IoC Cookbook - Overriding IoC Services

Overriding Tapestry IoC Services

Tapestry is designed to be easy to customize, and the IoC container is the key to that customizability.

One of Tapestry's most important activities is resolving injected objects; that is, when Tapestry is building an object or service and sees a constructor parameter or a field, it must decide what value to plug in. Most of the time, the injected object is a service defined elsewhere within the Tapestry IoC container.

However, there are cases where you might want to override how Tapestry operates in some specific way.

The strategy used to determine what object gets injected is defined inside Tapestry IoC itself; thus we can take advantage of several features of the Tapestry IoC container in order to take control over specific injections.

Contributing a Service Override

In most cases, services are injected by matching just the type; there is no @InjectService annotation, just a method or constructor parameter whose type matches the service's interface.

In this case, it is very easy to supply your own alternate implementation of a service, by *contributing a Service Override* in your module class (usually AppModule.java), like this:

AppModule.java (partial)

```
  @Contribute(ServiceOverride.class)
  public static void setupApplicationServiceOverrides(MappedConfiguration<Class,Object> configuration)
  {
    configuration.addInstance(SomeServiceType.class, SomeServiceTypeOverrideImpl.class);
  }
```

The name of the method is not important, as long as the @Contribute annotation is present on the method.

In this example, we are using `addInstance()` which will instantiate the indicated class and handle dependency resolution. (Be careful with this, because in some cases, resolving dependencies of the override class can require checking against the ServiceOverrides service, and you'll get a runtime exception about ServiceOverrides requiring itself!).

Sometimes you'll want to define the override as a service of its own. This is useful if you want to inject a Logger specific to the service, or if the overriding implementation needs a service configuration:

AppModule.java (partial)

```
  public static void bind(ServiceBinder binder)
  {
    binder.bind(SomeServiceType.class, SomeServiceTypeOverrideImpl.class).withId("SomeServiceTypeOverride");
  }

  @Contribute(ServiceOverride.class)
  public static void setupApplicationServiceOverrides(MappedConfiguration<Class,Object> configuration, @Local
SomeServiceType override)
  {
    configuration.add(SomeServiceType.class, override);
  }
```

Here we're defining a service using the module's `bind()` method.

Every service in the IoC container must have a unique id, that's why we used the `withId()` method; if we we hadn't, the default service id would have been "SomeServiceType" which is a likely conflict with the very service we're trying to override.

We can inject our overriding implementation of SomeServiceType using the special @Local annotation, which indicates that a service within the same module only should be injected (that is, services of the indicated type in other modules are ignored). Without @Local, there would be a problem because the override parameter would need to be resolved using the MasterObjectProvider and, ultimately, the ServiceOverride service; this would cause Tapestry to throw an exception indicating that ServiceOverride depends on itself. We defuse that situation by using @Local, which prevents the MasterObjectProvider service from being used to resolve the override parameter.

Decorating Services

Another option is to decorate the existing service. Perhaps you want to extend some of the behavior of the service but keep the rest.

Alternately, this approach is useful to override a service that is matched using marker annotations.

AppModule.java (partial)

```
  public SomeServiceType decorateSomeServiceType(final SomeServiceType delegate)
  {
    return new SomeServiceType() { . . . };
  }
```

This decorate method is invoked because its name matches the service id of the original service, "SomeServiceType" (you have to adjust the name to match the service id).

The method is passed the original service and its job it to return an *interceptor*, an object that implements the same interface, wrapping around the original service. In many cases, your code will simply re-invoke methods on the delegate, passing the same parameters. However, an interceptor can decide to not invoke methods, or it can change parameters, or change return values, or catch or throw exceptions.

Note that the object passed in as `delegate` may be the core service implementation, or it may be some other interceptor from some other decorator for the same service.

IoC Cookbook - Basic Services and Injection IoC cookbook IoC Cookbook - Patterns