IoC Cookbook - Basic Services and Injection

IoC cookbook IoC Cookbook - Overriding IoC Services

The starting point for Tapestry IOC services and injection is knowing a few conventions: what to name your classes, what packages to put them in and so forth.

In many cases, these conventions are just a little stronger: you may have to do some amount of extra configuration if you choose to go your own way.

Getting Started

As always, you'll first need to choose a package for your application, such as org.example.myapp.

By convention, services go in a sub-package named "services". Tapestry IOC Module class names have a "Module" suffix. Thus, you might start with a module class org.example.myapp.services.MyAppModule.

Simple Services

The simplest services don't have any special configuration or dependencies. They are defined as services so that they can be shared.

For example, the PropertyAccess service is used in multiple places around the framework to access properties of objects (its a wrapper around the Java Beans Introspector and a bit of reflection). This is defined in the TapestryIOCModule.

It's useful to share PropertyAccess, because it does a lot of useful caching internally.

The PropertyAccess service is defined inside TapestryIOCModule's bind() method:

```
  public static void bind(ServiceBinder binder)
  {
    . . .
    binder.bind(PropertyAccess.class, PropertyAccessImpl.class);
    binder.bind(ExceptionAnalyzer.class, ExceptionAnalyzerImpl.class);
    . . .
  }
```

This example includes ExceptionAnalyzer, because it has a dependency on PropertyAccess:

```
public class ExceptionAnalyzerImpl implements ExceptionAnalyzer
{
    private final PropertyAccess propertyAccess;
    public ExceptionAnalyzerImpl(PropertyAccess propertyAccess)
    {
        this.propertyAccess = propertyAccess;
    }

    . . .
}
```

And that's the essence of Tapestry IoC right there; the bind() plus the constructor is *all* that's necessary.

Service Disambiguation

In the previous example, we relied on the fact that only a single service implements the PropertyAccess interface. Had more than one done so, Tapestry would have thrown an exception when the ExceptionAnalyzer service was realized (it isn't until a service is realized that dependencies are resolved).

That's normally okay; in many situations, it makes sense that only a single service implement a particular interface.

But there are often exceptions to the rule, and in those cases, we must provide more information to Tapestry when a service is defined, and when it is injected, in order to disambiguate – to inform Tapestry which particular version of service to inject.

This example demonstrates a number of ideas that we haven't discussed so far, so try not to get too distracted by some of the details. One of the main concepts introduced here is *service builder methods*. These are methods, of a Tapestry IoC Module class, that act as an alternate way to define a service. You often used a service builder method if you are doing more than simply instantiating a class.

A service builder method is a method of a Module, prefixed with the word "build". This defines a service, and dependency injection occurs on the parameters of the service builder method.

The Tapestry web framework includes the concept of an "asset": a resource that may be inside a web application, or packaged inside a JAR. Assets are represented as the type Asset.

In fact, there are different implementations of this class: one for context resources (part of the web application), the other for classpath resources (packaged inside a JAR). The Asset instances are created via AssetFactory services.

Tapestry defines two such services, in the TapestryModule.

```
@Marker(ClasspathProvider.class)
public AssetFactory buildClasspathAssetFactory(ResourceCache resourceCache,

ClasspathAssetAliasManager aliasManager)
{
  ClasspathAssetFactory factory = new ClasspathAssetFactory(resourceCache, aliasManager);

  resourceCache.addInvalidationListener(factory);

  return factory;
}

@Marker(ContextProvider.class)
public AssetFactory buildContextAssetFactory(ApplicationGlobals globals)
{
  return new ContextAssetFactory(request, globals.getContext());
}
```

Service builder methods are used here for two purposes: For the ClasspathAssetFactory, we are registering the new service as a listener of events from another service. For the ContextAssetFactory, we are extracting a value from an injected service and passing *that* to the constructor.

What's important is that the services are differentiated not just in terms of their id (which is defined by the name of the method, after stripping off "build"), but in terms of their *marker annotation*.

The Marker annotation provides the discriminator. When the service type is supplemented with the ClasspathProvider annotation, the ClasspathAssetFactory is injected. When the service type is supplemented with the ContextProvider annotation, the ContextAssetFactory is injected.

Here's an example. Again, we've jumped the gun with this *service contributor method* (we'll get into the why and how of these later), but you can see how Tapestry is figuring out which service to inject based on the presence of those annotations:

```
public void contributeAssetSource(MappedConfiguration<String, AssetFactory> configuration,
    @ContextProvider
    AssetFactory contextAssetFactory,

    @ClasspathProvider
    AssetFactory classpathAssetFactory)
{
  configuration.add("context", contextAssetFactory);
  configuration.add("classpath", classpathAssetFactory);
}
```

This is far from the final word on injection and disambiguation; we'll be coming back to this concept repeatedly. And in later chapters of the cookbook, we'll also go into more detail about the many other concepts present in this example. The important part is that Tapestry *primarily* works off the parameter type (at the point of injection), but when that is insufficient (you'll know ... there will be an error) you can provide additional information, in the form of annotations, to straighten things out.

IoC cookbook   IoC Cookbook - Overriding IoC Services