1 - Resolving ambiguity by using @Qualifier

Specifying an unique name for @Bean annotation is necessary if

- the configuration provides more than one implementations for a bean
- or if we want to inject bean instances by name rather than by type.

To match a named bean to an injection point (or in other words to qualify a bean to an injection point), the bean's property name (at the injection point) should match with the bean definition name.

Example

```
public interface OrderService {

  String getOrderDetails(String orderId);

}


public class OrderServiceImpl1 implements OrderService {

  public String getOrderDetails(String orderId) {

      return "Order details from impl 1, for order id=" + orderId;

  }

}


public class OrderServiceImpl2 implements OrderService {

  public String getOrderDetails(String orderId) {

      return "Order details from impl 2, for order id=" + orderId;

  }

}
```

Using @Qualifier at injection point

```
public class OrderServiceClient {

  @Autowired
  @Qualifier("OrderServiceA")
  private OrderService orderService;

  public void showPendingOrderDetails () {
      for (String orderId : Arrays.asList("100", "200", "300")) {
          System.out.println(orderService.getOrderDetails(orderId));
      }
  }
}
```

Defining beans' name via @Bean#name

```
@Configuration
public class AppRunner {

  @Bean(name = "OrderServiceA")
  public OrderService orderServiceByProvider1() {
      return new OrderServiceImpl1();
  }

  @Bean(name = "OrderServiceB")
  public OrderService orderServiceByProvider2() {
```

```
        return new OrderServiceImpl2();
    }

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

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

Output:

```
Order details from impl 1, for order id=100
Order details from impl 1, for order id=200
Order details from impl 1, for order id=300
```

Using @Qualifier

```
@Configuration
public class AppRunner2 {
```

```
  @Bean
  @Qualifier("OrderServiceA")
  public OrderService orderServiceByProvider1() {
      return new OrderServiceImpl1();
  }

  @Bean(name = "OrderServiceB")
  public OrderService orderServiceByProvider2() {
      return new OrderServiceImpl2();
  }

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

  public static void main(String... strings) {
      AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppRunner2.class);
      OrderServiceClient bean = context.getBean(OrderServiceClient.class);
      bean.showPendingOrderDetails();
  }
}
```

Output

Order details from impl 1, for order id=100

Order details from impl 1, for order id=200

Order details from impl 1, for order id=300

2 - Resolving ambiguity by using @Resource

Spring supports Java SE Common Annotations (JSR-250). That means, we can use @Resource instead of using the combination of @Autowire and @Qualifier.

`javax.annotation.Resource`

Example

```
public interface OrderService {
  String getOrderDetails(String orderId);
}

public class OrderServiceImpl1 implements OrderService {

  public String getOrderDetails (String orderId) {
      return "Order details from impl 1, for order id=" + orderId;
  }
}

public class OrderServiceImpl2 implements OrderService {
```

```
  public String getOrderDetails (String orderId) {
      return "Order details from impl 2, for order id=" + orderId;
  }
}
```

Using @Resource at injection point

```
public class OrderServiceClient {

  @Resource(name = "OrderServiceA")
  private OrderService orderService;

  public void showPendingOrderDetails() {
      for (String orderId : Arrays.asList("100", "200", "300")) {
          System.out.println(orderService.getOrderDetails(orderId));
      }
  }
}
```

Defining beans and running app

```
@Configuration
public class AppRunner {

  @Bean(name = "OrderServiceA")
  public OrderService orderServiceByProvider1() {
```

```
    return new OrderServiceImpl1();
  }

  @Bean(name = "OrderServiceB")
  public OrderService orderServiceByProvider2() {
    return new OrderServiceImpl2();
  }

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

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

Output

```
Order details from impl 1, for order id=100
Order details from impl 1, for order id=200
Order details from impl 1, for order id=300
```

Note that @Resource can also be used without specifying any name (the qualifier) parameter. If no name is specified explicitly, the default name is derived from the field name or setter method.

3 - Resolving ambiguity by using @Inject and @Qualifier Annotations

Spring supports annotations specified by JSR 330 (Dependency Injection for Java). That means we can use javax.inject.Inject annotation along with @Qualifier annotation to resolve ambiguity.

Example

pom.xml

```
<dependency>
   <groupId>javax.inject</groupId>
   <artifactId>javax.inject</artifactId>
   <version>1</version>
</dependency>
```

Bean classes

```
public interface OrderService {
  String getOrderDetails(String orderId);
}

public class OrderServiceImpl1 implements OrderService {
```

```
  public String getOrderDetails (String orderId) {
      return "Order details from impl 1, for order id=" + orderId;
  }
}

public class OrderServiceImpl2 implements OrderService {

  public String getOrderDetails (String orderId) {
      return "Order details from impl 2, for order id=" + orderId;
  }
}
```

Using @Inject annotation

```
public class OrderServiceClient {

  @Inject
  @Qualifier("OrderServiceB")
  private OrderService orderService;

  public void showPendingOrderDetails() {
```

```
        for (String orderId : Arrays.asList("100", "200", "300")) {
            System.out.println(orderService.getOrderDetails(orderId));
        }
    }
}
```

Defining beans and running example application

```
@Configuration
public class AppRunner {
    @Bean(name = "OrderServiceA")
    public OrderService orderServiceByProvider1() {
        return new OrderServiceImpl1();
    }

    @Bean(name = "OrderServiceB")
    public OrderService orderServiceByProvider2() {
        return new OrderServiceImpl2();
    }

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

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

Output

```
Order details from impl 2, for order id=100
Order details from impl 2, for order id=200
Order details from impl 2, for order id=300
```

4 - Resolving ambiguity by using @Inject and @Named annotations

In the last example we saw how to use JSR 330 javax.inject.Inject annotation (in the place of @Autowire). Spring also supports javax.inject.Named annotation (also defined in JSR 330) to qualify a name for the dependency. That means we can use @Named annotation in the place of @Qualifier annotation.

Example

```
public interface OrderService {
  String getOrderDetails(String orderId);
}
public class OrderServiceImpl1 implements OrderService {
```

```
  public String getOrderDetails (String orderId) {
      return "Order details from impl 1, for order id=" + orderId;
  }
}

public class OrderServiceImpl2 implements OrderService {

  public String getOrderDetails (String orderId) {
      return "Order details from impl 2, for order id=" + orderId;
  }
}
```

Using @Inject and @Named annotations

```
public class OrderServiceClient {

  @Inject
  @Named("OrderServiceB")
  private OrderService orderService;

  public void showPendingOrderDetails() {
```

```
        for (String orderId : Arrays.asList("100", "200", "300")) {
            System.out.println(orderService.getOrderDetails(orderId));
        }
    }
}
```

Defining beans and running the example app

```
@Configuration
public class AppRunner {

    @Bean(name = "OrderServiceA")
    public OrderService orderServiceByProvider1() {
        return new OrderServiceImpl1();
    }

    @Bean(name = "OrderServiceB")
    public OrderService orderServiceByProvider2() {
        return new OrderServiceImpl2();
    }

    @Bean
    public OrderServiceClient createClient() {
        return new OrderServiceClient();
```

```
  }

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

Output

```
Order details from impl 2, for order id=100
Order details from impl 2, for order id=200
Order details from impl 2, for order id=300
```

5 - Using @Primary annotation

As we saw in Inject Spring Bean By Name example that if there are more than one instances available for an injection point then we have to use @Qualifier annotation to resolve ambiguity. As @Qualifier is used at injection point, there might be two situations where we don't want to or cannot use @Qualifier.

- Our autowiring mode is Autowire.BY_TYPE. Then of course we cannot use @Qualifier because we actually don't have user defined injection point specified as @Autowired or @Inject
- We want to do bean selection (i.e. resolve the ambiguity) at configuration time rather than during beans development time.

The solution to above problems is to use @Primary annotation.

@Primary indicates that a particular bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency. If exactly one 'primary' bean exists among the candidates, it will be the autowired value.

Example

In this example we are going to demonstrate how to use @Primary when autowiring mode is set to Autowire.BY_TYPE

```
public interface Dao {
  void saveOrder(String orderId);
}

public class DaoA implements Dao {

  @Override
  public void saveOrder(String orderId) {
      System.out.println("DaoA Order saved " + orderId);
  }
}

public class DaoB implements Dao {
```

```
  @Override
  public void saveOrder(String orderId) {
      System.out.println("DaoA Order saved " + orderId);
  }
}

public class OrderService {

  private Dao dao;

  public void placeOrder(String orderId) {
      System.out.println("placing order " + orderId);
      dao.saveOrder(orderId);
  }

  public void setDao(Dao dao) {
      this.dao = dao;
  }
}
```

Definition beans and running the example

```
@Configuration
```

```
public class AppConfig {


  @Bean(autowire = Autowire.BY_TYPE)
  public OrderService orderService() {
      return new OrderService();
  }

  @Bean
  public Dao daoA() {
      return new DaoA();
  }

  @Primary
  @Bean
  public Dao daoB() {
      return new DaoB();
  }

  public static void main(String[] args) throws InterruptedException {
      AnnotationConfigApplicationContext context =
              new AnnotationConfigApplicationContext(AppConfig.class);

      OrderService orderService = context.getBean(OrderService.class);
```

```
    orderService.placeOrder("122");


  }
}
```

Output

```
placing order 122
DaoA Order saved 122
```

6 - Java Generics as Autowiring Qualifiers

Spring 4.x is capable of using Java Generics types as an implicit qualification for dependency injection. Now we don't have to use @Qualifier or some specialized types to overcome this limitation in previous versions.

Consider List<String> and List<Integer>. They are of same type List but have different generics types, String and Integer. Spring first matches up the type and if the type has generics then matches up the generic type. If there is only one bean available (having same type and generics) for injection then performs injection otherwise throws error: NoUniqueBeanDefinitionException

Example

This example demonstrates that even though we have two instances of RateFormatter available for dependency injection, RateCalculator is still injected with the one matched by generic type.

```
import java.math.BigDecimal;
```

```
import java.text.NumberFormat;


public class RateFormatter<T extends Number> {


  public String format(T number){
      NumberFormat format = NumberFormat.getInstance();
      if(number instanceof Integer){
          format.setMinimumIntegerDigits(0);
      }else if(number instanceof BigDecimal){
          format.setMinimumIntegerDigits(2);
          format.setMaximumFractionDigits(2);
      }//others


      return format.format(number);
  }
}


public class RateCalculator {


  @Autowired
  private RateFormatter<BigDecimal> formatter;


  public void calculate() {
```

```
        BigDecimal rate = new BigDecimal(1053.75356);
        System.out.println(formatter.format(rate));
    }
}
```

Defining beans and running the example

```
@Configuration
public class Config {

    @Bean
    RateFormatter<Integer> integerRateFormatter() {
        return new RateFormatter<Integer>();
    }

    @Bean
    RateFormatter<BigDecimal> bigDecimalRateFormatter() {
        return new RateFormatter<BigDecimal>();
    }

    @Bean
    RateCalculator rateCalculator() {
```

```
        return new RateCalculator();
    }

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(Config.class);
        RateCalculator bean = context.getBean(RateCalculator.class);
        bean.calculate();
    }
}
```

Output

```
1,053.75
```