Injection in Detail

Injection in Tapestry IoC can be a complicated subject for a number of reasons:

- Injection can occur in many places: on fields, and on parameters to methods and constructors of certain objects.
- Parts of Injection are themselves defined in terms of Tapestry IoC services, many of which are extensible.

Despite this, injection generally *Just Works*: most of the time, you want Tapestry to inject a service, and only a single service implements the service interface.

This document discusses what to do when you hit a case that doesn't Just Work, or when you want to extend the injection logic in some way.

Some aspects of this discussion reflect Tapestry IoC used within a Tapestry web application: the tapestry-core module makes some extensions to injection.

Related Articles

- Injection FAQ
- Injection in Detail
- Environmental Services
- Injection

Injection Triggers

Injection is triggered in a number of ways:

- A field in a component class, autobuilt object, or service implementation class is annotated with @Inject.
- A method parameter to a service builder method, a decorator method, or a contribute method (in a Tapestry IoC module class).
- A constructor parameter to an autobuilt object, or a service implementation class.
- Any of the above with an @InjectService annotation.

These define the *point of injection*.

Injection also covers a related matter: providing special resources to a service or component (remember that pages are specialized components). For a service, the service's id (as a string) or extensible configuration (as a Collection, List or Map) may be provided. For a component, the component's id, locale, message catalog, or component resources may be provided.

Standard Injection Processing

This section describes standard injection, which applies at the IoC layer: autobuilt objects and service implementations. The steps for injection into Tapestry components are slightly different and are covered later.

So a the point of injection, Tapestry has identified a field or parameter that should be injected. At this point, Tapestry knows the following:

- The field name (if field injection). The parameter name is not available.
- The field or parameter type, as a Java class. In many cases, this will be enough to identify what object shall be injected.
- Any additional annotations on the field or parameter.

Tapestry proceeds with this information.

Check for @InjectService

Tapestry checks first for the @InjectService annotation. The value of this annotation is the service id to inject. When @InjectService is present at the point of injection, the search is complete, though the injection can fail (throwing an exception) if the service id indicated does not exist, or if the service's interface is not compatible with the field's type.

Check for service resources

This step applies only to IoC layer injection (not to injection into components).

When the @Inject annotation is *not present* at the point of injection, Tapestry checks to see if a resource can be injected. Services are global values, but resources are specific to the service under construction.

When the Inject annotation is present, this step is skipped (this is necessary when the object to be injected has a type that conflicts with a resource type, such as List or Class).

- org.slf4j.Logger – The Logger of the service being constructed (or the logger of the Module class being instantiated).
- ObjectLocator – For contribute methods, used to locate additional objects.
- ServiceResources – For service builder methods, an extended version of ObjectLocator.
- Class – The service interface type.
- OperationTracker – Used to track deeply nested operations so that errors can be reported sensibly.
- Object, or service interface type – Passed to decorator methods.
- Collection, List, Map – Assembled service configurations passed to service builder methods (or service class constructors).

- Configuration, OrderedConfiguration, MappedConfiguration – Configuration passed to contribute methods, to build service configurations.

  If field type does not match any of the available resource types, or the Inject annotation is present, logic continues to the next step.

  > Injection of resources into fields is triggered by the presence of the @InjectResource annotation, whereas injection of resources into parameters occurs when the Inject or InjectService annotation is *not* present. These rules are slightly tricky, which reflects a desire to avoid any annotations except when needed, and the fact that field injection came much later than parameter injection.

Service Lookup by Type and Annotations

Tapestry attempts to find a matching *service*.

First, it generates a set of services whose service interface is compatible with the injection type. This is based on assignability.

If the @Local annotation is present, then services not from the module containing the service being constructed will be eliminated.

Tapestry then works through the known marker annotations. For each marker annotation that is present at the point of injection, Tapestry eliminates services which *do not* have the marker. Thus, if multiple marker annotations are present, the final service must have *all of them*.

At the end, of this, Tapestry determines how many services match.

- If there is a single matching service, then the service to inject as been identified.
- If there are no matches, and there were no marker annotations at the point of injection, then the Tapestry continues to the next step.
- Otherwise there were either no matches, or too many matches: Tapestry will throw a RuntimeException.

MasterObjectProvider Lookup

This is the point at which Tapestry's extensibility comes into play. MasterObjectProvider is a service, with a configuration of ObjectProviders.

The MasterObjectProvider is also the point at which Tapestry's IoC layer injection, and Tapestry's component injection, unite.

As a chain-of-command, each of the following ObjectProviders will be considered and will attempt to identify the object to be injected.

> A common problem when extending injection is that contributions into the MasterObjectProvider configuration have to be handled carefully. Any dependencies of the contributed objects should be resolvable using only the early stages of the injection process, otherwise MasterObjectProvider will have to be instantiated in order to handle its own injection: Tapestry will detect this impossibility and throw an exception. In addition, the TypeCoercer Service is used by several ObjectProvider implementations, so the same restrictions apply to TypeCoercer service contributions.

Value ObjectProvider

Checks for the presence of the @Value annotation. If present, then the annotation's value is evaluated (to expand any symbol references), and the TypeCoercer service is used to convert the resulting String to the injection type (the field or parameter type).

Symbol ObjectProvider

Similar to the Value ObjectProvider: the @Symbol annotation's value (if present) is looked up using the SymbolSource service, and converted to the injection type via the TypeCoercer service.

Autobuild ObjectProvider

Checks to see if the @Autobuild annotation is present and, if so, autobuilds the value for the parameter. Of course, the object being built will itself be configured via injection.

ServiceOverride ObjectProvider

Checks any contributions to the ServiceOverride service. Contributions map a type to an object of that type. Thus, ServiceOverrides will override injections of services that are not qualified with a marker annotation.

Alias ObjectProvider (tapestry-core)

Uses the Alias service (API) to look for an object that can be injected.

> Deprecated in Tapestry 5.2 and removed in 5.4.

This is commonly used to override a built-in service by contributing an object with the exact same interface. This is an older and more complex version of the ServiceOverride provider.

Asset ObjectProvider (tapestry-core)

Checks for the @Path annotation.

If present, the annotation's value has embedded symbols expanded, and is converted into an Asset (which must exist).

The TypeCoercer can then convert the Asset to the injection type, for example, as Resource.

Service ObjectProvider (tapestry-core)

Looks for the @Service annotation; if present, the annotation's value is the exact service id to inject. This is necessary because injections into *component* fields are always triggered by the Inject annotation.

> ⚠ This is supported but no longer necessary, as the @InjectService annotation is now also supported for component fields.

SpringBean ObjectProvider (tapestry-spring)

Attempts to resolve a Spring bean purely by object type (Spring qualifiers are not supported). If no beans are assignable to the type, then processing continues. If exactly one is assignable, it is used as the injection value. If more than one bean is assignable, it is an error (and a list of matching beans names will be part of the thrown exception).

Service Lookup

If none of the ObjectProviders can identify the value to inject, a last step occurs: lookup by service type. If exactly *one* service matches the injection type, then that service is injected.

Otherwise, the lookup fails because either no services match, or more than one matches. An exception will be thrown with the details, including a list of matching services (if there is more than one match).

Post-Injection Methods

Autobuilt objects (services and the like, but *not* components) may have post-injection methods.

Any public method may have the @PostInjection annotation.

Such methods are invoked after constructor and/or field injection. Only **public methods** will be invoked. Any return value is ignored.

The method often takes no parameters; however if the method has parameters, these parameters are new points of injection.

Often this is used to perform additional setup, such as registerring a service as a listener of events produced by another service:

```
public class MyServiceImpl implements MyService, UpdateListener
{
  @PostInjection
  public void registerAsListener(UpdateListenerHub hub)
  {
    hub.addUpdateListener(this);
  }
}
```

Component Injection

Inside Tapestry components, injection occurs exclusively on *fields* and is always triggered by the @Inject (or @InjectService) annotation.

Component field injection is very similar to IoC layer, but with a different set of injectable resources.

Injection is the responsibility of the InjectionProvider service, which is a chain-of-command across a number of implementations.

Block InjectionProvider

Checks if the field type is Block. If so, determines the block id to inject (either from the field name, or from an @Id annotation, if present).

Default InjectionProvider

Uses the MasterObjectProvider service to provide the injectable value. The Service Lookup stage is skipped.

ComponentResources InjectionProvider

Injects fields of type ComponentResources.

CommonResources InjectionProvider

Injects fields with common resources:

- String: the components' complete id
- org.slf4j.Logger: Logger for the component (based on component class name)
- Locale: locale for the containing page (page locale is immutable)
- Messages: Component's message catalog
- ComponentResourceSelector: selector for the containing page (selector is immutable)

> **Added in 5.3**
>
> ComponentResourceSelector is new as of release 5.3. It encapsulates a locale plus additional application-specific data used for skinning and/or themeing.

Asset InjectionProvider

Triggered by the @Path annotation: the Path value has symbols expanded, and is then converted to an Asset.

Service InjectionProvider

Equivalent to the Service Lookup phase in an IoC layer injection.

@InjectService in Components

You may use the @InjectService annotation on component fields.