Java BAdIs, Part II: Concept Study on a Generic Framework

Applies to:

SAP NetWeaver Composition Environment 7.1 SP 3 (or higher)

Summary

This is Part II of my series of articles about Java BAdIs, which introduces a concept study on how to develop a generic BAdI framework. The presented framework builds upon the concept of EJB JNDI lookups as the base technology for developing a BAdI concept as stated in the previous article and leverages several key features of Java EE 5 and EJB 3.0 in particular; such as Java annotations and EJB interceptors.

Author(s): Matthias Steiner

Company: SAP AG

Created on: 04 November 2007

Author Bio

Matthias Steiner is a Solution Architect for the 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

This second article in a series about Java BAdIs builds upon the concept of EJB JNDI lookups as the base technology for developing a BAdI concept, as stated in the previous article. You will benefit most from the following content if you are familiar with Part I, so I'd like to strongly recommend that you read it (if you haven't done so) in order to start from common ground.

Overview

As the name already indicates, this article provides a **concept study** on how a generic and reusable BAdI framework can be developed with Java. The approach presented in this article is based on the SAP NetWeaver Composition Environment 7.1 SP3 (available as a Sneak Preview version on SDN's download section) and uses the following techniques:

- Custom Java Annotation
- EJB 3.0 (AOP-like) `@AroundInvoke` Interceptor
- Dynamic EJB invocation via Java Reflection API and JNDI look-ups

In order for all these elements to work together, some well-defined naming patterns, default values and coding conventions - adhering to the **Configuration by Exception** paradigm of EJB 3.0 - have been defined (more on this later).

To better illustrate the presented concept, the corresponding source code (a CE port from the previous article source code) has been published as a Software Component (SC), which can be downloaded here.

Development Components (DCs) Overview

Figure 1 illustrates the technical structure of the Software Components (SCs) and the contained Development Components (DCs). As you can see, two distinct SCs have been created: one with the namespace "demo.sap.com", which represents the source code as originally distributed by the software vendor. The other SC resides in the customer name namespace "foo.net" and contains the coding developed by an imaginary customer.

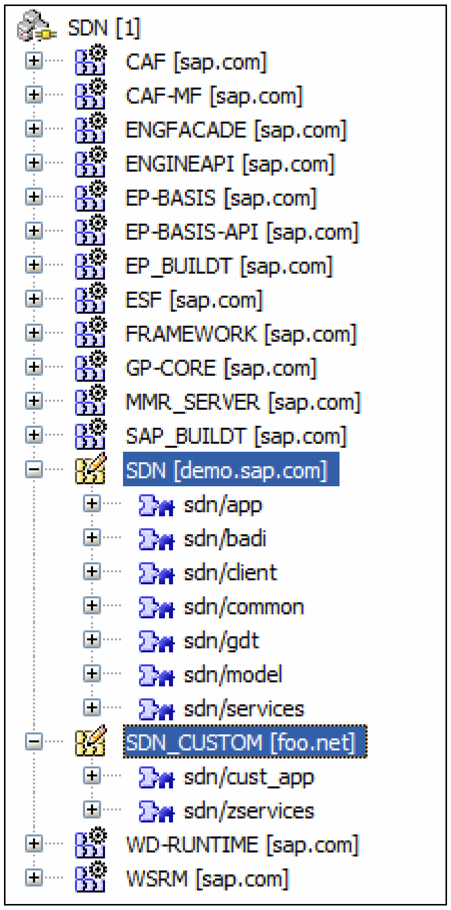

SDN [1]
CAF [sap.com]
CAF-MF [sap.com]
ENGFACADE [sap.com]
ENGINEAPI [sap.com]
EP-BASIS [sap.com]
EP-BASIS-API [sap.com]
EP_BUILDT [sap.com]
ESF [sap.com]
FRAMEWORK [sap.com]
GP-CORE [sap.com]
MMR_SERVER [sap.com]
SAP_BUILDT [sap.com]
SDN [demo.sap.com]
sdn/app
sdn/badi
sdn/client
sdn/common
sdn/gdt
sdn/model
sdn/services
SDN_CUSTOM [foo.net]
sdn/cust_app
sdn/zservices
WD-RUNTIME [sap.com]
WSRM [sap.com]

Figure 1 - Component overview

Name	Description
SDN [demo.sap.com]	
sdn/app	Java Enterprise Application assembly/deployable unit for all application development components
sdn/badi	The BAdI framework classes
sdn/client	A simple remote client to test-drive the application services
sdn/common	Provides commonly used utility classes
sdn/gdt	Provides classes from SAP's Global data type (GDT) catalogue
sdn/model	The domain model classes used by the application services
sdn/services	The application services provided as EJB 3.0 Stateless Session Beans
SDN_CUSTOM [foo.net]	
sdn/cust_app	Java Enterprise Application assembly/deployable unit for all custom development components
sdn/zservices	The custom services (EJB 3.0 Stateless Session Beans)

Note:

For easier re-distribution and installation I put all of the DCs into one SC in the provided source code. Still, I kept the distinct namespaces in order to emphasize the necessity of keeping the code lines separated from each other to facilitate easier lifecycle management.

Business Functionality

Before we dig deeper, let's have a quick recap on the "functionality" of the services provided. Like I stated last time, I kept the exemplary business service function rather easy in order not to confuse anybody with complicated business logic. So frankly speaking, all that the business service (implemented as a Stateless Session Bean) does is to return some technical information (the java system properties that is).

The method signature of this business service is shown below:

```
/**
 * Returns technical information about the host system.
 *
 * @param request - The <code>TechInfoRequest</code> information
 * @return The <code>TechInfoResponse</code>
 * @throws ServiceException - In case of an error
 */
 public TechInfoResponse getTechnicalInformation (TechInfoRequest request)
      throws ServiceException;
```

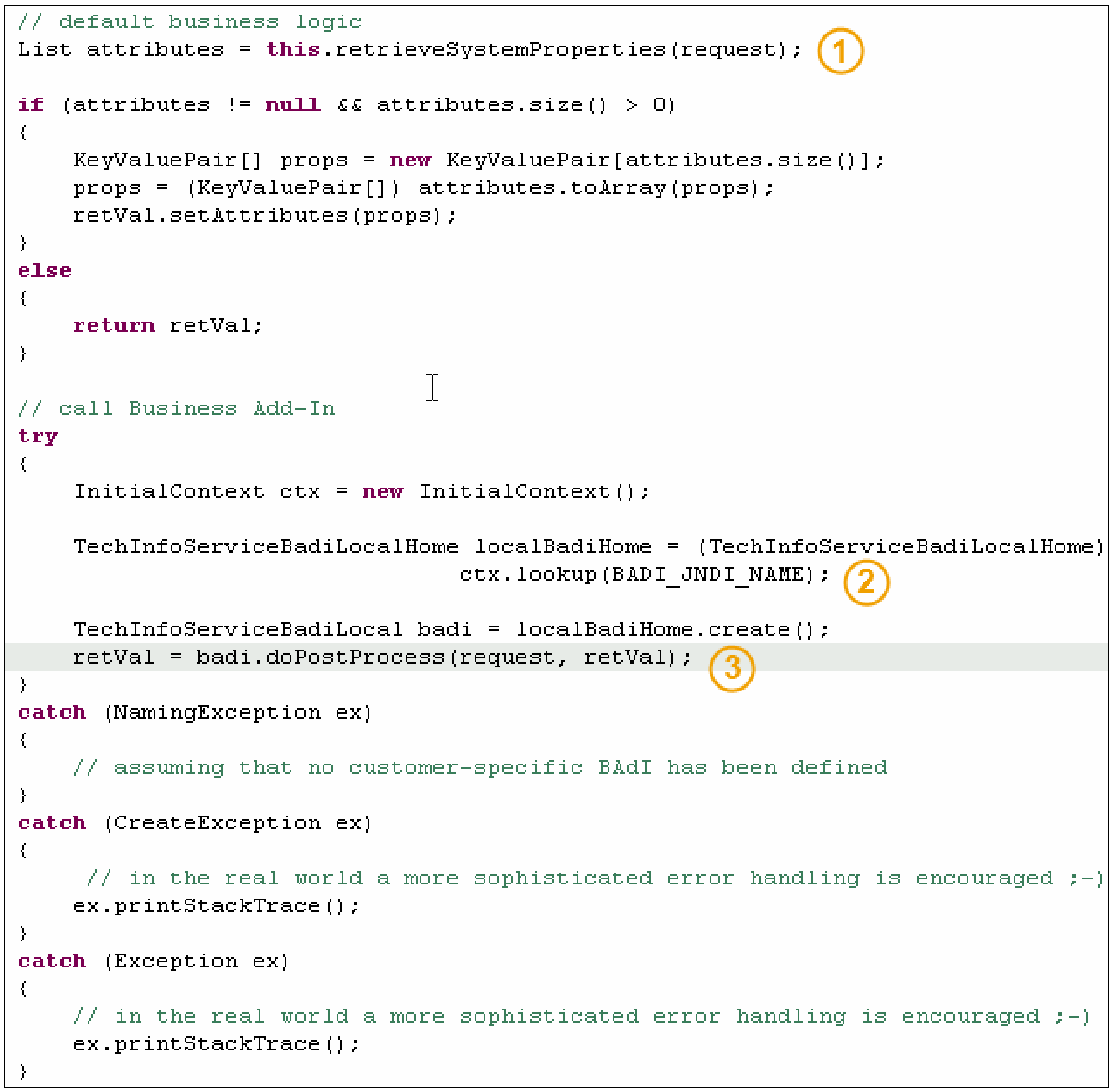
Two things should be noted/pointed-out here:

- we have dedicated objects for both the request and the response
- we throw a dedicated `ServiceException` in case of an error during the processing

Both aspects are basic design principles of a business service that is supposed to be exposed as an Enterprise Service (ES). For a better understanding of these design principles, please refer to this article about "The Anatomy of Java-based Enterprise Services".

BAdI Invocation

With this in mind, let's have a final look at how the BAdI call was developed in Part I, before we finally proceed to explore new and more exciting paths.



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

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);

    TechInfoServiceBadiLocal badi = localBadiHome.create();
    retVal = badi.doPostProcess(request, retVal);
}
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 - BAdI call coding

As you can see, most of the coding shown in figure 2 actually deals with calling the BAdI. Now, assume you are developing a complex application that contains numerous business services - coding passages as shown above would be cluttered all over the place. Let's further assume that - later-on during the development phase - you'd need to re-factor the BAdI invocation (e.g. by introducing a Dependency Injection framework a la Spring or GUICE)… you would have to change all these coding passages.

In modern software development, coding fragments, that are used throughout the application and which do not directly influence the business logic of the application are called "cross-cuttings concerns". The Aspect-oriented programming paradigm (AOP) tackles this issue by separating these cross-cutting concerns (also called aspects) from the rest of the application-specific coding by using interceptors.

Note:

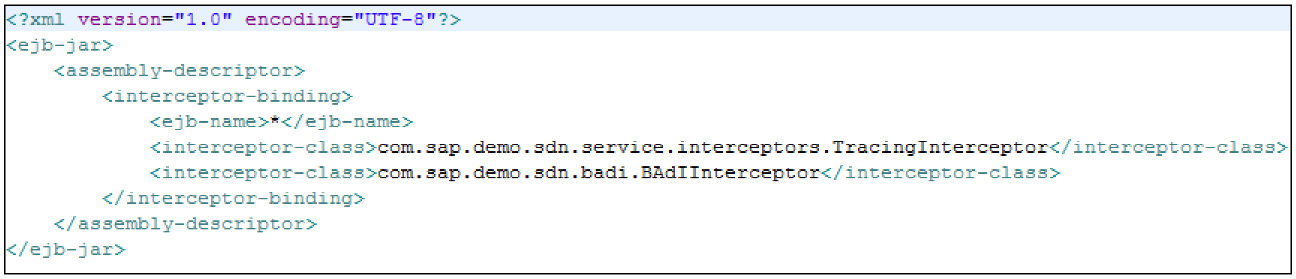
I'm aware of the fact that calling BAdI invocation a cross-cutting concern may cause debates, but please bear with me and let me continue. Let's save the discussion for later.

EJB 3.0 Interceptors

Fortunately, EJB 3.0 introduces an AOP-like interceptor model that allows us to intercept any method call of a Stateless Session Bean. (My colleague Vladimir Pavlov has written two blogs about EJB 3.0 Interceptors, which can be found here.) The interceptor type most suited to "out-source" the BAdI invocation would be the `@AroundInvoke` interceptor, because it's actually capable of performing tasks before and after the actual processing of the intercepted method call.

Several ways exist to "wire" an interceptor to an EJB Stateless Session Bean such as annotating the class/method or by using the well-known `ejb-jar.xml` deployment descriptor, which is the approach taken for this case study. By doing so, we have no dependency between the BAdI framework and the source code of the business service we would like to BAdI-enable. The reason why I mention this explicitly is that this allows integrating the BAdI framework in already existing applications without modifying the source code (more on this later).

Obviously, the below stated wiring should be optimized in a productive environment to only contain references to the EJBs that should actually be BAdI-enabled instead of using the "*" wildcard (which applies to all EJBs contained in the enclosing EJB archive.)



```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar>
    <assembly-descriptor>
        <interceptor-binding>
            <ejb-name>*</ejb-name>
            <interceptor-class>com.sap.demo.sdn.service.interceptors.TracingInterceptor</interceptor-class>
            <interceptor-class>com.sap.demo.sdn.badi.BAdIInterceptor</interceptor-class>
        </interceptor-binding>
    </assembly-descriptor>
</ejb-jar>
```

Figure 3 - ejb-jar.xml deployment descriptor

BAdI Types

The BAdI framework will provide a new `@AroundInvoke` interceptor, which alters the normal process flow of the EJB execution. Four different types/flavours of a BAdI are envisioned:

- Completely substitute the original business service with a customer-specific implementation (default type)
- Provide a pre-processing hook that alters the incoming input data of the business service and then dispatches to the original business service
- Provide a post-processing hook that alters the result of the original business service and returns the modified result transparently to the caller
- Wrap the entire business service invocation by proving both a pre- and a post-processing hook.

Java Reflection API - Coding Conventions

EJB interceptors use the Java Reflection API to access the data passed to the original business service (through the `InvocationContext` interface) in a generic approach, as the concrete service method signatures faced during runtime vary from instance to instance.

With regards to what has been said about business service operation signatures above (e.g. dedicated input and output structures), we take it for granted that the business services that should be BAdI-enabled adhere to these coding conventions, which tremendously reduces the complexity of dynamically invoking the corresponding BAdI processing methods.

The coding below shows how a concrete example of a pre-processing hook from the above shown business service could look like:

```
/**
 * Pre-processing hook for the <code>getTechnicalInformation</code>
 * operation.
 *
 * @param request - The original <code>TechInfoRequest</code>
 * @return The modified <code>TechInfoRequest</code>
 * @throws ServiceException - In case of an error
 */
 public TechInfoRequest doPreProcessing(TechInfoRequest request)
      throws ServiceException;
```

The `BAdIInterceptor` would then replace the original `TechInfoRequest` object with the one returned from the pre-processing hook and then continue with the normal process flow.

In case of the post-processing hook, the exemplary business service is passed the `TechInfoRequest` and the `TechInfoResponse` objects and is shown below:

```
/**
 * Post-processing hook for the <code>getTechnicalInformation</code>
 * operation.
 *
 * @param request - The original <code>TechInfoRequest</code>
 * @param response – The original <code>TechnInfoResponse</code>
 * @return The modified <code>TechnInfoResponse</code>
 * @throws ServiceException - In case of an error
 */
 public TechnInfoResponse doPostProcessing(TechInfoRequest request,
                                         TechInfoResponse response)
      throws ServiceException;
```

BAdI Interceptor

As you can see there's an obvious pattern in terms of the method signatures, which greatly simplifies the dynamic invocation of these hook methods via the Java reflection API. Let's have a closer look at the BAdI Interceptor coding now:

```
/**
 * Provides means to conveniently add Pre- and Post-processing hooks.
 *
 * @param invocationContext - The <code>InvocationContext</code> of the
 * intercepted operation
 * @return The original response from the intercepted service operation
 * @throws Exception - In case of an error in the intercepted service
 * operation
 */
@AroundInvoke
public Object executeBAdI(InvocationContext invocationContext)
   throws Exception
{

      Object retVal = null;

       // get some information about the intercepted operation
       Class<?> clazz = invocationContext.getMethod().getDeclaringClass();
       Method method = invocationContext.getMethod();

       // get a reference to the BAdI-Framework
       BAdIFramework badiFrmwork = BAdIFramework.getInstance();

       // check if the intercepted operation is BAdI-enabled
       boolean badiEnabled = badiFrmwork.isBAdIEnabled(invocationContext);

       if (! badiEnabled)
       {
           retVal = invocationContext.proceed();
           return retVal;
       }

       try
       {
           if (badiFrmwork.getBAdIConfiguration(invocationContext)
              .isSubstitute())
           {
               // substitute call
               retVal = badiFrmwork.substituteCall(invocationContext);
               return retVal;
           }

           // pre-process
           Object modifiedRequest = badiFrmwork.preProcess(invocationContext);


           // replace original request:
           Object[] requestParam = invocationContext.getParameters();
           requestParam[0] = modifiedRequest;

           // execute normal chain
```

```
            retVal = invocationContext.proceed();

            // post-process
            retVal = badiFrmwork.postProcess(invocationContext, retVal);

            // return final response
            return retVal;

        }
        catch (BAdIException badiEx) { … }
        catch (Exception ex) {… }
        finally { … }
    }
```

I've omitted the tracing/logging as well as the exception handling for simplicity (and due to the fact these are as well aspects, that can be dealt with via interceptors), but in general the coding shown above is very lean, yet complete. Let's have a quick code walk-through:

1. Some meta information concerning the intercepted class/method are obtained via Java Reflection API and the data provided by the `InvocationContext`
2. A reference from the (singleton) BAdIFramework class is obtained
3. A check is performed if the intercepted call is BAdI-enabled at all (here's where a custom `BAdIConfigProvider` can overwrite the default behaviour explained later on)
4. If the intercepted method call is not BAdI-enabled, the normal process flow is continued via `invocationContext.proceed();`
5. If the intercepted method call is BAdI-enabled, we act according to the BAdI type and either substitute the original call - by using different/custom specific JNDI-endpoint(s) – or putting pre- and/or post-processing hooks in place.

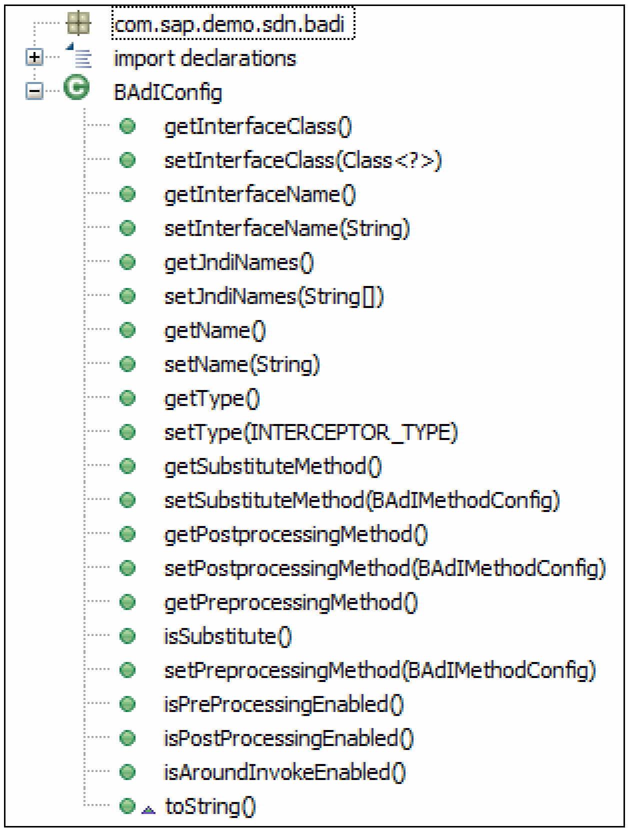
All the heavy lifting in terms of the actual hook method invocation is taken over by the `BAdIFramework` class, but as this is basically plain vanilla reflection API we'll skip this part (please consult the source code for further details) and look at the other remaining issue, which is the actual `BAdIConfiguration`.

The BAdI Configuration

The `BAdIConfiguration` provides several piece of information about the BAdI, such as:

- Name of the BAdI
- Name of the local EJB interface that comprise the processing hooks
- JNDI name(s) of the EJB implementation(s) to be executed (there can be more than one implementation, which would then be called consecutively)
- Interceptor type (substitute, pre, post or around invoke)
- Method names of the processing hooks

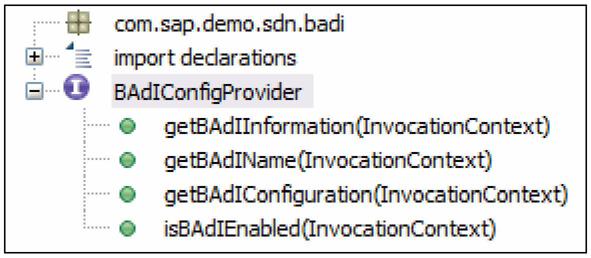
The corresponding class is shown below:



```
com.sap.demo.sdn.badi
import declarations
BAdIConfig
    getInterfaceClass()
    setInterfaceClass(Class<?>)
    getInterfaceName()
    setInterfaceName(String)
    getJndiNames()
    setJndiNames(String[])
    getName()
    setName(String)
    getType()
    setType(INTERCEPTOR_TYPE)
    getSubstituteMethod()
    setSubstituteMethod(BAdIMethodConfig)
    getPostprocessingMethod()
    setPostprocessingMethod(BAdIMethodConfig)
    getPreprocessingMethod()
    isSubstitute()
    setPreprocessingMethod(BAdIMethodConfig)
    isPreProcessingEnabled()
    isPostProcessingEnabled()
    isAroundInvokeEnabled()
    toString()
```

Figure 4 – The BAdIConfig class

In order to remain flexible in regards to the type of the BAdI configuration repository (XML, DB, etc.) to be used, an abstraction layer has been incorporated into the BAdIFramework in the form of an interface called `BAdIConfigProvider`:



```
com.sap.demo.sdn.badi
import declarations
BAdIConfigProvider
    getBAdIInformation(InvocationContext)
    getBAdIName(InvocationContext)
    getBAdIConfiguration(InvocationContext)
    isBAdIEnabled(InvocationContext)
```

Figure 5 – The BAdIConfigProvider interface

The default implementation is a hybrid, using the configuration attributes as defined by a newly introduced `BAdI` annotation and the JNDI name(s) as maintained in the `sap.application.global.properties` of the enclosing enterprise application.

This approach has been chosen for several reasons. First, annotating the business services with the `BAdI` annotation has the appealing effect that we appropriately document that this business service provides a BAdI as well as the related configuration. By using a Doclet we would be well suited to create a technical design document to hand-over. Second, the Configuration API (used to store the JNDI names of the EJBs to execute) comes free of charge, allows for changing the BAdI implementations to-be-used **at runtime** (without the need for down-time that is) and ships with administration support in the SAP NetWeaver Administrator (NWA).

Let’s have a glance at these components first and then see how they fit together to finalize our framework. Below is the source code of the BAdI annotation:

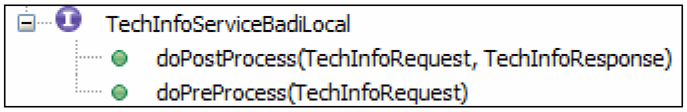
```
**
 * This <code>Annotation</code> marks a service operation (<code>Method</code>)
 * as being BAdI-enabled and allows to specify related configuration settings.
 * The <code>BAdI</code> <code>Annotation</code> is just the default way of
 * specifying the <code>BAdIConfig</code> and it's up to the
 * <code>BAdIConfigProvider</code> to provide alternative approaches of deriving
 * the <code>BAdIConfig</code>.
 */
@Target(ElementType.METHOD)
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface BAdI
{
    public enum INTERCEPTOR_TYPE { SUBSTITUTE, PRE, POST, AROUND }

    INTERCEPTOR_TYPE type() default INTERCEPTOR_TYPE.SUBSTITUTE;

    String name() default "";

    String interfaceName() default "";

    String[] operationNames() default "";
}
```

No rocket-science here either, just a very light-weight Java annotation (more information on how to write your own annotations can be found here.) The only thing of interest is probably that we need to use the `RUNTIME RetentionPolicy`, to ensure that the compiler does not discard the annotation, but keeps it available at runtime to be retrieved via Java Reflection API. Please note, that all the attributes have safe defaults to comply with the “Configuration by Exception” paradigm of EJB 3.0. For more information about the default values please consult the JavaDoc of the `BAdIBaseConfigProvider`.

So, let’s have a look at the exemplary business service to see how the `@BAdI` annotation is used with a concrete example. Please note that the coding below also shows the `@WebMethod` annotation as the operation is also exposed as a WebService operation.

```
/**
 * Returns technical information about the host system.
 *
 * @param request - The <code>TechInfoRequest</code> information
 * @return The <code>TechInfoResponse</code>
 * @throws ServiceException - In case of an error
 */
@WebMethod(operationName="getTechnicalInformation", exclude=false)
@BAdI(name="BAdI.TechInfoService",
      type=BAdI.INTERCEPTOR_TYPE.AROUND,
interfaceName="com.sap.demo.sdn.service.techinfo.TechInfoServiceBadiLocal")
    public TechInfoResponse getTechnicalInformation(@WebParam(name="request")
    TechInfoRequest request) throws ServiceException
```

Again, nothing too fancy… but it may be easier to digest if I show the corresponding BAdI interface:

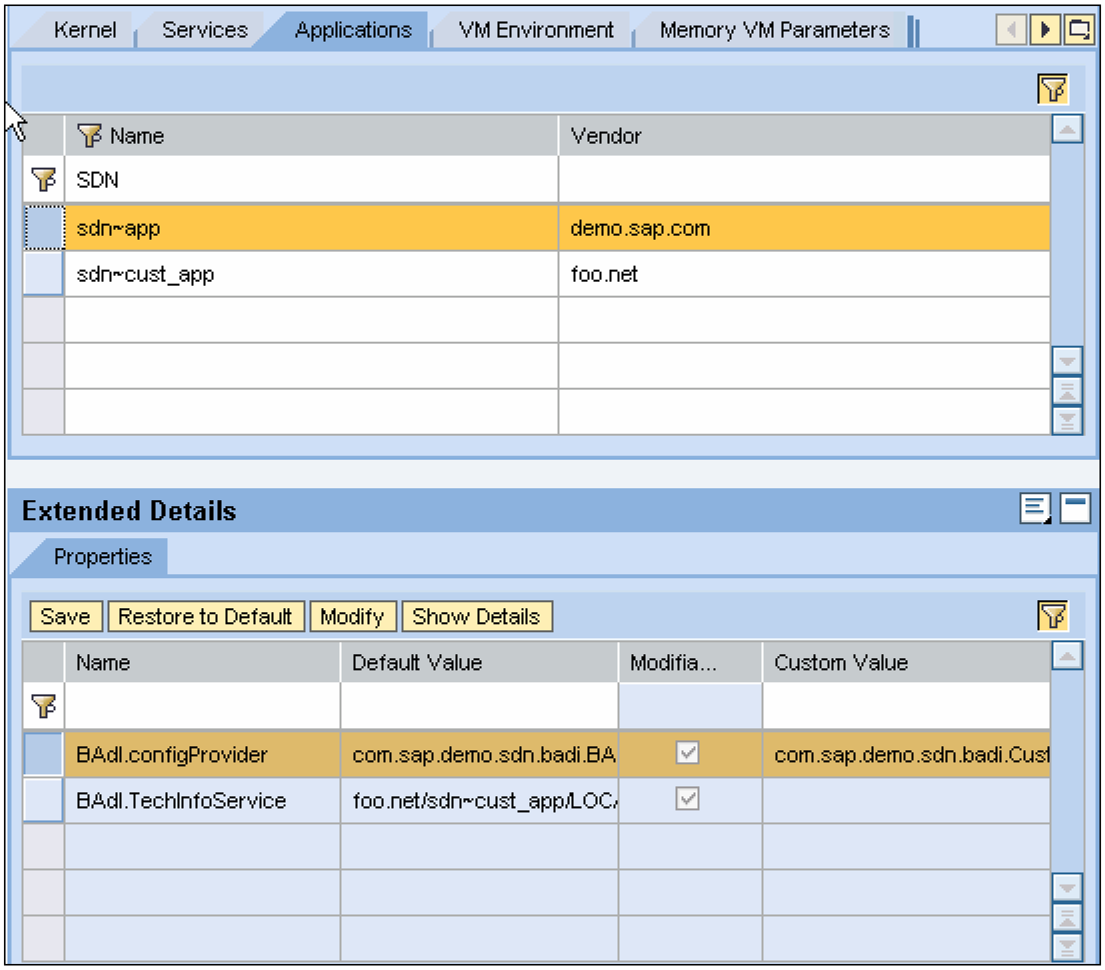


```
TechInfoServiceBadiLocal
    doPostProcess(TechInfoRequest, TechInfoResponse)
    doPreProcess(TechInfoRequest)
```

Figure 6 – BAdI Interface

As you can see, I've specified the fully-qualified class name of the EJB Local Interface that comprises the BAdI (actually I could have omitted the interface name as the default naming conventions would have "guessed" the right name.) As we have both a pre- and a post-processing hook I specified the `AROUND` interceptor type. I've omitted the operation Names since I've used the default values.

So far, so good… I specified the name of the BAdI to be "BAdI.TechInfoService". So, this is the key that is used to retrieve the JNDI name(s) of the implementation(s) to be executed. As stated above, these values can be changed at runtime via the NWA. The following screenshot shows the **Java System Properties** maintenance screen (which can be found under the **Configuration** -> **Infrastructure** tab):



Name	Vendor
SDN	
sdn~app	demo.sap.com
sdn~cust_app	foo.net

Name	Default Value	Modifia...	Custom Value
BAdI.configProvider	com.sap.demo.sdn.badi.BA	✓	com.sap.demo.sdn.badi.Cust
BAdI.TechInfoService	foo.net/sdn~cust_app/LOC/	✓	

Figure 7 – NWA Java System Properties editor

The second line here corresponds to the key stated above. If this value is modified and saved, the Configuration API informs the default implementation of the `BAdIConfigProvider` about the change via a special interface called `ApplicationPropertiesChangeListener`. This is just another reason why the default implementation is based on the Configuration API as, due to the fact that the framework gets notified about changes, we don't need to worry about a clever caching mechanism.

Custom Configuration Providers

Well, I stated that the `BAdIConfigProvider` implementation can be exchanged in order to store the BAdI configuration in any kind if repository, so let's quickly explain how this can be achieved. The first line in the screenshot above shows a property with the key "BAdI.configProvider". In order to plug-in a different `BAdIConfigProvider` (using a different repository, e.g. XML or a database) implementation one would simply need to maintain the fully-qualified class name here.

Note:

Please keep in mind that due to class-loading aspects the custom `BAdIConfigProviders` must reside in the enclosing enterprise application of the BAdI Framework DC.

Loose Ends

Before we reach the end of this article, all that is left for me to say is that since you read through all of this I really hope it was interesting for you and worth the time spent. I'd be happy to receive your comments and have a little debate, so please feel encouraged to provide your opinion and concerns in my accompanying blog post.

Known Concerns

Prior to releasing this article I've asked some of the most respected and knowledgeable colleagues of mine to review it, which resulted in interesting and diverse discussions about the nature of BAdIs and how an optimal solution for a generic BAdI framework should look. As I feel that these discussions are a valuable addition to the material presented, I'll state them below and comment on some of them.

1. **By exposing the dedicated request and response object entirely, the BAdI lacks any sense of (business) semantic.**

   True, it is a trade-off in favour of providing maximum flexibility. It is up to the implementation provider to ensure the integrity of the data. Furthermore, I would consider it a best practice to implement a BAdI with business semantics on mind, hence to better implement multiple semantic-scoped BAdIs instead of one complex implementation that addresses multiple business aspects.

2. **The design approach of defining four stereotypes (substitute, pre, post and around invocations) only allows to hook into the process on the corresponding "attachment points". Hence it is not possible to call a BAdI at a particular coding line within the business service (as it would be possible by manually invoking a Stateless Session Bean via JNDI-lookup.)**

   Also true, but when listening to the above concern I am tempted to argue that if one would want to call a BAdI within a service function then it sounds like the coding before and after the BAdI invocation are to some degree two distinct blocks that may be better split into two methods anyway.

3. **The design approach to assume pre-defined method signatures on business services that can potentially be BAdI-enabled is too strict and limits the usage of the framework.**

   Obviously, the limitation to assume dedicated input and output objects (Request and Response) only applies to the pre- and post-processing hooks. The method signature of the substitute BAdI stereotype simply reflects the existing business service signature and therefore is not subject to any *limitations. This is also the reason why this is the default stereotype. Actually, the substitute stereotype is the most powerful one and would allow to simulate the behaviour of other stereotypes (e.g. by sub-classing the original bean and using the* `super.businessService()` *at either the beginning or end of the implementation.)*

Furthermore, I would like to point out that I would not call the design approach of introducing dedicated request and response objects a limitation, but rather a well-chosen best-practice.

Related Content

- Java BAdIs (Part I)
- Source Code
- Extending Enterprise JavaBeans™ 3.0 Specification With Interceptors
- Anatomy of Java-based Enterprise Services
- EJB 3.0: Interceptors and Callbacks Made Easy - Part I
- EJB 3.0: Interceptors and Callbacks Made Easy - Part II

Copyright