Tapestry IoC Modules

You inform Tapestry about your services and contributions by providing a **module** class.

The module class is a plain Java class that you create to inform Tapestry about your services and contributions.

A system of annotations and naming conventions allow Tapestry to determine what services are provided by the module.

A module class exists for the following reasons:

- To *bind* service interfaces to service implementations
- To contribute configuration data *into* services
- To *decorate* services by providing *interceptors* around them
- To provide explicit code for building a service
- To set a default *marker* for all services defined in the module

All public methods of a module class must be meaningful to Tapestry (be one of the categories above). Any extra public methods result in startup exceptions (because the method may contain a typo).

Service Builder Methods

Service builder methods were the original way to define a service and provide the logic to construct it; although this is now more commonly (and succinctly) accomplished using the bind() method, there are still many cases where service builder methods are useful.

Service builder methods are public methods. They are often static. Here's a trivial example:

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

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

Any public method (static or instance) whose name starts with "build" is a service builder method, implicitly defining a service within the module.

Here we're defining a service around the Indexer service interface (presumably also in the org.example.myapp.services package).

Every service has a unique id, used to identify it throughout the Registry of services (the Registry is the combined sum of all services from all modules). If you don't provide an explicit service id, as in this example, the service id is drawn from the return type; this service has an id of "Indexer".

You can give a service an explicit id by adding it to the method name: buildIndexer(). This is useful when you do not want the service id to match the service interface name (for example, when you have different services that implement the same interface), or when you need to avoid name collisions on the method name (Java allows only a single method with a given name and set of parameters, even if the return types are different, so if you have two different service builder methods that take the same parameters, you should give them explicit service ids in the method name).

Tapestry IoC is case insensitive; later we can refer to this service as "indexer" or "INDEXER" or any variation thereof, and connect to this service.

Service ids must be unique; if another module contributes a service with the id "Indexer" (or any case variation thereof) a runtime exception will occur when the Registry is created.

We could extend this example by adding additional service builder methods, or by showing how to inject dependencies. See the service documentation for more details.

Autobuilding Services

Main article: Defining Tapestry IOC Services

An alternate, and usually preferred, way to define a service is via a module's bind() method. The previous example can be rewritten as:

```
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);
  }
}
```

For more details, see see Defining Tapestry IOC Services. In most cases, autobuilding is the *preferred* approach.

Generally speaking, you should always bind and autobuild your services. The only exceptions are when:

- You wish to do more than just instantiate a class; for example, to register the class as an event listener with some other service.
- There is *no implementation class*; in some cases, you can create your implementation on the fly using JDK dynamic proxies or bytecode generation.

The bind() method must be static; an exception is thrown if the bind() method exists but is an instance method.

Caching Services

You will occasionally find yourself in the position of injecting the same services into your service builder or service decorator methods repeatedly (this occurs much less often since the introduction of service autobuilding). This can result in quite a bit of redundant typing. Less code is better code, so as an alternative, you may define a *constructor* for your module that accepts annotated parameters (as with service builder injection).

This gives you a chance to store common services in instance variables for later use inside service builder methods.

```
public class MyModule
{
  private final JobScheduler scheduler;
  private final FileSystem fileSystem;

  public MyModule(JobScheduler scheduler, FileSystem fileSystem)
  {
    this.scheduler = scheduler;
    this.fileSystem = fileSystem;
  }

  public Indexer build()
  {
    IndexerImpl indexer = new IndexerImpl(fileSystem);

    scheduler.scheduleDailyJob(indexer);

    return indexer;
  }
}
```

Notice that we've switched from *static* methods to *instance* methods. Since the builder methods are not static, the MyModule class will be instantiated so that the methods may be invoked. The constructor receives two common dependencies, which are stored into instance fields that may later be used inside service builder methods such as buildIndexer().

This approach is far from required; all the builder methods of your module can be static if you wish. It is used when you have many common dependencies and wish to avoid defining those dependencies as parameters to multiple methods.

Tapestry IoC automatically resolves the parameter type (JobScheduler and FileSystem, in the example) to the corresponding services that implement that type. When there's more than one service that implements the service interface, you'll get an error (but additional annotations and configuration can be used to ensure the correct service injected).

For modules, there are two additional parameter types that are used to refer to *resources* that can be provided to the module instance (rather than *services* which may be injected).

- org.slf4j.Logger: logger for the module (derived from the module's class name)
- ObjectLocator: access to other services

Note that the fields are final: this is important. Tapestry IoC is thread-safe and you largely never have to think about concurrency issues. But in a busy application, different services may be built by different threads simultaneously. Each module class is a singleton, instantiated at most once, and making these fields final ensures that the values are available across multiple threads. Refer to Brian Goetz's Java Concurrency in Practice for a more complete explanation of the relationship between final fields, constructors, and threads ... or just trust us!

Care should be taken with this approach: in some circumstances, you may force a situation in which the module constructor is dependent on itself. For example, if you invoke a method on any injected services defined within the same module from the module class' constructor, then the service implementation will be needed. Creating service implementations requires the module builder instance ... that's a recursive reference.

Tapestry detects these scenarios and throws a runtime exception to prevent an endless loop.

Module Class Implementation Notes

Module classes are designed to be very, very simple to implement.

Again, keep the methods very simple. Use parameter injection to gain access to the dependencies you need.

Be careful about inheritance. Tapestry will see all *public* methods, even those inherited from base classes. Tapestry *only* sees public methods.

By convention, module class names end in Module and are final classes.

You don't *have* to define your methods as static. The use of static methods is only absolutely necessary in a few cases, where the constructor for a module is dependent on contributions from the same module (this creates a chicken-and-the-egg situation that is resolved through static methods).

Default Marker

Services are often referenced by a particular marker interface on the method or constructor parameter. Tapestry will use the intersection of services with that exact marker and assignable by type to find a unique service to inject.

Often, all services in a module should share a marker, this can be specified with a @Marker annotation on the module class. For example, the TapestryIOCModule:

```
@Marker(Builtin.class)
public final class TapestryIOCModule
{
  . . .
```

This references a particular annotation class, Builtin:

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

}
```

The annotation can be applied to method and constructor parameters, for use within the IoC container. It can also be applied to fields, though this is specific to the Tapestry web framework.

Field Injection

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

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.

Using this style, the previous example of a module class may be rewritten:

```
public class MyModule
{
  @Inject
  private JobScheduler scheduler;

  @Inject
  private FileSystem fileSystem;

  public Indexer build()
  {
    IndexerImpl indexer = new IndexerImpl(fileSystem);

    scheduler.scheduleDailyJob(indexer);

    return indexer;
  }
}
```

Tapestry IoC Overview | IOC | Defining Tapestry IOC Services