

Using JSR 330 standard annotations

Added in 5.3

JSR-330 annotations can be used for injection in Tapestry 5.3 and later.

The following table demonstrates that most of annotations in Tapestry IoC and JSR-330 are interchangeable. However, there are few differences in semantics.

Related Articles

- [Injection FAQ](#)
- [Injection in Detail](#)
- [Environmental Services](#)
- [Injection](#)

JSR-330 javax.inject	Tapestry org.apache.tapestry5.ioc.annotations	Comment
@Inject	@Inject	-
@Inject @Named	@InjectService	-
@Scope	@Scope	-
@Qualifier	-	Tapestry marker annotations don't need any qualifier annotations
@Singleton	-	By default all Tapestry services are singletons

Field Injection

Let's start with field injection. In Tapestry the injection into fields is triggered by [@Inject](#) or [@InjectService](#) annotations. When [@Inject](#) annotation is present on a field, Tapestry tries to resolve the object to inject by the type of the field. If several implementations of the same service interface are available in the registry, you have to disambiguate which implementation you want to be injected. This can be done by placing the [@InjectService](#) annotation on the injection point.

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

...

public class AuthenticationFilter implements ComponentRequestFilter {

    @InjectService("HttpBasic")
    private AuthenticationService basicAuthService;

    @InjectService("HttpDigest")
    private AuthenticationService digestAuthService;

    @Inject
    private Response response;

    ...

}
```

Now let's see the JSR-330 equivalent of the same service. As you can see the [@Inject](#) annotations are interchangeable. The difference is how to get a service by its unique id. For this purpose JSR-330 provides the [@Named](#) annotation which accompanies the [@Inject](#) annotation.

```
import javax.inject.Inject;
import javax.inject.Named;

...

public class AuthenticationFilter implements ComponentRequestFilter {

    @Inject @Named("HttpBasic")
    private AuthenticationService basicAuthService;

    @Inject @Named("HttpDigest")
    private AuthenticationService digestAuthService;

    @Inject
    private Response response;

    ...

}
```

Constructor Injection

For constructor injection the `@Inject` annotations are interchangeable. You can use either JSR-330 or Tapestry annotation to mark a constructor for injection. Note that at most one constructor per class may be marked as injection point.

However, the semantics of constructor injection are different in JSR-330 and Tapestry IoC. In JSR-330 a constructor is injectable only if the `@Inject` annotation is present.

```
public class Car {

    public Car() { ... }

    @Inject
    public Car(Engine engine) { ... }

}
```

In Tapestry the `@Inject` annotation for constructors is optional. All available constructors are candidates for injection: the constructor with the most parameters will be invoked.

```
public class Car {

    public Car() { ... }

    public Car(Engine engine) { ... }

}
```

When several constructors are available and you don't want the constructor with most parameters to be injectable, you need to place the `@Inject` annotation.

```
public class Car {

    public Car() { ... }

    @Inject
    public Car(Engine engine) { ... }

    public Car(Engine engine, Logger logger) { ... }

}
```

Injection Into Pages and Components

Inside Tapestry components, injection occurs exclusively on fields. So far the injection was triggered by the [@Inject](#) or [@InjectService](#) annotations. As of version 5.3 the injection points can also be marked with JSR-330 annotations. The following example demonstrates that.

```
public class Index {

    @Inject
    private Request request;

    @javax.inject.Inject
    private ComponentResources resources;

    @javax.inject.Inject
    @Named("FrenchGreeter")
    private Greeter greeter;

    @javax.inject.Inject
    @Symbol(SymbolConstants.PRODUCTION_MODE)
    private boolean productionMode;

    void onActivate() { ... }

}
```

Marker/Qualifier Annotations

Both JSR-330 and Tapestry IoC allow you to disambiguate services by marker or qualifier annotations, as shown in the following example.

```
public class Index {

    @Inject
    @French
    private Greeter greeter;

}
```

Again, there is a slight difference. In JSR-330 a qualifier annotation like [@French](#) in the example above needs to be annotated by the [@Qualifier](#) annotation.

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@javax.inject.Qualifier
public @interface French {
}
```

In Tapestry any annotation can be a marker annotation. You don't need to place something like the [@Qualifier](#) annotation on your marker annotation.

Method Injection

Injectable methods is a next slight difference. In JSR-330 a method is injectable if the [@Inject](#) annotation is present. In Tapestry the [@Inject](#) annotation is optional. An ordinary setter method is a candidate to perform injection.

```
public class Car {

    private Engine engine;

    public void setEngine(Engine engine) {
        this.engine = engine;
    }

}
```

When building a *Car* instance, Tapestry IoC will try to resolve a service of type *Engine*. If available, Tapestry will perform injection by invoking the setter method.

Besides that, module methods are injectable. Again, there is no need to mark the methods with `@Inject` annotation as Tapestry explicitly knows which module methods to invoke. In the following example you can see how to use `@Named` annotation to inject a service by id into a *contribute method*.

```
public class TapestryModule {

    @Contribute(BindingSource.class)
    public static void provideBindings(
        MappedConfiguration<String, BindingFactory> cfg,

        @Named("PropBindingFactory")
        BindingFactory propBindingFactory,

        @Named("MessageBindingFactory")
        BindingFactory messageBindingFactory ) {

        cfg.add(BindingConstants.PROP,
            propBindingFactory);
        cfg.add(BindingConstants.MESSAGE,
            messageBindingFactory);

    }

    ...
}
```

Scopes

By default, a JSR-330 injector creates an instance, uses the instance for one injection, and then forgets it. By placing the `@Scope` annotation you can tell the injector to retain the instance for possible reuse in a later injection. If you want a service to be a singleton, you need to use the `@Singleton` annotation.

In Tapestry, it is exactly the other way around. By default a service is a singleton. Once an instance is created, it is reused for injection. Another available scope is *perthread*, which exists primarily to help multi-threaded servlet applications. If a service has *perthread* scope, it is recreated for every incoming request.