Different ways of injecting dependencies

There are three possible ways of DI in Spring

Constructor-based: should be used for mandatory dependencies. In constructor, we should assign constructor args to final member fields.

Setter-based: Should be used for optional dependencies.

Field-based: Spring discourages the use of this because it would possibly hide mandatory fields from outside which would otherwise be assigned in the constructor. This would take away the advantage of properly initialized POJO, specially if intended to use outside of Spring container. Even though, we are mostly using field based injection in this series of tutorials to simplify the concept we want to deliver, we suggest the developers to always avoid using field-based DI in real project scenarios.

Examples

```
public class OrderService {
  public String getOrderDetails(String orderId) {
      return "Order details for order id=" + orderId;
  }
}
```

Constructor Based DI

```
@Configuration
public class ConstBasedDI {
```

```
    @Bean
    public OrderService orderService() {
        return new OrderService();
    }

    @Bean
    public OrderServiceClient orderServiceClient() {
        return new OrderServiceClient(orderService());
    }

    public static void main(String... strings) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(
                ConstBasedDI.class);
        OrderServiceClient bean = context.getBean(OrderServiceClient.class);
        bean.showPendingOrderDetails();
    }

    private static class OrderServiceClient {

        private OrderService orderService;

        // @Autowired is not needed
```

```
        OrderServiceClient(OrderService orderService) {
            this.orderService = orderService;
        }

        public void showPendingOrderDetails() {
            System.out.println(orderService.getOrderDetails("100"));
        }
    }
}
```

Output

```
Order details for order id=100
```

Constructor Based DI with Component Scan

Following example shows Constructor Based DI with @ComponentScan:

```
@Configuration
@ComponentScan({"com.piseth.java.school"})
public class ConstBasedDIWithScan {

    @Bean
    public OrderService orderService() {
        return new OrderService();
    }
```

```
    public static void main(String... strings) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(
                ConstBasedDIWithScan.class);
        OrderServiceClient bean = context.getBean(OrderServiceClient.class);
        bean.showPendingOrderDetails();
    }

    @Component
    public static class OrderServiceClient {

        private OrderService orderService;

        @Autowired
        OrderServiceClient(OrderService orderService) {
            this.orderService = orderService;
        }

        public void showPendingOrderDetails() {
            System.out.println(orderService.getOrderDetails("500"));
        }
    }
}
```

Output: Order details for order id=500

Setter Based DI

```
@Configuration
public class SetterBasedDI {

    @Bean
    public OrderService orderService() {
        return new OrderService();
    }

    @Bean
    public OrderServiceClient orderServiceClient() {
        return new OrderServiceClient();
    }

    public static void main(String... strings) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(
                SetterBasedDI.class);
        OrderServiceClient bean = context.getBean(OrderServiceClient.class);
        bean.showPendingOrderDetails();
    }
```

```
    private static class OrderServiceClient {

        private OrderService orderService;

        @Autowired
        public void setOrderService(OrderService orderService) {
            this.orderService = orderService;
        }

        public void showPendingOrderDetails() {
            System.out.println(orderService.getOrderDetails("200"));
        }
    }
}
```

Output: Order details for order id=200

Field Based DI

```
@Configuration
public class FieldBasedDI {

    @Bean
    public OrderService orderService() {
        return new OrderService();
    }
```

```
    @Bean
    public OrderServiceClient orderServiceClient() {
        return new OrderServiceClient();
    }

    public static void main(String... strings) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(
                FieldBasedDI.class);
        OrderServiceClient bean = context.getBean(OrderServiceClient.class);
        bean.showPendingOrderDetails();
    }

    private static class OrderServiceClient {

        @Autowired
        private OrderService orderService;

        public void showPendingOrderDetails() {
            System.out.println(orderService.getOrderDetails("300"));
        }
    }
}
```

Output: Order details for order id=300

Implicit constructor Injection

Starting Spring 4.3, it is no longer necessary to specify the @Autowired annotation for constructor based dependency injection.

As we saw in different ways of DI example project, that we had to use @Autowired on the constructor when using @ComponentScan. Starting Spring 4.3, @Autowired is no longer required on constructor.

Example

```
@Component
public class OrderService {

  public String getOrderDetails(String orderId) {
      return "Order details, for order id=" + orderId;
  }
}
```

Performing constructor base DI without @Autowired

```
@Component
public class OrderServiceClient {

  private OrderService orderService;
```

```
  //@Autowired is no longer required in Spring 4.3 and later.
  public OrderServiceClient(OrderService orderService) {
      this.orderService = orderService;
  }

  public void showPendingOrderDetails () {
      System.out.println(orderService.getOrderDetails("100"));
  }
}
```

Configuration class and running the example

```
@Configuration
@ComponentScan
public class AppRunner {

  public static void main(String... strings) {
      AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppRunner.class);
      OrderServiceClient bean = context.getBean(OrderServiceClient.class);
      bean.showPendingOrderDetails();
  }
}
```

Output: Order details, for order id=100

Implicit Constructor Injection In @Configuration Class

Just like in an ordinary bean class, we can also implicitly inject dependency in @Configuration classes (only in Spring 4.3 and later).

Example

```
@Component
public class Greeter {

  public void greet(String name){
      System.out.println("hi there, "+name);
  }
}
```

Implicit DI in @Configuration class constructor

```
@Configuration
@ComponentScan
public class AppRunner {
  private Greeter greeter;

  public AppRunner(Greeter greeter) {
      this.greeter = greeter;
  }
```

```
  public static void main(String... strings) {
      AnnotationConfigApplicationContext context =
              new AnnotationConfigApplicationContext(AppRunner.class);

      AppRunner appRunner = context.getBean(AppRunner.class);
      appRunner.greeter.greet("Piseth");
  }
}
```

Output: hi there, Piseth

Dependency injection in @Bean method parameters

In a @Configuration class, the methods annotated with @Bean may depend on other beans to initialize themselves. Other beans should be annotated with @Bean as well to be registered with the Spring container.

Spring provides a mechanism where we can pass such bean dependencies with @Bean method parameters. They are injected by the framework just like a arbitrary method's dependencies are resolved

There are following scenarios:

Injecting by type:

If there's only one bean instance available to be injected to the injection target point then it will be injected successfully by type.

Injecting by name:

If there are more than one instance of the same type available for a target injection point then there's a conflict (ambiguity). Spring doesn't know which particular instance to be injected in that case. If the name of parameter is same as bean's definition method (the method annotated with @Bean) name then the dependency is resolved by name.

The bean's definition method can provide a different name than the method name by using @Bean(name = ...), the injection point method's parameter name should match in that case as well.

Injecting by bean's name with matching @Qualifier:

If there's an ambiguity then it can also be resolved if the injection point method parameter add a @Qualifier annotation with matching target bean's name.

Injecting by matching @Qualifiers

Ambiguity can also be resolve by using @Qualifier on the both sides. This is important when a bean provider method has already indented to be exposed as a @Qualifier per business logic sense, so that a particular bean's implementation can be changed without updating all injection points.

Examples

Bean classes

```
public class BeanA {
  private String name;

  public BeanA(String name){
      this.name = name;
  }

  @Override
  public String toString() {
      return "BeanA{" +
              "name='" + name + '\'' +
              '}';
  }
}

public class BeanB {
  private BeanA beanA;

  BeanB (BeanA beanA) {
```

```
        this.beanA = beanA;
    }

    public BeanA getBeanA () {
        return beanA;
    }

    @Override
    public String toString() {
        return "BeanB{" +
                "beanA=" + beanA +
                '}';
    }
}

public class BeanC {
    private String name;

    public BeanC(String name){
        this.name = name;
    }

    @Override
```

```
  public String toString() {
      return "BeanC{" +
              "name='" + name + '\'' +
              '}';
  }
}
```

Injecting by type

```
public class InjectParameterByType {

    public static void main (String[] args) {
        AnnotationConfigApplicationContext context =
                        new AnnotationConfigApplicationContext(
                                        Config.class);
        BeanB beanB = context.getBean(BeanB.class);
        System.out.println("In the main method: " + beanB.getBeanA());
    }

    @Configuration
    public static class Config {

        @Bean
        public BeanA bean1 () {
```

```
            return new BeanA("a1");
        }

        @Bean
        public BeanB bean2 (BeanA theBean) {
            BeanB beanB = new BeanB(theBean);
            System.out.println("method bean2: beanB created = " + beanB +
                               "\n with constructor param BeanA = " + theBean);

            return beanB;
        }
    }
}
```

Output

```
method bean2: beanB created = BeanB{beanA=BeanA{name='a1'}}
 with constructor param BeanA = BeanA{name='a1'}
In the main method: BeanA{name='a1'}
```

Injecting by default bean names

```
public class InjectParameterByName {

    public static void main (String[] args) {
        AnnotationConfigApplicationContext context =
```

```
                new AnnotationConfigApplicationContext(
                        Config.class);
        BeanB beanB = context.getBean(BeanB.class);
        System.out.println("In the main method: " + beanB.getBeanA());
    }

    @Configuration
    public static class Config {

        @Bean
        public BeanA bean1 () {
            return new BeanA("a1");
        }

        @Bean
        public BeanA bean2 () {
            return new BeanA("a2");
        }

        @Bean
        public BeanB bean3 (BeanA bean1) {
            BeanB beanB = new BeanB(bean1);
            System.out.println("method bean3: beanB created = " + beanB +
```

```
                        "\n with constructor param BeanA: " + bean1);

            return beanB;
        }
    }
}
```

Output

```
method bean3: beanB created = BeanB{beanA=BeanA{name='a1'}}
 with constructor param BeanA: BeanA{name='a1'}
In the main method: BeanA{name='a1'}
```

Injecting by using @Qualifier at both places

```
public class InjectParameterByQualifier {

    public static void main (String[] args) {
        AnnotationConfigApplicationContext context =
                        new AnnotationConfigApplicationContext(
                                        Config.class);
        BeanB beanB = context.getBean(BeanB.class);
        System.out.println("In the main method: " + beanB.getBeanA());
    }

    @Configuration
```

```
    public static class Config {

        @Bean
        public BeanA bean1 () {
            return new BeanA("a1");
        }

        @Qualifier("myBean")
        @Bean
        public BeanA bean2 () {
            return new BeanA("a2");
        }

        @Bean
        public BeanB bean3 (@Qualifier("myBean") BeanA theBean) {
            BeanB beanB = new BeanB(theBean);
            System.out.println("method bean3: beanB created = " + beanB +
                               "\n with constructor param BeanA = " + theBean);

            return beanB;
        }
    }
}
```

Output

```
method bean3: beanB created = BeanB{beanA=BeanA{name='a2'}}
 with constructor param BeanA = BeanA{name='a2'}
In the main method: BeanA{name='a2'}
```

Injecting by using @Qualifier at injection point only

```
public class InjectParameterByQualifier2 {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(
                        Config.class);
        BeanB beanB = context.getBean(BeanB.class);
        System.out.println("In the main method: " + beanB.getBeanA());
    }

    @Configuration
    public static class Config {

        @Bean
        public BeanA bean1() {
            return new BeanA("a1");
        }
```

```
    @Bean
    public BeanA bean2() {
        return new BeanA("a2");
    }

    @Bean
    public BeanB bean3(@Qualifier("bean2") BeanA theBean) {
        BeanB beanB = new BeanB(theBean);
        System.out.println("method bean3: beanB created = " + beanB +
                "\n with constructor param BeanA = " + theBean);

        return beanB;
    }
  }
}
```

Output

```
method bean3: beanB created = BeanB{beanA=BeanA{name='a2'}}
 with constructor param BeanA = BeanA{name='a2'}
In the main method: BeanA{name='a2'}
```

Injecting multiple beans using @Qualifier

```
public class InjectParameterByQualifier3 {
```

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(
                    Config.class);
    BeanB beanB = context.getBean(BeanB.class);
    System.out.println("In the main method: " + beanB.getBeanA());
}

@Configuration
public static class Config {

    @Bean
    public BeanA bean1() {
        return new BeanA("a1");
    }

    @Bean
    @Qualifier("myBean")
    public BeanA bean2() {
        return new BeanA("a2");
    }
```

```
    @Bean
    public BeanC bean3() {
        return new BeanC("c1");
    }

    @Bean
    @Qualifier("myBean2")
    public BeanC bean4() {
        return new BeanC("c2");
    }

    @Bean
    public BeanB bean5(@Qualifier("myBean") BeanA theBean,
                       @Qualifier("myBean2") BeanC theBean2) {
        BeanB beanB = new BeanB(theBean);
        System.out.println("method bean5: beanB created = " + beanB +
                "\n with constructor param of type BeanA= " + theBean);
        System.out.println("method bean5: theBean2 instance (can also be in as
constructor " +
                "arg or some " +
                "other way): " + theBean2);

        return beanB;
    }
```

```
    }
}
```

Output

```
method bean5: beanB created = BeanB{beanA=BeanA{name='a2'}}
 with constructor param of type BeanA= BeanA{name='a2'}
method bean5: theBean2 instance (can also be in as constructor arg or some other way): BeanC{name='c2'}
In the main method: BeanA{name='a2'}
```