Java on Guice

Guice 1.0 User's Guide

Guice (pronounced "juice") is an ultra-lightweight, next-generation dependency injection container for Java 5 and later.

Introduction

The enterprise Java community exerts a lot of effort toward wiring objects together. How does your web application get access to a middle tier service, or your service to the logged in user or transaction manager? You'll find many general and specific solutions to this problem. Some rely on patterns. Others use frameworks. All result in varying degrees of testability and some amount of boilerplate code. You'll soon see that Guice enables the best of all worlds: easy unit testing, maximal flexibility and maintainability, and minimal repetition.

We'll use an unrealistically simple example to illustrate the benefits of Guice over some classic approaches which you're probably already familiar with. The following example is so simple in fact that, even though it will show immediate benefits, we won't actually do Guice justice. We hope you'll see that as your application grows, Guice's benefits accelerate.

In this example, a client depends on a service interface. This could be any arbitrary service. We'll just call it `Service`.

```
public interface Service {

 void go();
}
```

We have a default implementation of this service which the client should not depend directly on. If we decide to use a different service implementation in the future, we don't want to go around and change all of our clients.

```
public class ServiceImpl implements Service {

 public void go() {

   ...

 }

}
```

We also have a mock service which we can use in unit tests.

```
public class MockService implements Service {



 private boolean gone = false;

 public void go() {
  gone = true;
 }

 public boolean isGone() {
  return gone;
 }
}
```

Plain Old Factories

Before we discovered dependency injection, we mostly used the factory pattern. In addition to the service interface, you have a service factory which provides the service to clients as well as a way for tests to pass in a mock service. We'll make the service a singleton so we can keep this example as simple as possible.

```
public class ServiceFactory {

 private ServiceFactory() {}

 private static Service instance = new ServiceImpl();
```

```
 public static Service getInstance() {
  return instance;
 }

 public static void setInstance(Service service) {
  instance = service;
 }
}
```

Our client goes directly to the factory every time it needs a service.

```
public class Client {

  public void go() {
    Service service = ServiceFactory.getInstance();
    service.go();
  }
}
```

The client is simple enough, but the unit test for the client has to pass in a mock service and then remember to clean up afterwards. This isn't such a big deal in our simple example, but as you add more clients and services, all this mocking and cleaning up creates friction for unit test writing. Also, if you forget to clean up after your test, other tests may succeed or fail when they shouldn't. Even worse, tests may fail depending on which order you run them in.

```
public void testClient() {
  Service previous = ServiceFactory.getInstance();
  try {
    final MockService mock = new MockService();
    ServiceFactory.setInstance(mock);
    Client client = new Client();
    client.go();
    assertTrue(mock.isGone());
  }
  finally {
    ServiceFactory.setInstance(previous);
  }
}
```

Finally, note that the service factory's API ties us to a singleton approach. Even if `getInstance()` could return multiple instances, `setInstance()` ties our hands. Moving to a non-singleton implementation would mean switching to a more

complex API.

Dependency Injection By Hand

The dependency injection pattern aims in part to make unit testing easier. We don't necessarily need a specialized framework to practice dependency injection. You can get roughly 80% of the benefit writing code by hand.

While the client asked the factory for a service in our previous example, with dependency injection, the client expects to have its dependency passed in. Don't call me, I'll call you, so to speak.

```
public class Client {

  private final Service service;

  public Client(Service service) {
    this.service = service;
  }

  public void go() {
    service.go();
  }
}
```

This simplifies our unit test considerably. We can just pass in a mock service and throw everything away when we're done.

```
public void testClient() {
 MockService mock = new MockService();
 Client client = new Client(mock);
 client.go();
 assertTrue(mock.isGone());
}
```

We can also tell from the API exactly what the client depends on.

Now, how do we connect the client with a service? When implementing dependency injection by hand, we can move all dependency logic into factory classes. This means we need a factory for our client, too.

```
public static class ClientFactory {

  private ClientFactory() {}

  public static Client getInstance() {
    Service service = ServiceFactory.getInstance();
    return new Client(service);
  }
}
```

Implementing dependency injection by hand requires roughly the same number of lines of code as plain old factories.

Dependency Injection with Guice

Writing factories and dependency injection logic by hand for every service and client can become tedious. Some other dependency injection frameworks even require you to explicitly map services to the places where you want them injected.

Guice aims to eliminate all of this boilerplate without sacrificing maintainability.

With Guice, you implement modules. Guice passes a binder to your module, and your module uses the binder to map interfaces to implementations. The following module tells Guice to map `Service` to `ServiceImpl` in singleton scope:

```
public class MyModule implements Module {
 public void configure(Binder binder) {
  binder.bind(Service.class)
    .to(ServiceImpl.class)
    .in(Scopes.SINGLETON);
 }
}
```

A module tells Guice what we want to inject. Now, how do we tell Guice where we want it injected? With Guice, you annotate constructors, methods and fields with `@Inject`.

```
public class Client {

  private final Service service;

  @Inject
  public Client(Service service) {
    this.service = service;
  }

  public void go() {
    service.go();
  }
}
```

The `@Inject` annotation makes it clear to a programmer editing your class which members are injected.

For Guice to inject `Client`, we must either directly ask Guice to create a `Client` instance for us, or some other class must have `Client` injected into it.

Guice vs. Dependency Injection By Hand

As you can see, Guice saves you from having to write factory classes. You don't have to write explicit code wiring clients to their dependencies. If you forget to provide a dependency, Guice fails at startup. Guice handles circular dependencies automatically.

Guice enables you to specify scopes declaratively. For example, you don't have to write the same code to store an object in the `HttpSession` over and over.

In the real world, you often don't know an implementation class until runtime. You need meta factories or service locators for your factories. Guice addresses these problems with minimal effort.

When injecting dependencies by hand, you can easily slip back into old habits and introduce direct dependencies, especially if you're new to the concept of dependency injection. Using Guice turns the tables and makes doing the right thing easier. Guice helps keep you on track.

More Annotations

When possible, Guice enables you to use annotations in lieu of explicit bindings and eliminate even more boilerplate code. Back to our example, if you need an interface to simplify unit testing but you don't care about compile time dependencies, you can point to a default implementation directly from your interface.

```
@ImplementedBy(ServiceImpl.class)
public interface Service {
  void go();
}
```

If a client needs a `Service` and Guice can't find an explicit binding, Guice will inject an instance of `ServiceImpl`.

By default, Guice injects a new instance every time. If you want to specify a different scope, you can annotate the implementation class, too.

```
@Singleton
public class ServiceImpl implements Service {
  public void go() {
    ...
  }
}
```

Architectural Overview

We can break Guice's architecture down into two distinct stages: startup and

runtime. You build an `Injector` during startup and use it to inject objects at runtime.

Startup

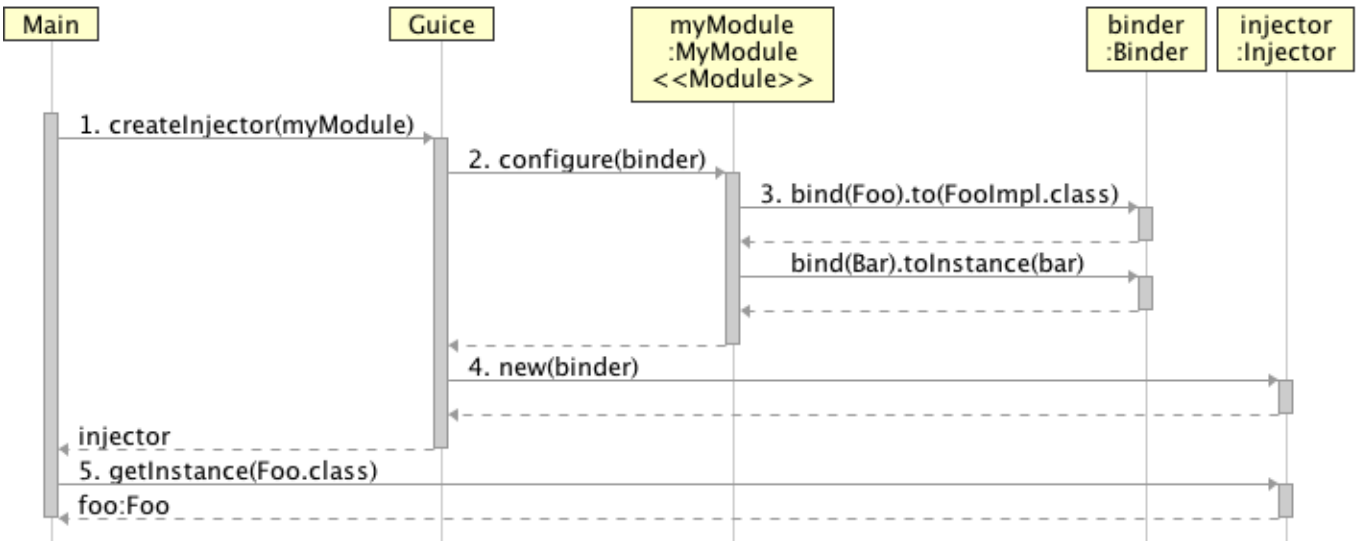
You configure Guice by implementing `Module`. You pass Guice a module, Guice passes your module a `Binder`, and your module uses the binder to configure bindings. A binding most commonly consists of a mapping between an interface and a concrete implementation. For example:

```
public class MyModule implements Module {
 public void configure(Binder binder) {
  // Bind Foo to FooImpl. Guice will create a new
    // instance of FooImpl for every injection.
    binder.bind(Foo.class).to(FooImpl.class);

    // Bind Bar to an instance of Bar.
    Bar bar = new Bar();
    binder.bind(Bar.class).toInstance(bar);
 }
}
```

Guice can look at the classes you tell it about during this stage and any classes those classes know about, and tell you whether or not you're missing any dependencies. For example, in a Struts 2 application, Guice knows about all of your actions. Guice can validate your actions and anything they transitively depend on, and fail early if necessary.

Creating an `Injector` entails the following steps:

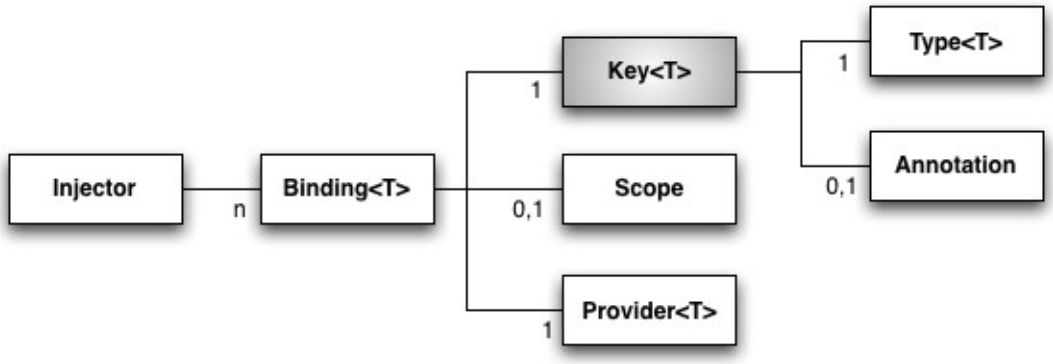


1. First, create an instance of your module and pass it to `Guice.createInjector()`.

2. Guice creates a `Binder` and passes it to your module.
3. Your module uses the binder to define bindings.
4. Based on the bindings you specified, Guice creates an `Injector` and returns it to you.
5. You use the injector to inject an object.

Runtime

We can now use the injector we created during the first stage to inject objects and introspect on our bindings. Guice's runtime model consists of an injector which contains some number of bindings.



A `Key` uniquely identifies each binding. The key consists of a type which the client depends on and an optional annotation. You can use an annotation to differentiate multiple bindings to the same type. The key's type and annotation correspond to the type and annotation at a point of injection.

Each binding has a provider which provides instances of the necessary type. You can provide a class, and Guice will create instances of it for you. You can give Guice an instance of the type you're binding to. You can implement your own provider, and Guice can inject dependencies into it.

Each binding also has an optional scope. Bindings have no scope by default, and Guice creates a new instance for every injection. A custom scope enables you to control whether or not Guice creates a new instance. For example, you can create one instance per `HttpSession`.

Bootstrapping Your Application

The idea of *bootstrapping* is fundamental to dependency injection. Always explicitly asking the `Injector` for dependencies would be using Guice as a service locator, not a dependency injection framework.

Your code should deal directly with the `Injector` as little as possible. Instead, you want to bootstrap your application by injecting one root object. The container can further inject dependencies into the root object's dependencies, and so on

recursively. In the end, your application should ideally have one class (if that many) which knows about the `Injector`, and every other class should expect to have dependencies injected.

For example, a web application framework such as Struts 2 bootstraps your application by injecting all of your actions. You might bootstrap a web service framework by injecting your service implementation classes.

Dependency injection is viral. If you're refactoring an existing code base with a lot of static methods, you may start to feel like you're pulling a never-ending thread. This is a Good Thing. It means dependency injection is making your code more flexible and testable.

If you get in over your head, rather than try to refactor an entire code base all in one shot, you might *temporarily* store a reference to the `Injector` in a static field somewhere or use static injection. Name the field's class clearly though: `InjectorHack` and `GodKillsAKittenEveryTimeYouUseMe` come to mind. Keep in mind that you you'll have to mock this class, and your unit tests will have to install an Injector here by hand, and remember to clean up afterwards.

Binding Dependencies

How does Guice know what to inject? For starters, a `Key` composed of a type and an optional annotation uniquely identifies a dependency. Guice refers to the mapping between a key and an implementation as a `Binding`. An implementation can consist of a single object, a class which Guice should also inject, or a custom provider.

When injecting a dependency, Guice first looks for an *explicit* binding, a binding which you specified using the `Binder`. The `Binder` API uses the builder pattern to create a domain-specific expression language. Different methods return different objects depending on the context limiting you to appropriate methods.

For example, to bind an interface `Service` to a concrete implementation `ServiceImpl`, call:

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

This binding matches the following the method:

```
@Inject
void injectService(Service service) {
  ...
}
```

> **Note:** In contrast to some other frameworks, Guice gives no special treatment to "setter" methods. Guice will inject any method with any number of parameters so long as the method has an `@Inject` annotation, even if the method is in a superclass.

DRY (Don't Repeat Yourself)

Repeating "`binder`" over and over for each binding can get a little tedious. Guice provides a `Module` support class named `AbstractModule` which implicitly gives you access to `Binder`'s methods. For example, we could extend `AbstractModule` and rewrite the above binding as:

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

We'll use this syntax throughout the rest of the guide.

Annotating Bindings

If you need multiple bindings to the same type, you can differentiate the bindings with annotations. For example, to bind an interface `Service` and annotation `@Blue` to the concrete implementation `BlueService`, call:

```
bind(Service.class)
  .annotatedWith(Blue.class)
  .to(BlueService.class);
```

This binding matches the following the method:

```
@Inject
void injectService(@Blue Service service) {
  ...
}
```

Notice that while `@Inject` goes on the method, binding annotations such as `@Blue` go directly on the parameter. The same goes for constructors. When using field injection, both annotations can apply directly to the field, as in this example:

```
@Inject @Blue Service service;
```

Creating Binding Annotations

Where did this `@Blue` annotation just mentioned come from? You can create such an annotation easily, although the standard incantation you have to use is unfortunately a little complex:

```
/**
 * Indicates we want the blue version of a binding.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER})
@BindingAnnotation
public @interface Blue {}
```

Luckily, we don't really have to understand it all just to use it. But for the curious, here's what all this boilerplate means:

- `@Retention(RUNTIME)` allows your annotation to be visible at runtime.
- `@Target({FIELD, PARAMETER})` is a courtesy to your users; it prevents `@Blue` from being applied to methods, types, local variables, and other annotations, where it would serve no purpose.
- `@BindingAnnotation` is a Guice-specific signal that you wish your annotation to be used in this way. Guice will produce an error whenever user applies more than one binding annotation to the same injectable element.

Annotations With Attributes

If you can get by with marker annotations alone, feel free to skip to the next section.

You can also bind to annotation instances, i.e. you can have multiple bindings with the same type and annotation type, but with different annotation attribute values. If Guice can't find a binding to an annotation instance with the necessary attribute values, it will look for a binding to the annotation type instead.

Say for example we have a binding annotation `@Named` with a single string attribute value.

```
@Retention(RUNTIME)
@Target({ FIELD, PARAMETER })
@BindingAnnotation
public @interface Named {
  String value();
}
```

If we want to bind to `@Named("Bob")`, we first need an implementation of `Named`. Our implementation must abide by the `Annotation` contract, specifically the implementations of `hashCode()` and `equals()`.

```
class NamedAnnotation implements Named {
  final String value;
  public NamedAnnotation(String value) {
    this.value = value;
```

```
  }
  public String value() {
    return this.value;
  }
  public int hashCode() {
    // This is specified in java.lang.Annotation.
    return 127 * "value".hashCode() ^ value.hashCode();
  }
  public boolean equals(Object o) {
    if (!(o instanceof Named))
      return false;
    Named other = (Named) o;
    return value.equals(other.value());
  }
  public String toString() {
    return "@" + Named.class.getName() + "(value=" + value + ")";
  }
  public Class<? extends Annotation> annotationType() {
    return Named.class;
  }
}
```

Now we can use this annotation implementation to create bindings to `@Named`.

```
bind(Person.class)
  .annotatedWith(new NamedAnnotation("Bob"))
  .to(Bob.class);
```

This may seem like a lot of work compared to string based identifiers used by other frameworks, but keep in mind that you can't do this at all with string-based identifiers. Also, you'll find that you get a lot of reuse out of binding annotations.

Since identifying a binding by name is such a common use case, Guice provides a production-worthy implementation of `@Named` in `com.google.inject.name`.

Implicit Bindings

As we saw in the introduction, you don't always have to declare bindings explicitly. In the absence of an explicit binding, Guice will try to inject and create a new instance of the class you depend on. If you depend on an interface, Guice will look for an `@ImplementedBy` annotation which points to the concrete implementation. Take the following explicit binding to a concrete, injectable class named `Concrete` for example. It basically says, bind `Concrete` to `Concrete`. That's explicit, but also a little redundant.

```
bind(Concrete.class);
```

Removing the binding above would not affect the behavior of this class:

```
class Mixer {

  @Inject
  Mixer(Concrete concrete) {
    ...
  }
}
```

So, take your pick: explicit or brief. In the event of an error, Guice will produce helpful messages either way.

Injecting Providers

Sometimes a client needs multiple instances of a dependency per injection. Other times a client may not want to actually retrieve an object until some time after the actual injection (if at all). For any binding of type `T`, rather than inject an instance of `T` directly, you can inject a `Provider<T>`. Then call `Provider<T>.get()` as necessary. For example:

```
@Inject
void injectAtm(Provider<Money> atm) {
  Money one = atm.get();
  Money two = atm.get();
  ...
}
```

As you can see, the `Provider` interface couldn't get much simpler so it doesn't get in the way of easy unit testing.

Injecting Constant Values

When it comes to constant values, Guice gives special treatment to several types:

- Primitive types (int, char, ...)
- Primitive wrapper types (Integer, Character, ...)
- Strings
- Enums
- Classes

First, when binding to constant values of these types, you needn't specify the type you're binding to. Guice can figure it out from the value. For example, given a binding annotation named `TheAnswer`:

```
bindConstant().annotatedWith(TheAnswer.class).to(42);
```

Has the same effect as:

```
bind(int.class).annotatedWith(TheAnswer.class).toInstance(42);
```

When it comes time to inject a value of one of these types, if Guice can't find an explicit binding for a primitive type, it will look for a binding to the corresponding wrapper type and vice versa.

Converting Strings

If Guice still can't find an explicit binding for one of the above types, it will look for a constant `String` binding with the same binding annotation and try to convert its value. For example:

```
bindConstant().annotatedWith(TheAnswer.class).to("42"); //
String!
```

Will match:

```
@Inject @TheAnswer int answer;
```

When converting, Guice will try to look up enums and classes by name. Guice converts a value once at startup which also means you get up front type checking. This feature comes in especially handy if the binding value comes from a properties file for example.

Custom Providers

Sometimes you need to create your objects manually rather than let Guice create them. For example, you might not be able to add `@Inject` annotations to the implementation class as it came from a 3rd party. In these cases, you can implement a custom `Provider`. Guice can even inject your provider class. For example:

```
class WidgetProvider implements Provider<Widget> {

 final Service service;

 @Inject
 WidgetProvider(Service service) {
  this.service = service;
 }

 public Widget get() {
  return new Widget(service);
 }
}
```

You bind `Widget` to `WidgetProvider` like so:

```
bind(Widget.class).toProvider(WidgetProvider.class);
```

Injecting the custom providers enables Guice to check the types and dependencies up front. Custom providers can reside in any scope independent of the scope of the objects they provide. By default, Guice creates a new provider instance for every injection. In the above example, if each `Widget` needs its own instance of `Service`, our code will work fine. You can specify a different scope for a custom factory using a scope annotation on the factory class or by creating a separate binding for the factory.

Example: Integrating With JNDI

Say for example we want to bind to objects from JNDI. We could implement a reusable custom provider similar to the one below. Notice we inject the JNDI `Context`:

```
package mypackage;

import com.google.inject.*;
import javax.naming.*;

class JndiProvider<T> implements Provider<T> {

  @Inject Context context;
  final String name;
  final Class<T> type;

  JndiProvider(Class<T> type, String name) {
    this.name = name;
    this.type = type;
  }

  public T get() {
    try {
      return type.cast(context.lookup(name));
    }
    catch (NamingException e) {
      throw new RuntimeException(e);
    }
  }

  /**
   * Creates a JNDI provider for the given
   * type and name.
   */
  static <T> Provider<T> fromJndi(
      Class<T> type, String name) {
    return new JndiProvider<T>(type, name);
```

```
  }
}
```

Thanks to generic type erasure, we must pass in the class at runtime. You could skip this step, but tracking down type casting errors later might be a little tricky (i.e. if JNDI returns an object of the wrong type).

We can use our custom `JndiProvider` to bind `DataSource` to an object from JNDI:

```
import com.google.inject.*;
import static mypackage.JndiProvider.fromJndi;
import javax.naming.*;
import javax.sql.DataSource;

...

// Bind Context to the default InitialContext.
bind(Context.class).to(InitialContext.class);

// Bind to DataSource from JNDI.
bind(DataSource.class)
  .toProvider(fromJndi(DataSource.class, "..."));
```

Scoping Bindings

By default, Guice creates a new object for every injection. We refer to this as having "no scope." You can specify a scope when you configure a binding. For example, to inject the same instance every time:

```
bind(MySingleton.class).in(Scopes.SINGLETON);
```

As an alternative, you can use an annotation on your implementation class to specify the scope. Guice supports `@Singleton` by default:

```
@Singleton
class MySingleton {
  ...
}
```

The annotation approach works with implicit bindings as well but requires that Guice create your objects. On the other hand, calling `in()` works with almost any binding type (binding to a single instance being an obvious exception) and overrides annotations when present. `in()` also accepts annotations if you don't want a compile time dependency on your scope implementation.

Specify annotations for custom scopes using `Binder.bindScope()`. For example, given an annotation `@SessionScoped` and a `Scope` implementation `ServletScopes.SESSION`:

```
binder.bindScope(SessionScoped.class, ServletScopes.SESSION);
```

Creating Scope Annotations

Annotations used for scoping should:

- Have a `@Retention(RUNTIME)` annotation so we can see the annotation at runtime.
- Have a `@Target({TYPE})` annotation. Scope annotations only apply to implementation classes..
- Have a `@ScopeAnnotation` meta-annotation. Only one such annotation can apply to a given class.

For example:

```
/**
 * Scopes bindings to the current transaction.
 */
@Retention(RUNTIME)
@Target({TYPE})
@ScopeAnnotation
public @interface TransactionScoped {}
```

Eagerly Loading Bindings

Guice can wait to load singleton objects until you actually need them. This helps speed up development because your application starts faster and you only initialize what you need. However, sometimes you always want to load an object at startup. You can tell Guice to always eagerly load a singleton like so:

```
bind(StartupTask.class).asEagerSingleton();
```

We frequently use this to implement initialization logic for our application. You can control the ordering of your initialization by creating dependencies on singletons which Guice must instantiate first.

Injecting Between Scopes

You can safely inject objects from a larger scope into an object from a smaller scope, or the same scope. For example, you can inject an Http session-scoped object into an HTTP request-scoped object. However, injecting into objects with larger scopes is a different story. For example, if you injected a request-scoped object into a singleton, at best, you would get an error due to not running within an HTTP request, and at worst your singleton object would always reference an object from the first request. In these cases, you should inject a `Provider<T>`

instead and use it to retrieve the object from the smaller scope as necessary. Then, you should be certain to never invoke this provider when you are outside of `T`'s scope (for example, when you are not servicing an HTTP request, and `T` is request-scoped).

Development Stages

Guice is aware that your application goes through different stages of development. You can tell it which stage the application is running in when you create a container. Guice currently supports "development" and "production." We've found that tests usually fall under one stage or the other.

During development, Guice will load singleton objects on demand. This way, your application starts up fast and only loads the parts you're testing.

In production, Guice will load all your singleton objects at startup. This helps catch errors early and takes any performance hits up front.

Your modules can also apply method interceptors and other bindings based on the current stage. For example, an interceptor might verify that you don't use your objects out of scope during development.

Intercepting Methods

Guice supports simple method interception using the AOP Alliance API. You can bind interceptors from your modules using `Binder`. For example, to apply a transaction interceptor to methods annotated with `@Transactional`:

```
import static com.google.inject.matcher.Matchers.*;

...

binder.bindInterceptor(
 any(),                              // Match classes.
 annotatedWith(Transactional.class), // Match methods.
 new TransactionInterceptor()        // The interceptor.
);
```

Try to shoulder as much of the filtering as is possible on the matchers rather than in the interceptor's body as the matching code runs only once at startup.

Static Injection

Static fields and methods make testing and reusing more difficult, but there are

times where your only choice is to keep a static reference to the `Injector`.

For these situations, Guice supports injecting less accessible static members. For example, HTTP session objects often need to be serializable to support replication, but what if your session object depends on a container-scoped object? We can keep a transient reference to the object, but how do we look it up again upon deserialization?

We've found the most pragmatic solution to be static injection:

```
@SessionScoped
class User {

  @Inject
  static AuthorizationService authorizationService;
  ...
}
```

Guice never performs static injection automatically. You must use `Binder` to explicitly request that the `Injector` inject your static members after startup:

```
binder.requestStaticInjection(User.class);
```

Static injection is a necessary evil, which makes testing more difficult. If you can find a way to avoid using it, you'll probably be glad you did.

Optional Injection

Sometimes your code should work whether a binding exists or not. In these cases, you can use `@Inject(optional=true)` and Guice can override your default implementation with a bound implementation when available. For example:

```
@Inject(optional=true) Formatter formatter = new
DefaultFormatter();
```

If someone creates a binding for `Formatter`, Guice will inject an instance from that binding. Otherwise, assuming `Formatter` isn't injectable itself (see *Implicit Bindings*), Guice will skip the optional member.

Optional injection applies only to fields and methods, not constructors. In the case of methods, if a binding for one parameter is missing, Guice won't inject the method at all, even if bindings to other parameters are available.

Binding to Strings

We try to avoid using strings whenever possible as they're prone to misspellings, not tool friendly, and so on, but using strings instead of creating custom annotations can prove useful for quick and dirty code. For these situations, Guice

provides `@Named` and `Names`. For example, a binding to a string name like:

```
import static com.google.inject.name.Names.*;

...

bind(named("bob")).to(10);
```

Will match injection points like:

```
@Inject @Named("bob") int score;
```

Struts 2 Support

To install the Guice Struts 2 plugin with Struts 2.0.6 or later, simply include `guice-struts2-plugin-1.0.jar` in your web application's classpath and select Guice as your `ObjectFactory` implementation in your `struts.xml` file:

```
<constant name="struts.objectFactory" value="guice" />
```

Guice will inject all of your Struts 2 objects including actions and interceptors. You can even scope your actions. You can optionally specify a `Module` for Guice to install in your `struts.xml` file:

```
<constant name="guice.module" value="mypackage.MyModule"/>
```

If all of your bindings are implicit, you can get away without defining a module at all.

A Counting Example

Say for example that we want to count the number of requests in a session. Define a `Counter` object which will live on the session:

```
@SessionScoped
public class Counter {

  int count = 0;

  /** Increments the count and returns the new value. */
  public synchronized int increment() {
    return count++;
  }
}
```

Next, we can inject our counter into an action:

```
public class Count {
```

```
  final Counter counter;

  @Inject
  public Count(Counter counter) {
   this.counter = counter;
  }

  public String execute() {
   return SUCCESS;
  }

  public int getCount() {
   return counter.increment();
  }
}
```

Then create a mapping for our action in our struts.xml file:

```
<action name="Count"
    class="mypackage.Count">
  <result>/WEB-INF/Counter.jsp</result>
</action>
```

And a JSP to render the result:

```
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
 <body>
  <h1>Counter Example</h1>
  <h3><b>Hits in this session:</b>
   <s:property value="count"/></h3>
 </body>
</html>
```

We actually made this example more complicated than necessary in an attempt to illustrate more concepts. In reality, we could have done away with the separate `Counter` object and applied `@SessionScoped` to our action directly.

JMX Integration

See com.google.inject.tools.jmx.

Appendix: How the Injector resolves injection requests

The injector's process of resolving an injection request depends on the bindings that have been made and the annotations found on the types involved. Here is a

summary of how an injection request is resolved:

1. Observe the Java type and the optional "binding annotation" of the element to be injected. If the type is `com.google.inject.Provider<T>`, perform resolution for the type indicated by `T` instead. Find a binding for this (type, annotation) pair. If none, skip to #4.
2. Follow transitive bindings. If this binding links to another binding, follow this edge and check again, repeating until we reach a binding which does not link to any other binding. We are now at the most specific explicit binding for this injection request.
3. If this binding specifies an instance or a `Provider` instance, we're done; use this to fulfill the request.
4. If, at this point, the injection request used an annotation type or value, we have failed and we produce an error.
5. Otherwise examine the Java type for this binding; if an `@ImplementedBy` annotation is found, instantiate the referenced type. If a `@ProvidedBy` annotation is found, instantiate the referenced provider and use it to obtain the desired object. Otherwise attempt to instantiate the type itself.