

# How to rebuild your build without reworking your work

Mark Claassen, Senior Software Engineer, Donnell Systems  
Joe Foster, Developer, UI Specialist, Donnell Systems

JavaOne 2015

CON3374

# Agenda

- Why we had to do this
- How we did this

# What started this...

- We needed to change our application's deployment model
- Run a native packager
  - Package
  - Signing
- Add some SSH commands
  - Different options required on Mac

No big deal

# Big deal

- Rapid prototype was easy
- Making it production ready was not
- Testing was tedious

# Getting out the next release

Re-engineering the build was not part of the original development project

How would we accomplish this without slowing us down?

# Where Our Build Stood

- ANT build system
  - Created in the Distant Past (early 2000s)
  - Somewhat Obtuse
  - Could have made it work, but no one would be able to understand it

## **Disclaim-splanation:**

There are places where the build is managed by a dedicated team.

## **Disclaim-splanation:**

There are places where the build is managed by a dedicated team.

We do not work at one of those places



# Fix the build

- Extensively modifying the existing build seemed short-sighted.
  - Plan for the future
- Rewrite ANT build
  - Time constraints
  - Still ANT
- Something else?

# Build System Requirements

- Not Reworking our Work
  - Avoid Refactoring
  - Keep Using Netbeans
- Readable / Maintainable
  - Prep for Jigsaw and JDK9 (if possible)
- Don't Restrict Development

# Gradle

# Gradle

- Declarative frameworks like ANT
  - Good for some things, not for others
- Gradle is not declarative
  - Build files are in Groovy
  - Uses the Gradle DSL
    - “Domain Specific Language”
- Faster by skipping up-to-date tasks
- Integrated way to extend the build

# Concerns with Gradle

- Relatively new, limited resources
- We would not have a “typical” Gradle build
- Examples used modern code structure paradigms
- Needed to reproduce ANT build outputs
- Examples often too trivial
- Gradle documentation is great

# We did it

And it is pretty awesome.

- Fast
- Reliable
- **Readable**
  - The build is something we use, not something we do
  - Need to be able to go back months later and understand it enough to change it

# Session Goals

Share what we learned

- Present some patterns we think you will find useful
- Reveal some traps to avoid.

Gradle has many advanced features, but the first step is to have a build.

**Be the example**

# We made a companion project

<https://github.com/markacx/session>

Why?

- Putting more than a few lines of code on these slides makes them too small
- Showing a lot of code can obscure the main point



# Examples on the slide are short

<https://github.com/markacx/session>

- Show the heart of the example on the slides
  - Little bit of hand waving...on purpose
- Can see these in context later
- The github project has a lot more
- The program is trivial, but the build is not

# What we will cover

- Gradle basics (in 4 slides)
- Lessons learned
  - Things that are not easily found in books
  - But you should still read a book.
- All from a small company perspective
  - For those of us who do it all!

Are Gradle builds readable? We will see.

# 4 Slide Gradle Primer

Let me explain. No, there is too much. Let me sum up...

# Gradle Basics

- Projects
  - sub-projects
  - tasks
- Task is the basic building block
  - Task Types (like “compile” and “jar”)
  - Tasks can depend on other tasks
    - jar would depend on compile
  - Inputs / Outputs
  - Do things - Actions

# Actions and the build lifecycle

- Initialization phase
- Configuration phase
  - Configuration action is executed for all tasks
- Execution phase
  - Determine if tasks are up-to-date. If not:
    - “Before” actions (doFirst)
    - The main task action, from the “Type”
    - “After” actions (doLast)

# Configuration Action

- Always run at the beginning for all tasks
- Configures the tasks for **possible** execution later
- Determines the Inputs/Outputs
  - What they are, but not their values
  - If these haven't changed, the execution actions can be skipped

# CopyTask

```
task copyStuff(type: Copy) {
    from resourcesDir
    into copyDir
    include 'README.txt'
    include 'picture.png'

    doLast {
        ant.fixCrLf (eol:"lf", srcdir : destinationDir,
            includes: 'README.txt')
    }
}
```

# CopyTask

```
task copyStuff(type: Copy) {  
    from resourcesDir  
    into copyDir  
    include 'README.txt'  
    include 'picture.png'  
  
    doLast {  
        ant.fixCrLf (eol:"lf", srcdir : destinationDir,  
                    includes: 'README.txt')  
    }  
}
```



# CopyTask

```
task copyStuff(type: Copy) {
    from resourcesDir
    into copyDir
    include 'README.txt'
    include 'picture.png'
    println "Hello World"
    doLast {
        ant.fixCrLf (eol:"lf", srcdir : destinationDir,
            includes: 'README.txt')
    }
}
```

# CopyTask

```
task copyStuff(type: Copy) {
    from resourcesDir
    into copyDir
    include 'README.txt'
    include 'picture.png'
    println "Hello World"
    doLast {
        ant.fixCrLf (eol:"lf", srcdir : destinationDir,
            includes: 'README.txt')
    }
}
```

# CopyTask

```
task copyStuff(type: Copy) {
    from resourcesDir
    into copyDir
    include 'README.txt'
    include 'picture.png'
    println "Hello World"
    doLast {
        ant.fixcrlf (eol:"lf", srcdir : destinationDir,
            includes: 'README.txt')
    }
}
```

# **The example project**

# Example layout

- Directory structure
  - Root
    - Viewer
    - Servlet

# Fundamental structure

- Directory structure
  - Root
    - Viewer
    - Servlet
    - **GBuilder (root project)**
      - viewer\*
      - servlet\*

\* Subdirectories each containing a Gradle project

# settings.gradle

```
rootProject.name = 'GBuilder'
```

```
include 'viewer,servlet'
```

```
rootProject.children.each {  
    it.buildFileName = it.name + ".gradle"  
}
```

# Source path

Directory structure Gradle expects

src

main

java

com

resources

com



# Customize the source path

In build.gradle [root project]

```
project(':viewer') {  
    ext.projectBase=myRootDirectory+'/Viewer'  
}
```

In viewer.gradle

```
compileJava {  
    source "$projectBase/src"  
}
```

# Packaging the modules

- Main application was in a native bundle
- We wanted our modules to be packaged like they were in the ANT build
- ANT had a macrodef to create jar files
  - 5 to 10 lines of XML for each module

# Viewer project directory

- Common source directory
- Everything compiled at once; packaged separately

**com.rebuild** (base package)

**app** (package with main application)

**ext** (package base for extensions)

moduleX (One of these for every “module”)

...

# Code the build

- Didn't want large cookie cutter sections
- I am a coder, so I coded it
- Create a method to create a jar file

# Don't code the build

Don't code the build

- Doing this takes power from the framework
- No automatic input / output handling

Every time my build ran, all the modules would be rebuilt

# Programmatic task creation

Instead of creating a method to create the jar files, create a method to create a Jar task

---

```
void createClientTask(archive,dirs) {  
    task([type: Jar,  
        group : 'client_sub_jar',  
        dependsOn:"compileJava"],  
        "create_viewer_" + archive) { ... }
```

# Calling the method

```
task init_client (description: 'Create client sub jar tasks') {  
  createClientTask(  
    'Module1',  
    ['com/rebuild/ext/module1/**'])  
}
```

---

```
void createClientTask(archive, dirs) { ... }
```

# Short and sweet

```
createClientTask('Module1', ['com/rebuild/ext/module1/**'])  
createClientTask('Module2', ['com/rebuild/ext/module2/**'])
```

...

```
tasks.each( {  
    if (it.group == 'client_sub_jar') {  
        jar.dependsOn += it  
    }  
})
```



# Configuration of main jar task

```
jar {  
    archiveName 'MyApp.jar'  
  
    include 'com/rebuild/app/viewer/**'  
    include 'com/rebuild/client/**'  
}
```

# Don't use properties (as much)

```
task makePackage (type: Zip) {  
    archiveName <zip property name>  
    ...  
}  
task copyPackage(type: Copy) {  
    from <zip property name>  
    ...  
}
```

# Refer to other tasks

```
task makePackage (type: Zip) {  
    archiveName "package.zip"  
    ...  
}  
task copyPackage(type: Copy) {  
    from makePackage  
}
```

# Control your inputs and outputs

## Our jar file specification

- **META-INF**
  - Include VERSION-INFO.txt
- **Manifest**
  - Release build (boolean)
  - Build time

# Jar example

```
metaInf {  
    from (resourcesDir) { include 'VERSION-INFO.txt' }  
}  
manifest {  
    attributes ( 'ReleaseBuild': isReleaseBuild,  
                'BuildTime': System.currentTimeMillis() )  
}
```

# Jar example - improved

```
metaInf {
    from (resourcesDir) { include 'VERSION-INFO.txt' }
}
manifest {
    attributes ( 'ReleaseBuild': isReleaseBuild )
}
doFirst {
    manifest.attributes ( 'BuildTime': System.currentTimeMillis() )
}
```

# Be careful what you wish for

Sometime Gradle can be too great.

Inputs and outputs are mostly automatic

# An Exec task

```
task execJava (type: Exec, dependsOn : copyPackage) {  
    def file = new File(distDir,"version.txt")  
    executable System.getProperty("java.home") + "/bin/java.exe"  
    args "-version"
```

```
doFirst {    errorOutput new ByteArrayOutputStream()    }
```

```
doLast {    file.text = errorOutput.toString()    }
```



# An Exec task

```
task execJava (type: Exec, dependsOn : copyPackage) {  
    def file = new File(distDir,"version.txt")  
    executable System.getProperty("java.home") + "/bin/java.exe"  
    args "-version"
```

## **outputs.file(file)**

```
doFirst {    errorOutput new ByteArrayOutputStream()    }  
  
doLast {    file.text = errorOutput.toString()    }
```

# An Exec task

```
task execJava (type: Exec, dependsOn : copyPackage) {  
    def file = new File(distDir,"version.txt")  
    executable System.getProperty("java.home") + "/bin/java.exe"  
    args "-version"  
    inputs.property('MyExecutable',executable)  
    inputs.property('MyArgs',args)  
    outputs.file(file)  
    doFirst {    errorOutput new ByteArrayOutputStream()    }  
  
    doLast {    file.text = errorOutput.toString()    }
```

# Don't Mess With Other Tasks

```
task makePackage (type: Zip, dependsOn: [jar,init_distribution]) {
    archiveName "package.zip"
    from ( tasks.matching( { task -> task.group == 'client_sub_jar' } ) )
}

task makeDistribution (dependsOn: makePackage) {
    doLast {
        def org = makePackage.outputs.getFiles()[0];
        org.renameTo(distDir)
    }
}
```

# Efficiency is not always efficient

What happen on subsequent executions?

- makePackage checks it inputs (same)
- makePackage checks it outputs
  - package.zip is not there!
  - makePackage will run every time
  - makeDistribution will run every time

Happened to us when signing the executable

# Careful with copy and paste

- All projects that made a local distribution needed to copy that to the main distribution point
  - These tasks were (almost) all the same.
- Create a task programmatically in the root project and add it to the list of tasks in the subproject

# Add a task to a sub-project

```
if (!proj.tasks.matching( {it.name == 'createDist'}).empty) {  
    proj.task( [type: Copy, group : 'Distribution',  
              dependsOn: ':' + proj.name +  
                ':createDist'] , "copyToDistribution") {  
        from proj.distDir  
        into new File(distributionBase)  
    }  
}
```

# “Overriding” tasks

Created tasks for each and then a parent task to depend on them

```
task copyToDistribution (type: Copy, dependsOn: [
    createDist,
    copyPackageToDistribution,
    copyOtherToDistribution ] )
```

# “Overriding” tasks

```
if (!proj.tasks.matching( {it.name == 'createDist'}).empty) {  
    if (proj.tasks.matching( {it.name == 'copyToDistribution'}).empty) {  
        proj.task(    [type: Copy, group : 'Distribution',  
        ...
```



## “<<” - shorthand for “doLast”

```
task justForShow << {  
    println "Just for show"  
}
```

```
task justForShow {  
    doLast { println "Just for show" }  
}
```

# “<<” - shorthand for “doLast”

```
task justForShow (description: "My Task") << {  
    println "Just for show"  
}
```

```
task justForShow (type: Copy, description: "My Task") << {  
    println "Just for show"  
}
```

# “<<” - shorthand for “doLast”

```
task justForShow (type: Copy, description: "My Task") << {  
    println "Just for show"  
  
    from resourcesDir  
    into distributionBase  
    include "MyShowFile.txt"  
}
```

# “<<” - shorthand for “doLast”

```
task justForShow (type: Copy, description: "My Task") {  
    doLast { println "Just for show" }
```

```
    from resourcesDir  
    into distributionBase  
    include "MyShowFile.txt"
```

```
}
```

```
task justForShow (type: Copy, description: "My Task", group : "show task" ) << {
```

# UI

Just a touch of JavaFX

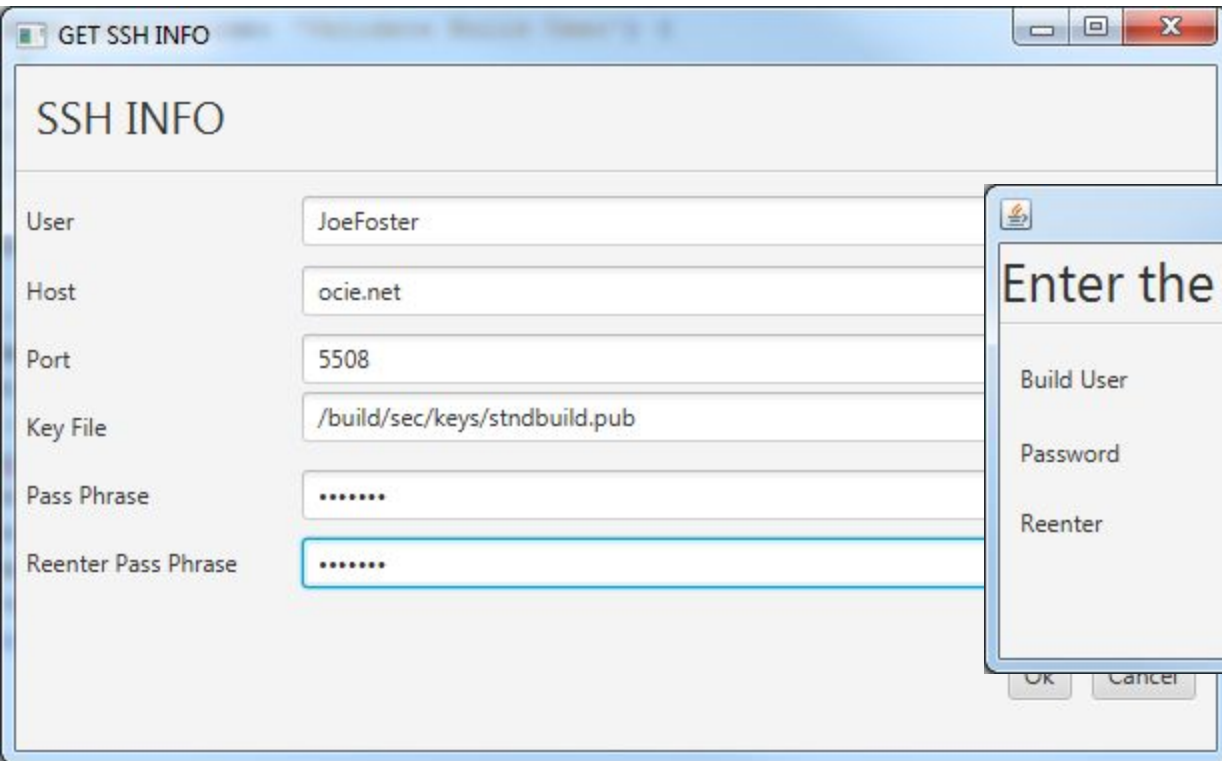
# Appropriate Use of UI

- Why UI in a build process?
  - Warm Fuzzy
  - Ad-Hoc Situations

# Our Build UI

- Needed Two Prompting Dialogs
  - SSH Config
  - Build Destination Verification
  
- Mandate to use JavaFX
  - Perfect entry point
  - Limited Impact to Product
  - Not end-user facing UI

# The Dialogs



GET SSH INFO

SSH INFO

User: JoeFoster

Host: ocie.net

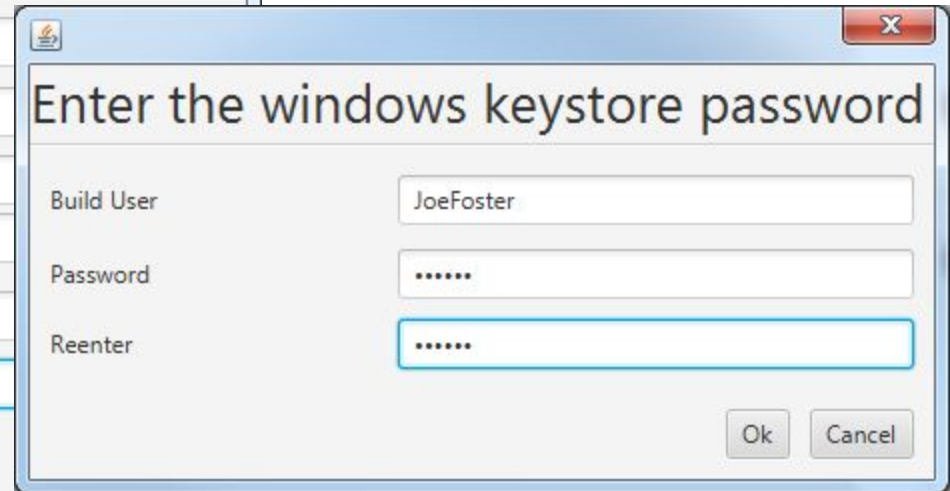
Port: 5508

Key File: /build/sec/keys/stndbuild.pub

Pass Phrase: .....

Reenter Pass Phrase: .....

Ok Cancel



Enter the windows keystore password

Build User: JoeFoster

Password: .....

Reenter: .....

Ok Cancel



# JavaFX in Gradle

- Initially Easy
- Tricky to get return values
- Issues with secondary usage

# How to get back to Gradle?

How to return from the dialogs

- Can't just close them
- `Platform.exit()`

# Subsequent Dialogs won't Open

- Can't close JavaFX and open it again

```
Caused by: java.lang.IllegalStateException: Application launch must not be called more than once  
    at com.sun.javafx.application.LauncherImpl.launchApplication(Unknown Source)  
    at com.sun.javafx.application.LauncherImpl.launchApplication(Unknown Source)  
    at javafx.application.Application.launch(Unknown Source)
```

- What can we do?

# Two (Frameworks) is better than One

- Swing and JavaFX
- JFXPanel
- Careful with the threading
  - Watch your `SwingUtilities.invokeLater()`s
  - and your `Platform.runLater()`s
  - ... or just use the system property

```
javafx.embed.singleThread = true
```

# Summary

# Summary

- To make your own Gradle build
  - Understand the lifecycle
  - Careful with your inputs and outputs
  - Avoid coding steps that can be done with a task
- Consider UI for ad-hoc scenarios
  - Beware of JavaFX pitfalls

**Don't recreate your build in Gradle, create a Gradle build**

# Oh, the possibilities...

- Refactoring can occur without worrying about the build
  - JDK 9
- Project conversion
- Go on vacation

# Contact Information

Mark Claassen	<a href="mailto:mclaassen@ocie.net">mclaassen@ocie.net</a>
Joseph Foster	<a href="mailto:jfoster@ocie.net">jfoster@ocie.net</a>

<https://github.com/markacx/session>



```

//Example project to go along with JavaOne 2015 conference session CON3374
//To run full build, install gradle. Then, with the gradle binary in your
//system PATH, run gradle createDist
//
// Root Project build.gradle

import org.gradle.api.Project

description = 'Root project'

logger.quiet("Java version of build: " + System.getProperty("java.version"))

//If we want to set defaults for our project that our private to us, maybe where our codesign program is, we can put in
//in this file. I don't put the private.gradle in version control.
File privateFile = new File(projectDir,'private.gradle')
if (privateFile.exists()) {
    apply from: 'private.gradle'
}

ext.myRootDirectory = new File(projectDir,'..').canonicalFile.absolutePath

if (!hasProperty('distributionBase')) {
    //We will use the current directory by default, but we can specify this in the private.gradle file and not use the default
    ext.distributionBase = new File(projectDir,'dist').canonicalFile.absolutePath
}
ext.isReleaseBuild = "true".equals(isReleaseBuildText.toLowerCase())
ext.resourcesDir = new File(projectDir,'resources')

apply from: 'lib-def.gradle'

logger.quiet "Base = ${distributionBase}"

subprojects {
    ext.distDir = new File(projectDir,'dist')
    ext.resourcesDir = resourcesDir

    //Find all tasks that have a compile step, and configure the compiler how we want
    tasks.withType(JavaCompile) {
        options.incremental = true
        options.warnings = false
        options.deprecation = false
        options.fork = true;
        options.forkOptions.with {
            memoryMaximumSize = "512M"
        }
    }
    //Find all tasks that have a Jar task, and configure the jar how we want
    tasks.withType(Jar) {
        metaInf {
            from (resourcesDir) {
                include 'VERSION-INFO.txt'
            }
        }
        manifest {
            attributes {
                "ReleaseBuild": isReleaseBuild,
                //Having BuildTime in the config section will force this task run every time. Instead, put it in doFirst section
                //BuildTime: System.currentTimeMillis()
            }
        }
        doFirst {
            //Doing this prevents a change in the build date from invalidating the jar file and triggering a rebuild
            manifest.attributes ("BuildTime": System.currentTimeMillis())
        }
    }
    //After everything has "evaluated" create a copyToDistribution task if there is not already one.
    afterEvaluate { (Project proj) ->
        if (!proj.tasks.matching( {it.name == 'createDist'}).empty()) {
            if (proj.tasks.matching( {it.name == 'copyToDistribution'}).empty()) {
                proj.task(type: Copy, group : 'Distribution',dependsOn: ":"+proj.name + ':createDist','copyToDistribution') {
                    from proj.distDir
                    into new File(distributionBase)
                    doLast {
                        println "Default' copyToDistribution task in project ${project.name}"
                    }
                }
            }
        }
        //Whether or not we created a task or not, if the project has a "createDist" task, make sure the root project's
        //createDist task depends on it, so we are sure it gets executed
        rootProject.tasks[createDist].dependsOn += proj.tasks[createDist]
        rootProject.tasks[createDist].dependsOn += proj.tasks[copyToDistribution]
    }
}
//This will never copy anything because the "configuration" is in the doLast section ("<<" means doLast)
//To fix, remove the "<<"
task justForShow (type: Copy) << {
    println "Just for show"
    from resourcesDir
    into distributionBase
    include "MyShowFile.txt"
}
//clean is defined in the Java plugin, but not in this file, so we just create a new task called 'clean' here
//Note, that in the viewer.gradle file, which uses the Java plugin, we do our delete in the configure action.
//Here, this is going to extend from default action, meaning that a delete called here will actual do the delete.
//Therefore, we put the delete in a doLast action
task clean {
    doLast {
        delete distributionBase
    }
}
task createDist (dependsOn: [':viewer:createDist','servlet:createDist',justForShow]) {
}

project(':viewer') {
    ext.projectBase=myRootDirectory+Viewer
}
project(':servlet') {
    ext.projectBase=myRootDirectory+Servlet
}

```

```

// Viewer Project viewer.gradle

import net.ocie.javaone2015.build.fx.*
import net.ocie.javaone2015.build.validation.*
import org.gradle.api.GradleException

apply plugin: 'java'

File copyDir = new File(distDir, 'copyDirectory')
File packageDir = new File(distDir, 'packageDirectory')

task validate(description: 'Validate Build User') {
    doFirst {
        //this will open a SSH configuration dialog, in Pure JavaFX.
        //because Platform.exit() is the only way to return to the Gradle context
        //we cannot open further javafx dialogs.
        //uncomment out the below line to see this behavior
        //def pureFXConfigOutput = PureFXGetSSHInfo.open(Window("This is the initial config"));

        //this limitation led us to develop a Swing wrapper for our JavaFX layout
        def fxmlURL = BuildValidationController.class.getResource("buildvalidation.fxml")
        def styleURL = BuildValidationController.class.getResource("buildvalidation.css")
        //we will pass in a string configuration object, and expect a ValidatorResponse back
        def window = new FXPanelFrame<String, ValidationResponse>("Validation Config String", fxmlURL, styleURL);
        window.setExtraProperty("title", "Enter the windows keystore password");
        window.show();
        def val = window.getReturnValue();
        if (!val.isSuccess()) {
            throw new GradleException("Build canceled by user")
        }
    }
}

//Using << means to put the code in braces in the doLast action. If this task gets edited to have a type,
//make sure you removed this so you can configure the task
task init_distribution << {
    distDir.mkdirs()
    copyDir.mkdirs()
    packageDir.mkdirs()
}

//This will configure the clean action in the Java plugin. Add some additional steps to clean the things we
//make in the init_distribution
clean {
    delete distDir
    delete copyDir
    delete packageDir
}

//This would not have to be a task. The createClientTask lines could just be part of the script, but this helps
//me keep everything in order
task init_client(description: 'Create client sub jar tasks') {
    createClientTask('Module1', ['com/rebuild/ext/module1'])
    createClientTask('Module2', ['com/rebuild/ext/module2'])
    createClientTask('Module3', ['com/rebuild/ext/module3'])
    //if modules 4 and 5 are so tied together that they can be bundled as one, we can do this too
    createClientTask('Module4n5', ['com/rebuild/ext/module4', 'com/rebuild/ext/module5'])
}

//We want to make sure these new tasks are executed. To do that, we need something to depend on them
//find all the tasks in the project, find the ones we just created, and make the are class depend on these being done
tasks.each {
    logger.info("Examining task " + it)
    if (it.group == "client_sub_jar") {
        jar.dependsOn += it
        logger.info("Adding " + it)
    }
}

//This would not have to be a task. The createServerTask lines could just be part of the script, but this helps
//me keep everything in order
task init_server(description: 'Create server sub jar tasks') {
    createServerTask('Module1', ['com/rebuild/ext/module1'])
    createServerTask('Module2', ['com/rebuild/ext/module2'])
    createServerTask('Module3', ['com/rebuild/ext/module3'])
    //if modules 4 and 5 are so tied together that they can be bundled as one, we can do this too
    createServerTask('Module4n5', ['com/rebuild/ext/module4', 'com/rebuild/ext/module5'])
}

tasks.each {
    logger.info("Examining task " + it)
    if (it.group == "server_sub_jar") {
        jar.dependsOn += it
        logger.info("Adding " + it)
    }
}

//Just include the java files, that way a resources in this structure won't be considered as inputs that might
//force a recompile
compileJava {
    source "$projectBase/src"
    include "**/*.java"
}

//Resources would be everything that isn't a java file
processResources {
    from "$projectBase/src"
    exclude "**/*.java"
}

//Configure the jar task from the Java plugin to not include the files in the modules
jar {
    archiveName 'MyApp.jar'
    //The module files are in other jar files, so exclude them from this one
    include 'com/rebuild/app/viewer/**'
    include 'com/rebuild/client/**'
}

task copyStuff(type: Copy) {
    from resourcesDir
    into copyDir
    include 'README.txt'
    include 'picture.png'
    println 'Hello World'
    doLast {
        ant.fixCrLf(eol: "lf", srcDir: destinationDir, includes: 'README.txt')
    }
}

task execJava(type: Exec, dependsOn: [compileJava, init_distribution]) {
    def file = new File(packageDir, "version.txt")
    executable System.getProperty("java.home") + "/bin/java.exe"
    args "-version"
}

//The arguments and executable are not considered part of the "inputs"
//Add them explicitly if a change in them should re-run the task
inputs.property("MyArgs", args)
inputs.property("MyExecutable", executable)

//Without setting an output, the task not have any output and so will run every time
outputs.file(file)

doFirst {
    //Running java with the -version option outputs the the version information to standard error
    errorOutput new ByteArrayOutputStream()
}

doLast {
    file.text = errorOutput.toString()
}

task makePackage(description: "Put files in zip file", type: Zip, dependsOn: [jar, init_distribution, copyStuff, execJava]) {
    archiveName "package.zip"
}

```

```

def subTasks = tasks.matching( { task -> task.group == 'client_sub_jar' } )
from subTasks
}
//This rename moves the file from the output of makePackage, forcing it to re-run every time
task copyPackage_bad (description : 'Do something to the zip file', dependsOn: makePackage) {
def org = makePackage.outputs.getFiles()[0];
org.renameTo(packageDir)
}
task copyPackage (type : Copy, description : 'Do something to the zip file', dependsOn: makePackage) {
from makePackage
into packageDir
}
task createDist (type: Copy, dependsOn: [execJava.jar.copyPackage,copyStuff]) {
from resourceDir
into distDir
include "ViewerDistribution.txt"
}

task copyPackageToDistribution(type: Copy, dependsOn: createDist) {
from distDir
into new File(rootProject.distributionBase)
include "$packageDir.name"
}
task copyOtherToDistribution(type: Copy, dependsOn: createDist) {
from distDir
into new File(rootProject.distributionBase)
include "$copyDir.name"
}
//Make this depend on the fine-grained tasks so one change doesn't require re-copying everything
task copyToDistribution (type: Copy, dependsOn: [createDist,copyPackageToDistribution,copyOtherToDistribution]) {
from distDir
into new File(rootProject.distributionBase)
exclude "$copyDir.name"
exclude "$packageDir.name"
doLast {
println "Overridden" copyToDistribution task in project ${project.name}
}
}
void createClientTask(archive,dirs) {
task[type: Jar, group : 'client_sub_jar', dependsOn:"compileJava","create_viewer_" + archive] {
from sourceSets.main.output

include dirs
exclude """/server""
archiveName "${archive}C.jar"
artifacts {
archives file: new File(buildDir.archiveName), name: archive, type: 'jar', classifier: 'extra'
}
}
tasks.jar.dependsOn += "create_viewer_" + archive
}
void createServerTask(archive,dirs) {
task[type: Jar, group : 'server_sub_jar', dependsOn:"compileJava","create_server_" + archive] {
from sourceSets.main.output

include dirs
exclude """/client""
archiveName "${archive}S.jar"
artifacts {
archives file: new File(buildDir.archiveName), name: archive, type: 'jar', classifier: 'extra'
}
}
tasks.jar.dependsOn += "create_server_" + archive
}
}

```

```
// Servlet Project servlet.gradle

apply plugin: 'java'

dependencies {
    compile libraries.Tomcat
}

task init_distribution << {
    distDir.mkdirs()
}

clean {
    delete distDir
}

compileJava {
    source "SprojectBase/src"
    include "**/*.java"
}

processResources {
    from "SprojectBase/src"

    exclude "**/*.java"
}

jar {
    archiveName = "Servlet.jar"
}

task createDist (type: Copy, dependsOn: jar) {
    from (jar.outputs)
    into distDir
}
}
```