# A Taste of Guice

6/17/2010

# An email filter

```
class Filter {
   private Whitelist whitelist       = new MysqlWhitelist();
   private SpamClassifier classifier = new FarmdSpamClassifier();
   private OutboundQueue queue       = new SmtpOutboundQueue();

   public void filter(Message msg) {
      int score = classifier.getScore(msg);

      if (whitelist.contains(msg.getSender()) ||
          classifier.getScore(msg) < 50) {
         queue.put(msg);
      }
   }
}
```

# Issues?

- Coupling
- How do we unit test that class?
  - We need a Farmd cluster, a mysql instance for the whitelist, an smtp server
  - If we're not careful, we'll be sending email out every time the test runs!

# Improvement (?): static factories

```
class Filter {
   private Whitelist whitelist = WhitelistFactory.get();
   private SpamClassifier classifier = SpamClassifier.get();
   private OutboundQueue queue = OutboundQueue.get();

   public void filter(Message msg) {
      ...
   }
}
```

# Factories

- For each service...

```
class WhitelistFactory {
   private static Whitelist instance;

   public static synchronized Whitelist get() {
      if (instance == null) {
         instance = new MysqlWhitelist();
      }
      return instance;
   }

   public static synchronized void set(Whitelist whitelist) {
      instance = whitelist;
   }
}
```

# Issues?

- “decoupling”. Compile-time dependency between Filter and MysqlWhitelist, etc.
- Need to look at Filter implementation to know it depends on quarantine, outbound queue, etc
- Hard to reuse Filter in different contexts.
- Lots of boilerplate: same factory code for every dependency
  - … and in tests

# Testing with factories

```
void testWhitelistPrecedence () {
   Whitelist previousWhitelist  = WhitelistFactory.get();
   SpamClassifier previousClassifier  = SpamClassifierFactory .get();
   OutboundQueue previousQueue  = OutboundQueueFactory .get();

   try {
      MockWhitelist whitelist  = new InWhiteList();
      MockClassifier classifier  = new MockClassifier ();
      MockOutboundQueue queue  = new MockOutboundQueue();

      WhitelistFactory .set(whitelist);
      ClassifierFactory .set(classifier);
      OutboundQueueFactory .set(queue);

      Message msg = new Message("sender@proofpoint.com" , "recipient@proofpoint.com" , ...);
      whitelist.add(msg.getSender ());
      classifier.setScore (msg, 100);

      Filter filter = new Filter();
      filter.filter(msg);

      assertTrue (queue.contains (msg));
   }
   finally {
      WhitelistFactory .set(previousWhitelist );
      ClassifierFactory .set(previousClassifier );
      OutboundQueueFactory .set(previousQueue );
   }
}
```

# Dependency Injection

```
class Filter {
   private final Whitelist whitelist;
   private final SpamClassifier classifier;
   private final OutboundQueue queue;

   public Filter(Whitelist whitelist, SpamClassifier classifier,
                 OutboundQueue queue) {
      this.whitelist = whitelist;
      this.classifier = classifier;
      this.queue = queue;
   }

   public void filter(Message msg) {
      int score = classifier.getScore(msg);

      if (whitelist.contains(msg.getSender()) ||
         classifier.getScore(msg) < 10) {
         queue.put(msg);
      }
   }
}
```

# Testing with Dependency Injection

```
void testWhitelistPrecedence() {
   MockWhitelist whitelist = new InWhiteList();
   MockClassifier classifier = new MockClassifier();
   MockOutboundQueue queue = new MockOutboundQueue();

   Message msg = new Message("sender@proofpoint.com",
                             "recipient@proofpoint.com", ...);
   whitelist.add(msg.getSender());

   classifier.setScore(msg, 100); // very spammy

   Filter filter = new Filter();
   filter.filter(msg);

   assertTrue(queue.contains(msg));
}
```

# DI advantages

- Testability
  - Easier to mock dependent services, less boilerplate
- Modularity
  - Filter depends only on interfaces
  - Implementation of dependent services can be packaged and developed independently
- Explicit dependencies
  - Just look at the public interface of the class

# What is Guice?

A dependency injection framework

Questions?

# DI with Guice

```
class Filter {
   private final Whitelist whitelist;
   private final SpamClassifier classifier;
   private final OutboundQueue queue;

   @Inject
   public Filter(Whitelist whitelist, SpamClassifier classifier,
                 OutboundQueue queue) {
      this.whitelist = whitelist;
      this.classifier = classifier;
      this.queue = queue;
   }

   public void filter(Message msg) {
      ...
   }
}
```

# DI with Guice

```
public class FilterModule extends AbstractModule {
   protected void configure() {
      bind(Whitelist.class)
         .to(MysqlWhitelist.class);

      bind(SpamClassifier.class)
         .to(FarmdSpamClassifier.class);

      bind(OutboundQueue.class)
         .to(SmtpOutboundQueue.class);
   }
}
```

```
public static void main(String[] args) {
   Injector injector = Guice.createInjector(new FilterModule());

   Filter filter = injector.getInstance(Filter.class);
   ...
}
```

# Some benefits of using Guice

- Declarative
  - Bindings can be defined in any order. Guice will figure out who depends on what and construct objects accordingly
- Meaningful errors
  - e.g., missing binding

```
Exception in thread "main" com.google.inject.ConfigurationException: Guice
configuration errors:

1) No implementation for SpamClassifier was bound.
  while locating SpamClassifier
    for parameter 1 at Filter.<init>(Filter.java:7)
  while locating Filter

1 error
at com.google.inject.InjectorImpl.getProvider(InjectorImpl.java:784)
at com.google.inject.InjectorImpl.getProvider(InjectorImpl.java:743)
at com.google.inject.InjectorImpl.getInstance(InjectorImpl.java:793)
at Main.main(Main.java:20)
```

# More benefits of using Guice

- Type safety (compared to frameworks such as Spring)

```
// doesn't compile!
bind(Whitelist.class).to(SmtpOutboundQueue.class)
```

```
FilterModule:10: cannot find symbol
symbol  : method to(java.lang.Class<SmtpOutboundQueue>)
location: interface com.google.inject.binder.AnnotatedBindingBuilder<Whitelist>
        bind(Whitelist.class).to(SmtpOutboundQueue.class);
                      ^
1 error
```

- Modularity

```
Guice.createInjector(new DatabaseModule(),
                     new FilterModule(),
                     new LoggingModule(),
                     ...);
```

# Handling multiple implementations

```
class LocallyCachedWhitelist {
   private final Database remote;
   private final Database local;

   @Inject
   public LocallyCachedWhitelist(Database remote,
                                 Database local) {
      this.remote = remote;
      this.local = local;
   }

   ...
}
```

# Handling multiple implementations

- Can't have more than one binding!

```
bind(Whitelist.class).to(LocallyCachedWhiteList.class);
bind(Database.class).to(LocalDatabase.class);
bind(Database.class).to(RemoteDatabase.class);
```

```
Exception in thread "main" com.google.inject.CreationException: Guice creation errors:

1) A binding to Database was already configured at FilterModule.configure(FilterModule.java:14).
  at FilterModule.configure(FilterModule.java:15)

1 error
at com.google.inject.internal.Errors.throwCreationExceptionIfErrorsExist(Errors.java:354)
at com.google.inject.InjectorBuilder.initializeStatically(InjectorBuilder.java:152)
at com.google.inject.InjectorBuilder.build(InjectorBuilder.java:105)
at com.google.inject.Guice.createInjector(Guice.java:92)
at com.google.inject.Guice.createInjector(Guice.java:69)
at com.google.inject.Guice.createInjector(Guice.java:59)
...
```

# Binding Annotations

- “Tagged” dependencies

```
class LocallyCachingWhitelist {
   @Inject
   public LocallyCachingWhitelist(@Remote Database remote,
                                  @Local  Database local) {
      ...
   }
}
```

```
void configure() {
   bind(Database.class)
      .annotatedWith(Local.class)
      .to(LocalDatabase.class);

   bind(Database.class)
      .annotatedWith(Remote.class)
      .to(RemoteDatabase.class);
}
```

# Binding Annotations

```
@Target({ ElementType.PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
@BindingAnnotation
public @interface Local
{
}
```

# Providers

```
bind(Database.class)
   .toProvider(MysqlDatabaseProvider.class);



class MysqlDatabaseProvider implements Provider<Database>
   private final Configuration config;

   @Inject
   public MysqlDatabaseProvider(Configuration config) {
      this.config = config;
   }

   public Database get() {
      return new MysqlDatabase(config.getUrl(),
                               config.getUser(),
                               config.getPassword());
   }
}
```

# Provider methods

```
class DatabaseModule extends AbstractModule {
   protected void configure() {
      ...
   }

   @Provides
   public Database getMysqlDatabase(Configuration config) {
      return new MysqlDatabase(config.getUrl(),
                               config.getUser(),
                               config.getPassword())
   }
}
```

# Scopes

- Policy for reusing objects
  - Default: no scope
    - A new instance is created for every injection site
  - Built-in: singleton
    - One instance per type within an injector
  - Extensions: RequestScope, SessionScope
  - Custom scopes

# Scopes

```
class FilterModule extends AbstractModule
{
   protected void configure() {
      bind(OutboundQueue.class)
         .to(SmtpOutboundQueue.class)
         .in(Scopes.SINGLETON);
   }
}
```

# Method injection

```
class Filter
{
   private Whitelist whitelist;
   private OutboundQueue queue;
   private SpamClassifier classifier;

   @Inject
   public void setWhitelist(Whitelist whitelist) {
      this.whitelist = whitelist;
   }

   @Inject
   public void setOutboundQueue(OutboundQueue queue) {
      this.queue = queue;
   }

   ...
}
```

# Constructor vs method injection

- In general, try to use constructor injection
  - Internal references can be marked as final
    - Immutability (remember the Java Concurrency talk?)
- Use method injection when
  - Don't want subclass to know about parent's dependencies
  - Guice can't create your object (e.g., it already exists)
  - Optional injection

# Optional injection

```
class Filter
{
   private final Whitelist whitelist;
   private final OutboundQueue queue;
   private final SpamClassifier classifier;

   private Logger logger = Logger.NULL;

   @Inject
   public Filter(Whitelist whitelist, SpamClassifier classifier,
                 OutboundQueue queue) {
      ...
   }

   @Inject(optional = true)
   public void setLogger(Logger logger) {
      this.logger = logger;
   }

   ...
}
```

# Binding instances

```
void configure() {
   bind(Database.class)
      .toInstance(new MysqlDatabase(...));

   ...
}


class MysqlDatabase implements Database {
   private Logger logger = Logger.NULL;

   public MysqlDatabase(String url, String user,
                        String password) {
      ...
   }

   @Inject
   public void setLogger(Logger logger) {
      this.logger = logger;
   }
}
```

# Binding constants

```
bindConstant()
    .annotatedWith(HttpPort.class)
    .to(System.getProperty("http.port"));


class HttpListener {
    @Inject
    public HttpListener(@HttpPort int port) {
        ...
    }
}
```

# Advanced features

- Type literals, keys
- Injecting providers
- Multibindings
- Private modules
- Integration with other frameworks
  - Jersey
  - Servlets
- Extensions and SPI

# Resources

- Project home
  - http://code.google.com/p/google-guice/
- Google Guice book
  - http://www.apress.com/book/view/1590599977
- Dependency Injection book
  - http://manning.com/prasanna/

# Questions?