Extension points and modules in OpenLMIS

Extension Module

Extension module consists of a basic Spring Application Context, which uses a Service's extension point(s) to define new behaviour, as well as a Gradle build system that supports building and publishing to a Maven repository (extension module is packaged and deployed as a JAR through Maven). Extension module utilizes 1 or more Extension Points (interfaces) exposed by the Service to modify behavior.

Every extension module resides in the Git repository named *openlmis-independentServiceName-extensionModuleName.*

Defining new Extension Point

Every extension point is an interface in independent service that can be implemented in extension module. In the independent service there is a class implementing default behavior for every extension point.

Creating an Extension Module

Extension module should be deployed as JAR and placed in the classpath of the Independent Service. It can override or extend default behavior of independent service by defining beans in its application context.
In the independent service there would be a class implementing interface with default methods. Any extension module can define a new class implementing the same interface and create bean from it. The code in independent service and extension module would look like that:

Independent service:

```
interface OrderQuantity {

      OrderQty calc(Product product, Period period,
            Facility facility);
}

class DefaultOrderQuantity implements OrderQuantity {

      OrderQty calc(Product product, Period period,
            Facility facility)  {
            //default logic implemented here
      }
}
```

Extension module:

```
class AMCOrderQuantity implements OrderQuantity {

      OrderQty calc(Product product, Period period,
            Facility facility)  {
            //custom logic implemented here
      }
}
```

We have found several ways of extending/overriding beans functionality through application context. In solution A and B @Configuration class should have @ComponentScan annotation so that it can scan packages from independent service and also packages from an extension module. Scanning packages from extension module can be done using wildcard characters.

A. Beans with @Primary annotation

While defining @Beans implementing interface from independent service in extension module it is possible to mark them as default by using @Primary annotation. This annotation indicates that a bean should be given preference when multiple candidates are qualified to autowire a single-valued dependency. Note that using @Primary at the class level has no effect unless component-scanning is being used. If a @Primary-annotated class is declared via XML, @Primary annotation metadata is ignored, and <bean primary="true|false"/> is respected instead. The important thing is that two beans **cannot** have the same name - then one bean will overwrite another and we cannot be sure which one will be chosen. The bean definitions would look like that:

```
<bean id="amcOrderQuantity" class="x.y.z.AmcOrderQuantity"
primary="true"/>
<bean id="defaultOrderQuantity" class="a.b.c.DefaultOrderQuantity"/>
```

This solution seems to be the simplest one both for users and implementers as it does not require much configuration.

B. Configuration classes with @Profile annotation

Another way to define which implementation of interface should be used, is using @Profile annotation on @Configuration class. Any @Component or @Configuration can be marked with @Profile to limit when it is loaded. It is possible to specify the active profile in your application.properties file:

```
spring.profiles.active=default
```

Sometimes it is useful to have profile-specific properties that add to the active profiles rather than replace them. The `spring.profiles.include` property can be used to unconditionally add active profiles.

For unit test it is possible to use @ActiveProfiles annotation, for example:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = { AppConfig.class })
@ActiveProfiles("AMC")
public class OrderQuantityUnitTest {

	@Autowired
	private OrderQuantity orderQuantity;

	@Test
	public void testAMCCalcOrderQuantity() {
		//...
	}
}
```

This solution was rejected, because it requires from user specifing an active profile in application.properties file and therefore modifying OpenLMIS codebase

C. Arbitrary method replacement

We have also found a solution that does not need defining an interface that can be implemented in extension module. It is possible to use “replaced-method” element to replace an existing method from given class with another. Then a class implementing MethodReplacer interface provides the new method definition. The bean definitions would look like that:

```
<bean id="orderQuantity" class="x.y.z.OrderQuantity">
    <replaced-method name="calc" replacer="amcOrderQuantity">
        <arg-type>String</arg-type>
    </replaced-method>
</bean>

<bean id="amcOrderQuantity" class="a.b.c.AmcOrderQuantity"/>
```

This solution was rejected, because it requires adding extending beans definitions into base applicationContext.xml. Also, as Spring documentation states, it is rarely used.

Extension Modules recommendations

1. Extension modules should be as small as possible. They can extend multiple interfaces from independent service, but it is better to keep this number low, to avoid situation of adding two extension modules extending the same extension point. It is not stated in Spring documentation what will happen if two beans have @Primary annotation.

2. Interfaces should be used where possible in independent services. It creates more possibilities of extending or altering service behavior, which helps us to achieve more flexibility.