Java BAdIs, Part I

Applies to:

SAP NetWeaver Application Server 7.0 Support Package 10 (or higher)

Summary

This article describes how the Business Add-in (BAdI) concept (for those who might be unfamiliar with the term, it's frankly speaking a concept to setup pre-defined extension points where customers can modify the processing in a modification-free manner) known from the ABAP programming language can be adapted to Java.

Author: Matthias Steiner

Company: SAP AG

Created on: 26 June 2007

Author Bio

Matthias Steiner is a Solution Architect for SAP NetWeaver Center of Excellence, where his role and responsibility is to evaluate SAP's cutting edge technologies and transform them into real solutions for SAP customers using the potential of enterprise SOA and realizing the full value of composite applications. He began his career with SAP in 2002 and has gathered significant experience in numerous enterprise-scaled development projects in both the ABAP and the Java world.

Table of Contents

Introduction

No matter how sophisticated a business service may be, customers will always have the need to adjust the "standard" behaviour to address their specific business requirements. It's an evergreen in software development: the need for a technique which provides some sort of extension point that allows for easy modification/enhancement of specific business functionality. In ABAP, the recommended technology for this purpose is BAdIs (Business Add-Ins).

BAdI – Definition

SAP Business Add-Ins (BAdIs) are one of the most important technologies used to adapt SAP software to specific requirements. BAdIs were introduced with Release 4.6 and replace function module exits. This technology is not limited to SAP applications. BAdI calls can be integrated in customer applications. These can then be enhanced by other customer applications. In the various SAP applications, BAdI calls are implemented at places where enhancements are appropriate. In the SAP ERP system, there were already approximately 5,000 BAdIs before Release 7.0.

Reference Link

The main characteristic of a BAdI is that it provides a mechanism to alter the functionality of a well-defined business function (e.g. a **BAPI**) without making changes to the delivered source code. This way, future upgrades of the original business function can be applied without losing the customer specific enhancements of the business function (which are wrapped inside the BAdI) or the need to merge the changes. This way, the two code-lines (originally delivered and customer-specific coding) are strictly separated but still integrated.

Well, even though a deep-dive into the technical details of BAdIs it out of scope of this article, I still need to talk about the fundamental concept of BAdIs in order to illustrate how this concept can be adapted to the Java (more precisely Java Enterprise Edition) world: a BAdI basically consists of a well-defined Interface (with a well-defined name), which can be implemented by the customer. At runtime, the system identifies the active implementation(s) (there can be multiple implementations which are called consecutively in random order) and executes it/them.

There is a technology perfectly suited for implementing something similar on the Java stack of the application server: Stateless Session Beans (EJB). EJBs (prior to Java EE 5 at least) always consist of an interface, an implementation and a home interface (and in case it is a remote enabled EJB there has to be an additional remote interface and a remote home interface). At runtime, the home interface of an EJB is retrieved by using a well-defined name via JNDI. (This article explains the fundamentals of accessing an EJB via JNDI.) The home interface is then used to obtain an implementation instance that is executed.

Getting Started

I created a rather easy example to illustrate the concept in order not to confuse anybody with complicated business logic, so frankly speaking all that the exemplary business service (implemented as a Stateless Session Bean) does is to return some technical information (the java system properties that is). The "standard" coding contains a post-processing hook that can be implemented by customers to alter the returned system information as required (e.g. hide security-sensitive information etc.)

The structure of the application to be used is illustrated below in Figure 1.

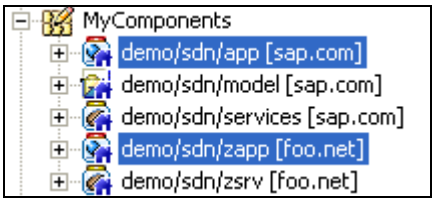

Figure 1 - Development Components Overview

As you can see I have defined two Enterprise Application DCs, one with the vendor prefix **sap.com** and one with the prefix **foo.net**. The first contains (well, the correct term would be assembles) the "standard" coding, where as the second DC assembles the customer specific coding.

Let's have a closer look at the specific coding lines that compose our java BAdI (Figure 2.)

```
// default business logic
List attributes = this.retrieveSystemProperties(request);  (1)

if (attributes != null && attributes.size() > 0)
{
    KeyValuePair[] props = new KeyValuePair[attributes.size()];
    props = (KeyValuePair[]) attributes.toArray(props);
    retVal.setAttributes(props);
}
else
{
    return retVal;
}

// call Business Add-In
try
{
    InitialContext ctx = new InitialContext();

    TechInfoServiceBadiLocalHome localBadiHome = (TechInfoServiceBadiLocalHome)
                                ctx.lookup(BADI_JNDI_NAME);  (2)

    TechInfoServiceBadiLocal badi = localBadiHome.create();
    retVal = badi.doPostProcess(request, retVal);  (3)
}
catch (NamingException ex)
{
    // assuming that no customer-specific BAdI has been defined
}
catch (CreateException ex)
{
     // in the real world a more sophisticated error handling is encouraged ;-)
    ex.printStackTrace();
}
catch (Exception ex)
{
    // in the real world a more sophisticated error handling is encouraged ;-)
    ex.printStackTrace();
}
```

Figure 2 - Business Service Coding

Let's break this down into the most relevant steps:

1. the system properties are obtained via the dedicated `retrieveSystemProperties()` function
2. a pre-defined JNDI name (BADI_JNDI_NAME) is used to lookup the EJBLocalHome of the TechInfoService BAdI (TechInfoServiceBadi)
3. the post-processing method (`doPostProcess()`) is called passing the original request and response objects

It really is that easy! We simply leverage the standard EJB features to obtain a reference to a well-defined business service via a well-defined JNDI key and we have the Java equivalent of a BAdI. So, what does that mean well-defined business service and well-defined JNDI name? *Good question, glad you asked!*

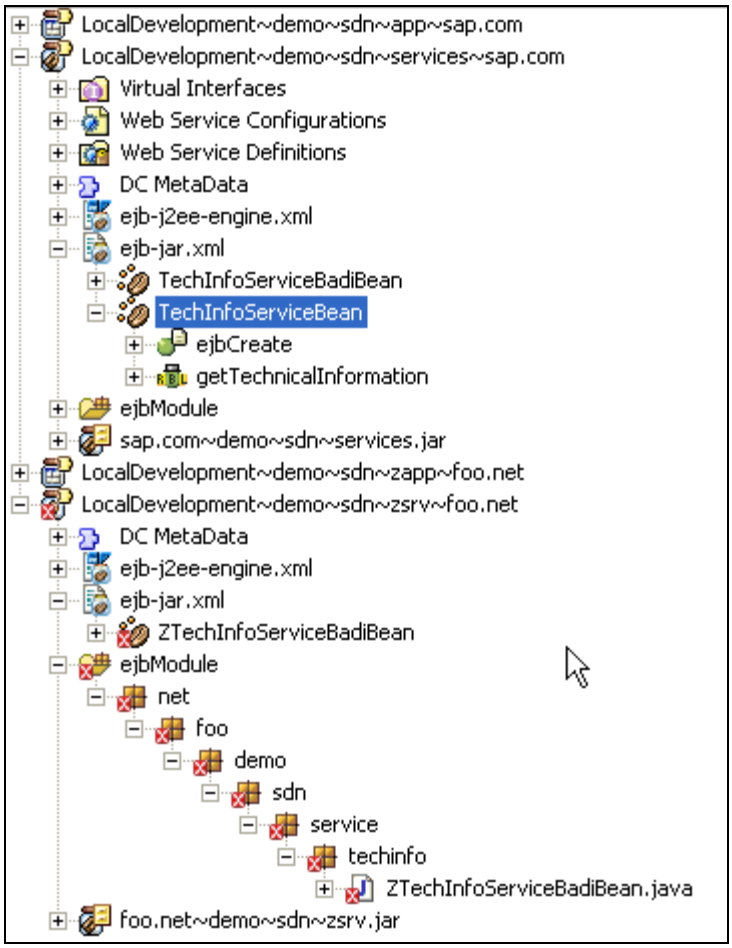

Figure 3 -EJB Development Component

As you can see in Figure 3 I have created two Stateless Session Beans in the EJB DC (`LocalDevelopment~demo~sdn~services~sap.com`): one for the business service itself (`TechInfoServiceBean`) and one for the BAdI (`TechInfoServiceBadiBean`). Both are exposed via the default `Public Parts` of an EJB DC.

The customer specific EJB DC (`LocalDevelopment~demo~sdn~zsrv~foo.net`) only contains the EJB implementation itself (`ZTechInfoServiceBadiBean`) and re-uses all the required EJB interfaces by defining a `Used-DC` reference to the EJB DC with the **sap.com** prefix. This is also the reason why you see the little red x icon here as the IDE is complaining that the EJB interfaces are missing... but don't worry, it will work just fine.

Note:

Please note that one also needs to maintain a sharing reference in the `application-j2ee-engine.xml` META artefact of the customer specific Enterprise Application in order to ensure that the required files are found by the `ClassLoader` at runtime.

Well, Class loading in J2EE servers is a very complicated topic, so please be aware that the entire concept presented here is specific to SAP NetWeaver.

The last missing piece of the puzzle is the JNDI name to be used for the look-up:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-j2ee-engine SYSTEM "ejb-j2ee-engine.dtd">
<ejb-j2ee-engine>
    <enterprise-beans>
        <enterprise-bean>
            <ejb-name>ZTechInfoServiceBadiBean</ejb-name>
            <jndi-name>TechInfoServiceBADI</jndi-name>
            <session-props/>
        </enterprise-bean>
    </enterprise-beans>
</ejb-j2ee-engine>
```

Figure 4 - ejb-j2ee-engine.xml file

Figure 4 shows the ejb-j2ee-engine.xml file of the customer specific EJB DC, where the JNDI name `TechInfoServiceBADI` is defined. Not surprisingly, this exactly matches the constant `BADI_JNDI_NAME` I have shown you in Figure 2.

Advanced features

Well, of course these are just the basics and there's lots of room for advanced features. Remember that I mentioned that in the ABAP world you could even provide multiple BAdI implementations that are executed consecutively? Of course, this can also be achieved on the Java stack by obtaining multiple EJBs with different JNDI names that all have the same Interface. One solution could be to use the `sap.global.application.properties` of the Enterprise Application and define a property that contains a comma-separated list of JNDI names that will be called consecutively. As you can imagine this is just the tip of the iceberg...

Source Code

Those who would like to see the concept in action can download the files here and test-drive the demo application. The package includes a `readme.txt` that should help you getting the Local Development Components imported in your SAP NetWeaver Developer Studio. The demo has been created with SAP NetWeaver 7.0 SP 11, but they should run fine on previous versions as well.

For the real Ahhh-effect I'd recommend deploying the application with the **sap.com** prefix first and test it with the WS Navigator using the namespace prefix `com.sap`. You should get 5 entries back. Then, deploy the application with the **foo.net** prefix as well, test it again... and compare the results!

> **Note:**
>
> If you deploy the foo.net application first you'll get a deployment error, because the required EJB interfaces are not yet present at the server.

Alternative Approaches

I feel obligated to tell you that the concept I presented here is not the only mechanism on how-to provide customer exits on the Java stack of SAP NetWeaver. In 2005, my colleague Richard Andrulis and I wrote an article about a very similar concept called Providing Plug-Ins for J2EE Applications that does not require EJBs, but works with plain POJOs.

Another approach could be to leverage a Contract First approach and use the Create Web Service Provider wizard provided by SAP NetWeaver Composition Environment 7.1 which allows you to generate a WebService implementation based on a WSDL file.

Related Content

- BAdI - Definition
- How to… EJB: Accessing EJB Applications Using JNDI
- Providing Plug-Ins for J2EE Applications

Copyright