Guice implementation

Your Rating: ☆☆☆☆☆ Results: ★★★☆☆ 135 rates

Abstract

This page details the goals and the status of our Guice implementation.

Rationale

Our main rationale for evaluating Guice is its standards support, specifically JSR-330 and its maturity, the broad adoption and large use base is evident of this.

JSR-330 provides a set of standard annotatons that enable our code base to benefit from using IoC without depending on a specific IoC container. These are:

- javax.inject.Inject
- javax.inject.Singleton
- javax.inject.Scope
- javax.inject.Named
- javax.inject.Qualifier

And the provider interface

- javax.inject.Provider

Javadoc for JSR-330

We also want support for life cycle callbacks using JSR-250, these are:

- javax.annotation.PostConstruct
- javax.annotation.PreDestroy

Javadoc for JSR-250

In addition we want to use request and session scoped components, unfortunately there are no standard annotations for this. The closest thing is the annotations from CDI, but these are not supported by other IoC frameworks. So we're sticking with the Guice annotations:

- com.google.inject.servlet.RequestScoped
- com.google.inject.servlet.SessionScoped

See the section below on the Guice Servlet Extension for more details

Feature Set

This is an example of how to use of the new features. Note how the example manager has no dependency on static facilities for accessing ModuleManager, the request scoped AggregationState and neither for using application properties.

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.inject.Inject;
import javax.inject.Named;
import javax.inject.Provider;
import javax.inject.Singleton;

import com.google.inject.servlet.RequestScoped;
import info.magnolia.module.ModuleManager;

@RequestScoped
public class AggregationState {
    // Implementation skipped for brevity
}

@Singleton
public class GuiceExampleManager {

    @Inject
    @Named("magnolia.develop")
    private boolean inDevelopmentMode;
    @Inject
    public Provider<AggregationState> aggregationStateProvider;
    public ModuleManager moduleManager;

    @Inject
    public GuiceExampleManager(ModuleManager moduleManager) {
        this.moduleManager = moduleManager;
    }

    @PostConstruct
    public void start() {
        if (inDevelopmentMode) {
            System.out.println("GuiceExampleManager started in development mode");
        } else {
            System.out.println("GuiceExampleManager started");
        }
    }

    @PreDestroy
    public void stop() {
        System.out.println("GuiceExampleManager stopped");
    }

    public void render() {
        // Gets the current aggregationState
        AggregationState aggregationState = aggregationStateProvider.get();
    }
}
```

Terms

- Container, responsible for locating a component. In Guice this is the class Injector.
- Component, anything that's configured in and managed by a container

An introduction to Guice

What makes Guice different from other IoC frameworks, and what has made it popular is:

- It uses annotations instead of XML configuration files
- It fails fast at startup and reports all configuration problems at once, this is known as static analysis
- It's fast and optimized for performance

Component life cycle

A component has a life cycle with these steps

1. instantiation, the constructor annotated with @Inject is called with arguments resolved by the container or the default no-args constructor is used
2. fields annotated with @Inject are filled
3. methods annotated with @Inject are called with arguments resolved by the container
4. @PostConstruct is called
5. Component is in use
6. @PreDestroy is called when the container is closed down

⚠ See the section below on JSR-250 for limitations on @PreDestroy

Bindings

Guice is a framework for resolving the appropriate dependency at a specific place. The place is usually a constructor argument or a field, Guice calls these injection points. They're usually an interface that Guice resolves to an implementation, but you can be more specific by using qualifier annotations such as @Named.

To find what to inject at a specific injection point it uses a set of bindings. A binding map from an injection point to a Provider (a Provider is an interface with a single method get() which returns an implementation).

Guice has a fluent builder style API for configuring bindings. This is called the Binding EDSL. It's used like this:

```
bind(Service.class).to(ServiceImpl.class)
```

This creates a binding from interface Service to its implementation ServiceImpl. It's important to note that is doesn't add a binding for any other interface that ServiceImpl implements.

Guice Binding EDSL

Lazy singletons

⚠ Guice doesn't really support lazy singletons. When you start Guice you specify if you're in DEVELOPMENT or PRODUCTION. In development all singletons are lazy by default unless registered as an eager singleton. In production it starts all singletons at startup.

There is an extension available that adds it using a new scope, @LazySingleton. You use it instead of @Singleton.

https://groups.google.com/group/google-guice/browse_thread/thread/e34e475609c2cec4?pli=1

https://code.google.com/p/google-guice/issues/detail?id=357

Injecting properties

Using @Named on a primitive value such as a string or an int we can inject properties. Guice will automatically convert a property of type string, which it considers to be a constant, to an int or a boolean when injecting.

We would also like to allow properties to change at runtime and allow reading them by injecting a Provider like this:

```
@Named("magnolia.develop")
Provider<Boolean> isDevelopmentMode;
```

⚠ However, Guice doesn't do conversion on values returned from providers. It only does this on bound constants. So we have to choose, either we can inject to int and boolean etc or we support properties to change at runtime.

ComponentProvider Abstraction Layer

We use an abstraction layer on top of Guice based on the Components class which exists in Magnolia since 4.3. It is a static class with a method called getSingleton(Class<?> class). A basic service locator pattern. The static method delegates to a ComponentProvider which does the actual lookup.

ComponentProviders can be arranged hierarchically having these properties:

- A component in a child can rely on a component in the parent
- A component of a certain type can be configured in both the parent and the child
- They do not share life cycle but a parent must not be closed if it has children that have not been closed.

We will still have a global ComponentProvider set in Components but it's recommended to not use unless you have to. Instead inject the ComponentProvider for the container that your component lives in.

Guice supports a model of hierarchical injectors but it doesn't fit with the properties we want since it doesn't allow a parent and a child injector to have bindings for the same type. We have solved this by creating an independent injector for each component provider and bridging the child to the parent by adding bindings for everything in the parent.

JSR-250 and @Destroy

⚠ Guice does not support JSR-250 but there are extensions for it. There has been a lot of discussion on adding native support for JSR-250 but there's no indication that this will happen. It's probably not that easy since Guice is based on bindings for providing something, somewhere and do nothing else. There's no real container under the hood that knows about a set of instances is supposed to manage.

The most known extension of these is GuiceyFruit and more recent claiming to extend the feature set of GuiceyFruit is Guice Mycila. We've used Mycila Guice and found later on that it does a good job but its unable to properly support @PreDestroy due to the limitations above. We've opened these tickets to report our problems:

- https://code.google.com/p/mycila/issues/detail?id=30
- https://code.google.com/p/mycila/issues/detail?id=31
- https://code.google.com/p/mycila/issues/detail?id=32

It's currently not clear if they can be fixed, we've suggested an alternative approach in ticket 32 that might do the trick.

Guice servlet extension

guice-servlet is an extension to Guice, provided by Google, that adds two new scopes, request and session scope. It's also a complete replacement of web.xml configuration. In web.xml you configure only their filter, GuiceFilter, and then use an API similar to the bindings API to add servlets, filters and mappings.

This is not very interesting for Magnolia as we have a similar design but we do want the scopes. However, the implementation of the scopes is hard coded to use GuiceFilter. Not very extendable.

Right now we have copied the implementation of the scopes and changed them use our WebContext for accessing the request and session objects. The only part of guice-servlet we're using is the annotations. An alternative approach would be to have our own custom annotations as this would eliminate dependence on Guice.

Magnolia Startup and Shutdown sequence

Startup

When Magnolia starts up it goes through a number of steps that start up services. In the example above the manager relies on properties having been loaded earlier and are available to it when it is created. Another assumption that typical components will make is that the content repositories are started.

To achieve this we have three main steps, or stages, that the application goes through at startup.

Step 1 Platform Container

In this step we focus on loading properties. Since properties can come from modules the ModuleManager is started in this step.

Step 2 System Container

In this step we start up fundamental system services that components starting in the next step rely on. This includes starting up content repositories.

Step 3 Main Container

Then we start the application for real. If we start in install/update mode this step is postponed until all update tasks have been executed. Components configured in modules are started here.

Step 4 Start modules

And finally we start up the modules by invoking ModuleLifecycle.start().

Shutdown

When Magnolia is shutdown it performs the same steps in reverse. First calling ModuleLifecycle.stop() on modules, then it closes Main, then System and finally Platform.

Components with methods annotated with @PreDestroy are called when the step in which they live is closed.

ComponentProvider is also an Object Factory

ComponentProvider is also an object factory where you give it a type, an interface or a class and it will look up the implementation to use and instantiate an object of this. It's used by Content2Bean and in many more places. The ComponentProvider will provide dependency injection when instantiating the implementation.

Since we have a global ComponentProvider, the type that is returned depends on which ComponentProvider is currently set.

Additional constructor arguments

ComponentProvider can also take a set of additional arguments that will be used of candidate constructor arguments in addition to those taken from Guice.

Since Guice does static analysis to make sure that everything configured can be instantiated it will fail on these classes. So we don't configure these in Guice. There's two possible solutions here:

1. Don't keep them in Guice and configure them separately in the module descriptor
2. There's a feature of Guice called Assisted Inject which is designed to solve this problem. It requires a factory for each type.

How our Vaadin integration uses IoC

Vaadin has a server side component model of what you see on the screen. This is kept in session and bound to the current user. We have a ComponentProvider for each user that contains components for the user interface. This ComponentProvider is a child of the main ComponentProvider.

Still to do

Scope annotations

Do we keep the annotations from guice-servlet or do we copy them as well and loose the dependency on guice-servlet.

@PreDestroy

The state of support for @PreDestroy is a problem, potentially a show stopper. See the tickets above for the latest status.

Injecting properties

We still need to research if it can be possible to have both reloadable properties and automatic conversion of Strings to Boolean, Integer etc.

How will modules contribute components

In Magnolia we want modules to be able to add and replace components. They need to be able to contribute components to both System and Main container. This has yet to be designed, there's two ways we can do this:

1. XML configuration in the module descriptor
2. In the module descriptor we can have a class that is called to add bindings

It must also be possible to customize the components that run in the platform container but via some other mechanism since the module manager isn't ready until that container is started.