org.springframework.context.annotation

Annotation Type Configuration

```
@Target(value=TYPE)
@Retention(value=RUNTIME)
@Documented
@Component
public @interface Configuration
```

Indicates that a class declares one or more [@Bean](#) methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime, for example:

```
@Configuration
public class AppConfig {

    @Bean
    public MyBean myBean() {
        // instantiate, configure and return bean ...
    }
}
```

Bootstrapping `@Configuration` classes

Via `AnnotationConfigApplicationContext`

`@Configuration` classes are typically bootstrapped using either `AnnotationConfigApplicationContext` or its web-capable variant, `AnnotationConfigWebApplicationContext`. A simple example with the former follows:

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.register(AppConfig.class);
ctx.refresh();
MyBean myBean = ctx.getBean("MyBean.class);
// use myBean ...
```

See the `AnnotationConfigApplicationContext` javadocs for further details, and see `AnnotationConfigWebApplicationContext` for web configuration instructions to a `Servlet` container.

Via Spring `<bean>` XML

As an alternative to registering `@Configuration` classes directly against an `AnnotationConfigApplicationContext`, `@Configuration` classes may be declared as normal `<bean>` definitions within Spring XML files:

```
<beans>
  <context:annotation-config/>
  <bean class="com.acme.AppConfig"/>
</beans>
```

In the example above, `<context:annotation-config/>` is required in order to enable `ConfigurationClassPostProcessor` and other annotation-related post processors that facilitate handling `@Configuration` classes.

Via component scanning

`@Configuration` is meta-annotated with `@Component`, therefore `@Configuration` classes are candidates for component scanning (typically using Spring XML's `<context:component-scan/>` element) and therefore may also take advantage of `@Autowired`/`@Inject` like any regular `@Component`. In particular, if a single constructor is present autowiring semantics will be applied transparently for that constructor:

```
@Configuration
public class AppConfig {

    private final SomeBean someBean;

    public AppConfig(SomeBean someBean) {
        this.someBean = someBean;
    }

    // @Bean definition using "SomeBean"

}
```

`@Configuration` classes may not only be bootstrapped using component scanning, but may also themselves *configure component scanning* using the `@ComponentScan` annotation:

```
@Configuration
@ComponentScan("com.acme.app.services")
public class AppConfig {
  // various @Bean definitions ...
}
```

See the `@ComponentScan` javadocs for details.

Working with externalized values

Using the `Environment` API

Externalized values may be looked up by injecting the Spring `Environment` into a `@Configuration` class — for example, using the `@Autowired` annotation:

```
@Configuration
public class AppConfig {

    @Autowired Environment env;

    @Bean
    public MyBean myBean() {
        MyBean myBean = new MyBean();
        myBean.setName(env.getProperty("bean.name"));
        return myBean;
    }
}
```

Properties resolved through the `Environment` reside in one or more "property source" objects, and `@Configuration` classes may contribute property sources to the `Environment` object using the `@PropertySource` annotation:

```
@Configuration
@PropertySource("classpath:/com/acme/app.properties")
public class AppConfig {

    @Inject Environment env;

    @Bean
    public MyBean myBean() {
        return new MyBean(env.getProperty("bean.name"));
    }
}
```

See the `Environment` and `@PropertySource` javadocs for further details.

Using the `@Value` annotation

Externalized values may be injected into `@Configuration` classes using the `@Value` annotation:

```
@Configuration
@PropertySource("classpath:/com/acme/app.properties")
public class AppConfig {

    @Value("${bean.name}") String beanName;

    @Bean
    public MyBean myBean() {
        return new MyBean(beanName);
    }
}
```

This approach is often used in conjunction with Spring's `PropertySourcesPlaceholderConfigurer` that can be enabled automatically in XML configuration via `<context:property-placeholder/>` or explicitly in a `@Configuration` class via a dedicated `static` `@Bean` method (see "a note on BeanFactoryPostProcessor-returning `@Bean` methods" of `@Bean`'s javadocs for details). Note, however, the explicit registration of a `PropertySourcesPlaceholderConfigurer` via a `static` `@Bean` method is typically only required if you need to customize configuration such as the placeholder syntax, etc. Specifically, if no bean post-processor (such as a `PropertySourcesPlaceholderConfigurer`) has registered an *embedded value resolver* for the `ApplicationContext`, Spring will register a default *embedded value resolver* which resolves placeholders against property sources registered in the `Environment`. See the section below on composing `@Configuration` classes with Spring XML using `@ImportResource`, see the `@Value` javadocs for details on working with `BeanFactoryPostProcessor` types such as `PropertySourcesPlaceholderConfigurer`.

Composing `@Configuration` classes

With the `@Import` annotation

`@Configuration` classes may be composed using the `@Import` annotation, similar to the way that `<import/>` works in Spring XML. Because `@Configuration` objects are managed as Spring beans within the container, imported configurations may be injected — for example, via constructor injection:

```
@Configuration
public class DatabaseConfig {

    @Bean
    public DataSource dataSource() {
        // instantiate, configure and return DataSource
    }
}

@Configuration
@Import({DatabaseConfig.class)
public class AppConfig {

    private final DatabaseConfig dataConfig;

    public AppConfig(DatabaseConfig dataConfig) {
        this.dataConfig = dataConfig;
    }

    @Bean
    public MyBean myBean() {
        // reference the dataSource() bean method
        return new MyBean(dataConfig.dataSource());
    }
}
```

Now both `AppConfig` and the imported `DatabaseConfig` can be bootstrapped by registering only `AppConfig` against the Spring context:

```
new AnnotationConfigApplicationContext(AppConfig.class);
```

With the `@Profile` annotation

`@Configuration` classes may be marked with the `@Profile` annotation to indicate they should be processed only if a given profile or profiles are active:

```
@Profile("development")
@Configuration
public class EmbeddedDatabaseConfig {

    @Bean
    public DataSource dataSource() {
        // instantiate, configure and return embedded DataSource
    }
}

@Profile("production")
@Configuration
public class ProductionDatabaseConfig {

    @Bean
    public DataSource dataSource() {
        // instantiate, configure and return production DataSource
    }
}
```

Alternatively, you may also declare profile conditions at the `@Bean` method level — for example, for alternative bean variants within the same configuration class:

```
@Configuration
public class ProfileDatabaseConfig {

    @Bean("dataSource")
    @Profile("development")
    public DataSource embeddedDatabase() { ... }

    @Bean("dataSource")
    @Profile("production")
    public DataSource productionDatabase() { ... }
}
```

See the `@Profile` and `Environment` javadocs for further details.

With Spring XML using the `@ImportResource` annotation

As mentioned above, `@Configuration` classes may be declared as regular Spring `<bean>` definitions within Spring XML files. It is also possible to import Spring XML configuration files into `@Configuration` classes using the `@ImportResource` annotation. Bean definitions imported from XML can be injected — for example, using the `@Inject` annotation:

```
@Configuration
@ImportResource("classpath:/com/acme/database-config.xml")
public class AppConfig {

    @Inject DataSource dataSource; // from XML

    @Bean
    public MyBean myBean() {
        // inject the XML-defined dataSource bean
        return new MyBean(this.dataSource);
    }
}
```

With nested `@Configuration` classes

`@Configuration` classes may be nested within one another as follows:

```
@Configuration
public class AppConfig {

    @Inject DataSource dataSource;

    @Bean
    public MyBean myBean() {
        return new MyBean(dataSource);
    }

    @Configuration
    static class DatabaseConfig {
        @Bean
        DataSource dataSource() {
            return new EmbeddedDatabaseBuilder().build();
        }
    }
}
```

When bootstrapping such an arrangement, only `AppConfig` need be registered against the application context. By virtue of being a nested `@Configuration` class, `DatabaseConfig` *will be registered automatically*. This avoids the need to use an explicit `@Import` annotation when the relationship between `AppConfig` and `DatabaseConfig` is already implicitly clear.

Note also that nested `@Configuration` classes can be used to good effect with the `@Profile` annotation to provide two options of the same bean to the enclosing `@Configuration` class.

Configuring lazy initialization

By default, `@Bean` methods will be *eagerly instantiated* at container bootstrap time. To avoid this, `@Configuration` may be used in conjunction with the `@Lazy` annotation to indicate that all `@Bean` methods declared within the class are by default lazily initialized. Note that `@Lazy` may be used on individual `@Bean` methods as well.

Testing support for `@Configuration` classes

The Spring *TestContext framework* available in the `spring-test` module provides the `@ContextConfiguration` annotation which can accept an array of *component* class references — typically `@Configuration` or `@Component` classes.

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {AppConfig.class, DatabaseConfig.class})
public class MyTests {

    @Autowired MyBean myBean;

    @Autowired DataSource dataSource;

    @Test
    public void test() {
        // assertions against myBean ...
    }
}
```

See the `TestContext framework` reference documentation for details.

Enabling built-in Spring features using `@Enable` annotations

Spring features such as asynchronous method execution, scheduled task execution, annotation driven transaction management, and even Spring MVC can be enabled and configured from `@Configuration` classes using their respective `@Enable*` annotation. See `@EnableAsync`, `@EnableScheduling`, `@EnableTransactionManagement`, `@EnableAspectJAutoProxy`, and `@EnableWebMvc` for details.

Constraints when authoring `@Configuration` classes

- Configuration classes must be provided as classes (i.e. not as instances returned from factory methods), allowing for runtime enhancements through a generated subclass.
- Configuration classes must be non-final (allowing for subclasses at runtime), unless the `proxyBeanMethods` flag is set to `false` in which case no runtime-generated subclass is necessary.
- Configuration classes must be non-local (i.e. may not be declared within a method).
- Any nested configuration classes must be declared as `static`.
- `@Bean` methods may not in turn create further configuration classes (any such instances will be treated as regular beans, with their configuration annotations remaining undetected).

Since:
3.0

Author:
Rod Johnson, Chris Beams, Juergen Hoeller

See Also:
`Bean`, `Profile`, `Import`, `ImportResource`, `ComponentScan`, `Lazy`, `PropertySource`, `AnnotationConfigApplicationContext`, `ConfigurationClassPostProcessor`, `Environment`, `ContextConfiguration`

Optional Element Summary

Optional Elements

Modifier and Type	Optional Element and Description
`boolean`	`proxyBeanMethods` Specify whether `@Bean` methods should get proxied in order to enforce bean lifecycle behavior, e.g.
`String`	`value` Explicitly specify the name of the Spring bean definition associated with the `@Configuration` class.

Element Detail

value

```
@AliasFor(annotation=Component.class)
public abstract String value
```

Explicitly specify the name of the Spring bean definition associated with the `@Configuration` class. If left unspecified (the common case), a bean name will be automatically generated.

The custom name applies only if the `@Configuration` class is picked up via component scanning or supplied directly as an `AnnotationConfigApplicationContext`. If the `@Configuration` class is registered as a traditional XML bean definition, the name/id of the bean element will take precedence.

Returns:
the explicit component name, if any (or empty String otherwise)

See Also:
`AnnotationBeanNameGenerator`

Default:
""

proxyBeanMethods

```
public abstract boolean proxyBeanMethods
```

Specify whether `@Bean` methods should get proxied in order to enforce bean lifecycle behavior, e.g. to return shared singleton bean instances even in case of direct `@Bean` method calls in user code. This feature requires method interception, implemented through a runtime-generated CGLIB subclass which comes with limitations such as the configuration class and its methods not being allowed to declare `final`.

The default is `true`, allowing for 'inter-bean references' via direct method calls within the configuration class as well as for external calls to this configuration's `@Bean` methods, e.g. from another configuration class. If this is not needed since each of this particular configuration's `@Bean` methods is self-contained and designed as a plain factory method for container use, switch this flag to `false` in order to avoid CGLIB subclass processing.

Turning off these method interception effectively processes `@Bean` methods individually like when declared on non-`@Configuration` classes, a.k.a. "@Bean Lite Mode" (see `@Bean`'s javadoc). It is therefore behaviorally equivalent to removing the `@Configuration` stereotype.

Since:
5.2

Default:
true