JavaConfig with Component Scan

To configure Spring container with our beans, we can mix XML's <context:component-scan> with JavaConfig configuration. We can even avoid XML altogether by using @ComponentScan.

With @ComponentScan we can still mix the factory approach . In factory approach we annotate classes with @Configuration having methods annotated with @Bean which return bean instances.

For component scanning to work we must annotate our @Configuration class with @ComponentScan and annotate our bean classes with one of the stereotype annotations

- Component
- Controller
- Repository
- Service

Classes annotated with one of the above are candidate for spring container registration when using scanning. The most important is Component annotation. The rest are specialization of Component. Each one is annotated with Component itself. They represent the roles in the overall application design.

you can annotate your component classes with @Component, but by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. It is also possible that @Repository, @Service, and @Controller may carry additional semantics in future releases of the

Spring Framework. Thus, if you are choosing between using @Component or @Service for your service layer, @Service is clearly the better choice.

Along with above annotations we can tag our beans with any of @Lazy, @DependsOn, @Scope etc to specify their specific behavior.

Spring Stereotype Annotations are not Inherited

None of the stereotype annotations are tagged with @Inherited. That means we cannot expect sub classes to inherit the stereotype annotations. We have to add them explicitly to each sub classes. Same is true for other annotations like Scope, Lazy etc. As per general Java concept, we can make use of method/fields/constructor level annotation from super classes, i.e. we don't have to repeat them in subclasses. In case of method overriding (typically setters in Spring), we have to explicitly add annotations in overridden methods, even though they are already present in the original super class methods.

Example

A singleton bean

```
@Component
public class MySingletonBean {

  @PostConstruct
  public void init() {
      System.out.println("initializing " +
              this.getClass().getSimpleName());
  }
}
```

Service beans

```
public interface MyService {
  String getMessage();
}


@Lazy
@Service("basic-service")
public class ServiceImplA implements MyService {


  @PostConstruct
  private void init() {
      System.out.println("initializing lazily " +
              this.getClass().getSimpleName());
  }


  @Override
  public String getMessage() {
      return "Message from " + getClass().getSimpleName();
  }
}


@Service
public class ServiceImplB implements MyService {
  @PostConstruct
  private void init(){
      System.out.println("initializing at start up " +
              this.getClass().getSimpleName());
  }
  @Override
  public String getMessage() {
      return "Message from "+getClass().getSimpleName();
  }
}
```

A prototype bean

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class MyPrototypeBean {

  @Autowired
  @Qualifier("basic-service")
  private MyService myService;

  public void doSomething(){
      System.out.println(myService.getMessage());
  }
}
```

@Configuration class with @ComponentScan

```
@Configuration
@ComponentScan("com.piseth.java.school.bean")
public class AppConfig {

  public static void main(String... strings) {
      AnnotationConfigApplicationContext context =
              new AnnotationConfigApplicationContext(AppConfig.class);
      System.out.println("-- Spring container started and is ready --");
      MyPrototypeBean bean = context.getBean(MyPrototypeBean.class);
      bean.doSomething();
  }
}
```

Output

```
initializing MySingletonBean
initializing at start up ServiceImplB
-- Spring container started and is ready --
initializing lazily ServiceImplA
Message from ServiceImplA
```

1-Specifying packages to be scanned with basePackages attribute of @ComponentScan

@ComponentScan#basePackages specifies packages to scan for annotated component. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation.

The basePackages attribute is an array of String so we can define multiple packages. e.g.

@ComponentScan(basePackages = {"com.piseth.java.school.client", "com.
piseth.java.school.service"})

Alternatively, we can specify a comma- or semicolon- or space-separated list of packages (since spring-context 4.1.1.RELEASE):

@ComponentScan(basePackages = "com.piseth.java.school.client,
com.piseth.java.school.service")

@ComponentScan(basePackages =
"com.piseth.java.school.client;com.piseth.java.school.service")

@ComponentScan(basePackages = "com.piseth.java.school.client
com.piseth.java.school.service")

@ComponentScan(basePackages = {"com.piseth.java.school.client
com.piseth.java.school.service"})

Example

Creating beans

package com.piseth.java.school.service;

```
@Service
public class RetailOrderService {

  public void placeOrder(String item) {
      System.out.printf("Retail order placed. Item: %s%n", item);
  }
}
package com.piseth.java.school.service;

@Service
public class WholeSaleOrderService {

  public void placeOrder(String item, int quantity) {
      if (quantity < 10) {
          throw new IllegalArgumentException(
                  "Quantity must be more than 10  for a wholesale order");
      }
      System.out.printf("Wholesale order placed. Item: %s Quantity: %s%n", item, 
quantity);
  }
}
```

```
package com.piseth.java.school.client;
@Component
public class Buyer {
  @Autowired
  private RetailOrderService orderService;

  public void buySomething() {
      orderService.placeOrder("Laptop");
  }
}
package com.piseth.java.school.client;

@Component
public class Wholesaler {
  @Autowired
  private WholeSaleOrderService wholeSaleOrderService;

  public void buySomethingInBulk() {
      wholeSaleOrderService.placeOrder("Car", 100);
  }
}
```

Main class

```
package com.piseth.java.school.app;


@Configuration
@ComponentScan(basePackages =
{"com.piseth.java.school.client","com.piseth.java.school.service"})
public class OnlineOrderApp {


  public static void main(String... strings) {
      AnnotationConfigApplicationContext context =
              new AnnotationConfigApplicationContext(OnlineOrderApp.class);
      System.out.println("-- Spring container started and is ready --");
      Buyer buyer = context.getBean(Buyer.class);
      buyer.buySomething();


      Wholesaler wholesaler = context.getBean(Wholesaler.class);
      wholesaler.buySomethingInBulk();
  }
}
```

Output

```
-- Spring container started and is ready --
Retail order placed. Item: Laptop
Wholesale order placed. Item: Car Quantity: 100
```

2 - Using basePackageClasses Attribute of @ComponentScan

@ComponentScan#basePackageClasses is a type-safe alternative to basePackages() for specifying the packages to scan for annotated components. The whole package of each class specified will be scanned.

Spring recommends to create a special no-op marker class or interface in each package that serves no purpose other than being referenced by this attribute.

Example

Creating beans in package com.piseth.java.school.service

```
package com.piseth.java.school.service;


@Service

public class RetailOrderService {


  public void placeOrder(String item) {

      System.out.printf("Retail order placed. Item: %s%n", item);

  }

}

package com.piseth.java.school.service;


@Service

public class WholeSaleOrderService {
  public void placeOrder(String item, int quantity) {
      if (quantity < 10) {
          throw new IllegalArgumentException(
                  "Quantity must be more than 10  for a wholesale order");
      }
      System.out.printf("Wholesale order placed. Item: %s Quantity: %s%n", item, 
quantity);
  }
}
```

Creating beans in package com.piseth.java.school.client

```
package com.piseth.java.school.client;


@Component
public class Consumer {
  @Autowired
  private RetailOrderService orderService;


  public void buySomething() {
      orderService.placeOrder("Laptop");
  }
}
package com.piseth.java.school.client;
```

```
@Component
public class Wholesaler {
  @Autowired
  private WholeSaleOrderService wholeSaleOrderService;

  public void buySomethingInBulk() {
      wholeSaleOrderService.placeOrder("Car", 100);
  }
}
```

Creating marker interface in both packages

```
package com.piseth.java.school.service;

public interface OnlineOrderService {
}
package com.piseth.java.school.client;

public interface OnlineOrderClient {
}
```

Main class

```
package com.piseth.java.school.app;
```

```
@Configuration
@ComponentScan(basePackageClasses = {OnlineOrderClient.class,
OnlineOrderService.class})
public class OnlineOrderApp {

  public static void main(String... strings) {
      AnnotationConfigApplicationContext context =
              new AnnotationConfigApplicationContext(OnlineOrderApp.class);
      System.out.println("-- Spring container started and is ready --");
      Consumer consumer = context.getBean(Consumer.class);
      consumer.buySomething();

      Wholesaler wholesaler = context.getBean(Wholesaler.class);
      wholesaler.buySomethingInBulk();
  }
}
```

Output

```
-- Spring container started and is ready --
Retail order placed. Item: Laptop
Wholesale order placed. Item: Car Quantity: 100
```

In above example if we don't create and specify special marker interfaces in each packages and just specify bean classes for basePackageClasses, that will work too and will produce the same output:

```
@Configuration

@ComponentScan(basePackageClasses = {RetailOrderService.class, Consumer.class})

public class OnlineOrderApp {

 ...

}
```

3 - Using Filters To Customize Scanning with @ComponentScan

By default, @ComponentScan scans all classes which are annotated with @Component, @Repository, @Service, @Controller, @Configuration, or a custom annotation that itself is annotated with @Component.

We can modify and extend this behavior by applying custom filters via @ComponentScan#includeFilters or @ComponentScan#excludeFilters attributes.

Following shows ComponentScan annotation snippet along with include/exclude filter attributes and their type (i.e. Filter) definition:

```
package org.springframework.context.annotation;

.....

.....

public @interface ComponentScan {

    .....

    ComponentScan.Filter[] includeFilters() default {};


    ComponentScan.Filter[] excludeFilters() default {};


    .....

    public @interface Filter {

        FilterType type() default FilterType.ANNOTATION;
        @AliasFor("classes")

        Class<?>[] value() default {};


        @AliasFor("value")

        Class<?>[] classes() default {};


        String[] pattern() default {};

    }

}
```

Following shows FilterType snippet:

```
package org.springframework.context.annotation;


public enum FilterType {

    public enum FilterType {


          //Filter candidates marked with a given annotation.

          ANNOTATION,


          //Filter candidates assignable to a given type.

          ASSIGNABLE_TYPE,


        //Filter candidates matching a given AspectJ type pattern expression.
          ASPECTJ,


        //Filter candidates matching a given regex pattern.

          REGEX,


          //Filter candidates using a given custom TypeFilter implementation.

          CUSTOM

}
```

4 - Using @ComponentScan to scan non component classes via includeFilters attribute

@ComponentScan#includeFilters can be used to scan classes which are not annotated with @Component and other stereotype annotation. It can be done by specifying a Filter with type as FilterType#ASSIGNABLE_TYPE and classes as non component classes to be scanned. For example:

```
@ComponentScan(includeFilters = @ComponentScan.Filter(
        type = FilterType.ASSIGNABLE_TYPE, classes = {OrderService.class}))
```

Example

A non component class

```
package com.piseth.java.school.example;

public class OrderService {

  public void placeOrder(String item) {
      System.out.printf("Retail order placed. Item: %s%n", item);
  }
}
```

A component class

```
package com.piseth.java.school.example;

@Component
public class Buyer {
  @Autowired
  private OrderService orderService;

  public void buySomething() {
      orderService.placeOrder("Laptop");
  }
}
```

Main class

```
package com.piseth.java.school.example;

@Configuration
@ComponentScan(includeFilters = @ComponentScan.Filter(
      type = FilterType.ASSIGNABLE_TYPE, classes = {OrderService.class}))
public class OnlineOrderApp {

  public static void main(String... strings) {
      AnnotationConfigApplicationContext context =
              new AnnotationConfigApplicationContext(OnlineOrderApp.class);
      System.out.println("-- Spring container started and is ready --");
      Buyer buyer = context.getBean(Buyer.class);
      buyer.buySomething();
  }
}
```

Output

```
-- Spring container started and is ready --
Retail order placed. Item: Laptop
```

5 - Using excludeFilters attribute of @ComponentScan to exclude component classes

@ComponentScan#excludeFilters can be used to exclude component classes from scanning. For example

```
@ComponentScan(basePackages =
"com.piseth.java.school.example.client;com.piseth.java.school.example.service",
        excludeFilters = @ComponentScan.Filter(
                type = FilterType.ASSIGNABLE_TYPE,
                classes = {WholeSaleOrderService.class, Wholesaler.class})
```

Example

Beans

```
package com.piseth.java.school.example.client;


@Component
public class RetailBuyer {
  @Autowired
  private RetailOrderService orderService;


  public void buySomething() {
      orderService.placeOrder("Laptop");
  }
}
```

```
package com.piseth.java.school.example.client;


@Component
public class Wholesaler {
  @Autowired
  private WholeSaleOrderService wholeSaleOrderService;


  public void buySomethingInBulk() {
      wholeSaleOrderService.placeOrder("Car", 100);
  }
}



package com.piseth.java.school.example.service;


@Service
public class RetailOrderService {


  public void placeOrder(String item) {
      System.out.printf("Retail order placed. Item: %s%n", item);
  }
}
package com.piseth.java.school.example.service;
```

```
@Service
public class WholeSaleOrderService {

  public void placeOrder(String item, int quantity) {
      if (quantity < 10) {
          throw new IllegalArgumentException(
                  "Quantity must be more than 10  for a wholesale order");
      }
      System.out.printf("Wholesale order placed. Item: %s Quantity: %s%n", item,
quantity);
  }
}
```

Main class

```
package com.piseth.java.school.example.app;

@Configuration
@ComponentScan(basePackages =
"com.piseth.java.school.example.client;com.piseth.java.school.example.service",
      excludeFilters = @ComponentScan.Filter(
              type = FilterType.ASSIGNABLE_TYPE,
              classes = {WholeSaleOrderService.class, Wholesaler.class})

)
```

```
public class OnlineOrderApp {


  public static void main(String... strings) {
      AnnotationConfigApplicationContext context =
              new AnnotationConfigApplicationContext(OnlineOrderApp.class);
      System.out.println("-- Spring container started and is ready --");
      RetailBuyer retailBuyer = context.getBean(RetailBuyer.class);
      retailBuyer.buySomething();


      boolean wholeSaleOrderServiceRegistered =
              context.containsBean("wholeSaleOrderService");
      System.out.println("wholeSaleOrderService registered: "+
wholeSaleOrderServiceRegistered);
  }
}
```

Output

```
-- Spring container started and is ready --
Retail order placed. Item: Laptop
wholeSaleOrderService registered: false
```

6 - Using @ComponentScan#includeFilters to scan non component classes based on annotations

Following example shows how to use @ComponentScan#includeFilters attribute along with @ComponentScan.Filter=FilterType.ANNOTATION to scan non-component classes based on user defined annotation. The annotation class is specified by @Filter#classes attribute.

Example

The annotation

```
package com.piseth.java.school;


@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.TYPE)

public @interface IncludeInScan {

  String value() default "";

}
```

Non-component class using the annotation

```
package com.piseth.java.school;


@IncludeInScan

public class OrderService {


  public void placeOrder(String item) {

      System.out.printf("Retail order placed. Item: %s%n", item);
  }
}
```

Another component class

```
@Component
public class Buyer {
  @Autowired
  private OrderService orderService;

  public void buySomething() {
      orderService.placeOrder("Laptop");
  }
}
```

Main configuration class

```
package com.piseth.java.school;

@Configuration
@ComponentScan(includeFilters = @ComponentScan.Filter(
      type = FilterType.ANNOTATION, classes = {IncludeInScan.class}))
public class OnlineOrderApp {

  public static void main(String... strings) {
      AnnotationConfigApplicationContext context =
              new AnnotationConfigApplicationContext(OnlineOrderApp.class);
      System.out.println("-- Spring container started and is ready --");
      Buyer buyer = context.getBean(Buyer.class);
      buyer.buySomething();
  }
}
```

Output

```
-- Spring container started and is ready --
Retail order placed. Item: Laptop
```

7 - Using @ComponentScan#excludeFilters to exclude classes from scanning based on annotations

Following example shows how to use @ComponentScan#excludeFilters attribute along with @ComponentScan.Filter=FilterType.ANNOTATION to exclude component classes from scanning based on user defined annotation. The annotation class is specified by @Filter#classes attribute.

Example

The annotation

```
package com.piseth.java.school.app;


@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface ExcludeFromScan {
  String value () default "";
}
```

Using the annotation on component classes

```
package com.piseth.java.school.service;


@Service
@ExcludeFromScan
public class WholeSaleOrderService {


  public void placeOrder(String item, int quantity) {
    if (quantity < 10) {
      throw new IllegalArgumentException(
          "Quantity must be more than 10  for a wholesale order");
    }
    System.out.printf("Wholesale order placed. Item: %s Quantity: %s%n", item,
quantity);
  }
}
package com.piseth.java.school.client;


@Component
@ExcludeFromScan
public class Wholesaler {
  @Autowired
  private WholeSaleOrderService wholeSaleOrderService;


  public void buySomethingInBulk() {
    wholeSaleOrderService.placeOrder("Car", 100);
  }
}
```

Other component classes

```
package com.piseth.java.school.service;


@Service
```

```
public class RetailOrderService {


  public void placeOrder(String item) {
      System.out.printf("Retail order placed. Item: %s%n", item);
  }
}
package com.piseth.java.school.client;


@Component
public class RetailBuyer {
  @Autowired
  private RetailOrderService orderService;


  public void buySomething() {
      orderService.placeOrder("Laptop");
  }
}
```

Main configuration class

```
package com.piseth.java.school.app;
```

```
@Configuration

@ComponentScan(basePackages =
"com.piseth.java.school.client;com.piseth.java.school.service",

      excludeFilters = @ComponentScan.Filter(

              type = FilterType.ANNOTATION,

              classes = ExcludeFromScan.class

      )


)

public class OnlineOrderApp {


  public static void main(String... strings) {

      AnnotationConfigApplicationContext context =

              new AnnotationConfigApplicationContext(OnlineOrderApp.class);

      System.out.println("-- Spring container started and is ready --");

      RetailBuyer retailBuyer = context.getBean(RetailBuyer.class);

      retailBuyer.buySomething();


      boolean wholeSaleOrderServiceRegistered =

              context.containsBean("wholeSaleOrderService");

      System.out.println("WholeSaleOrderService registered: " +
wholeSaleOrderServiceRegistered);
```

```
        boolean wholesaler =
                context.containsBean("wholesaler");
        System.out.println("Wholesaler registered: " + wholesaler);
    }
}
```

Output

```
-- Spring container started and is ready --
Retail order placed. Item: Laptop
WholeSaleOrderService registered: false
Wholesaler registered: false
```

8 - Registering beans within @Component classes

Classes annotated with any of the stereotype component annotations (@Component, @Service, @Repository etc), can also expose new bean definitions using @Bean annotation on their methods.

@Component classes are not CGLIB proxied

Classes annotated with @Configuration are CGLIB proxied. @Component classes, however, are not enhanced with CGILIB to intercept the bean method invocation, that means a direct call will not route through the container and will return a new instance every time.

In the component classes, we can use any valid annotation along with @Bean method and it's parameters, exactly the same way we use them in @Configuration classes (e.g. @Lazy, @Qualifier, @Primary etc).

Lite @Beans mode

When @Bean methods' enclosing class is not annotated with @Configuration like in above @Component case, they are said to operating in a 'lite' mode.

According to Spring reference doc :

Full @Configuration vs 'lite' @Beans mode?

When @Bean methods are declared within classes that are not annotated with @Configuration they are referred to as being processed in a 'lite' mode. For example, bean methods declared in a @Component or even in a plain old class will be considered 'lite'.

Unlike full @Configuration, lite @Bean methods cannot easily declare inter-bean dependencies. Usually one @Bean method should not invoke another @Bean method when operating in 'lite' mode.

Only using @Bean methods within @Configuration classes is a recommended approach of ensuring that 'full' mode is always used. This will prevent the same @Bean method from accidentally being invoked multiple times and helps to reduce subtle bugs that can be hard to track down when operating in 'lite' mode.

@Configuration classes can also be scanned

We can even use @Configuration classes instead of @Component classes to achieve the same. @Configuration basically itself is a component (it's annotated with @Component). In that case, the methods annotated with @Bean will be CGILIB enhanced and will be proxied.

AnnotationConfigApplicationContext and @Component classes

AnnotationConfigApplicationContext is an implementation of ApplicationContext to bootstrap the Spring container. Other than @Configuration classes, AnnotationConfigApplicationContext is capable to accept the class(s) annotated with @Component (or any other stereotype) as well, but as mentioned above doing so will cause the beans to operate in lite mode.

Possible use cases

The primary purpose of defining beans that way is: we want our component beans to act like a factory. Other than that, there might be some scenarios where we cannot modify the main @Configuration(s) files or we want to define/customize beans only within a particular module.

9 - @ComponentScan custom filtering

@ComponentScan, by default, look for the classes annotated with @Component, @Repository, @Service, @Controller, or any user defined annotation which has been meta-annotated with @Component. We can turn off this default behavior by using 'useDefaultFilters' element of @ComponentScan:

```
@ComponentScan(useDefaultFilters = false)
```

With or without useDefaultFilters = false, We may also want to supply our own custom filtering by using element 'includeFilters'

```
@ComponentScan(useDefaultFilters = false/true,  includeFilters =
{@ComponentScan.Filter{ ... })
```

Or we may want to provide 'excludeFilter':

```
@ComponentScan(useDefaultFilters = false/true,  excludeFilters =
{@ComponentScan.Filter{ ... })
```

@ComponentScan.Filter annotation

Following snippet shows the elements defined in nested @Filter annotation:

```
package org.springframework.context.annotation;
....
public @interface ComponentScan {
    ....
    Filter[] includeFilters() default {};
    Filter[] excludeFilters() default {};

    ....


    @Retention(RetentionPolicy.RUNTIME)

    @Target({})

    @interface Filter {

        FilterType type() default FilterType.ANNOTATION;


        @AliasFor("classes")

        Class<?>[] value() default {};


        @AliasFor("value")

        Class<?>[] classes() default {};


        String[] pattern() default {};

    }

}
```

The most important element is FilterType enum which has five values: ANNOTATION, ASSIGNABLE_TYPE, ASPECTJ, REGEX and CUSTOM.

Examples to understand the usage of different FilterType.

FilterType.ASSIGNABLE_TYPE example

FilterType.ASSIGNABLE_TYPE together with Filter.classes or Filter.value can be used to specify assignable classes to be searched in scanning. The 'assignable classes' means, the scanning returns sub types as well (per Class#isAssignableFrom(..) definition)

Consider very simple beans:

```
public class MyBean1 {
}
public class MyBean2 {
}
public class MyBean3 {
}
public class MyBean4 extends MyBean3{
}
```

We don't have to use @Component annotation with above beans, because we are going to turn off default scanning in our example.

We have also created a Util class to reuse code:

```
public class Util {
 public static void printBeanNames(ApplicationContext context){
     String[] beanNames = context.getBeanDefinitionNames();
     Arrays.stream(beanNames)
            .filter(n -> !n.contains("springframework"))
            .forEach(System.out::println);
 }
}
```

Here's the filter usage:

```
@Configuration
@ComponentScan(useDefaultFilters = false,
      includeFilters = {@ComponentScan.Filter(
              type = FilterType.ASSIGNABLE_TYPE, classes = {MyBean1.class,
MyBean3.class})})
public class FilterTypeAssignableExample {
  public static void main(String[] args) {
      ApplicationContext context =
              new AnnotationConfigApplicationContext(FilterTypeAssignableExample.class);
      Util.printBeanNames(context);
  }
}
```

Output

```
filterTypeAssignableExample
myBean1
myBean3
myBean4
```

Output includes the name of MyBean4 as well, that's because My Bean3. class. is Assignable From(My Bean4. class) returns true.

Also 'filterTypeAssignableExample' is in the output, Why? because 'this' @Configuration class is registered as a bean too (not because of scanning, but because of the constructor Annotation Config Application Context(Filter Type Assignable Example. class)).

Filer.pattern along with ASSIGNABLE_TYPE not supported

```
@Configuration
@ComponentScan(useDefaultFilters = false,
      includeFilters = {@ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE,
pattern = ".*1")})
public class FilterTypeAssignableExample2 {

  public static void main(String[] args) {
      ApplicationContext context =
              new AnnotationConfigApplicationContext(FilterTypeAssignableExample2.class);
      Util.printBeanNames(context);
  }
}
```

Output

```
Caused by: java.lang.IllegalArgumentException: Filter type not supported with String
pattern: ASSIGNABLE_TYPE
```

Filter.pattern can only be used with FilterType.REGEX (next example)

FilterType.REGEX example

Following regex pattern example will scan only beans classes ending with 1 or 2.

Note that we also have to exclude our FilterTypeAssignableExample2 (from last example) from being scanned because it has '2' at the end.

```
@Configuration
@ComponentScan(useDefaultFilters = false,
      includeFilters = @ComponentScan.Filter(type = FilterType.REGEX, pattern =
".*[12]"),
      excludeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE,
              classes = FilterTypeAssignableExample2.class)
)
public class FilterTypeRegexExample {

  public static void main(String[] args) {
      ApplicationContext context =
              new AnnotationConfigApplicationContext(FilterTypeRegexExample.class);
      Util.printBeanNames(context);
  }
}
```

Output

```
filterTypeRegexExample
myBean1
myBean2
```

Note that when specifying 'pattern' value along with the FilterType other than REGEX, we will have 'not supported' exception.

FilterType.ANNOTATION example

This is the default FilterType which causes the @Component annotation to be searched. We can specify our own custom annotation, in that case we have to specify 'classes' element as being our annotation class.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
}
@MyAnnotation
public class MyBean5 {
}
@Configuration
@ComponentScan(useDefaultFilters = false,
      includeFilters = {@ComponentScan.Filter(type = FilterType.ANNOTATION, classes =
MyAnnotation.class)})
public class FilterTypeAnnotationExample {
  public static void main(String[] args) {
      ApplicationContext context =
              new AnnotationConfigApplicationContext(FilterTypeAnnotationExample.class);
      Util.printBeanNames(context);
  }
}
```

Output

```
filterTypeAnnotationExample
myBean5
```

In above output MyBean1, MyBean2 etc are not included even though they are in the same package.

FilterType.CUSTOM example

FilterType.CUSTOM can be used for a custom programmatic filtering of the scanned classes. In that case, we have to assign an implementation of TypeFilter to the Filter.classes element.

In following example, scanning will only target the bean classes which are implementing java.lang.Runnable interface.

```
public class MyTypeFilter implements TypeFilter {
  private static final String RunnableName = Runnable.class.getName();

  @Override
  public boolean match(MetadataReader metadataReader,
                       MetadataReaderFactory metadataReaderFactory) throws IOException {
      ClassMetadata classMetadata = metadataReader.getClassMetadata();
      String[] interfaceNames = classMetadata.getInterfaceNames();
      if (Arrays.stream(interfaceNames).anyMatch(RunnableName::equals)) {
          return true;
      }
      return false;
  }
}
public class MyBean6 implements Runnable{
  @Override
  public void run() {
      //todo
  }
}
@Configuration
@ComponentScan(useDefaultFilters = false,
      includeFilters = @ComponentScan.Filter(type = FilterType.CUSTOM, classes =
MyTypeFilter.class)
)
public class FilterTypeCustomExample {


  public static void main(String[] args) {
      ApplicationContext context =
              new AnnotationConfigApplicationContext(FilterTypeCustomExample.class);
      Util.printBeanNames(context);
  }
}
```

Output

```
filterTypeCustomExample
myBean6
```

In above output, other example beans are not included even though they are in the same package.

There's one more FilterType i.e. ASPECTJ which is outside of Spring core. This filter type matches the beans based on AspectJ type pattern expression.