

Defining Tapestry IOC Services

Services consist of two main parts: a service interface and a service implementation.

The service interface is how the service will be represented throughout the rest of the registry. Since what gets passed around is normally a proxy, you can't expect to cast a service object down to the implementation class (you'll see a `ClassCastException` instead). In other words, you should be careful to ensure that your service interface is complete, since Tapestry IoC effectively walls you off from back doors such as casts.

Service Life Cycle

Every service has a very specific life cycle.

- **Defined:** The service has a definition (from some module) but has not yet been referenced.
- **Virtual:** The service has been referenced, so a proxy for the class has been created.
- **Realized:** A method on the proxy has been invoked, so the service implementation has been instantiated, and any decorators applied.
- **Shutdown:** The entire Registry has been shut down and with it, all the proxies have been disabled.

When the Registry is first created, all modules are scanned and the definitions for all services are created.

Services will be referenced by either accessing them using the Registry, or as dependencies of other realized services.

Tapestry IoC waits until the last possible moment to *realize* the service: that's defined as when a method of the service is invoked. Tapestry is *thread-safe*, so even in a heavily contested, highly threaded environment (such as a servlet container or application server) things *Just Work*.

Service Builder Methods

Tapestry doesn't know how to instantiate and configure your service; instead it relies on you to provide the code to do so, in a service builder method, a method whose name is (or starts with) "build":

```
package org.example.myapp.services;

public class MyAppModule
{
    public static Indexer build()
    {
        return new IndexerImpl();
    }
}
```

Here the service interface is `Indexer` (presumably inside the `org.example.myapp.services` package, since there isn't an import). Tapestry IoC doesn't know about the `IndexerImpl` class (the service implementation of the `Indexer` service), but it does know about the `build()` method.

That's one of the great innovations of Tapestry IoC: we don't try to encapsulate in XML or annotations all the different ways possible to create a service; those things are best expressed in Java code. For a simple case (as here), it would be hard for external configuration (again, in XML or Java annotations) to be shorter than "new `IndexerImpl()`".

The above paragraph was written before Binding and Autobuilding were introduced.

For more complex and realistic scenarios, such as injecting dependencies via the constructor, or doing more interest work (such as registering the newly created service for events published by some other service), the Java code is simply the most direct, flexible, extensible and readable approach.

Binding and Autobuilding

Tapestry IoC can also *autobuild* your service. Autobuilding is the *preferred* way to instantiate your services.

Every module may have an optional, static `bind()` method which is passed a [ServiceBinder](#). Services may be registered with the container by "binding" a service interface to a service implementation:

```
package org.example.myapp.services;

import org.apache.tapestry5.ioc.ServiceBinder;

public class MyAppModule
{
    public static void bind(ServiceBinder binder)
    {
        binder.bind(Indexer.class, IndexerImpl.class);
    }
}
```

You can make repeated calls to `ServiceBinder.bind()`, to bind additional services.

You might ask, "which is better, a builder method for each service, or a `bind()` method for the module?" For simple services, those that are just an instantiated instance with simple dependencies, binding is better than building. That covers at least 90% of all services, so bind away!

There are many cases, however, where constructing a service is more than just instantiating a class. Often the new service will (for example) be registered as a listener with some other service. In other cases, the implementation of the service is generated at runtime. These are where the service builder methods are most useful.

In terms of the evolution of the framework, service builder methods came first, and autobuilding was a later addition, inspired by the terseness of the [Guice](#) IoC container.

Following the convention over configuration principle, the autobuilding of services can be even less verbose. If a service interface is passed as a single argument to the `bind()` method, Tapestry will try to find an implementation in the same package whose name matches the name of the service interface followed by the suffix *Impl*.

```
package org.example.myapp.services;

import org.apache.tapestry5.ioc.ServiceBinder;

public class MyAppModule
{
    public static void bind(ServiceBinder binder)
    {
        binder.bind(Indexer.class);
    }
}
```

Service Ids

Every service will have a unique service id.

When using a service builder method, the service id is the *simple name* of the service interface.

This can be overridden by adding the `@ServiceId` annotation to the service builder method:

```
@ServiceId("FileSystemIndexer")
public static Indexer buildIndexer(@InjectService("FileSystem") FileSystem fileSystem)
{
    . . .
}
```

Another option is to add the service id to the method name, after "build", for example:

```
public static Indexer buildFileSystemIndexer(@InjectService("FileSystem") FileSystem fileSystem)
{
    . . .
}
```

Here, the service id is "FileSystemIndexer" not "Indexer".

For autobuilt services, the service id can be specified by placing the `@ServiceId` annotation directly on a service implementation class.

```
@ServiceId("FileSystemIndexer")
public class IndexerImpl implements Indexer
{
    ...
}
```

When the service is bound, the value of the annotation is used as id:

```
binder.bind(Indexer.class, IndexerImpl.class);
```

This id can be overridden again by calling the method [withId\(String\)](#):

```
binder.bind(Indexer.class, IndexerImpl.class).withId("FileSystemIndexer");
```

Injecting Dependencies

It's pretty unlikely that your service will be able to operate in a total vacuum. It will have other dependencies.

Dependencies are provided to a service in one of several ways:

- As parameters to the service builder method
- As parameters to the service implementation class' constructor (for autobuilt services)
- As parameters passed to the constructor of the service's module class (to be cached inside instance variables)
- Directly into fields of the service implementation

For example, let's say the Indexer needs a JobScheduler to control when it executes, and a FileSystem to access files and store indexes.

```
public static Indexer build(JobScheduler scheduler, FileSystem fileSystem)
{
    IndexerImpl indexer = new IndexerImpl(fileSystem);

    scheduler.scheduleDailyJob(indexer);

    return indexer;
}
```

Tapestry assumes that parameters to builder methods are dependencies; in this example it is able to figure out what services to pass in based just on the type (later we'll see how we can fine tune this with annotations, when the service type is not sufficient to identify a single service).

This is an example of when you would want to use the service builder method, rather than just binding the service interface to the implementation class: because we want to do something extra, in this case, register the new indexer service with the scheduler.

Note that we don't invoke those service builder methods ... we just "advertise" (via naming convention or annotation) that we need the named services. Tapestry IoC will provide the necessary proxies and, when we start to invoke methods on those proxies, will ensure that the full service, including its interceptors and its dependencies, are ready to go. Again, this is done in a thread-safe manner.

What happens if there is more than one service that implements the JobScheduler interface, or the FileSystem interface? You'll see a runtime exception, because Tapestry is unable to resolve it down to a *single* service. At this point, it is necessary to *disambiguate* the link between the service interface and *on* service. One approach is to use the [@InjectService](#) annotation:

```
public static Indexer build(@InjectService("JobScheduler")
    JobScheduler scheduler,

    @InjectService("FileSystem")
    FileSystem fileSystem)
{
    IndexerImpl indexer = new IndexerImpl(fileSystem);

    scheduler.scheduleDailyJob(indexer);

    return indexer;
}
```

If you find yourself injecting the same dependencies into multiple service builder (or service decorator) methods, you can [cache dependency injections](#) in your module, by defining a constructor. This reduces duplication in your module.

Disambiguation with Marker Annotations

In the previous example we were faced with a problem: multiple versions of the `JobScheduler` service. They had the same service interface but unique service ids. If you try to inject based on type, the service to inject will be ambiguous. Tapestry will throw an exception (identifying the parameter type and the matching services that implement that type).

The problem is that when injecting a `JobScheduler` into some other service we need to know which *one* to inject. Rather than using the service id, another approach is to use a *marker annotation*.

You may optionally link a service implementation with a marker annotation.

For example, maybe you have one `JobScheduler` implementation where the jobs are spread across a number of nodes in a cluster, and you have another `JobScheduler` where the jobs are all executed exclusively in the current process.

We can associate those two `JobSchedulers` with two annotations.

```
@Target({
    { PARAMETER, FIELD }}
@Retention(RUNTIME)
@Documented
public @interface Clustered
{
}

@Target({
    { PARAMETER, FIELD }}
@Retention(RUNTIME)
@Documented
public @interface InProcess
{
}

public class MyModule
{
    public static void bind(ServiceBinder binder)
    {
        binder.bind(JobScheduler.class, ClusteredJobSchedulerImpl.class).withId("ClusteredJobScheduler").withMarker(Clustered.class);
        binder.bind(JobScheduler.class, SimpleJobSchedulerImpl.class).withId("InProcessJobScheduler").withMarker(InProcess.class);
    }
}
```

Notice that the marker annotations have no attributes. Further, we support markers on fields (for use in Tapestry components) as well as parameters.

To get the right version of the service, you use one of the annotations:

```
public class MyServiceImpl implements MyService
{
    private final JobScheduler jobScheduler;

    public MyServiceImpl(@Clustered JobScheduler jobScheduler)
    {
        this.jobScheduler = jobScheduler;
    }

    . . .
}
```

The `@Clustered` annotation on the parameter is combined with the parameter type (`JobScheduler`) to find the exact service implementation.

Why is this better than using the service id? It's more refactoring-safe. Service ids can change, which can break your services. However, using an IDE to rename or move an annotation class or service interface will be able to update all the uses of the annotation or interface.

With a service builder method, you use the `@Marker` annotation:

```
@Marker(Clustered.class)
public JobScheduler buildClusteredJobScheduler()
{
    return . . .;
}
```

The `@Marker` annotation may also be placed on an implementation class, which means that you may omit the call to `withMarker()` inside the `bind()` method.

Finally, the point of injection may have multiple marker annotations; only services that are marked with *all* those markers will be considered for injection. Each marker annotation creates an increasingly narrow subset from the set of all possible services (compatible with the indicated dependency type).

Local Dependencies

A special marker interface, `@Local`, indicates a dependency that should only be resolved using services from within *the same module*.

`@Local` can also be combined with other marker annotations.

Injecting Dependencies for Autobuilt Services

With autobuilt services, there's no service builder method in which to specify injections.

Instead, the injections occur on *constructor* for the implementation class:

```
package org.example.myapp.services;

import org.apache.tapestry5.ioc.annotations.InjectService;

public class IndexerImpl implements Indexer
{
    private final FileSystem fileSystem;

    public IndexerImpl(@InjectService("FileSystem") FileSystem fileSystem)
    {
        this.fileSystem = fileSystem;
    }

    . . .
}
```

If the class has multiple constructors, the constructor with the *most* parameters will be invoked. Alternately, you may mark a single constructor with the `Inject` annotation, and Tapestry will use *that* constructor specifically, ignoring all other constructors.

Note how we are using final fields for our dependencies; this is generally a Good Idea. These services will often execute inside a multi-threaded environment, such as a web application, and the use of final fields inside a constructor ensures that the fields will be properly published (meaning, "visible to other threads") in accordance with the Java Memory Model.

Once thing that is not a good idea is to pass in another service, such as `JobScheduler` in the previous example, and pass `this` from a constructor:

```

package org.example.myapp.services;

import org.apache.tapestry5.ioc.annotations.InjectService;

public class IndexerImpl implements Indexer
{
    private final FileSystem fileSystem;

    public IndexerImpl(@InjectService("FileSystem") FileSystem fileSystem,

        @InjectService("JobScheduler") JobScheduler scheduler)
    {
        this.fileSystem = fileSystem;

        scheduler.scheduleDailyJob(this); // Bad Idea
    }

    . . .
}

```

Understanding why this is a bad idea involves a long detour into inner details of the Java Memory Model. The short form is that other threads may end up invoking methods on the `IndexerImpl` instance, and its fields (even though they are final, even though they appear to already have been set) may be uninitialized.

Field Injection

The `@Inject` and `@InjectService` annotations may be used on instance fields of a service implementation class, as an alternative to passing dependencies of the service implementation in via the constructor.

Note that only dependencies are settable this way; if you want resources, including the service's [configuration](#), you must pass those through the constructor. You *are* free to mix and match, injecting partially with field injection and partially with constructor injection.

Caution: injection via fields uses reflection to make the fields accessible. In addition, it may not be as thread-safe as using the constructor to assign to final fields.

```

package org.example.myapp.services;

import org.apache.tapestry5.ioc.annotations.InjectService;

public class IndexerImpl implements Indexer
{
    @InjectService("FileSystem")
    private FileSystem fileSystem;

    . . .
}

```

Defining Service Scope

Each service has a *scope* that controls when the service implementation is instantiated. There are two built in scopes: "singleton" and "perthread", but more can be added.

Service scope is specified using the `@Scope` annotation, which is attached to a builder method, or to the service implementation class. When this annotation is not present, the default scope, "singleton" is used.

singleton

Most services use the default scope, "singleton". With this scope a *proxy* is created when the service is first referenced. By reference, we mean any situation in which the service is requested by name, such as using the `@InjectService` annotation on a service builder method, or by using the [Registry](#) API from outside the container.

In any case, the service proxy will only create the service implementation when a method on the service interface is invoked. Until then, the service can be thought of as "virtual". As the first method is invoked, the service builder method is invoked, then any service decorations occur. This construction process, called "realization", occurs only once.

You should be aware when writing services that your code must be thread safe; any service you define could be invoked simultaneously by multiple threads. This is rarely an issue in practice, since most services take input, use local variables, and invoke methods on other services, without making use of non-final instance variables. The few instance variables in a service implementation are usually references to other Tapestry IoC services.

perthread

The perthread service scope exists primarily to help multi-threaded servlet applications, though it has other applications.

With perthread, the service proxy will delegate to a local service instance that is associated with the current thread. Two different threads, invoking methods on the same proxy, will ultimately be invoking methods on two different service instances, each reserved to their own thread.

This is useful when a service needs to keep request specific state, such as information extracted from the `HttpServletRequest` (in a web application). The default singleton model would not work in such a multithreaded environment. Using perthread on select services allows state to be isolated to those services. Because the dispatch occurs *inside* the proxy, you can treat the service as a global, like any other.

You will see that your service builder method is invoked more than once. It is invoked in each thread where the perthread service is used.

At the end of the request, the Registry's `cleanupThread()` method is invoked; it will discard any perthread service implementations for the current thread.

Caution: A common technique in Tapestry IoC is to have a service builder method register a core service implementation as an event listener with some event hub service. With non-singleton objects, this can cause a number of problems; the event hub will hold a reference to the per-thread instance, even after that per-thread instance has been cleaned up (discarded by the inner proxy). Simply put, this is a pattern to avoid. For the most part, perthread services should be simple holders of data specific to a thread or a request, and should not have overly complex relationships with the other services in the registry.

Defining the scope of Autobuilt Services

There are two options for defining the scope for an autobuilt service.

The service implementation class may include the `@Scope` annotation. This is generally the preferred way to specify scope.

In addition, it is possible to specify the scope when binding the service:

```
bind(MyServiceInterface.class, MyServiceImpl.class).scope(ScopeConstants.PERTHREAD);
```

Eager Loading Services

Services are normally created only as needed (per the scope discussion above).

This can be tweaked slightly; by adding the `@EagerLoad` annotation to the service builder method, Tapestry will instantiate the service when the Registry is first created.

This will cause the service builder method to be invoked, as well as any service decorator methods.

This feature is used when a service manages a resource, such as a thread, that needs to be created as soon as the application starts up. Another common example is a service that listens for events produced by a second service; the first service may need to be created, and start listening, before any of its service methods are invoked (which would normally trigger the instantiation of the service).

Many services may be annotated with `@EagerLoad`; the order in which services are created is not defined.

With the perthread scope, the service builder method will not be invoked (this won't happen until a service method is invoked), but the decorators for the service will be created.

Eager Loading Autobuilt Services

As with service scope, there are two options for indicating that an autobuilt service should be eagerly loaded.

The service implementation class may include the `@EagerLoad` annotation.

You may also specify eager loading explicitly when binding the service:

```
bind(MyServiceInterface.class, MyServiceImpl.class).eagerLoad();
```

Injecting Resources

In addition to injecting services, Tapestry will key off of the parameter type to allow other things to be injected.

- `java.lang.String`: unique id for the service
- [org.slf4j.Logger](#): logger for the service
- `java.lang.Class`: service interface implemented by the service to be constructed
- [ServiceResources](#): access to other services

No annotation is needed for these cases.

See also [service configuration](#) for additional special cases of resources that can be injected.

Note: resources may not be injected into fields, they are injectable only via method or constructor parameters.

Example:

```
public static Indexer build(String serviceId, Log serviceLog,
    JobScheduler scheduler, FileSystem fileSystem)
{
    IndexerImpl indexer = new IndexerImpl(serviceLog, fileSystem);

    scheduler.scheduleDailyJob(serviceId, indexer);

    return indexer;
}
```

The order of parameters is completely irrelevant. They can come first or last or be interspersed however you like.

Injecting in the `ServiceResources` can be handy when you want to calculate the name of a service dependency on the fly. However, in the general case (where the id of service dependencies is known at build time), it is easier to use the `@InjectService` annotation.

The Log's name (used when configuring logging settings for the service) consists of the module class name and the service id separated by a period, i.e. "org.example.myapp.MyModule.Indexer".

Further, `ServiceResources` includes an `autobuild()` method that allows you to easily trigger the construction of a class, including dependencies. Thus the previous example could be rewritten as:

```
public static Indexer build(ServiceResources resources, JobScheduler jobScheduler)
{
    IndexerImpl indexer = resources.autobuild(IndexerImpl.class);

    scheduler.scheduleDailyJob(resources.getServiceId(), indexer);

    return indexer;
}
```

This works the exact same way with `autobuilt` services, except that the parameters of the service implementation constructor are considered, rather than the parameters of the service builder method.

The `@InjectService` annotation takes precedence over these resources.

If the `@InjectService` annotation is not present, and the parameter type does not exactly match a resource type, then [object injection](#) occurs. Object injection will find the correct object to inject based on a number of (extensible) factors, including the parameter type and any additional annotations on the parameter.

Every once and a while, you'll have a conflict between a resource type and an object injection. For example, the following does not work as expected:

```
public static Indexer build(String serviceId, Log serviceLog,
    JobScheduler scheduler, FileSystem fileSystem,
    @Value("${index-alerts-email}")
    String alertEmail)
{
    IndexerImpl indexer = new IndexerImpl(serviceLog, fileSystem, alertEmail);

    scheduler.scheduleDailyJob(serviceId, indexer);

    return indexer;
}
```

It doesn't work because type `String` always gets the service id, as a resource (as with the `serviceId` parameter). In order to get this to work, we need to turn off the resource injection for the `alertEmail` parameter. That's what the `@Inject` annotation does:


```

public static Indexer build(String serviceId, Log serviceLog,
    JobScheduler scheduler, FileSystem fileSystem,
    @Inject @Value("${index-alerts-email}")
    String alertEmail)
{
    IndexerImpl indexer = new IndexerImpl(serviceLog, fileSystem, alertEmail);

    scheduler.scheduleDailyJob(serviceId, indexer);

    return indexer;
}

```

Here, the alertEmail parameter will receive the configured alerts email (see the [symbols documentation](#) for more about this syntax) rather than the service id.

Binding ServiceBuilders

Yet another option is available: instead of binding an interface to a implementation class, you can bind a service to a [ServiceBuilder](#), a callback used to create the service implementation. This is very useful in very rare circumstances.

Builtin Services

A few services within the Tapestry IOC Module are "builtin"; there is no service builder method in the [TapestryIOCModule](#) class.

Service Id	Service Interface
ClassFactory	ClassFactory
LoggerSource	LoggerSource
RegistryShutdownHub	RegistryShutdownHub
PerthreadManager	PerthreadManager

Consult the JavaDoc for each of these services to identify under what circumstances you'll need to use them.

Mutually Dependent Services

One of the benefits of Tapestry IoC's proxy-based approach to just-in-time instantiation is the automatic support for mutually dependent services. For example, suppose that the Indexer and the FileSystem needed to talk directly to each other. Normally, this would cause a "chicken-and-the-egg" problem: which one to create first?

With Tapestry IoC, this is not even considered a special case:

```

public static Indexer buildIndexer(JobScheduler scheduler, FileSystem fileSystem)
{
    IndexerImpl indexer = new IndexerImpl(fileSystem);

    scheduler.scheduleDailyJob(indexer);

    return indexer;
}

public static FileSystem buildFileSystem(Indexer indexer)
{
    return new FileSystemImpl(indexer);
}


```

Here, Indexer and FileSystem are mutually dependent. Eventually, one or the other of them will be created ... let's say its FileSystem. The buildFileSystem() builder method will be invoked, and a proxy to Indexer will be passed in. Inside the FileSystemImpl constructor (or at some later date), a method of the Indexer service will be invoked, at which point, the buildIndexer() method is invoked. It still receives the proxy to the FileSystem service.

If the order is reversed, such that Indexer is built before FileSystem, everything still works the same.

This approach can be very powerful. For example, it can be used to break apart untestable monolithic code into two mutually dependent halves, each of which can be unit tested.

The exception to this rule is a service that depends on itself *during construction*. This can occur when (indirectly, through other services) building the service tries to invoke a method on the service being built. This can happen when the service implementation's constructor invoke methods on service dependencies passed into it, or when the service builder method itself does the same. This is actually a very rare case and difficult to illustrate.

 [Tapestry IoC Modules](#)

 [IOC](#)

[Service Advisors](#) 