Concept IOC in Magnolia

Your Rating: Results: 120 rates

Implemented in 4.5

Introducing IoC in Magnolia. Implementation tracked in MAGNOLIA-2569@jira

- Rationale
  - Terminology
  - Background information
  - Benefits
  - Library or custom implementation ?
  - Relevant JSRs
- Some requirements
  - Scoped components
- Implementation
  - Containers
  - Main changes
    - Context listener
    - Properties
    - Filters
    - Content2Bean
    - Various
  - Lifecycle
  - Issues raised by this topic
    - Request-scoped components
- Further changes and conversion
  - In progress and TODO
    - Components registration
  - Guidelines for converting existing components

Rationale

Inversion of Control (or Dependency Injection) has been around for a while now, and is recognized and accept as a design principle which helps improving testability (and thus quality), code readability, maintenance, understandability, and countless other advantages.
MAGNOLIA-2569@jira

Terminology

- **IoC**: Inversion of Control. This is the "concept", the "programming model".
- **DI**: Dependency Injection. This is an implementation of the concept. We say that a Magnolia component uses/relies on DI, whereas we say that Magnolia itself uses/implements IoC.

Background information

Mandatory read: Martin Fowler's article! Wikipedia's article is probably also a good read, although maybe a touch too the theoretical.

- http://martinfowler.com/articles/injection.html
- http://en.wikipedia.org/wiki/Inversion_of_control
- Constructor-injection vs Setter-injection http://misko.hevery.com/2009/02/19/constructor-injection-vs-setter-injection/
- http://misko.hevery.com/code-reviewers-guide/
- http://misko.hevery.com/2008/08/01/circular-dependency-in-constructors-and-dependency-injection/
- http://stackoverflow.com/questions/2026016/google-guice-vs-picocontainer-for-dependency-injection

Benefits

- Testability
- Clearer APIs
- Better dependency management (which component uses which others), less singletons
- Less brittle code (singletons = global state)
- Cleaner lifecycle management of components (let the container manage it, instead of having some components arbitrary "start" others)
- More in the comments ! 😁

Library or custom implementation ?

A basic custom implementation would have been possible, but we quickly come in "reinventing the wheel" stuff. Guice and PicoContainer were the main contenders at Magnolia. Guice for its popularity, Pico for its practicality, and the fact that we have a potential bigger influence on the code base.
See IoC Container decision table.

- PicoContainer: http://picocontainer.org. See PicoContainer-specific notes
- Guice: http://code.google.com/p/google-guice/ - looks like we'd need some extensions to fulfill all our needs (lifecycle, jmx, ...), like guiceyfruit or guicebox: http://code.google.com/hosting/search?q=label:guice
- JSR-299 and Weld: http://seamframework.org/Weld
- Tapestry IoC http://tapestry.apache.org/tapestry5/tapestry-ioc/

Relevant JSRs

- JSR 299: Contexts and Dependency Injection for the JavaTM EE platform
- JSR 330: Dependency Injection for Java (`@inject` and other annotations)
  (http://www.adam-bien.com/roller/abien/entry/what_is_the_relation_between)

Some requirements

One of the points that was preventing us from introducing IoC in Magnolia was our observed components, which could not be "updated": until Magnolia 4.3, the singletons were re-instantiated when observation kicked in. Since MAGNOLIA-2553@jira, they are proxied, thus allowing code to keep references to these components provides a patch for this.

Not all of Magnolia needs to be "converted" in one goal, but there's a good chance changes to one component will lead to changes in its dependencies.

Here's a rough list of what we want to be able to achieve:

- Modules interdependency: a module class should be able to depend on another, i.e get the other's instance injected, as long as its module descriptor declares such descriptor. Use case: Forum's dependency on RSSAgg.
- Filters should be constructed via IoC and thus be able to declare dependencies, typically on module classes.
- Content2bean: any class loaded via c2b should ideally be instantiated via the IoC container: currently c2b uses `ClassFactory` instead of `ComponentProvider`, so this might be an issue.
- Replace usage of `SystemProperty` by a `@ConfigurationProperty("myProperty")` annotation on a private field of the component user.

- Scopes: components we currently use as "singletons" will probably be application-scoped. Scoped components are components whose lifecycle is tied to a different lifecycle than the "application", ie for webapps, typically the session and the request. This is usually implemented by nesting the containers, and storing them as webapp-, session- and request- attributes. Since they're (really) lightweight, there's virtually no added performance cost. However we'll need to think how we "mark" the scope of components.
- We still need some level of control on instance creations: for ex, we want to instantiate and IoC' RenderingModels, and let them depend on a module configuration class.
- Taglib: Guice.injectMe() - or something similar 😷
- Event/listener mechanism - nice to have - so components can be "ping"ed when others get reloaded, etc. See Concept - Event mechanism
- AOP ?
- Content2bean : init() - will that work with proxies/interceptor ?

Scoped components

Picocontainer introduced the notion of scoped containers (Guice has this too).

In the case of Magnolia, we might consider more custom tailored scopes (no specific idea yet, just throwing the thought out there)

Implementation

Containers

At startup time, we create a "root" container. This container is meant to contain only the components needed to start Magnolia up. (Classes and dependencies of ModuleManager, Content2Bean, MagnoliaConfigurationProperties - see below)

When then create a "main" container, composed of all registered components. This includes, amongst others, ConfigLoader.

Main changes

Context listener

`info.magnolia.init.MagnoliaServletContextInitializer` (replaces `info.magnolia.cms.servlets.MgnlServletContextInitializer`) is now solely responsible for instantiating the container(s), and starting it.
Certain components (`@AtStartup`) are started "eagerly", when the container is started.

Properties

`MagnoliaConfigurationProperties` is meant to replace `SystemProperty`. It aggregates all `PropertySource`. Each property source is "separate", one can identify where a property comes from.

`MagnoliaPropertiesResolver` is used by `DefaultMagnoliaConfigurationProperties` to determine which `magnolia.properties` files should be added to the other default sources.

`MagnoliaInitPaths` is a simpler wrapper around the 4 basic properties used to resolve the locations of `magnolia.properties` files, amongst others (also used to determine where Magnolia is deployed, in turn used to define paths to logs, temp, etc.)

It is possible to modify the `magnolia.properties` resolving mechanism by replacing the above component in the root container by custom implementations.

Filters

`MgnlMainFilter` is now "assisted" by `FilterManager`. `FilterManager` is a regular components (has dependencies, is retrieved from the IoC container), is holding the filter chain, and determines if the root filter is the configured filter chain, or the install filter chain.

`MgnlMainFilter` is the one filter configured in `web.xml`. It retrieves the `FilterManager` via `Components.getComponent()`, and merely delegates to the root filter provided by `FilterManager`. It is still responsible for push/popping the context for each request.

Content2Bean

Content2Bean now uses `ComponentProvider.newInstance()` when instantiating a bean/subbean. As such, any component configured in the repository can use DI

Various

Introduced a WebContextFactory, to simplify/clarify the way a WebContext is created, init'd, and set. By registering a different WebContextFactory, one can provide a different implementation of `WebContext` (hello WebLogic module), as well as an different implementation of `AggregationState` (hello STK)

Due to the changes regarding properties, introduced a `clear()` method on `SystemProperty`, to allow tests to clean up after themselves. Tests now need to use this method in their tearDown() methods, instead of `SystemProperty.getProperties().clear()`.

Lifecycle

A component's lifecycle (start/stop/dispose) can be managed by the IoC container. With PicoContainer, the lifecycle is typically "lazy": the component is started when first "requested". This makes sense in most cases. An "eager" lifecycle can make sense for "cached" components (i.e singletons), and we need this for certain components, typically the log configuration, module manager, repository provider, etc. See PicoContainer-specific notes for implementation details.

Issues raised by this topic

1. Content2Bean - currently modified the API to work around a circular dependency issue in Content2Bean - see MAGNOLIA-3525@jira.
2. Naming conventions - We currently use a lot of "-Manager" classes. Candidate suffixes for renames: -Service, -Provider)
3. Progressive conversion: maintaining a bastard codebase, where some components would be handled both by the IoC container and regular FactoryUtil calls.
4. scopes vs context (we have a lot of components which would be candidates for app-scope, except at some point or other, somewhere down their dependency hierarchy, they'll use, for instance, MgnlContext.getInstance. And worse, might fallback on the system context, or change their behaviour one way or the other, if no Context is set. These behaviours will somehow have to be wrapped and retrofitted in MgnlContext, see #3

Request-scoped components

At first sight, components like WebContext, AggregationState, and even the ServletRequest object are good candidate for the request-scope. The next almost-natural step would be to think: hey, but then I could make the Renderer dependent on AggregationState instead of relying on a threadlocal context to give it to me. Well, that isn't so easy. This could be solved only in one of 2 ways:

- Make your renderer also request-scoped. That would mean that the RenderingFilter (which depends on RenderingEngine) would also become request-scoped. Which would in turn mean all filters are request-scoped, and the whole chain get re-instantiated on every request (which could be optimized with caching/cloning of course, but still). And this breaks the filter api, since they're meant to "receive" the request in the doFilter() call instead.
- Depend on "provider" type of objects. Those could be app-scoped, and would know "how to" fetch the current AggregationState, for example. In the case of Magnolia, this would probably turn out to be something like (Web)ContextProvider, which would be a simple wrapper around the current context. This is better than nothing, as it could be easily mocked in tests.

If we go for the second approach, it sounds like we're going to end up with a "duplicate" of `MgnlContext` (minus all the static fluff). Should we thus deprecate (parts of) MgnlContext ?

Additionally, a choice would have to be made between:

1. a provider:

```
public interface WebContextProvider {
    WebContext get();
}
```

2. a proxy

```
public class (or intf) WebContextProxy implements WebContext {
    WebContext getDelegate();
    // .. implement all methods of WebContext and delegate to current threadlocal instance
}
```

Further changes and conversion

In progress and TODO

- Install container - the root and main containers currently hold components which are only used and needed during the install process. We could isolate those in a specific container.
- Scoped containers (session and request, perhaps more granular scopes?) - I still need to figure out exactly how picocontainer-web does this. Probably based on ThreadLocal, but haven't found out the crucial bit: where the container is actually instantiated and populated. Probably will need to startup one of the sample projects and use a debugger.
- Update task for MagnoliaServletContextInitializer (warning)
- Improve life cycle usage
- ConfigLoader isn't really needed anymore, merely just start 3/4 other components. (startup order?) - does it really need to happen with doInSystemContext ?
- Move loading of module descriptors to ModuleRegistry - make it lifecycle, ModuleManager should not care
- Modules lifecycle (currently started by MM)
- Module descriptor additions
- Check c2b instantiation for trees, pages, commands, ServletDispatchingFilter.
- Provide better hook for replacing the root container ? Comma-or-newline-separated list of classes, or do we need key=value ? Given the low amount of components, I think it's ok to have to re-declare ALL components, not just those you want to replace (which also allows **removing** some altogether)

Components registration

We're considering enforcing components registration via the module descriptor. Possibilities:

- Additional element in module descriptor: See MAGNOLIA-3517@jira and proposal below.
- We currently still use the `<properties>` of module descriptors. 1) Do we want to keep this for backwards-compatibility ? 2) There are a few "legacy" components that were never registered (not in mgnl-beans.properties, not in a module descriptor), and these were retrievable as singletons, because they were concrete classes. This currently does not work, and registration has become mandatory. Thoughts ?
- Scannotation: we could also scan the classpath for certain annotations to "magically" register components. This would probably increase startup time dramatically, which is not desired. However, in combination with a Maven plugin, this could be interesting: in dev mode (or whatever other optional way of enabling scannotation), we'd scan for components, and in production we'd use what's in the descriptor (i.e the Maven plugin generated a complete module descriptor based on said annotations)

Proposal for components registration in the module descriptor:

```
<component scope="..">info.magnolia.FooBar</component>
<component scope="..">/server/foo/bar</component>
<component scope="..">config:/server/foo/bar</component>
<component key="..">...</component>
<provider key?>...</provider>(~= factory)
<composer>....</composer>
```

- scope: not mandatory - values: tbd.
- key: to replace an existing component, otherwise defaults to value. **Not** specifying a key allows registering multiple implementations of a given interface.
- provider - similar to our current ComponentFactory, more explicit ?
- composer: if there is a need for a more complex component registration - this would then be container-specific

Guidelines for converting existing components

Quick check-list:

- deprecate getInstance() type of methods
- declare a dependency by:
    1. adding a `private final FooBar foobar` field to your class
    2. adding a `FooBar foobar` parameter to your constructor, and assign it to the `foobar` field you just added.

We should avoid using ClassFactory and Components as much as possible. Some components will still need to of course, for instance to instantiate rendering models. Commands, trees and other MVC pages could hopefully be entirely handled by Content2Bean.

Avoid field-injection if possible: field level injection (annotating a field with `@Inject` for example) has its use and is a pragmatic solution to some problems (we used it for example for `AbstractMgnlFilter`, to avoid having to have a constructor-dependency on `WebContainerResources` on all our filters. You should avoid it as much as possible, and favor constructor-dependency-injection instead:

- the constructor can be your "init" method - all dependencies are there and ready to use. (when using `@Inject`, you'll often need `@PostConstruc`
  too)
- dependency-fields can be marked final, making your code safer
- you'll need some magic to inject the value of that field in your tests

Some components will not be "convertible" as-is. In some instances, we will maybe need to introduce "provider" components. Some components might need to be renamed, and we will have to come up with naming conventions (what is a component, what is a "value object" - for instance, the module configuration beans are probably the latter, although we'll want to inject them)