }

    private def setupServiceLogging() {
        grailsApplication.serviceClasses.each { serviceClass ->
            serviceClass.metaClass.invokeMethod = { name, args ->
                delegate.log.info "> Invoke $name in ${delegate.class.name}"
                def metaMethod = delegate.metaClass.getMetaMethod(name, args)
                try {
                    def result = metaMethod.invoke(delegate, args)
                    delegate.log.info "< Completed $name with result '$result'"
                    return result
                } catch (e) {
                    delegate.log.error "< Exception occurred in $name"
                    delegate.log.error "< Exception message: ${e.message}"
                    throw e
                }
            }
        }
    }
}
```

Code written with Grails 1.2.

Original blog post written on January 18, 2010.

Multiple Environments

Grails supports different environments for configuring properties. Default we get a development, test and production environment, but we are free to add our own environments. We can define the new environment in for example `grails-app/conf/Config.groovy`. Next we can use the environment from the command line with the `-Dgrails.env=` option. Here we must use our newly created environment.

```
// grails-app/conf/Config.groovy
environments {
    development { ws.endpoint = 'http://localhost/ws.wsdl' }
    test { ws.endpoint = 'http://test/ws.wsdl' }
    production { ws.endpoint = 'http://ws.server/complete/ws.wsdl' }
    testserver1 { ws.endpoint = 'http://testserver1/test/ws.wsdl' }
    testserver2 { ws.endpoint = 'http://testserver2/test/ws.wsdl' }
}
```

To create a new WAR file with the settings for the testserver2 environment we type the following command:

```
$ grails -Dgrails.env=testserver2 war
```

Code written with Grails 1.2.

Original blog post written on January 23, 2010.

Execute Code for Current Environment

In Grails we can use the `Environment` class to execute code for specific environments. The method `executeForCurrentEnvironment()` takes a closure with the different environments and the code to be executed. This provides an elegant way to execute different blocks of code depending on the current environment.

The following sample is a simple controller where we set the value of the `result` variable depending on the environment. Besides the default environments - development, test, production - we can define our own custom environments. In the controller we provide the custom environment *myenv*.

```
package environment

import grails.util.Environment

class ExecuteController {

    def index = {
        def result

        Environment.executeForCurrentEnvironment {
            development {
                result = 'Running in DEV mode.'
            }
            production {
                result = 'Running in production mode.'
            }
            myenv {
                result = 'Running in custom "myenv" mode.'
            }
        }

        render result
    }
}
```

We we run the Grails application with `$ grails run-app` and go to the URL http://localhost:8080/app/exe
we get the following output: `Running in DEV mode.`. When we run `$ grails -Dgrails.env=myenv run-app` we get the following output in our browser: `Running in custom "myenv" mode.`.

Code written with Grails 1.2.2.

Original blog post written on May 25, 2010.

Use Different External Configuration Files

A Grails 3 application uses the same mechanism to load external configuration files as a Spring Boot application. This means the default locations are the root directory or `config/` directory in the class path and on the file system. If we want to specify a new directory to search for configuration files or specify the configuration files explicitly we can use the Java system property `spring.config.location`.

In the following example we have a configuration `application.yml` in the `settings` directory. The default base name for a configuration file is `application`, so we use that base name in our example. In this sample we use a YAML configuration file where we override the property `sample.config` for the Grails production environment.

```
# File: settings/application.yml
sample:
  config: From settings/.

---
environments:
  production:
    sample:
      config: From settings/ dir and production env.
```

Next we need to reconfigure the `run` and `bootRun` tasks, both of type `JavaExcec`, to pass on Java system properties we use from the command line when we use the Grails commands:

```
// File: build.gradle
...
tasks.withType(JavaExec).each { task ->
    task.systemProperties System.properties
}
```

Now we can start our Grails application with the Java system property `spring.config.location`. We add the `settings` directory as a search location for configuration files. We add both the directory as a local directory as well as a root package name in the class path:

```
$ grails -Dspring.config.location=settings/,classpath:/settings/ run-app
...
```

In the following example we have a configuration `config.yml` in the root of our project:

```
# File: config.yml
sample:
  config: From config.yml.

---
environments:
  production:
    sample:
      config: From config.yml dir and production env.
```

Now we start Grails and use the explicit file name as a value for the Java system property `spring.config.location`:

```
$ grails -Dspring.config.location=file:./config.yml run-app
...
```

We could specify multiple files separated by comma's. Or even combine it with directories like we used in the first example.

Written with Grails 3.0.8.

Original blog post written on September 28, 2015.

Using External Configuration Files Per Environment

Grails 3 is build on top of Spring Boot and this adds a lot of the Spring Boot features to our Grails application. For example in Spring Boot we can store configuration properties in an external file. A default Grails application already adds `application.yml` in the `grails-app/conf` directory. If we want to specify different values for a configuration property for each environment (like development, test and production) we can use `environment` section in `application.yml`. We know this from previous Grails versions with a Groovy configuration file `Config.groovy`. But we can also create different configuration files per environment and set the value for the configuration property in each file. We can use the following naming pattern for the file: `application-{env}.yml` or `application-{env}.properties`. These files need be in:

1. a `config` directory in the directory the application is running from
2. the root of the directory the application is running from
3. in a `/config` package on the classpath
4. in the root of the classpath

The order is important, because properties defined in the first locations override the properties in the last locations. When we place the files in `grails-app/conf` they get on the root of the classpath. We could also use for example `src/main/resources/config` to place extra configuration files on the classpath.

Let's see this in action with a simple Grails application. We write an implementation of the `CommandLineRunner` interface to show the value of the `sample.conf` configuration property when the application starts:

```
// File: grails-app/init/com/mrhaki/grails/EnvironmentPrinter.groovy
package com.mrhaki.grails

import grails.config.Config
import grails.core.GrailsApplication
import grails.util.Environment
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.CommandLineRunner
import org.springframework.stereotype.Component

@Component
class EnvironmentPrinter implements CommandLineRunner {

    @Autowired
    GrailsApplication grailsApplication

    @Override
    void run(final String... args) throws Exception {
        println "Running in ${Environment.current.name}"

        // Get configuration from GrailsApplication.
        final Config configuration = grailsApplication.config

        // Get the value for sample.config.
        final String sampleConfigValue = configuration.getProperty('sample.config')

        // Print to standard out.
        println "Value for sample.config configuration property = $sampleConfigValue"
    }

}
```

We define the configuration property in `grails-app/conf/application.yml`:

```
# File: grails-app/conf/application.yml
...
sample:
    config: Value from application.yml
...
```

Next we create a file `grails-app/conf/application-development.yml` which should be used when we run our Grails application in development mode:

```
# File: grails-app/conf/application-development.yml
sample:
    config: Value from application-development.yml
```

Besides YAML format we can also use the plain old properties format files. We create `grails-app/conf/application-production.properties` with a value for `sample.config` used when Grails runs in production mode:

```
# File: grails-app/conf/application-production.properties
sample.config = Value from application-production.properties
```

Finally we add a configuration file for a *custom* Grails environment:

```
# File: grails-app/conf/application-custom.yml
sample:
    config: Value from application-custom.yml
```

Now let's run the Grails application with different environments and see what value the `sample.config` property has:

```
$ grails run-app
| Running application...
Running in development
Value for sample.config configuration property = Value from application-development.yml.
...
$ grails -Dgrails.env=custom run-app
| Running application...
Running in custom
Value for sample.config configuration property = Value from application-custom.yml.
...
$ grails prod run-app
| Running application...
Running in production
Value for sample.config configuration property = Value from application-production.properties.
...
```

Written with Grails 3.0.7.

Original blog post written on September 24, 2015.

Change Base Name For External Configuration Files

With Grails 3 we get the Spring Boot mechanism for loading external configuration files. The default base name for configuration files is `application`. Grails creates for example the file `grails-app/conf/application.yml` to store configuration values if we use the `create-app` command. To change the base name from `application` to something else we must specify a value for the Java system property `spring.config.name`.

In the following example we start Grails with the value `config` for the Java system property `spring.config.name`. So now Grails looks for file names like `config.yml`, `config.properties`, `config-{env}.properties` and `config-{env}.yml` in the default locations `config` directory, root directory on the filesystem and in the class path.

```
$ grails -Dspring.config.name=config run-app
...
```

To pass the system properties when we use Grails commands we must change our `build.gradle` and reconfigure the run tasks so any Java system property from the command line are passed on to Grails:

```
...
tasks.findAll { task ->
    task.name  in ['run', 'bootRun']
}.each { task ->
    task.systemProperties System.properties
}
```

Remember that if we use this system property the default `grails-app/conf/application.yml` is no longer used.

Written with Grails 3.0.8.

Original blog post written on September 28, 2015.

Using Random Values For Configuration Properties

Since Grails 3 we can use a random value for a configuration property. This is because Grails now uses Spring Boot and this adds the `RandomValuePropertySource` class to our application. We can use it to produce random string, integer or lang values. In our configuration we only have to use `${random.<type>}` as a value for a configuration property. If the type is `int` or `long` we get a `Integer` or `Long` value. We can also define a range of `Integer` or `Long` values that the generated random value must be part of. The syntax is `${random.int[<start>]}` or `${random.int[<start>,<end>}`. For a `Long` value we replace `int` with `long`. It is also very important when we define an end value that there cannot be any spaces in the definition. Also the end value is exclusive for the range. If the type is something else then `int` or `long` a random string value is generated. So we could use any value after the dot (`.`) in `${random.<type>}` to get a random string value.

In the following example configuration file we use a random value for the configuration properties `sample.random.password`, `sample.random.longNumber` and `sample.random.number`:

```
# File: grails-app/conf/application.yml
...
---
sample:
    random:
        password: ${random.password}
        longNumber: ${random.long}
        number: ${random.int[400,420]}
...
```

Next we have this simple class that used the generated random values and displays them on the console when the application starts:

```
// File: src/main/groovy/com/mrhaki/grails/random/RandomValuesPrinter.groovy
package com.mrhaki.grails.random

import org.springframework.beans.factory.annotation.Value
import org.springframework.boot.CommandLineRunner
import org.springframework.stereotype.Component

@Component
class RandomValuesPrinter implements CommandLineRunner {

    @Value('${sample.random.password}')
    String password

    @Value('${sample.random.number}')
    Integer intValue

    @Value('${sample.random.longNumber}')
    Long longValue

    @Override
    void run(final String... args) throws Exception {
        println '-' * 29
        println 'Properties with random value:'
        println '-' * 29

        printValue 'Password', password
        printValue 'Integer', intValue
        printValue 'Long', longValue
    }

    private void printValue(final String label, final def value) {
        println "${label.padRight(12)}: ${value}"
    }

}
```

When we run our Grails application we can see the generated values:

```
$ grails run-app
...
-----------------------------
Properties with random value:
-----------------------------
Password    : 018f8eea05470c6a9dfe0ce3c8fbd720
Integer     : 407
Long        : -1201232072823287680
...
```

Written with Grails 3.0.8.

Original blog post written on September 28, 2015.

Pass Configuration Values Via Environment Variables

Since Grails 3 is based on Spring Boot we can re-use many of the Spring Boot features in our Grails application. For example in a Spring Boot application we can use environment variables to give configuration properties a value. We simply need to follow some naming rules: the name of the configuration property must be in uppercase and dots are replaced with underscores. For example a configuration property `feature.enabled` is represented by the environment variable `FEATURE_ENABLED`.

We create the following controller in a Grails 3 application with a `message` property. The value is set with the `@Value` annotation of the underlying Spring framework. With this annotation we tell the application to look for an (external) configuration property `sample.message` and assign it's value to the `message` property. If it cannot be set via a configuration property the default value is "gr8".

```
package com.mrhaki.grails

import org.springframework.beans.factory.annotation.Value

class SampleController {

    @Value('${sample.message:gr8}')
    String message

    def index() {
        render "Grails is ${message}!"
    }
}
```

If we run the application with `grails run-app` or `gradle run` the result of opening `http://localhost:8080/s` is `Grails is gr8!`.

Now we use the environment variable `SAMPLE_MESSAGE` to assign a new value to the `message` property:

```
$ SAMPLE_MESSAGE=great grails run-app
...
Grails application running at http://localhost:8080 in environment: development
```

Now when we access `http://localhost:8080/sample` we get `Grails is great!`. If we use Gradle to start our Grails application we can use `$ SAMPLE_MESSAGE=great gradle run`.

Written with Grails 3.0.7.

Original blog post written on September 22, 2015.

Pass JSON Configuration Via Command Line

We can use the environment variable `SPRING_APPLICATION_JSON` with a JSON value as configuration source for our Grails 3 application. The JSON value is parsed and merged with the configuration. Instead of the environment variable we can also use the Java system property `spring.application.json`.

Let's create a simple controller that reads the configuration property `app.message`:

```
// File: grails-app/controllers/mrhaki/grails/config/SampleController.groovy
package mrhaki.grails.config

import org.springframework.beans.factory.annotation.Value

class MessageController {

    @Value('${app.message}')
    String message

    def index() {
        render message
    }
}
```

Next we start Grails and set the environment variable `SPRING_APPLICATION_JSON` with a value for `app.message`:

```
$ SPRING_APPLICATION_JSON='{"app":{"message":"Grails 3 is Spring Boot on steroids"}}' grails run-app
| Running application...
Grails application running at http://localhost:8080 in environment: development
```

When we request the `sample` endpoint we see the value of `app.message`:

```
$ http -b :8080/message
Grails 3 is Spring Boot on steroids
$
```

If we want to use the Java system property `spring.application.json` with the Grails command we must first configure the `bootRun` task so all system properties are passed along:

```
// File: build.gradle
...
bootRun {
    systemProperties System.properties
}
...
```

With the following command we pass the configuration as inline JSON:

```
$ grails -Dspring.application.json='{"app":{"message":"Grails 3 is Spring Boot on steroids"}}' run-app
| Running application...
Grails application running at http://localhost:8080 in environment: development
```

Written with Grails 3.1.8.

Original blog post written on June 27, 2016.

One WAR to Rule Them All

If we work on a Grails project and we want to deploy our application as Web Application Archive (WAR) it is easy to create the file. To create a WAR file of our Grails application we simply invoke the command: `$ grails war`. Suppose we want to put our WAR file first on a system test application server, then a user acceptance test application server and finally the production server. We want this WAR file to be self contained and all code and configuration must be in the WAR file. We don't want to generate a WAR file for each environment separately, but a single WAR must be passed through the different environments. In this post we see how we can do this.

Suppose we have a Grails application and we define a *systemTest* and *userAcceptanceTest* environment next to the default *development*, *test* and *production* environments. We add these new environments to the environments block in `grails-app/conf/Config.groovy` and set a simple property *runningMode* with a different value for each environment.

```
// File: grails-app/conf/Config.groovy
...
environments {
    production {
        runningMode = 'LIVE'
    }
    development {
        runningMode = 'DEV'
    }
    test {
        runningMode = 'INTEGRATION TEST'
    }
    systemTest {
        runningMode = 'SYSTEM TEST'
    }
    userAcceptanceTest {
        runningMode = 'USER ACCEPTANCE TEST'
    }
}
...
```

Next we are going to change our `grails-app/views/index.gsp` and add a little code to show the value of the property *runningMode*. This way we can show which environment is used by the running WAR.

```
<%-- File: grails-app/views/index.gsp --%>
...
<h1>Application Status</h1>
<ul>
    <li>Running mode: ${grailsApplication.config.runningMode}</li>
    <li>App version: <g:meta name="app.version"></g:meta></li>
    <li>Grails version: <g:meta name="app.grails.version"></g:meta></li>
    <li>Groovy version: ${org.codehaus.groovy.runtime.InvokerHelper.getVersion()}</li>
    <li>JVM version: ${System.getProperty('java.version')}</li>
    <li>Controllers: ${grailsApplication.controllerClasses.size()}</li>
    <li>Domains: ${grailsApplication.domainClasses.size()}</li>
    <li>Services: ${grailsApplication.serviceClasses.size()}</li>
    <li>Tag Libraries: ${grailsApplication.tagLibClasses.size()}</li>
</ul>
...
```

Let’s package the application in a WAR file:

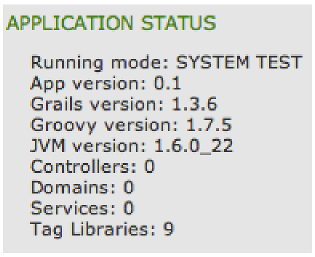
```
$ grails war
```

Next we can deploy the WAR file to our application servers. But how can we set the environment for our application, so we can see the right value of our configuration property *runningMode*? Answer: We need to set the system property `grails.env` with the correct value before we start the application server. The Grails application determines in which environment the application is running by looking at the system property *grails.env*.

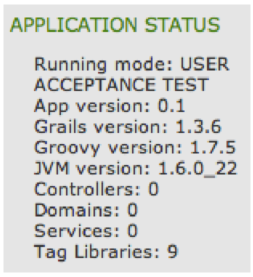
Suppose we use Tomcat as our servlet container for the Grails application. We defined separate Tomcat instances for each environment (system test, user acceptance test and production). Before we start an instance we can use the environment variable `CATALINA_-OPTS` to set the system property `grails.env`. For example for the system test Tomcat intance we define `CATALINA_OPTS` as:

```
$ export CATALINA_OPTS=-Dgrails.env=systemTest
```

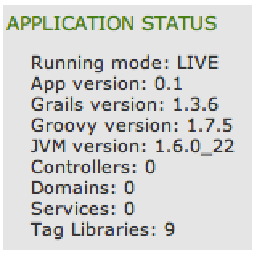
After we have defined the correct value we can install our single WAR file to the three Tomcat instances and start them. If we then open the index page of our application we can see in the left column the value or our configuration property *runningMode*:


APPLICATION STATUS

```
Running mode: SYSTEM TEST
App version: 0.1
Grails version: 1.3.6
Groovy version: 1.7.5
JVM version: 1.6.0_22
Controllers: 0
Domains: 0
Services: 0
Tag Libraries: 9
```


APPLICATION STATUS

```
Running mode: USER
ACCEPTANCE TEST
App version: 0.1
Grails version: 1.3.6
Groovy version: 1.7.5
JVM version: 1.6.0_22
Controllers: 0
Domains: 0
Services: 0
Tag Libraries: 9
```


APPLICATION STATUS

```
Running mode: LIVE
App version: 0.1
Grails version: 1.3.6
Groovy version: 1.7.5
JVM version: 1.6.0_22
Controllers: 0
Domains: 0
Services: 0
Tag Libraries: 9
```

And we see the different values for the different servers. So it is very easy to create a single WAR file, but with different configuration settings for different environments, because of the environments support in Grails. We only have to tell the application server via system property `grails.env` which environment settings need to be used.

All configuration settings are part of the application code and if we want to change a value we must rebuild the WAR file again. But what if we want to set configuration options for different environments outside of the application code? So if we want to set a configuration property for a specific environment we don't have to rebuild the WAR file? In this post we learn how to achieve this for a Grails application.

In Grails we can add extra configuration files by setting the `grails.config.locations` property in `grails-app/conf/Config.groovy`. We can assign a list of files available in the classpath or file system. Besides Groovy configuration scripts we can also define plain old Java property files. If we start with a new fresh Grails application we can see at the top of `grails-app/conf/Config.groovy` the code for this functionality in a comment block. To define the location of our environment specific configuration file per application server we read in the file location from a system property value. So we leave the placement of the configuration file up to the administrators of the application server, because we don't want to hard-code the file location in our application code. At the top of the `grails-app/conf/Config.groovy` file we set the value of `grails.config.locations`:

```
// File: grails-app/conf/Config.groovy
def CONFIG_LOCATION_SYS_PROPERTY = 'sample.app.config.file'
if (System.properties[CONFIG_LOCATION_SYS_PROPERTY]) {
    grails.config.locations = ["file:" + System.properties[CONFIG_LOCATION_SYS_PROPERTY]]
}
...
```

We change our index view and add extra code to show the value of a new configuration property: `nodeName`. The value for this property needs to be defined in the configuration file we assign via the system property `sample.app.config.location`.

```
<%-- File: grails-app/views/index.gsp --%>
...
<h1>Application Status</h1>
<ul>
    <li>Running mode: ${grailsApplication.config.runningMode}</li>
    <li>Node: ${grailsApplication.config.nodeName}</li>
    <li>App version: <g:meta name="app.version"></g:meta></li>
    <li>Grails version: <g:meta name="app.grails.version"></g:meta></li>
    <li>Groovy version: ${org.codehaus.groovy.runtime.InvokerHelper.getVersion()}</li>
    <li>JVM version: ${System.getProperty('java.version')}</li>
    <li>Controllers: ${grailsApplication.controllerClasses.size()}</li>
    <li>Domains: ${grailsApplication.domainClasses.size()}</li>
    <li>Services: ${grailsApplication.serviceClasses.size()}</li>
    <li>Tag Libraries: ${grailsApplication.tagLibClasses.size()}</li>
</ul>
...
```

Our application code changes are done and we can package the application as WAR file:

```
$ grails war
```

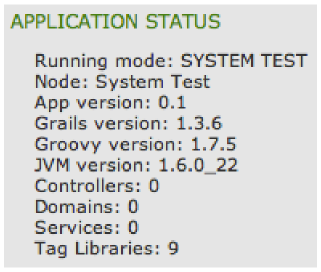
Next we create a Groovy script which sets the property `nodeName`. For each application server or environment we create a file. For example we create a file `sample-config.groovy` for the system test Tomcat instance of our previous post:

```
// File sample-config.groovy
nodeName = 'System Test'
```

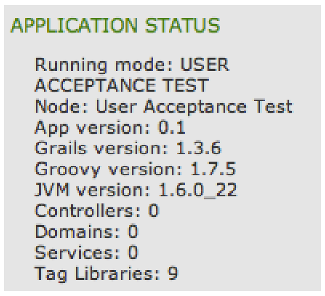
Before we start our application servers we must set the system property `sample.app.config.file`. We must reference our Groovy script which set the `nodeName` property.

```
$ export CATALINA_OPTS="-Dsample.app.config.file=sample-config.groovy"
```

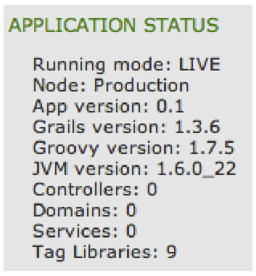
After we have defined the correct value we can install our single WAR file to the three Tomcat instances and start them. If we then open the index page of our application we can see in the left column the value or our configuration property *nodeName*:


APPLICATION STATUS

Running mode: SYSTEM TEST
Node: System Test
App version: 0.1
Grails version: 1.3.6
Groovy version: 1.7.5
JVM version: 1.6.0_22
Controllers: 0
Domains: 0
Services: 0
Tag Libraries: 9


APPLICATION STATUS

Running mode: USER ACCEPTANCE TEST
Node: User Acceptance Test
App version: 0.1
Grails version: 1.3.6
Groovy version: 1.7.5
JVM version: 1.6.0_22
Controllers: 0
Domains: 0
Services: 0
Tag Libraries: 9

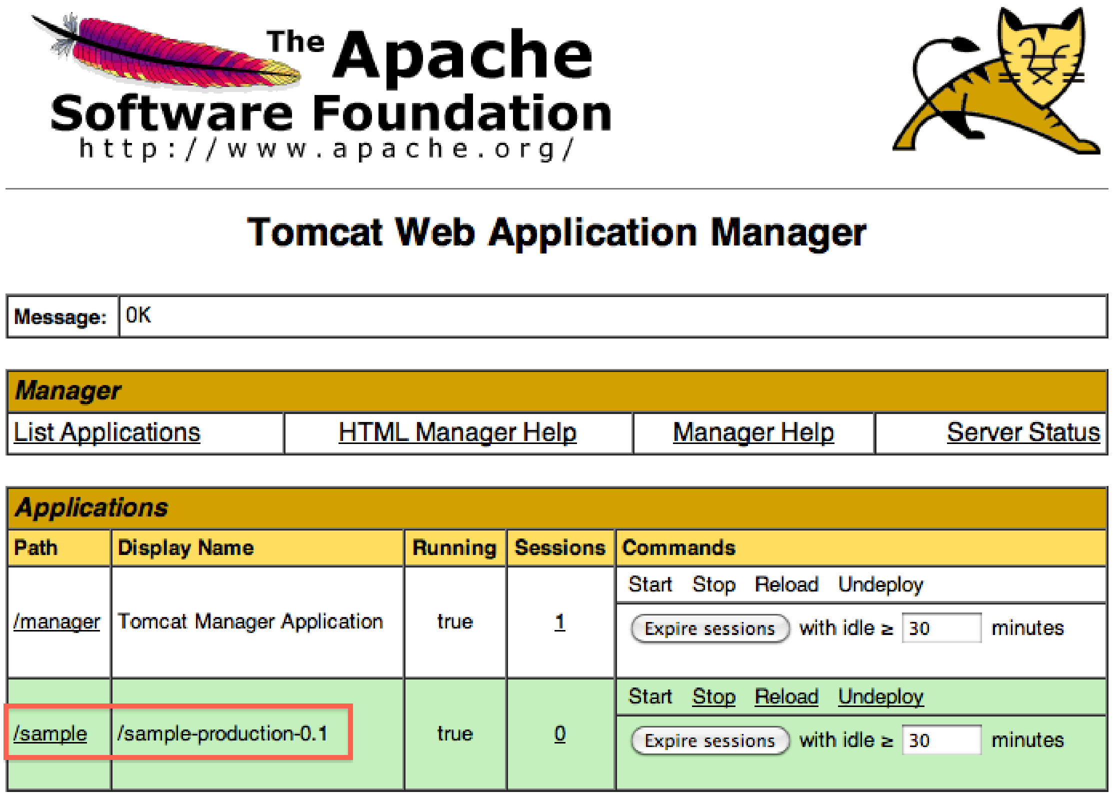

APPLICATION STATUS

Running mode: LIVE
Node: Production
App version: 0.1
Grails version: 1.3.6
Groovy version: 1.7.5
JVM version: 1.6.0_22
Controllers: 0
Domains: 0
Services: 0
Tag Libraries: 9

We see the correct value for each environment. Grails has built-in support for adding external configuration files to the application configuration. This makes it very easy to set configuration properties for separate environments and their values can be changed without building a new WAR file.

The goal is to have a single WAR file that can be deployed to several environments and still contains configuration properties per environment. We use `$ grails war` to create the

WAR file and if we deploy this WAR file to for example Tomcat we see the display name of our Grails application is set to */sample-production-0.1*:



Tomcat Web Application Manager

Message:	OK

Manager

List Applications	HTML Manager Help	Manager Help	Server Status

Applications

Path	Display Name	Running	Sessions	Commands
/manager	Tomcat Manager Application	true	1	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/sample	/sample-production-0.1	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes

The name consists of our application name, environment used to create the WAR file (by default Grails uses production when creating a WAR file) and the application version. Grails automatically sets this value when we package the application as WAR file. It can be confusing to see the environment *production* in the display name, so we set the value of the display name to a another value.

We first get the template `web.xml` Grails uses and set the value of `display-name` to a new value.

```
$ grails install-templates
```

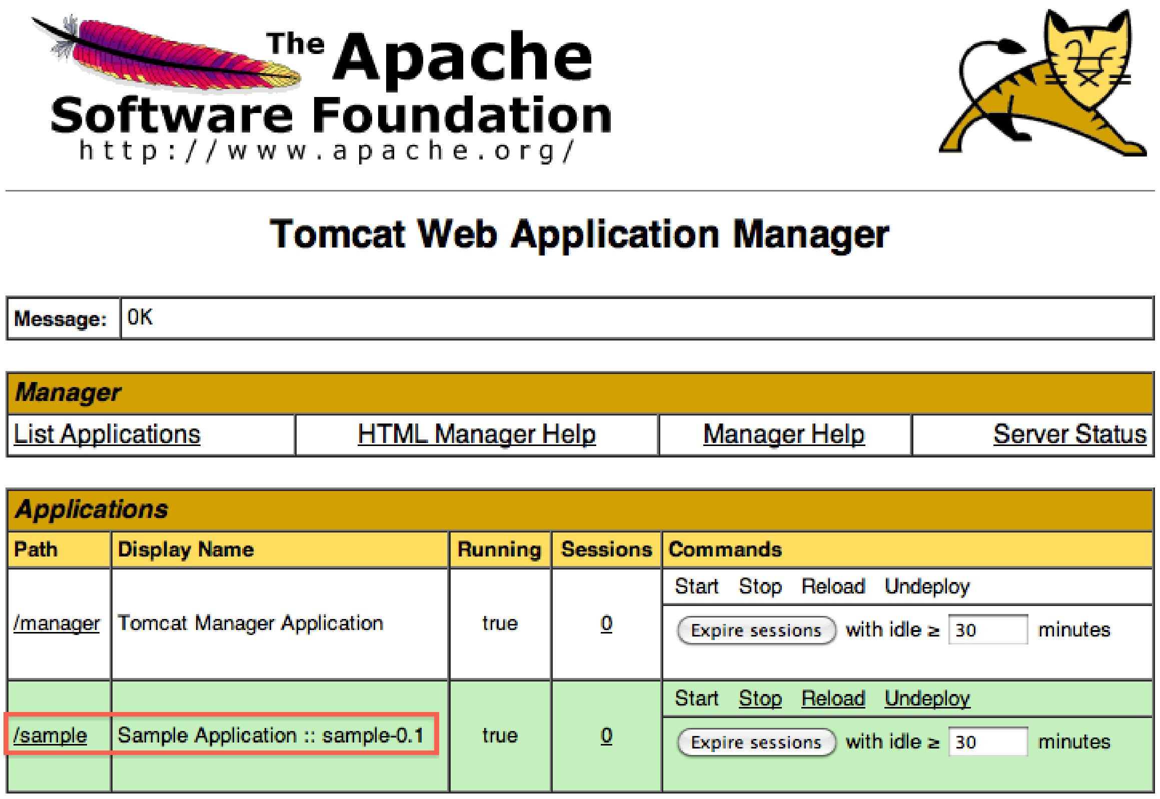
We open `src/templates/war/web.xml` and look for the `display-name` element. The value is */@grails.project.key@*. Grails uses the ANT replace task when building the WAR file to replace *@grails.project.key@* with application name, environment and application version. We want a custom value so we change the `display-name`:

```
...
<display-name>Sample Application :: @grails.app.name.version@</display-name>
...
```

We use the @…@ syntax, because we will use the ANT replace task, to add the application name and version to the generated `web.xml`. Next we create `scripts/_Events.groovy` and listen to the *WebXmlStart* event. Here we get a hold on the `web.xml` and use the ANT replace task to inject the application name and version.

```
// File: scripts/_Events.groovy
eventWebXmlStart = { webXmlFile ->
    ant.echo message: "Change display-name for web.xml"
    def tmpWebXmlFile = new File(projectWorkDir, webXmlFile)
    ant.replace(file: tmpWebXmlFile, token: "@grails.app.name.version@",
                value: "${grailsAppName}-${grailsAppVersion}")
}
```

We are ready to create the WAR file (`$ grails war`) and deploy it to our Tomcat instance. If we look at the display name we see our custom display name *Sample Application:: sample-0.1*:



Tomcat Web Application Manager

Message:	OK

Manager			
List Applications	HTML Manager Help	Manager Help	Server Status

Path	Display Name	Running	Sessions	Commands
/manager	Tomcat Manager Application	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/sample	Sample Application :: sample-0.1	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes

Original blog post written on February 04, 2011. Original blog post written on February 04, 2011. Original blog post written on February 04, 2011.

Customize the URL Format

Starting from Grails 2.0 we can change the URL format in our application. Default a camel case convention is used for the URLs. For example a controller *SampleAppController* with an action *showSamplePage* results in the following URL */sampleApp/showSamplePage*.

We can change this convention by creating a new class that implements the `grails.web.UrlConverter` interface. Grails already provides the custom UrlConverter `grails.web.HyphenatedUrlConverter`. This converter will add hyphens to the URL where there are uppercase characters and the uppercase character is converted to lowercase. Our sample controller and action result in the following URL with the `HyphenatedUrlConverter`: */sample-app/show-sample-page*.

Because Grails already provides this UrlConverter it is very easy to configure. We only need to change our configuration in *grails-app/conf/Config.groovy*. We add the key *grails.web.url.converter* with the value *hyphenated*:

```
// File: grails-app/conf/Config.groovy
...
grails.web.url.converter = 'hyphenated'
...
```

But we can implement our own class with the `grails.web.UrlConverter` interface to define our own URL format to be used in the Grails application. The interface only has one method *String toUrlElement(String)* we need to implement. The input argument is the name of the controller or action that needs to be converted. We cannot see if the value is a controller or action value, the conversion rules will be applied to both controller and action values. The following class is a sample implementation. The controller or action name is first converted to lowercase. Next we add the extension *-grails* to the controller or action. We make sure the conversion is not already done by checking if the extension is not already in place. This check is necessary because Grails will invoke our UrlConverter several times to map it to the correct controller and action names. And without the check the extension would be added again and again and again, resulting in a 404 page not found error.

```
// File: src/groovy/customize/url/format/CustomUrlConverter.groovy
package customize.url.format

import grails.web.UrlConverter
import org.apache.commons.lang.StringUtils

class CustomUrlConverter implements UrlConverter {
    private static final String GRAILS_EXTENSION = '-grails'

    String toUrlElement(String propertyOrClassName) {
        if (StringUtils.isBlank(propertyOrClassName)) {
            propertyOrClassName
        } else {
            String lowerPropertyOrClassName = propertyOrClassName.toLowerCase()
            String extendedPropertyOrClassName =
                    addGrailsExtension(lowerPropertyOrClassName)
            extendedPropertyOrClassName
        }
    }

    private String addGrailsExtension(String propertyOrClassName) {
        if (propertyOrClassName.endsWith(GRAILS_EXTENSION)) {
            propertyOrClassName
        } else {
            propertyOrClassName + GRAILS_EXTENSION
        }
    }
}
```

We have our custom UrlConverter. Now we need to configure our Grails application to use it. This time we don't change the configuration *grails-app/conf/Config.groovy*, but we add our custom implementation to the Spring configuration in *grails-app/conf/spring/resources.groovy*. If we use the name with the value of the constant `grails.web.UrlConverter.BEAN_NAME` for our implementation then Grails will use our custom UrlConverter. We can remove any *grails.web.url.converter* from *Config.groovy*, because it will not be used.

```
// File: grails-app/conf/spring/resources.groovy
import static grails.web.UrlConverter.BEAN_NAME as UrlConverterBean

beans = {
...
    "${UrlConverterBean}"(customize.url.format.CustomUrlConverter)
...
}
```

We are done. If we start our application then we use the URL */sampleapp-grails/showsamplepage-grails* to access the controller *SampleAppController* and the method *showSamplePage()* in the controller.

Code written with Grails 2.0.

Original blog post written on December 27, 2011.

Add Additional Web Application to Tomcat

In this post we learn how to add an additional WAR file to the Tomcat context when we use `$ grails run-app`. Recently I was working on a Alfresco Web Quick Start integration with Grails (current progress on GitHub). Alfresco offers inline editing of the web content with the help of an extra Java web application. This web application needs to be accessible from the Grails application, so at development time I want to deploy this web application when I use `$ grails run-app` at web context */awe*. But this scenario is also applicable if for example we use SoapUI to create a WAR file with mocked web services and we want to access it from our Grails application.

I found two sources via Google: Grails, Tomcat and additional contexts and Run a Java web application within grails. It turns out Grails fires a *configureTomcat* event when we use `$ grails run-app`, with a Tomcat instance as argument. We can configure this Tomcat instance with additional information. With the `addWebapp()` method we can add an additional context to Tomcat. We can use a directory or WAR file and we must define the context name we want to use. And furthermore we can add extra directories that are added to the classpath of the additional web application. We must create a `WebappLoader` instance from Tomcat's classloader and then we can use the `addRepository()` method to add directories to the classpath.

For my use case I had a web application packaged as a WAR file: `awe.war` and it must be deployed with the context */awe*. Furthermore extra configuration for the web application is done with a XML file found in the classpath, so we add an extra directory to the classpath of the web application.

```
import org.apache.catalina.loader.WebappLoader

eventConfigureTomcat = { tomcat ->
    // Use directory in the project's root for the
    // WAR file and extra directory to be included in the
    // web application's classpath.
    def aweWebappDir = new File(grailsSettings.baseDir, 'awe')
    def aweSharedDir = new File(aweWebappDir, 'shared')

    // Location of WAR file.
    def aweWarFile = new File(aweWebappDir, 'awe.war')

    // Context name for web application.
    def aweContext = '/awe'

    // Add WAR file to Tomcat as web application
    // with context /awe.
    def context = tomcat.addWebapp(aweContext, aweWarFile.absolutePath)
    // Context is not reloadable, but can be
    // if we want to.
    context.reloadable = false

    // Add extra directory to web application's
    // classpath for referencing configuration file needed
    // by the web application.
    def loader = new WebappLoader(tomcat.class.classLoader)
    loader.addRepository(aweSharedDir.toURI().toURL().toString())
    loader.container = context

    // Add extra loader to context.
    context.loader = loader
}
```

We start our development Grails environment with `$ grails run-app` and the web application is available at `http://localhost:8080/awe/`, next to the Grails' application path.

Code written with Grails 1.3.7.

Original blog post written on April 05, 2011.

Use a Different jQuery UI Theme with Resources Plugin

The resources plugin is a great way to manage resources in our Grails application. We define our resources like Javascript and CSS files with a simple DSL. The plugin will package the resources in the most efficient way for us in the final application.

The jQuery UI library has support for theming. We can use the default theme(s), but we can also create our own custom theme with for example the jQuery UI ThemeRoller site.

If we use the jQuery UI plugin and want to use a different than the default theme we must change our configuration for the resources plugin. We override the theme that is set by default and point it to our new custom theme. We can change `grails-app/conf/Config.groovy` or a separate resources artifact file. We add an *overrides* section and use the same *id* attribute value as set by the jQuery UI plugin. The *url* attribute points to the location of the custom jQuery UI ThemeRoller CSS file.

```
// File: grails-app/conf/Config.groovy
grails.resources.modules = {
    core {
        dependsOn 'jquery-ui'
    }
    // Define reference to custom jQuery UI theme
    overrides {
        'jquery-theme' {
            resource id: 'theme', url: '/css/custom-theme/jquery-ui.custom.css'
        }
    }
}
```

Code written with Grails 1.3.7.

Original blog post written on October 25, 2011.

Use A Different Logging Configuration File

Since Grails 3 the logging configuration is in a separate file. Before Grails 3 we could specify the logging configuration in `grails-app/conf/Config.groovy`, since Grails 3 it is in the file `grails-app/conf/logback.groovy`. We also notice that since Grails 3 the default logging framework implementation is Logback. We can define a different Logback configuration file with the environment configuration property `logging.config`. We can set this property in `grails-app/conf/application.yml`, as Java system property (`-Dlogging.config=<location>`) or environment variable (`LOGGING_CONFIG`). Actually all rules for external configuration of Spring Boot apply for the configuration property `logging.config`.

In the following example configuration file we have a different way of logging in our Grails application. We save it as `grails-app/conf/logback-grails.groovy`:

```
// File: grails-app/conf/logback-grails.groovy
import grails.util.BuildSettings
import grails.util.Environment

import org.springframework.boot.ApplicationPid

import java.nio.charset.Charset

// Log information about the configuration.
statusListener(OnConsoleStatusListener)

// Get PID for Grails application.
// We use it in the logging output.
if (!System.getProperty("PID")) {
    System.setProperty("PID", (new ApplicationPid()).toString())
}

conversionRule 'clr', org.springframework.boot.logging.logback.ColorConverter
conversionRule 'wex', org.springframework.boot.logging.logback.WhitespaceThrowableProxyConverter

// See http://logback.qos.ch/manual/groovy.html for details on configuration
appender('STDOUT', ConsoleAppender) {
    encoder(PatternLayoutEncoder) {
        charset = Charset.forName('UTF-8')
        pattern =
                '%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} ' + // Date
                        '%clr(%5p) ' + // Log level
                        '%clr(%property{PID}){magenta} ' + // PID
                        '%clr(---){faint} %clr([%15.15t]){faint} ' + // Thread
                        '%clr(%-40.40logger{39}){cyan} %clr(:){faint} ' + // Logger
                        '%m%n%wex' // Message
    }
}

root(WARN, ['STDOUT'])

if(Environment.current == Environment.DEVELOPMENT) {
    root(INFO, ['STDOUT'])

    def targetDir = BuildSettings.TARGET_DIR
    if(targetDir) {

        appender("FULL_STACKTRACE", FileAppender) {

            file = "${targetDir}/stacktrace.log"
            append = true
            encoder(PatternLayoutEncoder) {
                pattern = "%level %logger - %msg%n"
            }
        }
        logger("StackTrace", ERROR, ['FULL_STACKTRACE'], false )
    }
}
```

We use this configuration file with the following command:

```
$ LOGGING_CONFIG=classpath:logback-grails.groovy grails run-app
...
2015-09-28 16:52:38.758  INFO 26895 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : I\
nitializing Spring FrameworkServlet 'grailsDispatcherServlet'
2015-09-28 16:52:38.758  INFO 26895 --- [           main] o.g.w.s.mvc.GrailsDispatcherServlet      : F\
rameworkServlet 'grailsDispatcherServlet': initialization started
2015-09-28 16:52:38.769  INFO 26895 --- [           main] o.g.w.s.mvc.GrailsDispatcherServlet      : F\
rameworkServlet 'grailsDispatcherServlet': initialization completed in 11 ms
2015-09-28 16:52:38.769  INFO 26895 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : T\
omcat started on port(s): 8080 (http)
...
```

Written with Grails 3.0.8.

Original blog post written on September 28, 2015.

Add a DailyRollingFileAppender to Grails Logging

In Grails we can add new Log4J appenders to our configuration. We must add the new appender definition in the `log4j` closure in `conf/Config.groovy`. We define new appenders with the `appenders()` method. We pass in a closure where we define our appenders. The name of the appender is an identifier we can use when we want to define the log level and packages that need to be logged by the appender.

In the following sample configuration we create a DailyRollingFileAppender so each day a new log file is created and old log files are renamed with the date in the filename. Then we use the `root()` method to pass a closure telling we want to log all messages at level INFO to the appender.

```
// File: conf/Config.groovy
import org.apache.log4j.DailyRollingFileAppender

log4j = {
    appenders {
        appender new DailyRollingFileAppender(
            name: 'dailyAppender',
            datePattern: "'.'yyyy-MM-dd",  // See the API for all patterns.
            fileName: "logs/${appName}.log",
            layout: pattern(conversionPattern:'%d [%t] %-5p %c{2} %x - %m%n')
        )
    }

    root {
        info 'dailyAppender'
    }
}
```

Code written with Grails 1.1.2.

Original blog post written on December 13, 2009.

Use Log4j Extras Companion RollingFileAppender

Apache Extras Companion for Log4j contains a RollingFileAppender, which can be configured to automatically compress old log files. We can even save the old, archived log files to another directory than the active log file. In this post we learn how we can add and configure the `RollingFileAppender` in our Grails application.

First we must define our dependency on the Log4j Extras Companion libary. We open `BuildConfig.groovy` in the directory `grails-app/conf/` and add to the `dependencies` section the following code:

```
// File: grails-app/conf/BuildConfig.groovy
...
grails.project.dependency.resolution = {
...
    dependencies {
        compile 'log4j:apache-log4j-extras:1.0'
    }
}
...
```

Next we can configure the appender in `grails-app/conf/Config.groovy`:

```
// File: grails-app/conf/Config.groovy
import org.apache.log4j.rolling.RollingFileAppender
import org.apache.log4j.rolling.TimeBasedRollingPolicy

...
log4j = {
    def rollingFile = new RollingFileAppender(name: 'rollingFileAppender',
            layout: pattern(conversionPattern: "%d [%t] %-5p %c{2} %x - %m%n"))
    // Rolling policy where log filename is logs/app.log.
    // Rollover each day, compress and save in logs/backup directory.
    def rollingPolicy = new TimeBasedRollingPolicy(
            fileNamePattern: 'logs/backup/app.%d{yyyy-MM-dd}.gz',
            activeFileName: 'logs/app.log')
    rollingPolicy.activateOptions()
    rollingFile.setRollingPolicy rollingPolicy

    appenders {
        appender rollingFile
    }

    root {
        // Use our newly created appender.
        debug 'rollingFileAppender'
    }
}
...
```

We use TimeBasedRollingPolicy, which is quite powerful. We can configure the rollover period using a date/time pattern. If the `fileNamePattern` ends with `.gz` the contents of the log file is compressed. Finally we decouple the active log file name from the location where the archived log files are saved with the property `activeFileName`.

Code written with Grails 1.3.6.

Original blog post written on February 17, 2011.

Use TimeAndSizeRollingAppender for Logging

In a previous post we learned how to use the Log4j Extras Companion RollingFileAppender. In this post we learn how to use TimeAndSizeRollingAppender. This appender has a lot of nice features among rolling over the log file at a time interval **and** we can limit the number of rolled over log files. With this combination we can keep a history of log files, but limit how many log files are saved.

First we must download the JAR file with the appender and save it in the `lib` directory of our Grails application. Next we can configure the appender in `grails-app/conf/Conf.groovy`:

```
// File: grails-app/conf/Config.groovy
import org.apache.log4j.appender.TimeAndSizeRollingAppender
...
log4j = {
    appenders {
        appender new TimeAndSizeRollingAppender(name: 'timeAndSizeRollingAppender',
                     file: 'logs/app.log', datePattern: '.yyyy-MM-dd',
                     maxRollFileCount: 7, compressionAlgorithm: 'GZ',
                     compressionMinQueueSize: 2,
                     layout: pattern(conversionPattern: "%d [%t] %-5p %c{2} %x - %m%n"))
    }

    root {
        // Use the appender.
        debug 'timeAndSizeRollingAppender'
    }
}
...
```

We configured the appender to rollover daily, compress the contents of the archived log files after 2 rollovers, and only save 7 archived log files.

Code written with Grails 1.3.6.

Original blog post written on February 17, 2011.

Change Context Path of a Grails Application for Jetty

By default a Grails application context path is set to the application name. The context path is the **bold** part in the following URL: *http://localhost:8888**/appname**/index*. We can change the context path with properties for when we use `$ grails run-app` to run the application. We can run `$ grails -Dapp.context=/app run-app` to change the context path to */app*. Or we can set the property `app.context = /app` in `application.properties`. Or we can add `grails.app.context = '/app'` to our `conf/Config.groovy` configuration file. These properties will all affect the context path when we run `$ grails run-app`.

But what if we want to deploy our application to Jetty on a production server and use a custom context path? We have to add an extra file to `web-app/WEB-INF` with the name `jetty-web.xml`. Here we can configure Jetty specific settings for the application and one of the settings is the context path. If the application is deployed to Jetty or we run the application with `$ grails run-app` the context path is used that we set in `jetty-web.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- File: web-app/WEB-INF/jetty-web.xml -->
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN" \
"http://jetty.mortbay.org/configure.dtd">
<Configure class="org.mortbay.jetty.webapp.WebAppContext">
    <Set name="contextPath">/app</Set>
</Configure>
```

Code written with Grails 1.1.2.

Original blog post written on December 14, 2009.

Change Version For Dependency Defined By BOM

Since Grails 3 we use Gradle as the build system. This means we also use Gradle to define dependencies we need. The default Gradle build file that is created when we create a new Grails application contains the Gradle dependency management plugin via the Gradle Grails plugin. With the dependency management plugin we can import a Maven Bill Of Materials (BOM) file. And that is exactly what Grails does by importing a BOM with Grails dependencies. A lot of the versions of these dependencies can be overridden via Gradle project properties.

To get the list of version properties we write a simple Gradle task to print out the values:

```
// File: build.gradle
...
task dependencyManagementProperties << {
    description = 'Print all BOM properties to the console'

    // Sort properties and print to console.
    dependencyManagement
        .importedProperties
        .toSorted()
        .each { property -> println property }
}
...
```

When we run the task we get an overview of the properties:

```
$ gradle dependencyManagementProperties
:dependencyManagementProperties
activemq.version=5.12.3
antlr2.version=2.7.7
artemis.version=1.1.0
aspectj.version=1.8.8
atomikos.version=3.9.3
bitronix.version=2.1.4
cassandra-driver.version=2.1.9
commons-beanutils.version=1.9.2
commons-collections.version=3.2.2
commons-dbcp.version=1.4
commons-dbcp2.version=2.1.1
```

```
commons-digester.version=2.1
commons-pool.version=1.6
commons-pool2.version=2.4.2
crashub.version=1.3.2
derby.version=10.12.1.1
dropwizard-metrics.version=3.1.2
ehcache.version=2.10.1
elasticsearch.version=1.5.2
embedded-mongo.version=1.50.2
flyway.version=3.2.1
freemarker.version=2.3.23
gemfire.version=8.1.0
glassfish-el.version=3.0.0
gradle.version=1.12
groovy.version=2.4.6
gson.version=2.3.1
h2.version=1.4.191
...
BUILD SUCCESSFUL

Total time: 1.316 secs
```

For example if we want to change the version of the PostgreSQL JDBC driver that is provided by the BOM we only have to set the Gradle project property `postgresql.version` either in our build file or in the properties file `gradle.properties`:

```
// File: build.gradle
...
// Change version of PostgreSQL driver
// defined in the BOM.
ext['postgresql.version'] = '9.4.1208'
...
dependencies {
    ...
    // We don't have to specify the version
    // of the dependency, because it is
    // resolved via the dependency management
    // plugin.
    runtime 'org.postgresql:postgresql'
}
...
```

Another way to change the version for a dependency defined in the BOM is to include a dependency definition in the `dependencyManagement` configuration block. Let's see what it looks like for our example:

```
// File: build.gradle
...
dependencyManagement {
    imports {
        mavenBom "org.grails:grails-bom:$grailsVersion"
    }
    dependencies {
        // Dependencies defined here overrule the
        // dependency definition from the BOM.
        dependency 'org.postgresql:postgresql:9.4.1208'
    }
    applyMavenExclusions false
}

dependencies {
    ...
    // We don't have to specify the version
    // of the dependency, because it is
    // resolved via the dependency management
    // plugin.
    runtime 'org.postgresql:postgresql'
}
...
```

To see the actual version that is used we can run the task `dependencyInsight`:

```
$ gradle dependencyInsight --dependency postgres --configuration runtime
:dependencyInsight
org.postgresql:postgresql:9.4.1208 (selected by rule)

org.postgresql:postgresql: -> 9.4.1208
\--- runtime

BUILD SUCCESSFUL

Total time: 1.312 secs
```

This is just another nice example of the good choice of the Grails team to use Gradle as the build system.

Written with Grails 3.1.6

Original blog post written on May 11, 2016.

Validation and Data Binding

Add Extra Valid Domains and Authorities for URL Validation

Grails has a built-in URL constraint to check if a String value is a valid URL. We can use the constraint in our code to check for example that the user input http://www.mrhaki.com is valid and http://www.invalid.url is not. The basic URL validation checks the value according to standards RFC1034 and RFC1123. If want to allow other domain names, for example server names found in our internal network, we can add an extra parameter to the URL constraint. We can pass a regular expressions or a list of regular expressions for patterns that we want to allow to pass the validation. This way we can add IP addresses, domain names and even port values that are all considered valid. The regular expression is matched against the so called authority part of the URL. The authority part is a hostname, colon (:) and port number.

In the following sample code we define a simple command object with a String property `address`. In the `constraints` block we use the URL constraint. We assign a list of regular expression String values to the URL constraint. Each of the given expressions are valid authorities, we want the validation to be valid. Instead of a list of values we can also assign one value if needed. If we don't want to add extra valid authorities we can simple use the parameter `true`.

```
// Sample command object with URL constraint.
class WebAddress {
    String address

    static constraints = {
        address url: ['129.167.0.1:\\d{4}', 'mrhaki']

        // Or one String value if only regular expression is necessary:
        // address url: '129.167.0.1:\\d{4}'

        // Or simple enable URL validation and don't allow
        // extra hostnames or authorities to be valid
        // address url: true
    }
}
```

Code written with Grails 2.2.4

Original blog post written on October 13, 2013.

Combining Constraints with Shared Constraints

In our Grails applications we might have fields that need the same combination of constraints. For example we want all email fields in our application to have a maximum size of 256 characters and must apply to the email constraint. If we have different classes with an email field, like domain classes and command objects, we might end of duplicating the constraints for this field. But in Grails we can combine multiple constraints for a field into a single constraint with a new name. We do this in `grails-app/conf/Config.groovy` where we add the configuration property `grails.gorm.default.constraints`. Here we can define global constraints with can be used in our Grails application.

Let's add a custom email constraint in our application:

```
// File: grails-app/conf/Config.groovy
...
grails.gorm.default.constraints = {
    // New constraint 'customEmail'.
    customEmail(maxSize: 256, email: true)
}
...
```

To use the constraint in a domain class, command object or other validateable class we can use the `shared` argument for a field in the `constraints` configuration. Suppose we want to use our `customEmail` constraint in our `User` class:

```
// File: src/groovy/com/mrhaki/grails/User.groovy
package com.mrhaki.grails.User

@grails.validation.Validateable
class User {
    String username
    String email

    static constraints = {
        // Reference constraint from grails.gorm.default.constraints
        // with shared argument.
        email shared: 'customEmail'
    }
}
```

Code written with Grails 2.3.7.

Original blog post written on March 17, 2014.

Custom Data Binding with @DataBinding Annotation

Grails has a data binding mechanism that will convert request parameters to properties of an object of different types. We can customize the default data binding in different ways. One of them is using the `@DataBinding` annotation. We use a closure as argument for the annotation in which we must return the converted value. We get two arguments, the first is the object the data binding is applied to and the second is the source with all original values of type `SimpleMapDataBindingSource`. The source could for example be a map like structure or the parameters of a request object.

In the next example code we have a `Product` class with a `ProductId` class. We write a custom data binding to convert the `String` value with the pattern `{code}-{identifier}` to a `ProductId` object:

```
package mrhaki.grails.binding

import grails.databinding.BindUsing

class Product {

    // Use custom data binding with @BindUsing annotation.
    @BindUsing({ product, source ->

        // Source parameter contains the original values.
        final String productId = source['productId']

        // ID format is like {code}-{identifier},
        // eg. TOYS-067e6162.
        final productIdParts = productId.split('-')

        // Closure must return the actual for
        // the property.
        new ProductId(
            code: productIdParts[0],
            identifier: productIdParts[1])

    })
    ProductId productId

    String name

}

// Class for product identifier.
class ProductId {
    String code
    String identifier
}
```

The following specification shows the data binding in action:

```
package mrhaki.grails.binding

import grails.test.mixin.TestMixin
import grails.test.mixin.support.GrailsUnitTestMixin
import spock.lang.Specification
import grails.databinding.SimpleMapDataBindingSource

@TestMixin(GrailsUnitTestMixin)
class ProductSpec extends Specification {

    def dataBinder

    def setup() {
        // Use Grails data binding
        dataBinder = applicationContext.getBean('grailsWebDataBinder')
    }

    void "productId parameter should be converted to a valid ProductId object"() {
        given:
```

```
        final Product product = new Product()

        and:
        final SimpleMapDataBindingSource source =
            [productId: 'OFFCSPC-103910ab24', name: 'Swingline Stapler']

        when:
        dataBinder.bind(product, source)

        then:
        with(product) {
            name == 'Swingline Stapler'

            with(productId) {
                identifier == '103910ab24'
                code == 'OFFCSPC'
            }
        }
    }

}
```

If we would have a controller with the request parameters `productId=OFFCSPC-103910ab24&name=Swingline%` the data binding of Grails can create a `Product` instance and set the properties with the correct values.

Written with Grails 2.5.0 and 3.0.1.

Original blog post written on April 27, 2015.