

GETTING STARTED WITH DEPENDENCY INJECTION AND GUICE

Justin Lee

Member of Technical Staff @ mongodb.com
<http://antwerkz.com> / [@evanchooly](https://twitter.com/evanchooly)

SO. INJECTION.

Also known as Inversion of Control
though some make a finer distinction

Normally you give the class what it needs

You have to know how to construct the class

And all of its dependencies

which probably have their own dependencies

that have dependencies

that have dependencies

that have dependencies

WHEW!

SO. WHAT CAN WE DO ABOUT THAT?

We can inject our dependencies!

Right. But what does that even mean?

Well, it can mean a few things

In short, the dependencies are declared explicitly
somehow

and the dependencies are given to the class

XML (Spring)

Annotations (Guice and CDI)

We're gonna focus on guice
obviously

WHAT IS GUICE?

Injection framework developed at Google
by Bob Lee (@crazybob)

(No relation)

Variety of annotations

Extensible

Light weight (ish)

DIFFERENT KINDS OF INJECTION

Field

Constructor

Method

@Assisted

@Named

@com.google.inject.Inject

(@javax.inject.Inject -- CDI)

FIELD INJECTION

```
public class Needy {  
    @Inject  
    private MailService mailer;  
    @Inject  
    private Invoice invoice;  
  
    ...  
}
```

FIELD INJECTION

```
public class Needy {  
    @Inject  
    private MailService mailer;  
    @Inject(optional = true)  
    private Invoice invoice;  
  
    ...  
}
```


CONSTRUCTOR INJECTION

```
public class Needy {  
    private MailService mailer;  
    private Invoicer invoicer;  
  
    @Inject  
    public Needy(MailService mailer, Invoicer invoicer) {  
        this.mailer = mailer;  
        this.invoice = invoice;  
    }  
  
    ...  
}
```

METHOD AND NAMED INJECTION

```
public class TwitterAnalyzer {  
    private String apiKey;  
  
    @Inject  
    public setApiKey(  
        @Named("twitterKey") String apiKey) {  
        this.apiKey = apiKey;  
    }  
  
    ...  
}
```

DEFINING THE DEPENDENCIES

```
public class GuiceModule extends AbstractModule {  
    @Override  
    protected void configure() {  
        bind(BlahBlah.class).to( BlahBlahImpl.class );  
    }  
  
    @Provides  
    @Singleton  
    public MailService create() {  
        return new MailService("mail.example.com", 1500,  
                                "myusername", "mypassword");  
    }  
}
```

PUTTING IT ALL TOGETHER

```
public static void main(String[] args) {  
    Injector injector = Guice.createInjector(new GuiceModule());  
    Needy needy = injector.getInstance(Needy.class);  
    needy.whatever();  
}
```

This all sounds great

but big deal

Why?

TO THE CODE!

PROVIDERS

Sometimes you need more than one instance

Thread handlers

Multiple connections

```
@Inject Provider<YourType> provider;
```

```
public class Processor {  
    @Inject  
    private Provider<Transformer> provider;  
  
    ...  
  
    public execute(CustomerData data) {  
        provider.get().process(data);  
    }  
}
```

ON DEMAND INJECTIONS

Loading external class definitions
SPI

```
public class Processor {  
    @Inject  
    private Injector injector;  
  
    ...  
  
    public execute(CustomerData data) {  
        injector.getInstance(Transformer.class).process(data);  
    }  
}
```


FACTORIES

Assisted Injections

Hybrids between injection and traditional factories

Factory interface

Let's borrow an example from Cedric

<http://beust.com/weblog/2012/08/21/advanced-dependency-injection-with-guice/>

```
public class Person {  
    // Fields omitted  
  
    public Person(String name, Address address, GeoService gs) {  
        this.name = name;  
        this.address = address;  
        this.geoService = gs;  
    }  
  
    public boolean livesNear(Person otherPerson) {  
        return geoService.isNear(address, otherPerson.getAddress(),  
            2 /* miles */);  
    }  
}
```

```
public interface PersonFactory {  
    Person create(String name, Address address);  
}
```

```
@Inject  
public Person(@Assisted String name,  
              @Assisted Address address, GeoService geoService)  
{  
    this.name = name;  
    this.address = address;  
    this.geoService = geoService;  
}
```

```
public class MyModule extends AbstractModule {
    @Override
    protected void configure() {
        install(new FactoryModuleBuilder()
            .build(PersonFactory.class));
    }
}

-----

private PersonFactory personFactory;

// ...

Person p = personFactory.create("Bob",
    new Address("1 Ocean st"));
```

CODE

STATIC INJECTIONS

smelly

statics are unnecessary in a DI world

Can not dynamically vary by environment

A good intermediate step

<https://code.google.com/p/google-guice/wiki/AvoidStaticState>

STATIC INJECTIONS

```
public class MyModule extends AbstractModule {
    @Override
    protected void configure() {
        binder().requestStaticInjection(MyUtil.class);
    }
}

public class MyUtil {
    @Inject
    private static MailService service;

    public static void alert() {
        service.sendAlert();
    }
}

...

MyUtil.alert();
```

TESTING

Biggest win for DI

Vary by deployment environment

dev, staging, production

unit testing

integration testing

(continuous integration)

Different modules

TESTNG

```
@Guice(modules = {ModuleA.class, ModuleB.class})
public class BaseTest {
    @Inject
    private Foo;
}

public class ModuleA {
    @Provides
    public Foo foo() {
    }
}
```

RECAP

Centralize construction

Minimize impact of code changes

Easy substitution of implementations

Environmental based configurations

Testing

GETTING STARTED WITH DEPENDENCY INJECTION AND GUICE

Justin Lee

Member of Technical Staff @ mongodb.com
<http://antwerkz.com> / [@evanchooly](https://twitter.com/evanchooly)
<http://github.com/evanchooly>