HTML resource management enhancements to replace g.resource etc.

Resources Plugin - Reference Documentation

Authors: Marc Palmer (marc@grailsrocks.com), Luke Daley (ld@ldaley.com), Peter N. Steinmetz (ndoc3@steinmetz.org)

Version: 1.2.11

Table of Contents

1 Overview

This plugin provides the Resources framework for Grails.

The issues that the Resources framework tackles are:

- Web application performance tuning is difficult
- Correct load order for resources that depend on others
- Deferring inclusion of JavaScript to the end of the document
- The need for a standard way to for Grails plugins to expose static resources
- The need for an extensible processing chain to optimize resources
- Preventing inclusion of the same resource multiple times
- The need for identical behaviour in development and production

The Resources plugin achieves these goals by introducing new artefacts and processing the resources using the server's local file system.

In tandem with Sitemesh layouts and the Grails plugin dependency system, you gain the ability to express dependencies on specific UI libraries, and the ability to specify the resources that a page needs anywhere in the page GSPs or even in rendered GSP templates.

The Resource framework tags confer a whole new level of abstraction and power to application and plugin developers. It makes possible new patterns for managing resources and smart handling of JavaScript and CSS code.

It adds artefacts for declaring resources, for declaring "mappers" that can process resources, and a servlet filter to serve processed resources.

1.1 Quick Start

To demonstrate the power of the framework, here's a quick demonstration of usage with the jQuery, jQuery-Ui and Blueprint plugins which exposes resource modules.

1.1.1 Make sure jQuery plugin is installed

For Grails 1.4, it is installed by default. For older versions:

```
grails install-plugin jquery*
```

1.1.2 Install jQuery UI and Blueprint plugins

Just run:

```
grails install-plugin jquery-ui
grails install-plugin blueprint
```

1.1.3 Edit your Sitemesh layout

You need to add the layoutResources tag twice to your page, for the <head> resources and end-of-body resources.

Your grails-app/views/layouts/main.gsp:

```
<html>
   <head>
      <g:layoutTitle/>
      <r:layoutResources/>
   </head>
   <body>
      <g:layoutBody/>
      <r:layoutResources/>
   </body>
</html>
```

1.1.4 Edit your GSP page to include jQuery

All you have to do here is add the require to your GSP page. Anywhere will do, but in <head> makes sense, and add a bit of code:

```
<html>
   <head>
      <meta name="layout" content="main"/>
      <r:require modules="jquery-ui, blueprint"/>

<r:script>
          $(function() {
              $('#form').dialog('open');
          });
      </r:script>
   </head>
   <body>
      <div id="form">
           Hello World
      </div>
   </body>
</html>
```

1.1.5 View the page source

When you run this and request the GSP page, you should view the source and look at what is happening in the page. Use Safari or Chrome resource inspector to see what files were requested and when.

You should have a page that is rendered with the Blueprint CSS framework, pulling in both jQuery and jQuery-UI, with jQuery-UI JS deferred to the end of the page, jQuery loaded in the head, and the "document ready" JS code fragment to set up the page is rendered right at the end of the body after jQuery UI has loaded.

All with almost zero effort!

1.1.6 Now optimize your application

Installing the "cached-resources" and "zipped-resources" plugins and running your app again, you will then find that your resources are cached in the browser "eternally" and transferred with zip encoding.

Simply run these commands:

```
grails install-plugin zipped-resources
grails install-plugin cached-resources
```

No code changes are necessary for the simple use case.

For details of these plugins please see the documentation for zipped-resources and cached-resources.

More resources-aware plugins will be released. You can search by tag "resources" on the Grails portal with this link.

2 Concepts

The resources framework provides 5 primary functions to a Grails application:

1. Bundling of resources into modules with dependency management.
2. Processing of resources prior to serving them.
3. Tag library to render links to resources in modules.
4. Tag library to render links to other resources which are not in modules.
5. Service of resources to clients based on appropriate requests.

In the general sense, a resource is an item that a client accesses, whether from the application or a different location. Typical items served as resources by an application are CSS, Javascript, icon, and image files. With the resources plugin tags resource, img and external replacing the corresponding grails tags in the `g:` namespace as of Grails 1.4, this more general meaning is important, as different functions of the plugin view resources in different ways. The following sections will provide an overview of how resources are handled by the resource plugin with respect to each of these functions. Please see the other sections of this User Guide for details regarding each of the tags and objects mentioned.

Bundling of Resources into Modules with Dependency Management.

In the resources framework, you define resource modules that have a name and contain one or more resources. The modules are declared in separate files named like ModuleResources.groovy (forming a Resources artefact) or in a separate modules block within Config.groovy.

The resources can either be local to the application, and declared with the usual map of attributes, such as `dir:` and `file:`, or external to the application, with an absolute url containing `//` .

When resources in a module are required on a groovy server page (GSP), for example a css file in a set, this requirement is indicated by including a require tag for the module in the header. Links to the resources themselves are then included in the GSP by including one or two layoutResources tags. The first is placed in the header and the second at the bottom of the page to render any items which should be deferred to the end of the page, such as javascript fragments.

Dependencies can be specified between modules, so that a page which requires resources in one module, which in turn requires other resources in other modules, need only require the former and all dependent modules will automatically be included on the page, in the proper order.

Processing of Resources Prior to Serving Them.

Another important function of the Resources plugin is to process resources prior to serving them to clients. For example, CSS files can be minified and combined to improve server performance.

This processing is performed by Mappers, which can generally perform any type of modification of the resources, including renaming and combining files. An example is combination of css files, which can also include re-writing links to the css files so combined. The specific mappers applied to a resource can be specified based on the type of resource, type of mapper, or on a per resource basis.

Another form of resource processing provided by the Resources plugin is aggregation of pieces of Javascript. Segments of script can be embedded within a GSP using the script tag. These will then be rendered into the head or end of the page when the layoutResources tag is invoked. The location (head or end of the page) of both javascript fragments and other resources is controlled by defining their `disposition` attribute.

Tag Library to Render Links to Resources in Modules.

The plugin includes several tags for rendering links to resources that are declared in modules. The resource tag is the primary tag used for this purpose and the uri for a resource can be specified as a combination of attributes, such as directory and file, or a relative uri. The img tag is a specialized tag for rendering links to image files which can be used for images which are declared in modules.

Tag Library to Render Links to Resources Not Declared in Modules.

The Resources plugin also provides tags which can be used to render links to resources which are not declared in a module. This function assumes greater importance since Grails 1.4, as the `g:resource`, `g:external`, and `g:img` tags now call the corresponding tags in the Resources plugin.

Resources which haven't been declared in a module fall into two classes, those which **are** served by the application and those which refer to a resource **not** served by the application. The former have a context relative URI whereas the latter have an absolute URI, which the plugin defines as a URI that contains the characters `//` .

The primary tag to render links to either type of resource is the external tag. This tag can take a URI (relative or absolute) or a combination of other attributes to specify the resource, such as directory and file. The img tag may be used with similar arguments to render a link to an image file.

Service of Resources to Clients.

A final major function of the Resources plugin is to serve content which is not provided by the dynamic controllers, which serve URIs of the form /controller/action/id. The plugin provides two methods of resolving URIs to serve these non-dynamic, or static, resources.

The primary method is by resolving URIs within a specific static URI subspace, normally those URIs under `static/`. When such a URI is encountered, the plugin first attempts to determine if the URI corresponds to a resource declared in a module. If it finds one, it serves that. If it cannot find such a resource, it will next examine the file system under webapp and if a file with corresponding path and name is found, it will serve that. In the process of doing so, it will create a copy of the file and process it in the normal manner for a resource of its type, assigning it to a virtual 'adhoc' module.

The second method of serving resources is designed to serve resources requested with legacy URIs. These are URIs which are not under the static URI subspace or the form /controller/action/id served by the dynamic controllers. They typically would be generated by a mechanism other than the Resources plugin tags, for example, by older Grails applications. These are referred to as 'ad-hoc' or 'legacy' resources. In order to recognize these URIs, a pattern matching them must be configured, and by default the patterns `/images/*`, `*.css`, and `*.js` will be processed in this way. When such resources are requested, the client is redirected to a processed copy of the resource which is created in the `/static/` URI subspace. This method is not recommended for general use, but only for accomodating legacy URIs.

Use With Security

When the resources plugin is used to serve resources, it is important to understand that resources in the static URI subspace are served without application of security rules by SpringSecurity. This implies that any resources served from the static URI subspace are provided with no security rules applied.

Importantly, this implies that any files accessible in the web-app base directory will be accessible under a corresponding path under <ctx>/static, so be careful which files are available under the web-app base directory.

Performance

While the Resources plugin contains a number of optimizations for improving server performance, such as minifying and combining css files, it will not likely perform well at serving large numbers of files or very large files, such as movies. This is because each resource which can be served or processed as an ad-hoc resource creates entries in memory, and very large numbers of such entries could lead to memory exhaustion. Additionally, resources are served to clients through a number of stream buffers, which will limit performance for very large files. In such cases it is likely better to provide a controller with appropriate methods to serve the resources at controller/action style URIs.

3 Declaring resources

You can declare your resources in your Application's Config.groovy or in a Resources artefact in your application or plugins.

This is done using a simple DSL in both cases. You define the names of the modules you have and the resources within them, and dependencies between modules.

3.1 The resource DSL

There are few methods to call in the DSL. At the top level method calls taking a single Closure argument are translated into module names. The code in the nested closure represents the module definition.

A module definition can call dependsOn, defaultBundle and resource methods:

```
modules = {
    core {
        dependsOn 'jquery, utils'
        defaultBundle 'ui'

resource url:'/js/core.js', disposition: 'head'
        resource url:'/js/ui.js'
        resource url:'/css/main.css',
        resource url:'/css/branding.css'
        resource url:'/css/print.css', attrs:[media:'print']
    }

utils {
        dependsOn 'jquery'

resource url:'/js/utils.js'
    }

forms {
        dependsOn 'core,utils'
        defaultBundle 'ui'

resource url:'/css/forms.css'
        resource url:'/js/forms.js'
    }
}
```

The above DSL defines three resource modules; 'core', 'utils' and 'forms'. The resources in these modules will be automatically bundled out of the box according to the module name, resulting in fewer files. You can override this with **bundle:'someOtherName'** on each resource, or call defaultBundle on the module

3.1.1 The dependsOn method

To declare that a module depends on another, you use dependsOn. It accepts a string or list of names. The string can be comma-delimited to specify multiple dependencies. You can also call dependsOn as many times as you like:

```
modules = {
    core {
        dependsOn 'jquery, utils'
        dependsOn 'other'
        dependsOn(['heavy', 'metal'])
    }
}
```

3.1.2 The resource method

Declaring a resource is done using the "resource" method. This takes either a single string URI or a map of arguments that define the URL of the resource and any other specific options you require.

The single-string argument variant is a simple shortcut for resources that are safe to use all defaults:

```
modules = {
    ui {
        resource '/css/forms.css'
        resource '/js/forms.js'
    }
}
```

The more powerful map argument form accepts the following arguments:

- url - Required. The app-relative URL of the resource as a String, or a g:resource-style map of dir, file and plugin attributes.
- exclude - Optional. A comma-delimited String or a List of names of mappers/operations to exclude. The special "*" value excludes all mappers and operations - the resource will be passed-through as is. You can exclude specific operations such as "minify" if the resource has already been minified.
- bundle - Optional. The name of the bundle to put the resource into. See "Bundling" for more details
- disposition - Optional. The disposition of the resource. If not specified, it will default to a value appropriate for the type of the resource. For JavaScript this default is "defer". To force code into the <head>, set it to "head".
- attrs - Optional. Map of attribute names and values to pass through to the linking tag when the resource is rendered. e.g. use to pass attrs:media:'print' for print-only CSS, or "type". Note that supplying "attrs" will prevent bundling of the resource. Passing in "type" (which is passed through to external tag) allows you to tell the framework that the resource is a specific type irrespective of the file extension. For example if you have a file that ends ".less" but you want the framework to treat it like CSS when rendering links, you need to set attrs:[type:"css"].
- id - Optional. Module-unique id of the resource to be used when overriding resource properties. Defaults to the url of the resource if none supplied.
- linkOverride - Optional. The URL to use when rendering links for the resource, instead of the processed resource's URL. Allows your external resources to participate in dependency management, and some niche CDN requirements.
- wrapper - Optional. A Closure that will be used to render the resource link. The correct markup for linking will be passed in. Used primarily for MSIE workarounds i.e: wrapper: { s -> "<!--if lt IE 8>$s<![endif]-->" }. Note that supplying a wrapper will prevent bundling of that resource.
- plugin - Optional. If applicable, the name of the plugin a resource is being defined in.

Examples:

```
modules = {
    core {
        resource url:[dir:'css/blueprint',file:'screen.css'],
attrs:[media:'screen, projection'], bundle:'core-ui'
        resource url:[dir:'css/blueprint',file:'ie.css'],
attrs:[media:'screen, projection'],
            wrapper: { s -> "<!--[if lt IE 8]>$s<![endif]-->" }

resource id:'main-js', url:'js/coreutils-min.js', disposition: 'head',
exclude:'minify'

resource url:'js/lib.js', linkOverride:'http://mycdn.com/js/lib.js'
    }

ui {
        resource url:'/css/forms.css', bundle:'core-ui'
        resource url:'/js/forms.js'
    }
}
```

```
modules = {
    'font-awesome' {
        resource url: [plugin: 'font-awesome-resources', dir: 'css', file:
'font-awesome.css']
    }
}
```

3.1.3 The defaultBundle method

By default, modules with more than one resource will be auto-bundled using the module name as a bundle name. You can override this by setting the default bundle name for the module by calling defaultBundle. This name can be the same as bundles used in other modules, to bundle resoures across module boundaries.

The method takes a String or boolean as parameter. A string sets the default bundle name, and passing in a boolean false will turn off all default bundling for that module.

```
modules = {
    core {
        defaultBundle 'core-ui'

resource url:[dir:'css/blueprint',file:'screen.css'], attrs:[media:'screen,
projection']
        resource url:[dir:'css/blueprint',file:'ie.css'],
attrs:[media:'screen, projection'],
            wrapper: { s -> "<!--[if lt IE 8]>$s<![endif]-->" }

resource id:'main-js', url:'js/coreutils-min.js', disposition: 'head',
exclude:'minify'

resource url:'js/lib.js', linkOverride:'http://mycdn.com/js/lib.js'
    }

ui {
        defaultBundle 'core-ui'

resource url:'/css/forms.css', bundle:'theme'
        resource url:'/js/forms.js'
    }
}
```

Here you can see that the fallback bundle for both modules is set to "core-ui" and individual resources can still override this.

3.2 Resource artefacts

The best place to put these declarations is in a resource artefact. These have a filename ending in Resources.groovy and live in grails-app/conf.

Note that these are Groovy ConfigSlurper scripts that are therefore environment-aware, so you can declare different resources for e.g. dev, test and production.

Example grails-app/conf/MyAppResources.groovy:

```
modules = {
    core {
        dependsOn 'jquery, utils'
        defaultBundle 'ui'

resource url:'/js/core.js', disposition: 'head'
        resource url:'/js/ui.js'
    }

utils {
        dependsOn 'jquery'

resource url:'/js/utils.js'
    }

forms {
        dependsOn 'core,utils'
        defaultBundle 'ui'

resource url:'/css/forms.css'
        resource url:'/js/forms.js'
    }
}
```

3.3 Config.groovy

Applications can also define resources in Config.groovy if they wish. Simply assign the DSL to the **grails.resources.modules** property.

Example in grails-app/conf/Config.groovy:

```
grails.resources.modules = {
    core {
        dependsOn 'jquery, utils'
        defaultBundle 'ui'

resource url:'/js/core.js', disposition: 'head'
        resource url:'/js/ui.js'
    }

utils {
        dependsOn 'jquery'

resource url:'/js/utils.js'
    }

forms {
        dependsOn 'core,utils'
        defaultBundle 'ui'

resource url:'/css/forms.css'
        resource url:'/js/forms.js'
    }
}
```

3.4 Bundling

Bundling files is the process of concatenating multiple files of the same type into one, so that there are fewer requests made to the server. This can significantly improve page load times even when using ETag or Last-Modified caching schemes.

In a typical application you might bundle most or all of your CSS into one file, and all your common JS into one file, with per-page JS files for page-specific features.

This works very well in combination with the cached-resources plugin an mapper which hashes your files. It means that your clients will cache the (larger) bundled files "forever" but you needn't worry if your next deployment changes one file within a bundle - the bundle will be rebuilt with a new hash and downloaded to the client, as the URL for the resource will have changed.

There is of course a trade-off here, in that a change to one JS file in a bundle will trigger a download of the entire JS bundle for all clients.

It's important to understand that bundling is automatically performed per-module, but you can set it to bundle resources across modules - even those you have not defined yourself. For example using the jquery-ui plugin, you can force it to bundle all the jQuery and jQuery UI JS code into one file, mixed in along with your own code - all in the correct dependency ordering.

The framework will never bundle files that have different dispositions together - this makes no sense. Bundles are therefore automatically named to include the disposition (you may not see this if another mapper such as cached-resources is renaming the files).

Bundling can be controlled in a number of ways. The logic runs as follows:

1. If a resource declaration has an explicit bundle property set, this is used.
2. If there is no explicit bundle property set on the resource, it will use the default for the module
3. The default for the module is "bundle_" plus the name of the module unless defaultBundle has been called
4. If defaultBundle has been called in the module with a string, that name is used for the resource's bundle
5. If defaultBundle has been called in the module with a boolean false, no bundling will occur on the resource, only resources with a bundle specified will be bundled.

NOTE If a resource declares "attrs" or "wrapper" it will not be bundled at all. This is because bundling does not make sense in this situation unless all the attrs and wrapper logic match.

4 Using resources

Now that you know how to declare resources, you need to use them in your page.

There are several tags for this purpose, but the primary means is to use the require tag to indicate which modules you need, and the layoutResources tag to perform the rendering of the resources.

The require tag causes the framework to look up all the resources required to satisfy your module dependencies. However nothing is rendered at that point.

The layoutResources tag is called to render the resources themselves (and internally it calls into external tag for each resource). This tag has special behaviour, in that the first time you call it, it automatically renders only resources with disposition "head". The second time you call it, it automatically renders only resources with disposition "defer".

4.1 Linking to CSS, JavaScript etc.

So you need to add two calls to layoutResources tag to your GSP page or sitemesh layout. Normally you will place it in your sitemesh layout:

Your grails-app/views/layouts/main.gsp:

```
<html>
   <head>
      <g:layoutTitle/>
      <r:layoutResources/>
   </head>
   <body>
      <g:layoutBody/>
      <r:layoutResources/>
   </body>
</html>
```

You can of course include any common modules you require in your sitemesh layout using <r:require> but they must appear before the first <r:layoutResources/>.

In your GSP pages you invoke the require tag as many times as required, even inside GSP templates that you include with g:render, and you can include resources conditionally - something that can be really hard to do without resource dependency management.

Example GSP page:

```
<html>
   <head>
      <meta name="layout" content="main"/>
      <r:require modules="jquery-ui, blueprint"/>
      <g:if test="${customerBranding}">
          <r:require module="theme_${customer.theme}"/>
      </g:if>
   </head>
   <body>
      <div>
           Hello World
      </div>
   </body>
</html>
```

4.2 Linking to images

When you need to render an <img> tag that you wish to be subject to the Resources processing chain (e.g. to make it eternally cacheable) you should use the <r:img> tag:

```
<r:img uri="images/logo.png" width="100" height="50"/>
<r:img dir="images" file="logo.png" width="100" height="50"/>
```

Usually this will produce a link to an undeclared resource. However you can declare images in modules and specify extra attributes in "attrs" to supply e.g. the width and height:

```
modules = {
    images {
        resource url:'images/logo.png', attrs:[width:100, height:50, alt:'Our
logo'], disposition:'inline'
        resource url:'images/icon/add.png', attrs:[width:32, height:32,
alt:'Add'], disposition:'inline'
    }
}
```

Once you have done this, using <r:img> to reference them would automatically set the width, height and other attributes.

The disposition "inline" is optional - as long as you don't have any other modules that "dependOn" the images module and link using r:resource or r:img, you won't need this.

4.3 Linking to resources explicitly, bypassing modules

Sometimes you may need to link to a specific resource, or produce a URL pointing to the specific resource without rendering links to all the modules it depends on - or outside of a context where you can call <r:layoutResources/>. You may even wish to link to an undeclared resource, but still want it to be subject to processing on the fly.

To link to CSS or other resources that are not declared in a module you use <r:external>:

```
<r:external uri="js/custom.js"/>
<script type="text/javascript">
    var urlOfCSSToLoadInJSCode = '${r.external(uri:"css/custom.css"
).encodeAsJavaScript()}';
</script>
<r:external uri="icons/favicon.ico"/>
```

This would output the <script> and <link> tags required to include those resources, without rendering their dependencies first - whether the resource is declared in a module or not. The example shows how you might pass the URL of a resource to some JavaScript code for use later at runtime.

4.4 Including pieces of JavaScript code generated at runtime

Often in an application, especially those using custom Grails tags and rich UIs, you will need to render fragments of JavaScript code while the page is being rendered. It is not always possible to know what JS code there will be in the page until the render process is finished.

The script tag allows you to specify sections of JavaScript text during page rendering, but if using Sitemesh layouts, you will be able to have these fragments appear either in <head> or deferred to the end of the page, just like other JavaScript resources.

This integrates with the disposition mechanism, allowing you to throw your JavaScript into a specific location:

```
<r:script>
    window.alert('This is the end of the page!');
</r:script>

<r:script disposition='head'>
    window.alert('This is the head of the page!');
</r:script>
```

What happens is the <r:script> tag stashes your fragment in the request attributes until <r:layoutResources/> is called - when it is pulled out of the attributes and rendered according to the current disposition being rendered.

This is ideal for other custom Grails GSP taglibs to use to write out their JS code used to e.g. set up UI elements:

```
class MyCustomTagLib {
    def datePicker = {
        out << r.script(disposition:'head') {
            out << '$("#'+attrs.id.encodeAsJavaScript()+').datePicker();'
        }
    }
}
```

The order of code is preserved, and all such fragments are rendered **after** all the modules required by the page (in that disposition) have been included.

5 Overriding resources

When plugins provide resource module declarations, they may not choose the same resource attributes that you would like to use - in particular plugins will often auto-bundle resources by module name, or have dependencies that you'd like to tweak.

This is easy to achieve with the Resources framework, as your application can override the defaultBundle, dependsOn and resources of any module using the "overrides" clause:

```
modules = {
    overrides {
        jquery {
            // We want jquery bundled in with our other code
            defaultBundle 'monolith'
        }

otherModuleToTweak {
            dependsOn 'something-else-we-added'

resource id:'main-css', bundle: 'my-bundle'
        }
    }
}
```

All you do is re-define the module inside your "overrides" clause (yes this mean no module can be called "overrides"), and provide new values for defaultBundle, dependsOn and individual resource attributes.

To override the attributes of resources, the original resource needs to have an id declared on it, or you can use the URI of the original resource if no id was provided.

Arguments passed to the resource call in "overrides" are merged into those for the original resource, before being processed. Therefor you can change or clear any values previously declared.

6 Creating custom mappers

The resource processing chain uses ResourceMapper artefacts to do the work.

You can create your own ResourceMapper artefacts inside your application or plugins to perform tasks such as;

- Moving and renaming resources
- Changing the contents of resource files (compress, minify, compile etc)
- Set HTTP response headers when requests are processed

Resource mappers only get the chance to map (process) each resource once - the final resource is then stored on disk to serve future requests.

6.1 Defining a mapper

Defining a resource mapper is easy. You create a file with ResourceMapper.groovy as the file suffix, in the grails-app/resourceMappers folder.

The name of the mapper is extracted from the filename like any other Grails artefact, for example "TestResourceMapper" yields a mapper with name "test".

Example grails-app/resourceMappers/TestResourceMapper.groovy:

```
import org.grails.plugin.resource.mapper.MapperPhase

class TestResourceMapper {

def phase = MapperPhase.MUTATION

def map(resource, config) {
        def file = new File(resource.processedFile.parentFile,
"_${resource.processedFile.name}")
        assert resource.processedFile.renameTo(file)
        resource.processedFile = file
        resource.updateActualUrlFromProcessedFile()
    }

}
```

The only method a mapper must implement is **map(resource, config)** which is passed the ResourceMeta object that represents the resource and the mapper-specific config variables pulled out of `grails.resources.mappers.<mappername>`. NOTE: The name of the mapper is always in all lower case.

This method can do whatever it needs to the resource's file, provided it calls the updateActualUrlFromProcessedFile() method if the resource moves, unless you patch ResourceMeta.actualUrl manually.

You can change other properties of the resource, such as change the content type of the resource, add or modify tagAttributes (which are passed through when rendering the link for the resource).

That's all you need to do to create a mapper. The best way to learn how they work is to study the source of Cached-Resources and Zipped-Resources plugins.

6.2 Mapper phases and priority

The example mapper shows the "phase" property being set to MapperPhase.MUTATION. The MapperPhase enumeration provides the possible mapping phases in the order in which they occur during processing of resources:

```
enum MapperPhase {
    GENERATION, // create new assets = compile less files
    MUTATION, // alter/improve assets (may mean creating new/deleting
aggregated resources) = spriting
    COMPRESSION, // reducing the file size but maintaining semantics = minify
    LINKNORMALISATION, // convert all inter asset references into a normal
form = css links
    AGGREGATION, // combining mutiple assets into one = bundling
    RENAMING, // moving of physical assets = hashing
    LINKREALISATION, // convert normalised inter asset references into real
form = css links
    ALTERNATEREPRESENTATION, // attach different representations of the asset
= gzipping
    DISTRIBUTION, // moving assets to their hosting environment = s3, cdn
    ABSOLUTISATION, // update inter asset references to their distributed
equivalent = css links
    NOTIFICATION // let the world know about the new resources = cache
invalidation
}
```

In most cases it will be enough to specify your phase and operation (see next section). However in some cases there may be issues where multiple mappers in the same phase must operate in a specific order. In those cases a "priority" property can be set to specify the priority integer.

Mappers will be executed in phase order, in order of ascending priority within each phase. The default priority if not specified is equivalent to zero.

6.3 Operation

The optional "operation" property allows you to specify the name of the kind of work performed by the mapper. Users can then prevent any mappers of this operation from executing on their resources.

The common example is to specify **exclude:"minify"** in a resource declaration to prevent any kind of minifying mapper from being applied to a resource that is already minified.

A similar operation called e.g. "compress" could be used to prevent duplicate zipping of resources that may have been pre-compressed (such as images).

```
import org.grails.plugin.resource.mapper.MapperPhase

class TestResourceMapper {

def phase = MapperPhase.COMPRESS

def operation = "compression"

def map(resource, config) {
        // Zip the file here
    }

}
```

Note the operations and mapper names occupy the same namespace so that the "exclude" argument on resource declarations can apply to either.

Resources will fail fast at runtime if an operation is specified on any mapper, where there is also a mapper with a name the same as the operation.

6.4 Processing only the right types of files

Often a mapper is only meant to target certain file types or file patterns. Make it easier for your users by operating correctly out of the box by specifying defaultExcludes and/or defaultIncludes for your mapper, which will filter the resources passed to your mapper:

```
import org.grails.plugin.resource.mapper.MapperPhase

class TestResourceMapper {

def phase = MapperPhase.MUTATION

static defaultExcludes = [
        '**/*.png',
        '**/*.gif',
        '**/*.jpg',
        '**/*.jpeg',
        '**/*.gz',
        '**/*.zip'
    ]
    static defaultIncludes = [ '/images/**' ]

def map(resource, config) {
        …
    }
```

These defaults can be overidden by the user with the **grails.resources.<mappername>.includes** and **grails.resources.<mappername>.excludes** Config variables.

6.5 Adding response headers and intercepting requests

Resource mappers also have the opportunity to take part in response handling so that they can adjust the response headers if necessary. You may need to adapt the handling of the current request to the request headers supplied.

Note that this cannot be used to change how the resource is mapped - the mapping is performed once only, but the response headers can be customized every time the file is requested.

As an example, the Zipped-Resources plugin uses this mechanism to set the Content-Encoding header:

```
import org.grails.plugin.resource.mapper.MapperPhase

class ZipResourceMapper {

static phase = MapperPhase.ALTERNATEREPRESENTATION

/**
     * Rename the file to a hash of it's contents, and set caching headers
     */
    def map(resource, config) {
        // Do the zipping
        ...

// Set up response headers
        resource.requestProcessors << { req, resp ->
            resp.setHeader('Content-Encoding', 'gzip')
        }
    }
}
```

Here the mapper adds a Closure that takes the request and response objects to the resource's requestProcessors list. These will be called in the order they are defined on the resource.

7 Writing plugins that use Resources

There are generally two kinds of plugins that provide Resources artefacts; those that expose resources such as JavaScript libraries, and those that supply custom mappers e.g. to minify resources.

Plugin dependencies

If your plugin exposes resources, it does not need to depend on the "resources" plugin. In fact if it is intended to interoperate with applications that do **not** use the Resources framework, you should not depend on the "resources" plugin. Defining XXXXResources.groovy files does not require dependency on the plugin at all.

If you are exposing a mapper, you will need to have a dependency on the "resources" plugin so that you can reference the MapperPhase enum.

Versioning

When exposing resources, it is becoming a convention to ensure that your plugin has a version number that matches the version of the libraries you are exposing - with point releases or other suffixes to denote interim plugin releases where the library is the same but the plugin has changed.

This is important because it allows other applications that use the library you expose to specify which version of it they require, without having to mentally map the library version to yet another plugin version.

Avoiding bloat

It's important to remember that the dependencies introduced by a mapper plugin end up in the user's WAR deployment. Therefore mapper plugins should have minimal dependencies, and where possible focus on a specific niche of mapping - for example avoid creating something like a monolithic "minified-resources" plugin that bundles every known minifier library together.

Naming conventions for mapper plugins

It is recommended that developers establish sound naming conventions in the community such that it is easy for users to tell what a mapper plugin will do for them, in a way consistent with other mapper plugins of similar type.

For example there may be YUI and a Google CSS minifier plugins. These should be clearly named e.g:

- yui-css-minified-resources
- google-css-minified-resources

In addition the mappers must be clearly named, i.e. YuiCssMinifyResourceMapper and GoogleCssMinifyResourceMapper. Such that the user can exclude them selectively with code such as **exclude:'yuicssminify'**.

8 Debugging

Due to the processing of resources, it can be tricky to diagnose problems when something is not working - your styling may not be coming through correctly, or you are getting weird JS errors.

Usually these are due to incorrect dependencies in your modules, but you may also have actual bugs in your JS or CSS.

When you're using mappers that completely rename or aggregate files, it can be hard to tell where the problem lies.

There are a number of features in Resources that make it easier to debug such issues.

First: Get familiar with your browser's resource inspector. All modern browsers have a developer mode which lets you inspect the resources loaded by a page - including the request and response headers and the content itself. This is indispensable. For Safari you need to explicitly turn this on. For Firefox you may need to install firebug.

The custom response header

Every resource served by the plugin in development mode adds the **X-Grails-Resources-Original-Src** header to all resources served.

The value shows you the original filename of the resource. In the case of bundled resources, the first part is the bundle name, and then follow all the names of the resources appended to the file, in the order they appear.

The quick debug option - turn off processing for the current request

Any URL in your application can have the query parameter **_debugResources=y** added to it, and the request will perform no processing. So for example if you are browsing **http://localhost:8080/myapp/admin** and need to bypass resources, just change the URL in your browser to **http://localhost:8080/myapp/admin?_debugResources=y**

Dependency management will still be in effect (otherwise your app would break) but the resources will not be bundled or have other mappers applied. Resources will be served with a timestamp - this allows you to set breakpoints on resources and yet also force a refresh in your browser if you find that your browser still caches resources even after you have edited them and no caching headers are set. If this occurs, you can force a timestamp refresh by adding **_refreshResources=y** to the query params.

The nuclear debug option - turn on resource debugging all the time

You can turn off resource processing completely with the trivial Config variable, so that it is as if every request you make has _debugResources=y appended to it:

```
grails.resources.debug = true
```

You can specify this value per-environment in your Config.

Inspecting your dependency data

A utility function in the grailsResourceProcessor bean can be used to output the current resource metadata that the plugin has - all the modules and resources and their dependencies.

Simply add a "println grailsResourceProcessor.dumpResources()" somewhere in your application or GSP and it will come out.

Turning on debug logging

There are copious amounts of debug logging output by the plugin that should make it relatively easy to track down problems. This information includes the order of mapper execution that will be used, the dependency order of your resource module definitions, and all the steps of processing each resource.

To enable debug logging add this to your Config logging section:

```
debug "org.grails.plugin.resource"
```

9 Configuration

There are various configuration options to control processing of your resources.

All of these config variables can of course be set per-environment.

Setting the location of processed static resources

By default the plugin writes processed resources to a temporary directory.

You can change this to a directory that exists in a reliable location and hence can be used to server resources via e.g. Apache, by assigning a full filesystem path value to **grails.resources.work.dir**

Change the /static/ URI prefix: grails.resources.uri.prefix

By default the plugin serves the modified static resources from URIs beginning with <appcontext>/static/.

You can change this by assigning a value to **grails.resources.uri.prefix**

Debug mode: grails.resources.debug

Setting **grails.resources.debug=true** will force debug mode all the time, as if you added _debugResources=y to every request.

Dependency-only mode: grails.resources.processing.enabled

You can turn off resource processing completely with the trivial Config variable:

```
grails.resources.processing.enabled = false
```

Like the _debugResources option, this will perform no processing at all. It will also not require a writable file system on the server, but obviously precludes any processing of resources.

You do however retain the dependency management abilities.

Disabling specific resource mappers in different environments

You can disable any mapper using the standard environment-specific configuration of Grails:

```
grails.resources.mappers.bundle.enabled = false
grails.resources.mappers.hashandcache.enabled = false
```

Controlling the scope of the adhoc filter: grails.resources.adhoc.patterns

The ad-hoc resource filter is mapped using Servlet SDK filter mappings, which are more restricted that Ant patterns. You can specify folder (xxx/**) or file type (**.xxx) mappings only.

You may not want the adhoc filter, which is only used for legacy resources (which are not linked to using resource tags), to intercept everything in your application.

The default value is:

```
grails.resources.adhoc.patterns = ["/images/*", "*.css", "*.js"]
```

Controlling the includes and excludes of the adhoc filter: grails.resources.adhoc.includes/excludes

While the grails.resources.adhoc.patterns setting gives you coarse control over which legacy URIs are intercepted, you can get full Ant-style include/exclude patterns using the **grails.resources.adhoc.includes** and **grails.resources.adhoc.excludes** variables.

They both accept a list of Ant-style patterns.

Including specific resource patterns per-mapper: grails.resources.<mappername>.includes

You can control the list of matching resources on a per-mapper basis. Mappers provide sensible defaults but you may have new content types (for example a new CSS variant such as LESS) that were not known at the time the mapper was written.

So for example to add .less files to processing by the CSS rewriters (necessary for correct behaviour with bundle mapper):

```
grails.resources.mappers.cssrewriter.includes = ['**/*.css', '**/*.less']
grails.resources.mappers.csspreprocessor.includes = ['**/*.css', '**/*.less']
```

You can also use this to ensure that only resources under a specific folder are processed.

Note that the value you provide **replaces** the default list supplied by the mapper.

Excluding specific resource patterns per-mapper:grails.resources.<mappername>.excludes

There may be times when a mapper is processing too many files - those that may have already been processed or that may become damaged by the mapping (i.e. incompatibilities due to relative code loading by JS code). Simply add the resource names or patterns to that mapper's excludes:

```
grails.resources.mappers.cssrewriter.excludes = ['unsafe/**']
grails.resources.mappers.csspreprocessor.excludes = ['unsafe/**']
grails.resources.mappers.bundle.excludes = ['unsafe/**/*.css']
```

Note that the value you provide **replaces** the default list supplied by the mapper.

Excluding specific mapper all together

You can exclude any (also custom) mapper from processing all together by using grails.resources.mappers.<mappername>.enabled = true | false.

For example:

```
grails.resources.mappers.cssrewriter.enabled = false
```

Disabling CSS rewriting: grails.resources.rewrite.css

It is possible to turn off all CSS rewriting by setting this value to false. This is not recommended as it will usually break bundling unless all your links are relative and you don't have any mappers that move your resources relative to your CSS.

10 Security

The resources framework is now installed by default for all Grails applications. When security is in place to protect resources provided by the web application, there are several important points to bear in mind.

Resources served under <ctx>/static are not secureable.

Due to the manner in which the Resources plugin processes resources, the SpringSecurity plugin doesn't apply any rules to them. Thus access to resources under the reserved URL subspace, normally, `/static/`, cannot be restricted with security rules.

Files in the web-app base directory are always accessible under /static.

Again due to the type of processing performed by the Resources plugin, any files under the web-app base directory (with the exception of those under WEB-INF) will be accessible under corresponding paths under `/static`. For example, `/images/img1.jpg` would be accessible under `/static/images/img1.jpg`. This is true whether the `/images` directory is considered an ad-hoc pattern or not.