OpenbizIoC
Openbiz Framework - an Inverse of Control container
Phase-Design

Updated Mar 18, 2013 by agus.suhartono

Openbiz Inverse of Control container

Openbiz is a metadata centric application framework. In order to generate business object from metadata xml files, Openbiz uses the concept of Inverse of Control (IoC).

What is IoC ¶

The following paragraph is copied from wikipedia http://en.wikipedia.org/wiki/Inversion_of_control.

In software engineering, Inversion of Control (IoC) is an abstract principle describing an aspect of some software architecture designs in which the flow of control of a system is inverted in comparison to procedural programming.

In traditional programming the flow of the business logic is controlled by a central piece of code, which calls reusable subroutines that perform specific functions. Using Inversion of Control this "central control" design principle is abandoned. The caller's code deals with the program's execution order, but the business knowledge is encapsulated by the called subroutines.

IoC becomes well known to many programmers due to its wild adoption in many application frameworks including:

- Spring Framework. This is the most popular framework in Java community.
- Pico Container (Java)
- StructureMap (.Net)

Spring vs Openbiz

Like Spring framework, IoC is the foundation of the Openbiz framework, but the framework does a lot more than a IoC container. The following paragraphs make comparisons of IoC capability between Spring and Openbiz. SP stands for Spring Framework, OB stands for Openbiz Framework.

The container

SP: BeanFactory. BeanFactory is the actual representation of the Spring IoC container that is responsible for containing and otherwise managing the aforementioned beans. ApplicationContext is recommended as ApplicationContext includes all functionality of the BeanFactory.

OB: ObjectFactory plays the same role as BeanFactory in Spring. BizSystem is recommended as it includes all functionality of the ObjectFactory.

The objects managed by container

SP: beans

- XML based configuration. A general configuration that has has defined schema
  1. bean definition (name, class, ...)
  2. behavior configuration elements in types of either property and collection
  3. references to other beans
  4. other attributes
  5. support inheritance
- bean scope
  1. singleton. Scopes a single bean definition to a single object instance per Spring IoC container.
  2. prototype. Scopes a single bean definition to any number of object instances.
  3. request. Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition.
  4. session. Scopes a single bean definition to the lifecycle of a HTTP Session
  5. global session. Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context

OB: meta objects

- XML based configuration. Has defined schema for certain object types (data object, form object, view object, service object, widget object). Any xml schema can be supported with custom implementation
  1. object definition (name, class, ...)
  2. behavior configuration elements in types of either property and collection
  3. references to other beans
  4. other attributes
  5. support inheritance in certain object types

- Object scope
  1. prototype. Any number of object instances can be created
  2. request. A single object is created during the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of an object.
  3. session. A single object is created during the lifecycle of a HTTP Session

Dependency Injections

Dependency injection is the main method to implement Inversion of Control. The basic principle behind Dependency Injection (DI) is that objects define their dependencies (that is to say the other objects they work with) only through constructor arguments, arguments to a factory method, or properties which are set on the object instance after it has been constructed or returned from a factory method. Then, it is the job of the container to actually inject those dependencies when it creates the object

SP: DI method

- Constructor injection. Constructor-based DI is effected by invoking a constructor with a number of arguments, each representing a dependency.
- Setter injection. Setter-based DI is realized by calling setter methods on your beans after invoking a no-argument constructor. The Spring team generally advocates the usage of setter injection, since a large number of constructor arguments can get unwieldy, especially when some properties are optional.

OB: DI method

- Constructor injection with single metadata array as constructor argument. The metadata array as constructor argument doesn't have the limitation of Spring constructor injection

Examples

Email Service in SP and OB

SP: define a mailSender bean with fixed XML schema

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="...">
    <context:component-scan base-package="com.javacodegeeks.spring.mail" />
    <bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
        <property name="host" value="smtp.gmail.com"/>
        <property name="port" value="25"/>
        <property name="username" value="myusername@gmail.com"/>
        <property name="password" value="mypassword"/>
        <property name="javaMailProperties">
            <props>
                <prop key="mail.transport.protocol">smtp</prop>
                <prop key="mail.smtp.auth">true</prop>
                <prop key="mail.smtp.starttls.enable">true</prop>
                <prop key="mail.debug">true</prop>
            </props>
        </property>
    </bean>
</beans>
```

OB: define a mailSender service object.

- customized XML schema
- reference object for mail logging points to email.do.EmailLogDO

```
<?xml version="1.0" standalone="no"?>
<PluginService Name="mailSender" Class="service.emailService">
   <Accounts>
      <!-- use SMTP server -->
      <Account Name="System" Host="smtp.yourcompany.com" FromName="admin" FromEmail="admin@yourcompany.com" IsSMTP
      <!-- use default unix sendmail function -->
      <Account Name="SystemNotifier" Host="" FromName="System Notification" FromEmail="notify@yourcompany.com" IsSI
   </Accounts>
   <Logging Type="DB" Object="email.do.EmailLogDO" Enabled="Y" />
</PluginService>
```

Example Database access - CRUD operations on a table "customer"

Assume SP + Hibernate is used to implement the logic. Briefly, a developer needs to code the following files

Configurations

- data source bean xml
- hibernate session factory bean xml
- customer bean xml for data access object (DAO)
- hibernate mapping file customer.hbm.xml that mapps database tables to a java object

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="com.mkyong.customer.model.Customer"
        table="customer" catalog="mkyongdb">

        <id name="customerId" type="long">
```

```
            <column name="CUSTOMER_ID" />
            <generator class="identity" />
        </id>
        <property name="name" type="string">
            <column name="NAME" length="45" not-null="true" />
        </property>
        <property name="address" type="string">
            <column name="ADDRESS" not-null="true" />
        </property>
        <property name="createdDate" type="timestamp">
            <column name="CREATED_DATE" length="19" not-null="true" />
        </property>
    </class>
</hibernate-mapping>
```

Java files

- customer data access object (DAO) java.
- customer model java. this is the class declared in hibernate mapping file.

OB needs to code the following files Configurations metadata

- add data source (if not yet added) in a shared application.xml
- data object xml

```
<?xml version="1.0" standalone="no"?>
<BizDataObj Name="CustomerDO" class="BizDataObj" DBName="Default" Table="customer" SearchRule="" SortRule="" Other
    <BizFieldList>
        <BizField Name="Id" Column="CUSTOMER_ID" Type="Number"/>
        <BizField Name="name" Column="NAME" Type="Text"/>
        <BizField Name="address" Column="ADDRESS" Type="Text"/>
        <BizField Name="createdDate" Column="CREATED_DATE" Type="Datetime" ValueOnCreate="{date('Y-m-d H:i:s')}"/>
    </BizFieldList>
</BizDataObj>
```

Comment by project member agus.suhartono, May 14, 2012

Yes MetaObject? concept really great IoC. - with array (the power of PHP) as constructor parameter, number of object that can passed is unlimited - object that passed is not real object, but just name of MetaObject? aka reference of real object, with ObjectFactory? real object can called.