org.springframework.transaction.annotation

Annotation Type EnableTransactionManagement

```
@Target(value=TYPE)
 @Retention(value=RUNTIME)
 @Documented
 @Import(value=TransactionManagementConfigurationSelector.class)
public @interface EnableTransactionManagement
```

Enables Spring's annotation-driven transaction management capability, similar to the support found in Spring's `<tx:*>` XML namespace. To be used on `@Configuration` classes as follows:

```
@Configuration
@EnableTransactionManagement
public class AppConfig {

    @Bean
    public FooRepository fooRepository() {
        // configure and return a class having @Transactional methods
        return new JdbcFooRepository(dataSource());
    }

    @Bean
    public DataSource dataSource() {
        // configure and return the necessary JDBC DataSource
    }

    @Bean
    public PlatformTransactionManager txManager() {
        return new DataSourceTransactionManager(dataSource());
    }
}
```

For reference, the example above can be compared to the following Spring XML configuration:

```
<beans>

    <tx:annotation-driven/>

    <bean id="fooRepository" class="com.foo.JdbcFooRepository">
        <constructor-arg ref="dataSource"/>
    </bean>

    <bean id="dataSource" class="com.vendor.VendorDataSource"/>

    <bean id="transactionManager" class="org.sfwk...DataSourceTransactionManager">
        <constructor-arg ref="dataSource"/>
    </bean>

</beans>
```

In both of the scenarios above, `@EnableTransactionManagement` and `<tx:annotation-driven/>` are responsible for registering the necessary Spring components that power annotation-driven transaction management, such as the TransactionInterceptor and the proxy- or AspectJ-based advice that weave the interceptor into the call stack when `JdbcFooRepository`'s `@Transactional` methods are invoked.

A minor difference between the two examples lies in the naming of the `PlatformTransactionManager` bean: In the `@Bean` case, the name is *"txManager"* (per the name of the method); in the XML case, the name is *"transactionManager"*. The `<tx:annotation-driven/>` is hard-wired to look for a bean named "transactionManager" by default, however `@EnableTransactionManagement` is more flexible; it will fall back to a by-type lookup for any `PlatformTransactionManager` bean in the container. Thus the name can be "txManager", "transactionManager", or "tm": it simply does not matter.

For those that wish to establish a more direct relationship between `@EnableTransactionManagement` and the exact transaction manager bean to be used, the `TransactionManagementConfigurer` callback interface may be implemented - notice the `implements` clause and the `@Override`-annotated method below:

```
@Configuration
@EnableTransactionManagement
public class AppConfig implements TransactionManagementConfigurer {

    @Bean
    public FooRepository fooRepository() {
        // configure and return a class having @Transactional methods
        return new JdbcFooRepository(dataSource());
    }

    @Bean
    public DataSource dataSource() {
        // configure and return the necessary JDBC DataSource
    }

    @Bean
    public PlatformTransactionManager txManager() {
        return new DataSourceTransactionManager(dataSource());
    }

    @Override
    public PlatformTransactionManager annotationDrivenTransactionManager() {
        return txManager();
    }
}
```

This approach may be desirable simply because it is more explicit, or it may be necessary in order to distinguish between two `PlatformTransactionManager` beans present in the same container. As the name suggests, the `annotationDrivenTransactionManager()` will be the one used for processing `@Transactional` methods. See `TransactionManagementConfigurer` Javadoc for further details.

The `mode()` attribute controls how advice is applied; if the mode is `AdviceMode.PROXY` (the default), then the other attributes control the behavior of the proxying.

If the mode() is set to `AdviceMode.ASPECTJ`, then the `proxyTargetClass()` attribute is obsolete. Note also that in this case the `spring-aspects` module JAR must be present on the classpath.

Since:

`3.1`

Author:

`Chris Beams`

See Also:

`TransactionManagementConfigurer`, `TransactionManagementConfigurationSelector`, `ProxyTransactionManagementConfiguration`, `AspectJTransactionManagementConfiguration`

Optional Element Summary

Optional Elements

Modifier and Type	Optional Element and Description
`AdviceMode`	**`mode`** Indicate how transactional advice should be applied.
`int`	**`order`** Indicate the ordering of the execution of the transaction advisor when multiple advices are applied at a specific joinpoint.
`boolean`	**`proxyTargetClass`** Indicate whether subclass-based (CGLIB) proxies are to be created (`true`) as opposed to standard Java interface-based proxies (`false`).

Element Detail

proxyTargetClass

```
public abstract boolean proxyTargetClass
```

Indicate whether subclass-based (CGLIB) proxies are to be created (`true`) as opposed to standard Java interface-based proxies (`false`). The default is `false`. **Applicable only if `mode()` is set to `AdviceMode.PROXY`.**

Note that setting this attribute to `true` will affect *all* Spring-managed beans requiring proxying, not just those marked with `@Transactional`. For example, other beans marked with Spring's `@Async` annotation will be upgraded to subclass proxying at the same time. This approach has no negative impact in practice unless one is explicitly expecting one type of proxy vs another, e.g. in tests.

Default:

`false`

mode

```
public abstract AdviceMode mode
```

Indicate how transactional advice should be applied. The default is `AdviceMode.PROXY`.

See Also:

`AdviceMode`

Default:

`org.springframework.context.annotation.AdviceMode.PROXY`

order

```
public abstract int order
```

Indicate the ordering of the execution of the transaction advisor when multiple advices are applied at a specific joinpoint. The default is `Ordered.LOWEST_PRECEDENCE`.

Default:

`2147483647`