The Danet Workflow Component

Maintenance Guide

Dr. Michael Lipp, Danet GmbH
Dr. Christian Weidauer, Danet GmbH
Holger Schlüter, Danet GmbH
Horst-Günther Barzik, Danet GmbH

For version 2.1.2

The Danet Workflow Component: Maintenance Guide

by Dr. Michael Lipp, Dr. Christian Weidauer, Holger Schlüter, and Horst-Günther Barzik

Table of Contents

Introduction

...

Since this is the manual for developers, it refers to the user API as well as to the internal API of the workflow engine. The JavaDoc of the internal API is not in the binary distribution, because it is relevant to developers of the engine itself only. You can generate the documentation by running "**ant doc**" in the `$DIST/src` subdirectory of the source distribution. You can then start browsing the JavaDoc with `$DIST/dist/doc/workflow/api/index.html`.

Chapter 1. Development Environment

1.1. Personal Properties

For compilation the file `"ant.properties"` has to be created in directory `"WfMOpen/personal-props"`. This file contains settings for your personal development environment. To find out which properties have to be set or adapted, look at the file `"WfMOpen/config-props/ant.properties"` which contains defaults (if there are reasonable defaults) or hints about what you have to define in your personal properties file. Note that the file in `"WfMOpen/config-props"` is under CVS control and likely to change from release to release, while the file in `"WfMOpen/personal-props"` is not and will retain its contents even when you update (the message here is: don't change `"WfMOpen/config-props/ant.properties"` unless you are modifying the build system or have to introduce some new developer dependant property - if you just want to get things running, make changes to `"WfMOpen/personal-props/ant.properties"`).

To log debug information from the application the file `config-props/log4j-appl.properties` has to be copied to directory `"WfMOpen/personal-props"` and can be adapted to your needs. Please note that the appender `"ui-messages"` and the category `"ui"` must not be removed. Otherwise the WebClient user will not get feedback (errors, warnings, info) anymore.

Compilation will fail if there is no oracle driver. The Oracle JDBC driver requires strange handling of BLOBs (they should have put sequence diagrams in the JDBC specification). We have provided a wrapper that allows transparent handling of BLOBs for all types of databases. In order to compile this, however, you need some proprietary classes from the oracle driver. As Oracle does not permit the redistribution of its JDBC driver, you will have to get one yourself.

The driver must be renamed to `oracle-*-JDBC-*.jar` (`classes12.zip` really isn't it), e.g. `oracle-8.1.7-JDBC-thin-JDK1.2.x.jar`, and put in the `tools` directory.

1.2. Building

After setting your personal properties, you can build WfMOpen by invoking "ant" in the directory `WfMOpen/src`.

Useful targets are:

`ear`	Builds the ear (and all libraries needed to assemble the ear). The results are created in `WfMOpen/dist/...` using the same layout as the binary distribution.
`deploy`	Builds the ear and deploys it in a running JBoss. The deployment is "temporary", i.e. the ear is not copied to JBoss; so after restarting JBoss, you have to deploy again.
`dist`	Builds everything in `WfMOpen/dist/...`.

1.3. JBoss

Since newer versions of the packages `org.w3c.dom`, `org.xml.sax`, `org.xml.sax.ext` and `org.xml.sax.helpers` are used the Endorsed Standards Override Mechanism of the JDK [http://java.sun.com/j2se/1.4.1/docs/guide/standards/index.html] starting JBoss needs to be supported. Therefore the system property `java.endorsed.dirs` needs to be set to the directory `$DIST/tools/endorsed`.

Furthermore, it might be necessary to define a proxy for your http connection. To provide this in-

formation to the java virtual machine start JBoss as follows (see "J2SDK Networking Properties" [http://java.sun.com/j2se/1.4.2/docs/guide/net/properties.html]):

```
JAVA_OPTS="-Djava.endorsed.dirs=$DIST/tools/endorsed -DproxySet=true -Dhttp.proxyHost=<PROXY HOST> -Dhttp.proxyPort=<PORT>" ./run.sh
```

Apart from the JBoss installation specific settings (see Section 1.1, “Personal Properties” [1]) an additional security service has to be installed. Therefore the content of file `"WfMOpen/src/de/danet/an/workflow/resources/login-conf.xml.insert"` needs to be inserted in file `"$JBOSS_HOME/server/default/conf/login-config.xml"` in front of the closing `</policy>` tag.

Chapter 2. Design Patterns and Practices

2.1. Separation of business logic

2.1.1. Introduction

If you use EJBs as base components for your system, you soon find that the source code file becomes rather long. The reason is that an EJB has to handle at least three tasks:

- provide the distribution logic (roughly everything related to the home interface)
- provide persistence
- implement business logic

One way to reduce the amount of code assembled in EJBs has traditionally been to separate the persistence related aspects in a DAO (Data Access Object). This pattern has the additional advantage of allowing the persistence mechanism to be exchanged without modifying the code of the EJB. The implementation of our workflow components, however, uses a different approach.

2.1.2. Our approach

If you design a system that is not distributed, you usually start with the implementation of your domain classes. While they may not exactly match the classes you used during analysis, they are usually quite close which keeps the learning curve low for new team members. So, why not stick to that approach and put the domain classes in the center of the design, using EJBs only to add a distribution and persistence mechanism to the domain classes.

According to this approach, you will find the following package pattern in our design:

`...domain`	Defines the domain classes with an `interface` *`DomainClass`* `...` and provides an incomplete implementation as `class` `AbstractDomainClass ...`. (Sometimes it is possible to implement the class completely. In these cases the class is named `class` `Default`*`DomainClass`*`...`.) The abstract implementation provides all business logic that is independent of a particular distribution and persistence mechanism. Classes in the domain package reference each other using the interface only.
`...ejbs[.subpkg]`	Provides the implementation of distribution and persistence for the domain classes using EJBs. The EJBs' remote interfaces extend the corresponding domain interfaces and the EJBs extend the abstract domain classes.

This pattern has the advantage of clearly separating business logic from the distribution and persistence implementation. You could perfectly well reuse the abstract classes from the domain package for e.g. a JINI based implementation. Of course, you have to declare that the methods defined in the domain interfaces throw `RemoteException`. But defining the possible distribution cuts is a design issue, not an implementation issue. The coding of the EJBs is reduced to the distribution and persistence aspects, and this is what EJBs are about anyway.

Another caveat is the usage of `equals()` (and implicitly `hashCode()`). You may never use `equals()` to determine the equality of two remotely accessible objects in the implementation of

your domain classes. The objects you want to compare are not necessarily instances of the domain classes. Due to the distributable nature of your application, either object may be the local stub of a remote object and stubs usually do not implement `equals()` properly. This has the less obvious side effect that the behavior of remotely accessible domain classes as members of sets or keys of hash table is probably not what you would expect. Again, this is a side effect of making your application distributable and not a drawback of our design pattern.

2.1.3. Compromises

As outlined above, we would like to keep the domain layer completely independent of the distribution and persistence mechanism without noticeable changes to a naive domain class implementation. Others have tried this before — it's not possible. We have to make some compromises in order to be independent of a specific persistence implementation and to support an EJB based implementation of persistence.

2.1.3.1. Making domain classes containers

Entity EJBs are "reused" objects. I.e. if we create or load an entity bean, the corresponding Java object is not created. Rather an existing object is taken from a pool and then "filled" with the data of the bean created or loaded. This is not the way you think about objects in the domain, i.e. if you design a stand-alone in-memory system.

Considering the overhead that may be associated with creating Java objects, however, it is not a bad idea to have a pool of objects that can be reused to represent different instances. Such a container object can have one of two states:

- It is "pooled", i.e. it exists but is currently not used to hold an instance.
- It is "ready", i.e. its state has been set to represent an instance of an application object.

The change of state is controlled by the derived class that implements the persistence. The pooled object is notified about the change of state by calls to the following three functions:

`protected void init (...)`	This method is called on a transition from the pooled state to the ready state if the container is to be used for a newly created instance of a domain object. If the primary key for storage is supplied by the domain object, the corresponding persistent attribute must have been initialized after the call to `init(...)`. Note that `refresh(...)` will still be called subsequently.
`protected void refresh(...) {}`	This method is called on every transition from the pooled state to the ready state. For a newly created instance of a domain object, `init(...)` will have been called before `refresh(...)`. The implementation of this method may assume that all persistent attributes have been assigned the state that corresponds to the domain object to be represented. The purpose of this method is to setup any attributes that are not persistent but depend on persistent attributes (e.g. cached values derived from persistent values by some calculation).
`protected void dispose() {}`	This method is called on every transition from the ready state to the pooled state. The purpose of this method is to release any resources that depend on the domain object represented. These resources must be reevaluated in the next call to `refresh(...)` anyway. So it is a good idea to release them in the pooled state.

2.1.3.2. EJB binding

For persistence implemented using EJBs, the methods will be called as follows. `init(...)` is called in `ejbCreate`. If the primary key is supplied by the EJB, it will be called after the primary

key has been computed. `refresh(...)` is called in `ejbPostCreate` and in `ejbLoad` after all values have been read from the database. `dispose(...)` is called in `ejbRemove` and `ejbPassivate`.

2.1.3.3. Persistent attributes

Usually attributes of domain classes are simply attributes that have some Java type and are declared `private`. This pattern cannot be maintained when persistence is implemented by a derived class as the derived class cannot access the attributes (unless you do some nasty tricks with native methods). Making the attributes `protected` is a start (though not desirable from the OO point of view). But it is not sufficient.

The problem is that the derived class cannot track changes of the persistent attributes and thus cannot optimize the storing of changes. There are tricks using byte-code post-processing used e.g. by JDO. We have, however, based our solution on the approach taken by EJB 2.0 EntityBeans. (We have not considered using EJB 2.0 EntityBeans because they have not yet been commonly available when the project was started.)

2.1.3.3.1. Declaring persistent attributes

Persistent attributes have to be declared as virtual attributes by the domain class. To declare a virtual attribute, you simply declare abstract getter and setter methods for the attribute. As an additional convention, we require the methods to start with `getPa` and `setPa`, where "Pa" stands for "persistent attribute", of course. So if you want to have a persistent attribute `name`, you declare the methods `protected abstract String getPaName();` and `protected setPaName(String newValue);` (the project environment provides the emacs command `jde-x-gen-pers-attr` for a semi-automatic template based generation).

While we would like to declare the accessor methods "private" this is, of course, not possible, so "protected" is the next best choice. The accessor method must always be declared protected. If you decide to make a persistent attribute publicly available, define an additional method `public String getPaName() { return super.getPaName(); }`. This is why we have introduced the convention to insert the "Pa": it keeps the identifier `getName` free for other uses.

2.1.3.3.2. Implementing virtual attributes

Basically, implementing a persistent attribute in the derived class that provides the persistence layer is easy. You simply declare a private attribute of the desired type (as a convention, the attribute must start with "pa", e.g. `paName`) and implement the getter and setter methods (the project environment provides the emacs command `jde-x-gen-impl-pers-attr` for a semi-automatic template based generation). The hard part is managing the value of the attribute.

The EntityBean-based solution calls `init(...)` in `ejbCreate` or `ejbPostCreate` which should assign initial values to the persistent attributes. It uses the values assigned in the subsequent SQL INSERT-statement. The values are also used in `ejbStore` for the database update. `ejbLoad` assigns the values read from the database to the attributes (and calls `refresh(...)`).

While this is easy for simple types like `int`, `String` etc., the situation is far more complex when complex attributes such as lists or maps are used.

2.1.3.3.3. Handling Relations

One type of complex attributes are collections of other persistent objects. These attributes describe relations to other persistent objects. The recommended way to handle relations is to make the persistent attribute read-only (i.e. do not define a `setPaXXX` method). The attribute is then implicitly modified by calling an abstract factory method that creates the new persistent object and updates the relation describing attribute at the same time.

Another abstract method `destroy` may then be used to remove the object and the relationship. The method may either be a parameterless method of the created object or a method of the creating class that takes the object to be destroyed as parameter.

Of course, this pattern can only be used for primary (containment) relations.

2.1.3.3.4. Handling Complex Values

There are various ways to handle collections or maps with complex values that are not relations to other persistent objects. The easiest (but not necessarily the most efficient) way is to store the values as BLOBs in the data base.

A more efficient solution may be to implement a persistent collection or map type that tracks any changes made to it and defines a method that updates the data base. The class `de.danet.an.util.persistentmaps.JDBCPersistentMap` is an example of such a class.

If you use such a persistence implementing class, make sure that the persistent attribute is initialized before `init(...)` is called or that it is passed to `init(...)` as argument.

2.2. Generating XML views

Due to the usage of Cocoon for generating HTML pages, there is a frequent demand for XML representations of "business classes" such as `WfProcess` or `WfActivity`.

Our first attempt to handle the generation of such XML representations has been to define methods for the business objects that return an XML representation. The major drawback of this approach is that it softens the distinction between model and view.

In information exchange, an XML representation might be considered to represent the complete state of a model object while the associated DTD corresponds to the object's type definition (class). In the programming environment, however, the object's type is primarily defined by its class and its state is represented by the instance in main memory. The XML representation is mostly a particular view of that object. The view can change as the XML format (DTD) used matures. Even worse, there may be two or more XML representations of the same object because standard bodies or companies couldn't agree on one format for a particular business domain. We have therefore decided to consider XML a "snapshot" view of a business object.

As a consequence, XML representations are not generated by the business objects. Rather there are static methods collected in a class called `DOMGenerator` [../util/DOMGenerator.html] that accept business objects as input and return a DOM tree that represents the state of the object.

Chapter 3. Using Cocoon

The sample web based clients provided with the workflow component are based on the cocoon [http://jakarta.apache.org/cocoon/index.html] framework.

Cocoon provides a very good framework. As many frameworks, however, it lacks usage directives for various application scenarios. We have established such a scenario for our front-ends. This scenario is described in the next section.

In order to further ease the development of web pages and to establish certain patterns for cocoon pages, we have provided supporting logicsheets and stylesheets. These logic- and stylesheets allow easy exploitation of the data offered by the generated XML and the implemented query API. They are described in the sections following the description of the front-end scenario. Note that most parts of the library can be used independent of the front-end scenario.

3.1. A web based front-end model

Most frameworks for web based application frond-ends try to re-establish some kind of model view controller (MVC) components as known from GUI toolkits. We do not feel that this is the best solution, as the initial situation found when using web servers, browsers and the mixtures of languages does not provide a nicely structured environment that lends itself to such an approach.

We rather pursue a very pragmatic approach that takes the pieces we get, especially from Cocoon, and puts them together to form something that enables us to write applications effectively. We do not aim at providing the global solution, just something usable for WfMOpen and applications that are structured alike.

3.1.1. Basic layout

We separate the application UI in the core pages and a decorating context. This separation can be found in most web applications. Usually, the decorating context displays links that support navigation, i.e. the links support the selection of the currently displayed core page. But of course, the decoration may also display additional information such as currently logged in users or other statistical information.

The application sitemap and the decorating displays are considered an integration framework for components. A component can be anything that produces only core pages and obeys the framework rules described below. In the WfMOpen management application, the staff management WAR is such a component. The WAR for engine management combines both a component that provides the core pages and the integration framework for this application.

In an ideal world (or at least in an object oriented GUI framework such as Swing), the generation of the core pages could be implemented without any knowledge about the context. Our approach defines a specific context as follows. Core pages should assume that they are displayed as a frame (referred to as "core content frame" subsequently) of a frameset. Any link provided by core pages should be targeted at the frame they are displayed in (i.e. the core content frame) unless they want the context to be changed when the link is clicked (this is further explained below). Thus if the core page is a simple page, links should have no target specification at all. If the core page is a frameset, the links in the HTML code displayed as frames should target `_parent` (if they want the next core page to replace the frameset, not just the current frame or an explicitly targeted other frame in the frameset).

Every link in a core page should give a hint about the context it wants to be displayed in. This information must be provided as a request parameter `context-info=...` of the URI (see Section 3.3.5, “Adding a parameter to a URL” [12] for the description of a helper that adds parameters to URIs in stylesheets). The information provided should allow the sitemap to initiate the generation of appropriate content for the frame(s) that make up the context. Thus the values provided should not be too fine grained (many values leading to the same frame content) nor should it be too coarse grained (making the displayed context too unspecific). Special consideration must be given to the context controlling values if something is designed as reusable component (e.g. the staff manage-

ment WAR in WfMOpen), as the values provided by a ready-made component cannot be adjusted[1].

Of course, the framework cannot really change the context if the link in a core page is targeted at the core content frame. Therefore, if an application wants the context display to be updated, it must wrap the URI in an update context request. Such a request has the form `<application-base-path>/update-context?wrapped-request="..."`. The request must be targeted at the parent of the core content frame. Do not use the target `_top`, as the application may itself be part of an even larger display. Thus, if the core page is a simple page, links should have the target `_parent`. If the core page is a frameset, the links in the HTML code displayed in frames should target `parent.parent`[2] (if they want the next core page to replace the frameset).

The sitemap of the application (in its role as framework provider) recognizes the request to display a core content page with a new context and re-generates the frameset with the appropriate context frames and a core content frame using the request wrapped in the parameter[3].

Although the parameter `context-info` will normally only be used in an explicit update context request, it should be part of any URI associated with a link on a core page. This is required due to the possibility to open a link in a new window. In this case, the browser will display the core page only in the new window without any context. Although this may sometimes be desirable, in general it is not, as it contradicts the layout specified for the application above. We therefore recommend to add JavaScript to every core page that detects its being opened without a frameset context and trigger a redisplay using the request URI (or better, a specific reload URI, see below) as `wrapped-request` parameter of an update context request. If this procedure is to work under all circumstances, every URI must include the context information[4].

3.2. XSP development

3.2.1. XSP structure

While XSPs are a good starting point for XML generation, they can lead to code that is very hard to maintain. The main topic is the gap between the Web/HTML layer and the Java layer.

We therefore follow some very strict guidelines in order to keep things properly separated and documented.

- XSPs are kept as small as possible. They make a small number of calls and simply combine the results to the produced XML output. A typical page looks like this:

  ```
  <xsp:page language="java"
  ```

[1] Of course, decisions about what is to be displayed in the context can be based on other information derived from the URI provided for the core content frame. But this assigns other information (besides the context-info parameter) the status of external interface data (from the components point of view) which must be taken into account in the further development of the component.

[2] You need JavaScript to specify this as target.

[3] Do not forget to url-encode the request URI before using it as parameter value. In the stylesheet, you may use the helper described in Section 3.3.4, “Encoding a link parameter” [12] for doing the encoding.

[4] Obviously this works only if the URI causes a complete core page to be generated. If the core page consists of a frameset and a link targets only a frame of this frameset, then the core content frame of the newly opened window may have the proper context, but will show only one frame from the frameset of the originating core page. To avoid this, we recommend to use JavaScript in the `href` attribute of the anchors that do not target the core content frame as this usually prevents the browser UI from offering the possibility to open the link in a new window.

```
            xmlns:xsp="http://apache.org/xsp"
            xmlns:xsp-session="http://apache.org/xsp/session/2.0"
            xmlns:xsp-request="http://apache.org/xsp/request/2.0"
            xmlns:misc="http://an.danet.de/xsp/misc"
            xmlns:StaffMgmt="http://an.danet.de/xsp/WfMOpen/staffmgmt"
            create-session="true">
  <page>
    <misc:generate-mappings key="staff.member.detail">
      <misc:parameter name="properties">
        <StaffMgmt:I18N_PROPS/>
      </misc:parameter>
    </misc:generate-mappings>

    <body>
      <StaffMgmt:staffMemberDetail>
        <misc:parameter name="staffMemberKey">
          <xsp-request:get-parameter name="smk"/>
        </misc:parameter>
      </StaffMgmt:staffMemberDetail>
    </body>
  </page>
</xsp:page>
```

The page combines the generation of a keymap with the generation of the main XML content.

- XSPs never contain Java code. All calls to Java are mapped by a logicsheet which is usually called `library.xsl`. The logicsheet uses its own namespace which should relate to the purpose of the functions it provides (`http://an.danet.de/xsp/WfMOpen/staffmgmt` in the example above).

  The logicsheet should only contain straight forward mappings to Java methods. The tags used by the logicsheet to provide the Java method should exactly match the name of the Java function. Parameters should have the same names as the parameters of the Java function. Except for additional attributes like `session`, the function provided by the library logicsheet is thus implicitly documented by the javadoc of the corresponding Java method.

  All classes that provide methods used by the library logicsheet should be located in the same directory as the library logicsheet.

  A special logicsheet, the "misc" logicsheet is an exception to the rule because it is a builtin logicsheet that provides both mappings to Java code and utilities on the XSL level. The naming conventions in the "misc" logicsheet therefore follow the conventions used in other Cocoon logicsheets. See Section 3.2.2, “Support Logicsheet” [9] for more information about the misc logicsheet.

- If the result produced by an XSP depends on request parameters, the mapping between the request parameter names and the corresponding Java function parameter is made in the XSP, as shown in the example above.

- To handle requests from a page that wants an action to be executed before the next page is displayed, we have developed a special action package. It is extensively documented in the javadoc of the package `de.danet.an.util.cocoon.action`.

3.2.2. Support Logicsheet

The supporting logicsheet is called the "misc" logicsheet and uses the namespace `http://an.danet.de/xsp/misc` for its templates. Any output generated by this logicsheet lives in the `util` namespace `http://an.danet.de/cocoon/util` unless otherwise noted.

The functionalities of the "misc" logicsheet are split in two parts. The first part is the "real" logicsheet which must be registered in the cocoon configuration as

```
<builtin-logicsheet>
```

```
  <parameter name="prefix" value="misc"/>
  <parameter name="uri" value="http://an.danet.de/xsp/misc"/>
  <parameter name="href"
        value="resource://de/danet/an/staffmgmt/c2client/library/misc.xsl"/>
</builtin-logicsheet>
```

The value of `href` has to be adapted appropriately, of course.

There are, however, some convenience templates that cannot be defined in the "real" logicsheet because they are used with call-template instead of matching. As Cocoon applies logicsheets one after the other, templates from another logicsheet are not available for calling. Thus we cannot call a template from the "misc" logicsheet in the application specific library logicsheet.

The "misc" helpers that are used with `call-template` are therefore collected in a file `misc-import.xsl`, which — as the name suggests — should be imported in the logicsheets where you want to use those templates.

The features provided by the "misc" logicsheet are described in the subsections following.

3.2.2.1. *Setup*

The misc logicsheet matches the `<page>` element and inserts some XML elements at the start and the end of the `<page>` subtree. The XML thus generated looks like this:

```
...
<page xmlns:u=http://an.danet.de/cocoon/util>
  <u:reload-url href="..."/>
  <u:link-base-url href="...">

  <!-- Any XML between <page> and </page> from the XSP -->

  <u:messages>
    <u:message>UI message</u:message>
  </u:messages>
</page>
...
```

The reload URL can be used to reload the page as displayed. Its computation is described in the javadoc of `de.danet.an.util.cocoon.CocoonUtil` method `defaultReloadUrl`. The link base url should be used as base for all URLs generated in the target HTML code. `<error>` elements are generated for all error messages added during request processing with `CocoonUtil.addError`.

3.2.2.2. `<get-method-param>`

Tries to find a parameter value for a Java method call first as attribute, then as child element `<misc:parameter>` of the current node. See the item below for an example.

3.2.2.3. `<generate-mappings>`

Calls `generateMappings` in `de.danet.an.util.cocoon.Mapping`. Generates mappings for internationalization. The template matches XML like:

```
...
<misc:generate-mappings key="some.key.scope">
  <misc:parameter name="properties">
    some_file.properties
  </misc:parameter>
</misc:generate-mappings>
...
```

and generates mappings for the keys thus selected. The result may e.g. look like this:

```
...
<mappings xmlns="http://an.danet.de/cocoon/util">
  <mapping key="addMember">Hinzufügen</mapping>
  <mapping key="listCaption">Mitarbeiterübersicht</mapping>
</mappings>...
```

3.2.2.4. <generate-preferences>

Calls `generatePreferences` in `de.danet.an.util.cocoon.prefs.UserPrefs`. Generates the user specific preference entries. The template matches XML like:

```
...
<misc:generate-preferences key="some.key.scope"/>
...
```

and generates preferences for the keys thus selected. The result may e.g. look like this:

```
...
<preferences xmlns="http://an.danet.de/cocoon/util">
  <preference key="sort-info">[0name|0ascending]</preference>
</preferences>...
```

3.3. Global Stylesheet

The global stylesheet provides several useful templates that can support the implementation of a page specific stylesheet.

3.3.1. General page setup

The global stylesheet includes a template that matches the root of the document to be transformed. This template generates the HTML "frame", i.e. the `<HTML>`, a `<HEAD>` block with a title and a link to the global stylesheet and a `<BODY>` with the collected errors as initial content. It then calls `xsl:apply-templates` for `//body` and closes all tags.

3.3.2. Accessing internationalized text

The template `g:get-mapping` can be used to retrieve a text string for a given key.

Parameters:

`key` the lookup key for the mapping.

The string is taken from a list of selected mapping entries (see Section 3.2.2.3, “`<generate-mappings>`” [10]) which have been generated from a mapping file `I18n_xx.properties` (with "xx" being the language identifier). The language dependant file is selected according to the current language setting of the environment.

3.3.3. Accessing user preferences

The template `g:get-preference` can be used to retrieve a user specific preference value for a given key.

Parameters:

`key` the key to the preference entry.

`default-Value` Optional default value which is returned, if key cannot be found within the preference list.

The entry is taken from a list of user specific preference entries (see Section 3.2.2.4, "`<generate-preferences>`" [11]) which have been retrieved from the database.

3.3.4. Encoding a link parameter

The template `g:url-encode` can be used to encode a string that is to be used as parameter in e.g. the `href` attribute of the anchor element. Parameters in an `href` are encoded as key value pairs (e.g. `href="action?key1=value1&key2=value2"`) where the keys and values must be `x-www-form-urlencoded`. This template does the encoding.

Parameters:

`value` the string to be encoded.

3.3.5. Adding a parameter to a URL

The template `g:add-param-to-href` appends the given name/value pair to a given URL. A "`?`" or "`&`" is inserted as separator as appropriate. The template calls `g:url-encode` for the given name and value before appending them.

Parameters:

`href` the link base URL.

`name` the request parameter name.

`value` the request parameter value.

3.3.6. Replacing a parameter within a URL

The template `g:replace-param-in-href` replaces the given parameter value within the given URL. If the parameter does not exist, it will be added. The parameter syntax of this template is identical with `g:add-param-to-href` (see Section 3.3.5, "Adding a parameter to a URL" [12]).

3.3.7. Generating links

There is a special template `g:inline-link` available for creating text or image links. Usually this template is called by referencing a link node with an attribute `hint` set to `inline`.

Parameters:

`text` the link text (for text type links).

`image` the link image (for image type links).

Note

The image attribute has precedence against the text. If both attributes are supplied, the text is used as additional `ALT` information.

`target` the link target window.

`href` the link destination URL. If the attribute `href` is not defined for the current node but the ancestor has a link-base-url node with such an attribute and the parameter *`params`* is set, the attribute `href` of the that node is used. Otherwise, the parameter `href` of the top `reload-url` is used (see Section 3.2.2.1, "*Setup*" []) .

`params` additional parameters for the link. The given string is appended as is, i.e. all names and values must be `x-www-form-urlencoded`.

`disable` flag indicating, if link is to be disabled.

3.3.8. Generating forms and fields

When creating a new form, the template `g:form-setup` should be called to set up some basics. The initialization includes the setting of the action attribute with all its href parameters, creation of the hidden fields, described by the href parameters and the setting of the accepted charset.

Parameters:

`href` the action URI to be taken. If the attribute `href` is not defined for the current node but the ancestor has a link-base-url node with such an attribute and the parameter *`params`* is set, the attribute `href` of that node is used. Otherwise, the parameter `href` of the top `reload-url` is used (see Section 3.2.2.1, "*Setup*" []).

`params` additional parameters for the link.

Note that name/value pairs passed in *`href`* or *`params`* are converted to hidden fields. As they come as part of a link or link parameters, they are assumed to be `x-www-form-urlencoded` and will be decoded when the hidden field is created.

There are several templates for the generation of input fields. All those templates are best used for matching a given dialog hint with a specific type. First of all, there is the template `g:text-input` for generating single- or multi-row input fields as well as password fields. Usually this template is called by referencing a dialog hint for a node that should be displayed for editing.

Example:

```
...
        <WorkflowProcess Name="Account anlegen" type="account_neu" key="1">
        <dialog-hint attribute="Name" type="text" maxlength="50"/>
        <dialog-hint attribute="type" type="text"/>
        <dialog-hint attribute="key" type="text"/>
        </WorkflowProcess>...
```

Parameters:

`label` label text to be displayed above the input field.

`size` size specification for the input field.

`name` the name of the field. By default, this value is taken from the attribute `attribute` of the node.

`value` default text value for the field. If this template is applied to a dialog hint, the default value is taken from the attribute with the same name of the parent node (see example above). If no such exists, the text value of the node is taken.

`rows` number of rows to be diplayed. If the number of rows is greater than 1, a text area is created.

`type`	type of field. If the node has an attribute `type` with the value `textInput`, the type is set to `text`, otherwise it is set to `password`.
`style`	reference to a css style declaration (default: `inputAlignV`).
`attribLength`	input length limitation. Normally taken from attribute `maxlength`.

Next, the template `g:choice` can be used to create a selection of different values as an input. Usually this template is called by referencing a dialog hint of type `choice` for a node that should be displayed for selection.

Parameters:

`name`	the name of the field.

The selection values are thereby taken from the child nodes "option". The display values are created by mapping these values.

3.3.9. Accessing the reload-url

For an easy access to the often needed `reload-url` (see Section 3.2.2.1, "*Setup*" []), this value is provided in a variable called `g:reload-url`.

3.3.10. Creating a new sort string

A sort string, describing the current sort order for all sortable topics, is build like this: `{0topicX;0orderX}{1topicY;1orderY}...` with a maximum of 10 (0 to 9) different items supported. Each topic forms (in combination with access method used in xsl:sort; see Section 3.4, "Sorting table columns" [18]) an expression to select the appropriate attribute (or node value) to build the sort order. If a topic is chosen for toggling the sort order, not only the sort order for that topic has to be switched but also the sequence of all topics is changed, setting this topic at the first place. The task of creating such a new sort string is done by the template `g:get-new-sort-string`.

Parameters:

`sort-info`	The current sort info string.
`sort-topic`	The topic that is chosen to be toggled.
`url-encode`	Flag, indicating if the result sort string should be url-encoded.

3.3.11. Creating a table header entry for a sortable column

This template `g:create-sortable-header-entry` can be used to create a header entry for a sortable column in a table. This entry consists of a given header text and the appropriate sort link image (depending on the current sort order an priority of the topic). The link performs the action of setting the user preference value for the current sort-info with the corresponding sort string (see Section 3.3.10, "Creating a new sort string" [14]).

Parameters:

`header-Text`	The header text to be printed.

`sort-info`	The current sort info string.
`sort-topic`	The topic that is displayed for toggling as it is named within the sort info string.
`key-prefix`	Prefix of the (generated) preference entries "sort-info". This is usually the same prefix as used for the mapping entries.

3.3.12. Creating a tab menu

This template `g:create-tab-menu` can be used to create a tab menu to be placed (usually) at the top of a window. Given information about all the entries (items) and which entry should be selected, a table with a single row is created, that holds each link element of the tab menu in a separate cell. The links call the URL provided within the item definition or reload the current URL (reload-url) with an additional request parameter `context-info`, defining the current frame context. The request parameter `context-info` should either be provided within the item's URL or the last element of the item's key path is taken by default.

To create a tab menu, a call to this template is all that has to be done. This defines the tab entries and the default selection.

By default, the tab marked as selected is chosen using the value of the stylesheet parameter `selected-tab`. If another tab entry should be selected, it can be defined as an element named `<g:selected-tab>` (`xmnlns:g="http://an.danet.de/cocoon/global"`) within the input xml data. If both element and parameter are provided, the text value of the xml element takes precedence. Thus, the template parameter is usually used only for initialization purpose.

> **Note**
>
> The mapping of the frame context to the appropriate tab selection has to be defined within the sitemap configuration. It is good practice to create the element named `<g:selected-tab>` (`xmnlns:g="http://an.danet.de/cocoon/global"`) within the page correspondend xsp, depending on the selected tab information provided by the sitemap configuration.

Parameters:

`items`	A list of all tab entries, described as a string with the following format: `"key1<,url1>;key2<,url1>;..."`. The key is mapped, using the mapping mechanism as described in Section 3.3.2, “Accessing internationalized text” [11].
`selected-key`	The key name of the current selected tab entry. If no match is found, no tab is selected. For multiple matches, each matching tab is selected.

3.3.13. Formatting date and time

The template `g:format-date-time` can be used to build a formatted string for a given date and time. Its computation is described in the javadoc of `de.danet.an.util.cocoon.CocoonUtil` method `formatDateTime`.

3.3.14. Additional string operations

The template `g:ends-with` can be used to test if a given string ends with another string. This is an extension to the XPATH 1.0 string core functions.

3.3.15. Generating drop down select box

The template `g:choice` can be used to create a selection of different values as an input. This template can be applied by referencing a dialog hint of type `list` for a node that should be displayed for selection or it may be called by name.

The following is an example of how to create a select box of all activities for a given workflow process. This select box includes an empty option and if any `process` child nodes of `process-summaries` exists, a wildcard option will be included, too. The `'account application issued'` is as *`selected-text`* selected initially in the drop down select box.

example1.xml:

```
...
        <process-summaries>
          <dialog-hint type="list"/>
          <process key="1" state="open.running.running">
            <dialog-hint attribute="key" type="text" maxlength="30"/>
          </process>
        </process-summaries>
        <WorkflowProcess Name="apply account" mgr="account_new" key="1">
          ...
          <Activities>
            <dialog-hint type="list"/>
            <Activity key="1" Name="account application issued">
              <dialog-hint attribute="key" type="text"/>
            </Activity>
            <Activity key="2" Name="account application handled">
              <dialog-hint attribute="key" type="text"/>
            </Activity>
          </Activities>
          ...
        </WorkflowProcess>...
```

example1.xsl:

```
...
        <xsl:apply-templates select="./WorkflowProcess/Activities/dialog-hint">
          <xsl:with-param name="name" select="'activityKey'"/>
          <xsl:with-param name="style" select="'data'"/>
          <xsl:with-param name="empty-item" select="'true'"/>
          <xsl:with-param name="wildcard-item" select="//process-summaries/process
          <xsl:with-param name="selected-text" select="'account application issued
          <xsl:with-param name="use-choice-label-template" select="'true'"/>
        </xsl:apply-templates>
          ...
        <xsl:template match="WorkflowProcess/Activities/Activity/dialog-hint"
          mode="choice-label">
          <xsl:value-of select="../@Name"/>
        </xsl:template>
          ...
```

result: example1.html

```
...
        <select name="activityKey" size="1" class="data">
          <option value=""></option>
          <option value="*">*</option>
          <option selected="true" value="1">account application issued</option>
          <option value="2">account application handled</option>
        </select>
          ...
```

The example below shows all the `ValidStates` and the `state` of the `WorkflowProcess` in drop down select box using the template `g:choice`.

example2.xml:

```
...
        <mappings xmlns="http://an.danet.de/cocoon/util">
          ...
          <mapping key="state.open$running$running">In Bearbeitung</mapping>
          ...
        </mappings>
          ...
        <WorkflowProcess Name="apply account" mgr="account_new" key="1">
          ...
          <State>
            <dialog-hint type="key" maxlength="20"/>open.running.running
          </State>
          <ValidStates>
            <dialog-hint type="list"/>
            <State>
               <dialog-hint type="key"/>open.not_running.suspended
            </State>
            <State>
               <dialog-hint type="key"/>closed.terminated
            </State>
        </ValidStates>
          ...
        </WorkflowProcess>...
```

example2.xsl:

```
...
        <xsl:apply-templates select="./ValidStates/dialog-hint">
          <xsl:with-param name="name" select="'MP8'"/>
          <xsl:with-param name="style" select="'data'"/>
          <xsl:with-param name="key-prefix" select="'state.'"/>
          <xsl:with-param name="selected-item" select="./State"/>
        </xsl:apply-templates>
        ...
```

result: example2.html

```
...
        <select name="MP8" size="1" class="data">
          <option selected="true" value="open.running.running">In Bearbeitun
          <option value="open.not_running.suspended">Unterbrochen</option>
          <option value="closed.terminated">Beendet</option>
        </select>
        ...
```

Parameters:

`name`	The name of the select box.
`style`	Reference to a css style declaration.
`key-prefix`	Prefix used for the mapping entries. It is mandatory if the dialog-hint type of this item is `key` and the text value is different from its mapping entry. As shown in `example2.xml`, `State` has the dialog-hint type of `key`, its text value `open.running.running` is different from the mapping entry of `state.open$running$running`. If `state.` is given as the key-prefix, the mapped result is `In Bearbeitung` and used as text of the `option` in the select box.

`key-delim`	Delimiter used for tranformation of the mapping entries, it is optional and has a default value of `$`. As shown in `example2.xml`, the mapping entry of `state.open$running$running` has `$` as *`key-delim`*.
`selected-item`	Item to be initially selected in the drop down box, it is optional (default: `/..`). Depending on the dialog-hint type of this node (`key` or `text`), either the mapped text value of this key or the text value of this node is shown in the drop down box. If this node is not included in the node of the parameter *`items`*, it is added to the drop down box automatically.
`selected-text`	given text to be selected in the drop down box, it is optional (default: `/..`). If this text is not identic to the mapped text value or text value of any node of the parameter *`items`*, it is added to the drop down box automatically. This parameter is an alternative to *`selected-item`*. If neither *`selected-item`* nor *`selected-text`* is given, no item is initially selected in the drop down box.
`items`	list of nodes used to create different values in the select box. This value is mandatory if this template `g:choice` is called and optional if this template is applied. In the latter case the following nodes are selected: `../*/dialog-hint[@type='text' or @type='key']`).
`use-choice-label-template`	flag, indicating if the choice labels of the given items which are shown in the select box should be different from the text values of the individual item node. If true, then one customized template matching the dialog-hint with mode `choice-label` must be created, in which the choice label is determined, see `example1`.
`empty-item`	flag indicating if an empty option in the select box should be generated.
`wildcard-item`	If set to a non empty node set, a wildcard option in the select box should be generated.

Note that the template `g:choice` can be called or applied. The difference is applying this template to a given node assumes this node is a dialog-hint node and has an attribute `type` with the value of `list`, otherwise this template will not be called and as a result no select box will be rendered.

3.4. Sorting table columns

In order to make columns of a table sortable, the following steps have to be taken:

- Since sorting is performed using the user preferences (see Section 3.3.11, “Creating a table header entry for a sortable column” [14]), make sure that `de.danet.an.util.cocoon.prefs.UserPrefs` is part of the action list of the appropriate sitemap file.

  Furthermore, generation of the preference entries has to be enabled within the appropriate xsp file (see Section 3.2.2.4, “`<generate-preferences>`” [11]).

- The current sort information should be stored in a variable, using the following template:

```
...
<xsl:call-template name="g:get-preference">
  <xsl:with-param name="key" select="'sort-info'"/>
  <xsl:with-param name="defaultValue">
    <xsl:value-of select="concat('{0topic;0order}', ...)"/>
  </xsl:with-param>
</xsl:call-template>
...
```

defining a default sort order and priority for each sortable column (see Section 3.3.11, “Creating a table header entry for a sortable column” [14]).

For each sortable table column, the column header should be build as follows:

```
...
<xsl:call-template name="g:create-sortable-header-entry">
...
```

Within the template, creating all table rows is usually performed as follows: `<xsl:for-each select="...">`. For each sortable column (i.e. entry in the sort-info string), the following xsl:sort instruction has to be added (example for selection of the declared attribute with sorting priority "0"; for a generic approach to select the column value, `xalan:evaluate` may be used instead to build the select expression):

```
<xsl:sort
 select="@*[name()=substring-before(substring-after($sort-info,concat($g:lb
 order="{substring-before(substring-after($sort-info,';0'),'}')}"
 data-type="text"/>
```

with the "0" value increased according to the descending priority of the entries.

Chapter 4. Implementation Guidelines

4.1. Logging

Logging is based on the logging-commons [http://jakarta.apache.org/commons/logging.html] library.

4.1.1. General usage

Logs, i.e. instances of `org.apache.commons.logging.Log` *MUST* be defined as

```
private static final org.apache.commons.logging.Log logger
        = org.apache.commons.logging.LogFactory.getLog(DefiningClass.class);
```

`DefiningClass` is the class that defines `logger` as attribute. Usage of the attribute name `logger` for the log is mandatory.

4.1.2. Using priorities

When assigning priorities to messages, keep in mind that these messages are read and evaluated by the system administrator.

`FATAL` The `FATAL` priority designates very severe error events that will presumably lead the application to abort. Due to the archtitecture of the application, it is hard to think of any circumstances where this situation can arise.

`ERROR` The `ERROR` priority designates error events that might still allow the application to continue running. This priority should be used to inform the system administrator that the program could not proceed as expected during development, e.g. because some resource is unexpectedly not available.

`WARN` The `WARN` priority designates potentially harmful situations. This priority should be used when the application can circumvent an unexpected situation but can't be sure if the solution found is what the user expected.

`INFO` The `INFO` priority designates informational messages that highlight the progress of the application at coarse-grained level.

`DEBUG` The `DEBUG` priority designates fine-grained informational events that are most useful to debug an application. Usually, debug log messages *MUST* be removed from the code after successful termination of the module implementation task. The necessity to remove debugging messages is not moderated by log4j's ability to filter messages. Debug messages are often used to find an error during implementation and amount to a lot of code lines that make code less readable. Leaving them "just in case" thus reduces code quality.

There may, however, be circumstances in which certain messages can be useful even after completion of the implementation. In those cases, debug log messages may be left in the code. The availability of such debugging support and the log4j category that enables it *MUST* be documented in javadoc or one of the manuals.

4.1.3. Logging exceptions

Stack traces from exceptions contain valuable debug information, as the included line numbers lead

directly to the cause of the error. Stack traces are, however, rather long and make reading the log file difficult.

In order to avoid unnecessary stack traces in the log, we define the following rule: stack traces are logged at the point where information from the stack trace is discarded.

This rule forbids logging in catch blocks that simply re-throw the exception, as we may assume that the exception will be logged in the calling code (if the rule has been applied correctly):

```
    ...
} catch(SomeException sx) {
    // do something
    throw sx;
}
```

On the other hand side, the rule requires logging exceptions in catch blocks that ignore a specific exception as well as in catch blocks that create a new exception based on a caught exception:

```
    ...
} catch(SomeExceptionOne sx) {
    // cannot happen because ...
    logger.error (sx.getMessage(), sx); // just in case
} catch(OtherException ox) {
    logger.error (ox.getMessage(), ox);
    throw new SomeNewException (ox.getMessage());
}
```

The latter case needs further refinement. As an exceptional case, a caught exception should not be logged if it is embedded in a newly created and thrown exception as "causing exception" and this wrapper exception is known to output the causing exception when printed. An example for this kind of exceptions is the `javax.ejb.EJBException`:

```
    ...
} catch(SomeException sx) {
    // maybe do something, but do not log!
    throw new EJBException (sx);
}
```

Of course, exceptions should not be logged if they are expected to occur in the context and indicate a certain result:

```
    ...
} catch(ObjectNotFoundException onfe) {
    // object does not exist
    return false;
}
```

4.2. UI Messages

The messages generated in the Danet Workflow Component (UI Messages) can be logged in the user interface. To archieve it, first of all, a session related logging context must be establisched by calling the method `setUILogContext` of `de.danet.an.util.cocoon.CocoonUtil`. After that, if any message needs to be logged in the user interface, calls the methods of `de.danet.an.util.cocoon.CocoonUtil` (e.g. `logErrorMapped`), then the messages are logged to categories handled by `de.danet.an.util.log4j.ListAppender` (see Section 4.1, “Logging” [21]). In the end, the method `generateMessageList` of `CocoonUtil` must be called to generate the XML representation of the UI messages. This XML output will be transformed HTML using the template `u:message` of the global stylesheet `global.xsl`.

Our provided support logicsheet (see Section 3.2.2, “Support Logicsheet” [9]) offers an example how to log UI messages.

4.3. Transaction Handling

While implementing EJBs, it is import to keep the semantics of transaction handling in mind. By default we use container managed transactions. This implies that the EJB container commits a transaction automatically for successful completion of an EJB method. In addition the EJB container rolls back automatically the transaction, if an exception is thrown and the exception is either of type `RuntimeException` or of type `RemoteException`. These exceptions are of a so-called category *system exception*, in opposite to the category of *application exception*s.

Throwing an application exception instead requires *dooming* of a possibly open transaction by the EJB itself. This is done by calling the method `setRollbackOnly()` of the associated EJB context. Subsequently, EJBs within that transaction may check the transaction state by calling the method `getRollbackOnly()` of the associated EJB context.

Note that a client's perspective on transactions is by default different from the perspective taken when implementing an EJB. An EJB acting as client of other EJBs usually has an associated transaction context, i.e. rolling back will undo previous calls to the same or other EJBs. A call from an EJB client has no transaction context unless explicitly established (and it is a general recommendation not to use transaction contexts in a client). Thus every call is committed individually.

Transactions and separation of business logic

Due to the layered approach (see Section 2.1, “Separation of business logic” [3]) a method may be fully implemented in the domain layer. It has to be kept in mind, however, that the domain layer does not know about transaction handling. While things usually work as intended as far as system level exceptions are concerned, special care must be taken for application exceptions.

If an application exception should cause a transaction rollback, the EJB must therefore implement a wrapper:

```
      public void m() {
    try {
        super.m();
    } catch (ApplicationExceptionXYZ e) {
        context.setRollbackOnly();
        throw e;
    }
}
```

As an exceptional alternative (see below), the domain level implementation may assume that the persistence layer provides a way to reset the persistent attributes and may define an abstract method `setRollbackOnly()`. This method is called whenever the domain layer interrupts execution (i.e. throws an exception) and cannot reset already modified persistent attributes to consistent values. The domain layer should, however, reset the values itself whenever possible (or even better, avoid modifications before doing checks that may lead to exceptions) as this is usually more efficient and matches the spirit of a domain level implementation better.

When to roll back

According to the EJB specification concepts, application exception should — in general — not roll back transactions. On the other hand side, however, a client may assume that a call to an EJB method of the workflow API does not leave the workflow engine in an inconsistent state.

As a consequence, we should roll back if any modifications have been made to data before the application exception has been thrown. Note that application exceptions are often thrown before modifications as they result from checks being made before the method is executed.

If the necessity to roll back cannot be related with an exception type, the domain level must inform the persistence level using an abstract method as described above.

Caveats

Note that changing the transaction behavior of a method may have a significant impact on the workflow engine. EJB method implementations assume — unless other knowledge exists — that application exceptions do not roll back transactions. Subsequent calls to EJBs' method from an EJB acting as client are usually part of a transaction. Changing a called method's semantics such that it triggers transaction rollback will cause the complete sequence of calls to be rolled back and thus affect the observed behavior of the calling method.

4.4. How to write a tool

An activity can be implemented by an application program (Tool) which links to entity Workflow Application. To write a tool, first of all, you need to define an application within the process definition in the scope of package or the dedicated process, see the following example 1.

example 1:

```
...
      <WorkflowProcess Id="example1">
        <ProcessHeader/>
        <Applications>
        <Application Id="MailTool">
          <Description>Tool to send mail
          </Description>
          <FormalParameters>
            <FormalParameter Id="recipient" Mode="IN">
              <DataType>
                <BasicType Type="STRING"/>
              </DataType>
            </FormalParameter>
            <FormalParameter Id="message" Mode="IN">
              <DataType>
                <BasicType Type="STRING"/>
              </DataType>
            </FormalParameter>
          </FormalParameters>
          <ExtendedAttributes>
            <ExtendedAttribute Name="Implementation">
              <vx:ToolAgent Class="de.danet.an.workflow.tools.MailTool">
                <vx:Property Name="DefaultSender">anyone@abc.com</vx:Property>
              </vx:ToolAgent>
            </ExtendedAttribute>
          </ExtendedAttributes>
        </Application>
        </Applications>
      <DataFields>
        <DataField Id="recipient" IsArray="FALSE">
          <DataType>
            <BasicType Type="STRING"/>
            </DataType>
          <InitialValue>mao@danet.de</InitialValue>
          <Description/>
        </DataField>
        <DataField Id="message" IsArray="FALSE">
          <DataType>
            <BasicType Type="STRING"/>
            </DataType>
          <InitialValue>account setted up</InitialValue>
          <Description/>
        </DataField>
        </DataFields>
                ...
        <Activities>
          <Activity Id="feedback" Name="account request feedback">
            <Implementation>
              <Tool Id="MailTool">
```

```
            <ActualParameters>
              <ActualParameter>recipient</ActualParameter>
              <ActualParameter>message</ActualParameter>
            </ActualParameters>
          </Tool>
        </Implementation>
        ...
      </Activity>
      ...
    </Activities>
  </WorkflowProcess>
  ...
```

An application has parameters named FormalParameters, differed by their Id, Mode and DataType; Mode indicates if the parameter is input- or output-parameter for this application. The implementation class of an application is declared in an `ExtendedAttribute` named `Implmentation` as `vx:ToolAgent`. It can have properties for which a `set` method is to be implemented in the class. Next you can refer the defined application in the dedicated activity, see the example 1 above. To call the application the parameters named ActualParameters are to be defined. They must match the FormalParameters of the application definition in the correct order. The values of ActualParameters must not be identical with the Id of the FormalParameters, but a corresponding DataField identified by its Id must be defined in the process definition. After you have defined the process definition, you need to implement the class which is declared in the `vx:ToolAgent`. This class must implement the interfaces `java.lang.Serializable` and `de.danet.an.workflow.spis.aii.ToolAgent` which defines two methods: `invoke` and `terminate`. In the `invoke` method the work is performed, the process data is updated with the result of the work and then the `doFinish` method of the `WorkflowEngine` is called with the given activity and the upated process data. The work can be terminated in the method of `terminate`. See the following example 2.

Note

Please do not attempt to perform any method on the activity except `key` and `uniqueKey` before doing the work; otherwiese, the activity becomes part of the EJB transaction and is locked, i.e. all accesses (even display in the management client) are deferred until the work is performed completely.

example 2:

```
...
    public void invoke(Activity activity, FormalParameter[] formPars,
                       Map map) {
      WorkflowEngine wfe = (WorkflowEngine)EJBUtil.createSession
        (WorkflowEngineHome.class, "java:comp/env/ejb/WorkflowEngine");
      wfe.doFinish (activity, sendMail());
    }

    private ProcessData sendMail () {
      // do sending mail
      Session mailSession = (Session) EJBUtil.lookupJNDIEntry
          ("java:comp/env/mail/WfEngine");
      // create a message
      Message msg = new MimeMessage(mailSession);
      ...
      // send the message
      Transport.send(msg);
      // build return value
      ProcessData resData = new DefaultProcessData();
      if (status != null) {
          resData.put (status, "OK");
      }
      return resData;
    }
    ...
```

Chapter 5. Documentation

...

Chapter 6. Tips & Tricks

6.1. Testing stylesheet layout

While developing stylesheet files, it is often useful to test the correct behaviour of the new code without the need to deploy the application. To achive this, the first step is to create a file with the raw input data for the stylesheet which can be done by calling `http://localhost:8080/workflow/action?debug=raw&...` instead of `http://localhost:8080/workflow/action?...` (parameters as needed for the URL to be tested). By using this target, the raw data, created by the XSP is sent as the response and you can save this file for later use.

Retrieving the action URL

To achive a proper layout for every invoked link (which may in fact only lead to a part of a frame), all `action` URLs are relocated to `display` URLs (with an enhanced parameter list). Thus, to retrieve the needed `action` URL, copy the link location within your browser (before activating it) and paste the value to the new browser window. Then edit this URL as described above.

Next, you use the data file, created above to test the stylesheet code by calling `ant` with the target `stylesheet-test` in the client directory of you context (e.g. `...workflow/clients/c2client`). The parameters of this call are `-Dsrc=<PATH TO RAW DATA FILE> -Dstylesheet=<STYLESHEET FILENAME WITHOUT PATH>`. What you get is the result of the stylesheet processing as a file `...build/stylesheet-test/out.html` that can loaded with any browser.

Index

G