Unified file resource loading cascade

Rationale

Currently we have implemented the new logic of resource loading which work both for the regular resources and for the configuration files (Yaml). The files are served from the following origins in the order of priority:

- Resources installed into 'resources' workspace
- Files from <magnolia.resources.dir>/....
- Files from classpath

It was agreed that by default at least the configuration file paths must match the pattern described here: Conventions for config/templating in filesystem.

The same logic has been implemented separately for the serving (web) resources, for configuration, and in the past already for Groovy scripts, template loaders. We want to provide a common solution that could be re-used by everything.

A workaround/PoC has been implemented in the resources module, but needs to be handled by ResourcePath/Origin APIs. See MAGNOLIA-6106 - Getting issue details... STATUS for status.

New API & clients

With MAGNOLIA-6106 - Getting issue details... STATUS we introduced a new API to find, traverse and load resources. The main entry point is via the `Origins` factory. See `info.magnolia.config.source.ConfigurationSourceFactory` for an example usage.

Configuration

The configuration mechanism introduced for 5.4 uses the new API.

Resources app

Mika has been working on an app (based off of a first draft by Sasha) which lets users browse origins. Currently on the `feature/resources-app` branch.

Note: https://vaadin.com/web/matti/blog/-/blogs/connecting-large-amounts-of-data-to-ui - might be useful in actual implementation of the container (the LazyList concept).

See MGNLRES-140 - Getting issue details... STATUS .

Web resources

Web resources are currently served via a number of mechanisms:

1. ClasspathSpoolServlet, serves `/.resources` from the classpath (during install/startup as well in `FilterManagerImpl`)
2. Rendering mechanism servers `/resources` via a combination of UriToRepositoryMapping (that's how `/resources` is mapped and the AggregationState is populated with the corresponding node), a "fake" template definition, and renderers
3. `/docroot` and similar folders are served by the container, because our filter chain (or parts of it) bypasses them.

We aim to drop support for #1 (i.e the new implementation should transparently replace it). We also aim to deprecate #2.

See MGNLRES-144 - Getting issue details... STATUS

Processed resources

This is one of the points that may make replacing the current resource rendering a little less obvious than anticipated. Unless there are strong arguments against it, we'd like to deprecated support for "processed resources" for the following reasons.

- we can only think of 2 use-cases:
  - aggregation (render a folder of resources a single one, i.e include subnodes into a single large css)
  - simple variable substitution (i.e to avoid repeating color codes)
- other use-cases are better served by appropriate tools (e.g SASS, minification, ... which could easily be implemented as a filter)
- we could implement aggregation at the servlet level if needed (if request is for a folder, traverse and aggregate content in the response)
- While the current mechanism allows support for model classes, I can't think of a reasonable use-case, especially provided that there is no content to work with, AFAIK (i.e just the resource node itself, so the rendering model can't really know where it's attached anyway)

Can you think of use-cases for processed resources than don't fall in these 2 categories ?
- aggregation (request for a "folder" and aggregate all resources under it in one large css response)
- simple variable substitution (e.g to avoid repeating color codes)

After discussions, some compelling points to keep "processed resources" in one form or another:

- It is (sometimes ? by some people ?) preferable to keep the "decision" about further processing close to the resource itself
- Sass, less, and other post-process tools
  - GJ: if the resource is called `foobar.scss`, what's a use-case for serving it non-processed ?
  - At the same time, the request *should* be for `foobar.css`, so how do we "redirect" to the `foobar.scss` resource ?
- Minification
  - The file's extension could be used to drive a minification filter, either:
    - The request is done to `foobar.min.js`, but the resource is `foobar.js` - we have a "redirect" mechanism, and the filter minifies the response
    - The request and the resource is `foobar.min.js` - no need for a "redirect" mechanism, but it'll be odd for developers to edit a file called `foobar.min.js` which is *not* minified
- Site awareness (one use-case for model classes)
  - A resource serves different content based on which site it's served for, or based on some user property. (e.g the css contains something like `background-color: ${model.branding.color}`).
  - I don't have a good substitute for this

The "redirect" mechanism mentioned above could be similar to what we did for personalization, and to what we should do with AggregationState now that we have IoC: have a tiny request-bound object that keeps track of the changes. (minify filter sees a request for `foo.min.js`, "changes" the requested resource to `foo.js`. The resource-serving-filter serves `foo.js` based on that. The response comes back down the filter change, gets minified by the minify filter)

An idea to "revive" this feature with the new Origins API:

- An origin-dependent mechanism is introduced to retrieve a `Metadata` object for a given `ResourcePath`
- This metadata could be a simple key/value map of strings
- For JCR Origin, these would be jcr properties, quite obviously
- For filesystem (and classpath), we could load them off another file, e.g `theresource.css.meta` (where *.meta would be ignored by this origin for everything else). This file could be a simple yaml or even java.properties file.
- The app could display this metadata
- How would the app edit the metadata? The same it edits resources, I guess, i.e *just* on JCR if i'm not mistaken.
- That said, the above is *just* an idea for the case where we *really* need to re-introduced processed resources. I'd rather first explore all (possibly better) other approaches.

Backwards compatibility

If we want the new mechanism to also serve `/resources` , we'll probably want to implement a filter instead of a servlet:

- it could decide whether to serve the resources using the Origins API, or continuing the filter chain to let it "process" the resource. (if node exists and is a "processed resource" then continue chain, otherwise just serve the resource as is the filter was an end point)

Security

See clarification about `pathPattern` below. The servlet should implement some minimal security checks to ensure we don't expose unwanted resources. This could be implemented by wrapping the `Origin` instance to simplify client code.

Template scripts

We currently have a set of custom loaders, wrapped by `info.magnolia.freemarker.FreemarkerConfig#getTemplateLoader`, and default configuration (bootstrapped by `mgnl-bootstrap/core/config.server.rendering.freemarker.xml`) which essentially implement this fallback mechanism for FreeMarker. We could replace this by using the Origins API, but that depends on Core split or we'll have a circular dependency(at the very least it needs the extraction of FreeMarker features out of core, since resource-loader depends on core)

Groovy module

The Groovy module provides a similar class loading mechanism that allows loading Groovy scripts from the repository. Groovy natively supports loading scripts from files or the classloader. `info.magnolia.module.groovy.support.classes.MgnlGroovyClassLoader` could probably be rewritten to use the Origins API.

Imaging module

I believe the imaging module has the capability to serve and process images from the classpath. That support could be extended to use the Origins API.

Clarifications

Web resources

- What happens to resources in `/mgnl-resources`. Especially our own in admin central.
- If we want to replace `ClasspathSpool` with `ResourcesServlet` in `FilterManagerImpl` as well, how do we solve the dependency cycle between core and resource-loader

Cache

- Cached resources have to be by-passed (issue discovered by integration tests)
  - ❓ *insert details here please*
- As for everything else, cache will not be "magically" invalidated when resources are changed. Has to be manually flushed
  - In the future, a notification/event system could help invalidate cache entries

PathPattern, Origin.traverse, observe, and usage of Predicates

Origins currently use a `pathPattern` which was inherited from what we do in configuration. It is used to "filter" which resources can be served etc. I think it has to go:

- It is used inconsistently
- When navigating the resources hierarchies, it is unclear whether the pattern should be used or not. (i.e folders likely won't match the pattern we have for `.yaml` config files, but should still be traversed)

Client code should do their own checks (i.e should I really serve this `.class` through ?), but that could also be implemented by wrapping the `Origin` insta nce to simplify said client code. Platform could provide such a wrapper.

The `traverse` and `observe` methods should also clarify their use of `Predicate`:

- The predicate can be used to determine if we should continue traversing the hierarchy
- The predicate can be used to determine if the `Function` should be applied on a resource
- The `Function` can also check this itself.

The simplest, most self-documenting way to clarify this is, IMO, to have 2 `Predicates` in the method parameters instead of one.

It becomes however debatable - should we also have a `Predicate` in the `listChildren()` method for example ?

Security

- Loading of the node by (Jcr)Origin is done through SystemContext, no user ACLs are applied yet.
    - We don't want to apply ACLs to (just) the JcrResourcePath, since that'd create inconsistencies (some users would see the FileResourcePath, others would see the JcrResourcePath, for the same "input" path)
- We originally thought we'd simply rely on URISecurity for resources, to avoid creating another "fake-workspace-mapping for the sake of ACLs" (which is how UriSecurity was implemented, as far as the security "app" was concerned, in pre-5.0 times)
    - When *serving* the resource, the filter/servlet should check "manually" for the current user's permissions; we can use the ACLs set for the `resources` workspace, regardless of where the resource is actually coming from
    - In Security App, we could add a tab (next to uri/web security) for resources
    - The choose dialog for "resources" ACLs should now be the new resources app and not the jcr browser. (regardless of whether its a separate tab or not)