Ideas for a multi-tenant and multi-module content model

DRAFT DRAFT DRAFT - please consider this "wild ideas" so far - nothing's decided so far and not much is implemented

See SLING-4386 for an initial prototype that uses these ideas. It's not exactly like described below, that issue has the details.

Overview

This is an early 2015 brainstorm for a content model that fosters the following features for Sling-based applications:

- Support multiple tenants by default
- Clearly separate functional modules
- Allow modules to be easily added and removed
- Allow different tenants to use different versions of those modules
- Keep track of which tenants use which modules and versions

A "functional module" (short name "module") is a collection of scripts and servlets used to process HTTP requests.

The Goal

Could we run the whole Web on this thing is an interesting goal to keep in mind.

Not that we are *that* ambitious, and compromising other things to reach that exotic goal wouldn't be right, but why not aim high?

Open Questions

- Should the tenant-specific content move under /tenants? Why not, although in terms of content repository maintenance it might be useful to have all the "big things" in a single subtree.

Tree structure

/content

This is the publishable content, what's meant to appear in HTTP responses.

Resources under /content always belong to a tenant, identified by the name of the resource that's immediately under /content, like for example

- `/content/www.example.com/search` which is the /search resource of the `www.example.com` tenant
- `/content/foo.com/cars/fiat/punto` which belongs to the `foo.com` tenant

We probably need to allow several subtrees to point to the same tenant for example using paths like `/content/foo.com-mobile` as the root of another subtree called "mobile" that also belongs to the `foo`.com tenant.

/tenants

Once a /content resource is accessed, Sling looks here for the attributes of the tenant to which the content belongs.

These attributes tentatively include (T is the current tenant ID):

- `/tenants/T/assemblies` - the list of active assemblies (see below) for this tenant
- `/tenants/T/conf` - configuration overrides for this tenant
- `/tenants/T/state` - tenant-specific state like workflows, jobs etc.
- `/tenants/T/home` - tenant identity, list of user groups that belong to the tenant, etc.

/assemblies

Note that as of February 5th, 2015 this is not used in the SLING-4386 prototype - the tenants point directly to the modules instead of pointing to named assemblies. Easy to add if needed. A good reason might be to make it easy to make changes to a group of related tenants, by just changing properties on a assembly that they share.

Resolving scripts to process requests is only possible via assemblies that define which modules are active for a given tenant.

Assemblies can be named after tenants, for example a "foo.com" assembly might assemble modules for the foo.com website.

Assemblies that are used by multiple tenants can also have symbolic names like "oldstyle.blog".

Assemblies can also be created to partially override scripts and servlets from a parent assembly, "custom.oldstyle.blog" for example could replace a few scripts and servlets from "oldstyle.blog".

The advantage as compared to the current free-form Sling script resolution is that an assembly completely defines which scripts and servlets are used by a given tenant. The script resolver forbids direct references from /content to /modules, it only allows content resources to reference assemblies.

Mapping assemblies to modules is based on module paths, our "oldstyle.blog" for example could define that the /modules/blog/v42 and /modules/comments/v54 are active, which makes the scripts and servlets provided by these modules available to tenants for which the "oldstyle.blog" assembly is active.

/modules

Modules contain the scripts and pointers to servlets which build a consistent set of functionality.

Modules can have dependencies on others, the blog V42 module for example can declare that it requires the sling.post capability with a version >= 4.2. An assembly that points to blog V42 without including the module that provides the required sling.post capability is then considered invalid.

Multiple versions of modules can coexist, but a given assembly can only point to one of them.

Modules are stored under paths like /modules/blog/VVV where VVV is the module's version.

The folder structure of a module is standardized, with subpaths like

- osgi/install for installable OSGi resources (not sure yet how those are made module and version-specific, see open issues below)
- osgi/config for OSGi configurations (same comment)
- content/default for default content, templates that are meant to be copied under /content when used
- content/public for public content like stylesheets, button images etc.
- content/admin for content related to administrative features
- scripts for the module's scripts and pointers to OSGi servlets

The /modules subtree is readonly and immutable on production systems.

The new script resolver requires resource type subpaths to start with the module name. For example, all resource types handled by the "blog" module must start with `blog/`

Module names starting with "`sling`" are reserved.

Benefits

Reasonable modifications to the existing Sling script resolver should allow for tenant-specific script resolution.

The assemblies intermediate layer gives a much better overview on what scripts and servlets are used by a given tenant.

Having each module version in its own subtree makes it easy to add and remove them. Module versions that are not referenced in any assembly can be safely removed.

Standardizing the subtree of a module makes them easier to understand and organize.

Open issues

To enforce access to servlets via assemblies only, mounting a servlet requires creating resources under /assemblies. This is different from the current Sling where just registering a Servlet as an OSGi service mounts it. We might provide a bridge to ease migration of existing applications.

Ideally each assembly should have its own set of OSGi services. How this happens is not defined yet but Amdatu for example has support for multi-tenancy at the OSGi level (http://www.amdatu.org/components/multitenancy.html) and we might also use our own `org.osgi.framework.hooks.service.FindHook` to implement this.