OSGi Component / Felix Declarative Service Style Guide

Quick Index:

- Component Annotations
- Reference Annotations
    - Notes on ReferencePolicy.DYNAMIC
        - Dynamic references must be volatile.
        - Dynamic, mandatory, unary references are a lie.
- Component Declaration Ordering

When writing an OSGi component, additional annotations, declarations, and methods are used to facilitate the OSGi lifecycle. Keeping this in a consistent order helps people find the parts they need to work on. Please adhere to these guidelines:

Component Annotations

Components require the `@Component` annotation.

1. Components ***must*** declare a `name` property set to a custom, author-supplied string literal if the component is to be referenced *outside* the declaring bundle.
    a. Consider using the constant `NAME` if it is local to the component, or ending in `COMPONENT_NAME` if defined in another class or constants file.
        i. We used to call these constants `PID` when we referred to them as "service PIDs" or "config PIDs". Be sure to replace `PID` with `NAME` when working with old code.
    b. The string literal constant should follow the pseudo packaging format `"org.forgerock.openidm.someplace.somecomponent"` - use human-friendly names for `somplace` and `somecomponent` that tries to encapsulate the area of code and purpose of the component.
2. Components ***must not*** declare a `name` property if the component is not referenced as a service or is only used *inside* the declaring bundle.
3. Components ***must*** declare a `service` property if
    a. they are to be registered as a service
    b. they do not implement an interface
    c. they implement multiple interfaces but the author wants control over which interfaces the component is registered

Components ***must*** declare `@ServiceVendor(ServerConstants.SERVER_VENDOR_NAME)` and `@ServiceDescription` if the component is to be referenced *outside* the declaring bundle.

Components ***may*** omit `@ServiceVendor` and `@ServiceDescription` if the component is only used *inside* the declaring bundle.

Components ***must*** declare `@RouterPrefix` to provide the handlers route if they implement `RequestHandler`, `SingletonResourceProvider`, or `@CollectionResourceProvider`.

Components ***must*** declare `@RouterVersion` when registering multiple resource versions of `RequestHandler`, `SingletonResourceProvider`, or `@CollectionResourceProvider` handlers.

Reference Annotations

References are the OSGi term for class or object dependencies. Using SCR, we can include references to indicate dependency injection in a few different ways:

1. field injection
2. constructor injection
3. method injection

Service dependency declarations ***must*** be annotated with `@Reference`.

1. Use constructor injection when references are *unary* and *static*.
2. Use field injection when references are *multiple* or *dynamic* (*Hint: Don't use dynamic references at all.*)
3. Use bind methods and method injection any time when you need to transform (add business logic) to the event of a reference becoming available to the component.

Notes on ReferencePolicy.DYNAMIC

Dynamic references must be `volatile`.

This is enforced by the `maven-bundle-plugin`. The rationale is this: The DYNAMIC ReferencePolicy indicates that the reference can be dynamically be set by the OSGi Framework at runtime. This means that, for memory visibility at the site of usage of the referenced field, that field must be `volatile` to guarantee visibility to changes across threads. Even if your code is not multi-threaded, there is high likelihood that the OSGi Framework *is* and that it may update your field's reference in a different thread than your code is executing.

Dynamic, mandatory, unary references are a lie.

This is the "`ReferencePolicy.DYNAMIC` is a lie" rule. Earlier versions of IDM development did not properly understand the meaning of a dynamic reference policy. As stated above, this means that the OSGi Framework can dynamically set the value of the reference when a new one is available. Early development took that to mean that, as a dependency was updated or changed, your component could have its reference on that dependency *swapped out* at runtime without deactivating the component. This is partially true. The missing ingredient in previous thinking was that *reference cardinality trumps reference policy*. The default cardinality of a reference is `ReferenceCardinality.MANDATORY`, or 1:1, which means that the component must have at least one and only one reference to the referenced component. If there are zero references available to satisfy the dependency, your component will not be activated (or will deactivate if the reference is marked to go away). So *adding `ReferencePolicy.DYNAMIC` to a mandatory, unary reference is a lie*. It will not be treated as dynamic at all. The proper way to use dynamic references are with `ReferenceCardinality.OPTIONAL` or `ReferenceCardinality.MULTIPLE` references. Do not add `ReferencePolicy.DYNAMIC` to unary references.

Component Declaration Ordering

When writing a component, declare the elements of your class in this order:

1. static final declarations (loggers, constants, etc)
2. service dependency field declarations annotated with @Reference
3. service dependency field declarations requiring bind/unbind methods and the `@Reference`'d bind/unbind methods themselves should follow immediately after annotated service declarations with these caveats
    a. **do not** write bind/unbind methods just to perform assignment/unassignment - use constructor injection instead
    b. **do** write bind/unbind methods when you need to assign the bind method parameter into a different type as the @Reference interface, such as inserting referenced objects with multiple cardinality into a Map
    c. **do** write bind/unbind methods when you need to perform logic other than or in addition to assignment, such as registering a listener or calling an event notification method which cannot be done in a constructor
4. `final` and non-@Reference instance variables
5. @Activate method or constructor
6. @Modified method
7. @Deactivate method
8. helpers called from activate/modified/deactivate
9. all other methods (interface implementation(s))