# Documentation for module XmlMerge

## Purpose

The XmlMerge module is a pragmatic library to merge XML documents.

edit purpose

## Introduction

The aim of the XmlMerge module is to merge XML documents. Merging means producing a new document out of several source documents. Merging XML documents can be useful in many situations, such as adding modularity to configuration files, deployment descriptors or build files. XMLMerger internally relies on JDOM.

Here is a merge example:

| original | | patch | | result |
|---|---|---|---|---|
| `<root> <a> <b/> </a> <d id="0"/> <d id="1"/> </root>` | + | `<root> <a> <c/> </a> <d id="1" newAttr="2"/> </root>` | = | `<root> <a> <b /> <c /> </a> <d id="0" /> <d id="1" newAttr="2" /> </root>` |

To obtain such a merge, here is the code:

```
public String merge(String original, String patch) {
        Configurer configurer = new
PropertyXPathConfigurer("xpath.1=/root/d \n matcher.1=ID");
        return new ConfigurableXmlMerge(configurer).merge(new String[] {
original, patch} );
}
```

In the sample above, we configure that for the merging of the part `/root/d` one should use the `ID` matcher.

The design is configurable and extensible in order to fulfill any requirement in the behavior of the merge. The rest of this document explains how to use the module and how to extend it.

Note that the design is focused towards flexibility and extensibility and not performance.

**Quick Reading Guide**: if you want to get a quick overview read only the following sections:

- How to use
- Original and Patch
- Processing model
- Operations
- Aliases for built-in operations
- Configuring with XPath and Properties

## Module contents

The module contains the following stuff:

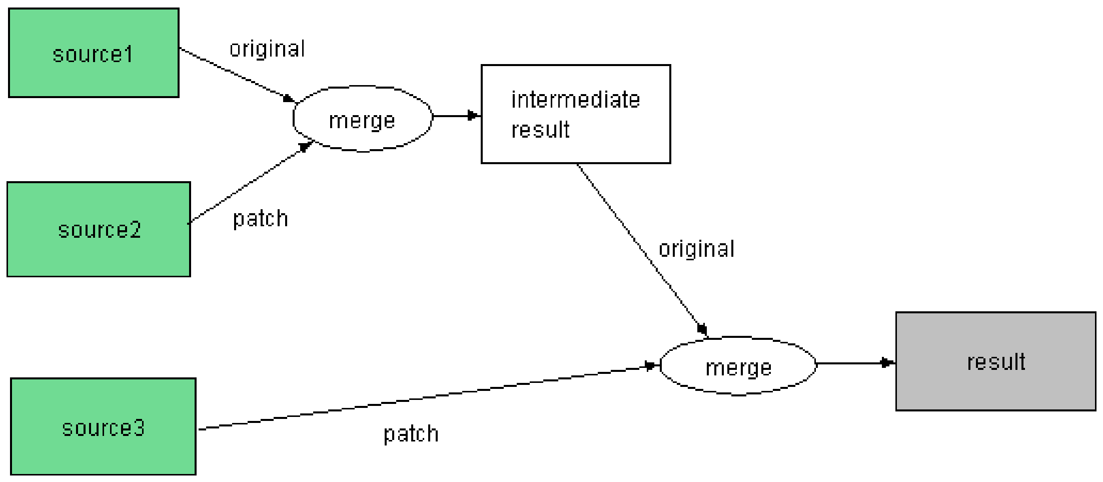
- Interfaces and infrastructure supporting the concepts the module is based on (in fact this forms the API and SPI).
- Default implementations of these interfaces.
- Convenience support for configuring your merge using XPath (outside of the XML documents) or with XML attributes within the XML documents to merge.
- Tool to merge XML files from the command-line.
- Ant task for merging XML files from ant scripts.

- SpringFramework resource implementation merging XML files read from other resources.
- Web application to rapidly demonstrate the module.

## Important concepts

### Original and Patch

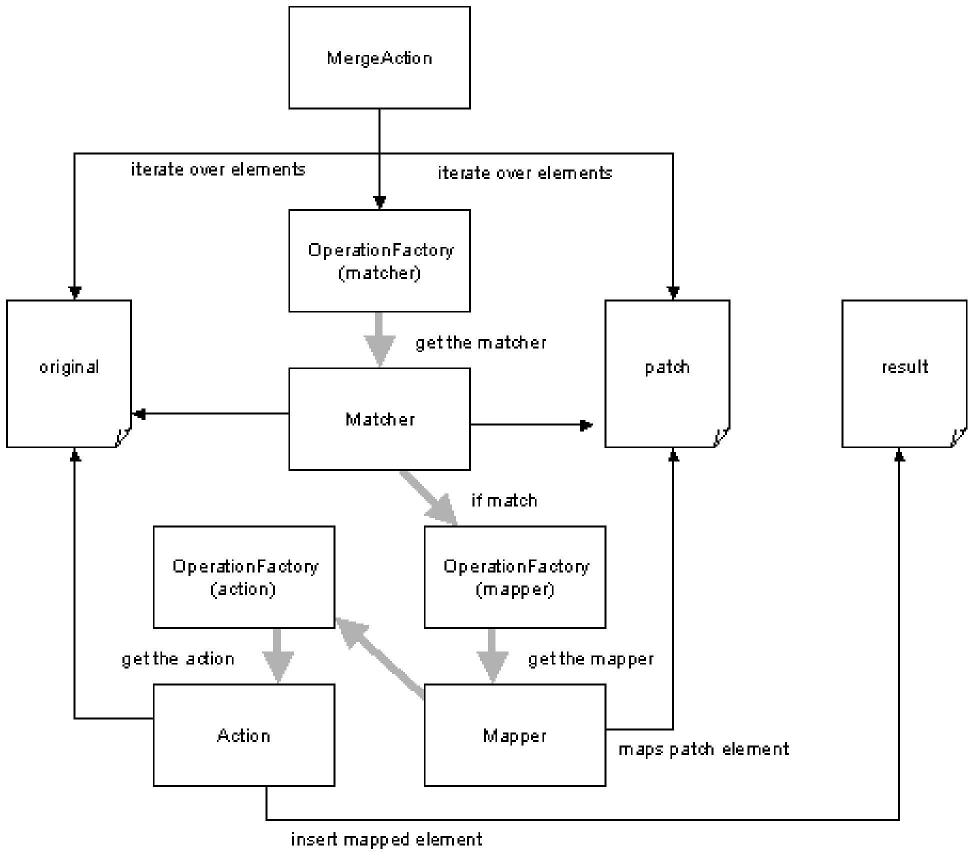
The *sources* are The XML documents (as java.lang.String, java.io.InputStream or org.w3c.dom.Document) that we want to merge.



Several sources can be given, but the merge is always performed two-by-two, using an *original* and a *patch* document. For example, when merging three documents, the *result* of the merge of the two first documents is used as *original* for merging with the third.

### Processing model

The natural way of merging documents is to recursively traverse the elements of each *original* and *patch* document and apply the following process to each element.



In the picture above, in the boxes OperationFactory (matcher), OperationFactory (action) and OperationFactory (mapper) one can plug-in particular implementations. There is a simplification in the picture: *action* works on the parent node, the original node, and the patch element that was already mapped.

## Core Concepts as Java Interfaces

*See also the javadocs.*

The XmlMerge infrastructure is based on the following concepts. For each of them, a Java interface is defined in the module.

## Operations

- **Matcher**. A *matcher* implements a way to compare two XML elements (one of the original document and one of the patch document) and decides if they match.
- **Action**. When two elements match, an *action* is applied on both to produce the *result* element and the *result element* is inserted in the *result* document. An action can also be destructive, i.e. it can insert nothing in the result.
- **Mapper**. Before applying the action, the *patch* element is optionally transformed by a mapper to give it the right form to appear in the *result* document.

NB: If an element of the *original* document does not *match* any *patch* element, it is nevertheless passed to the action with `null` as *patch* element. Respectively, if the *patch* element does not match any *original* element, the action is applied with `null` as *original* element.

## Configuration with Factories

Used terminology

- **Operation**. Concepts and marker interfaces covering Matcher, Action and Mapper for using factories.
- **OperationFactory**. Provides the corresponding operation for a pair of *original* and *patch* element. The implementation of the factory decides according to its configuration which implementation of the operation must be applied to the pair of elements.
- **MergeAction**. Sub-interface of the **Action** interface. This is the kind of action implementing the traversing of the element's sub-elements and applying the corresponding matcher, mapper and action. Hence, a **MergeAction** is configured by dependency injection with the **OperationFactory** objects providing the mappers, matchers and actions for the sub-elements. Note that a **MergeAction** is also responsible to pass the factories to the merge actions applied to the sub-elements.

- **XmlMerge**. Entry-point to perform the merge, it provides the `merge` methods. One can configure it by injecting the **MergeAction** and **Mapper** applied to the root element.
- **Configurer**. Interface for convenience classes configuring the root merge action and root mapper of an XmlMerge instance; thus, it can also configure the operation factories. This way, with only a few lines of code, one can use XmlMerge. The **ConfigurableXmlMerge** wrapper class automatically applies a **Configurer** on an XmlMerge instance.

## Built-in implementations

### Operations

The module provides implementations of operations that are commonly used:

#### Matchers

- **TagMatcher**. The original and patch elements match if the tag name is the same.
- **IdentityMatcher**. The original and patch elements match if the tag name are the same and the *id* attribute value are the same.
- **SkipMatcher**. The original and patch elements never match. Useful to force inserting the elements.

#### Mapper

- **IdentityMapper**. "Do nothing" mapper, it returns an exact copy of the element.
- **NamespaceFilterMapper**. Maps by removing all elements and attributes of a given namespace. Useful with the **AttributeOperationFactory** which allow defining the actions to apply as attributes in the patch document.

#### Actions

- **OrderedMergeAction**. Default merge action. It traverses parallelly the original and patch elements, the matching pairs are determined in the order of traversal. This is generally sufficient for all usage because most of original and patch documents will have elements in the same order.

- **ReplaceAction**. Replaces the original with the patch element or creates the element if not in original.
- **OverrideAction**. Replaces with the patch element only if it exists in the original.
- **CompleteAction**. Copy the patch element only if it does not exist in the original.
- **DeleteAction**. Copy the original element only if it does not exist in the patch. If it exists in the patch, then nothing is added to the result.
- **PreserveAction**. Invariantly copies the original element regardless of the existence of patch element.
- **InsertAction**. Inserts the patch element after elements of the same already existing in the result. Use with the **SkipMatcher** to merge on one level and keep the same relative order of elements.
- **DtdInsertAction**. Inserts the patch element in the result according to the order specified in the original document's DTD. Use with the **SkipMatcher** to merge on one level and make the document valid.

## Aliases for Built-In Operations

For convenience in configuration, the built-in operations have short aliases, so that we can refer to them using the aliases instead of the full class names:

- Matchers: `TAG`, `ID`, `SKIP`.
- Mappers: `IDENTITY`
- Actions: `MERGE`, `REPLACE`, `OVERRIDE`, `COMPLETE`, `PRESERVE`, `INSERT`, `DTD`.

These constants are defined in the classes **StandardMatchers**, **StandardMappers**, **StandardActions**.

## XmlMerge Implementation

The `DefaultXmlMerge` class applies the `OrderedMergeAction`, `TagMatcher` and `IdentityMapper` to all elements.

## Operation Factories

Three implementations of the operation factory are provided:

- **StaticOperationFactory**. Returns the same operation for all element pairs. Used when the same behavior applies to all elements of the document.
- **XPathOperationFactory**. Configured with a map of {XPath, Operation}, it returns the operation of the first XPath matching the element path.
- **AttributesOperationFactory**. Configured with attributes in the patch element.

## Configuring your Merge

You have currently three ways to configure an XmlMerge instance:

### Programming the Configuration

This is the most powerful but tedious way to configure. You create the instances of the root merge action, root mapper and factories programmatically. Example:

```
<root>          <root>          <root>
  <a/>            <a>             <a>
  <c/>              <b/>            <b/>
</root>    +      </a>      =     </a>
                  <c>             <c/>
                    <d/>        </root>
                  </c>
                </root>
```

original          patch          result

```
public void testXPathOperationFactory() throws Exception {

        String[] sources = {
                "<root><a/><c/></root>",
                "<root><a><b/></a><c><d/></c></root>" };

        XmlMerge xmlMerge= new DefaultXmlMerge();

        MergeAction mergeAction = new OrderedMergeAction();

        XPathOperationFactory factory = new XPathOperationFactory();
        factory.setDefaultOperation(new CompleteAction());

        Map map = new LinkedHashMap();
        map.put("/root/a", new OrderedMergeAction());

        factory.setOperationMap(map);

        mergeAction.setActionFactory(factory);

        xmlMerge.setRootMergeAction(mergeAction);

        String result = xmlMerge.merge(sources);

        String expected =
        "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + NL +
        "<root>"+ NL +
        "  <a>"+ NL +
        "    <b />"+ NL +
        "  </a>"+ NL +
        "  <c />"+ NL +
        "</root>";

        assertEquals(expected.trim(), result.trim());
}
```

Note that this kind of configuration can be interesting in conjunction with the SpringFramework, since these components can be configured in Spring configuration files.

## Configuring with XPath and Properties

The most usual way is to configure XmlMerge with the **PropertyXPathConfigurer** which uses a `Properties` object.

The properties define XPath entries and the associated matchers, mappers and actions. Syntax:

```
xpath.pathName=XPath
```

`matcher.pathName=Matcher alias or class name` `mapper.pathName=Mapper alias or class name` `action.pathName=Action alias or class name`

By default, the **OrderedMergeAction**, **IdentityMapper** and **TagMatcher** is used for all elements.

Example:

`test.properties`:

```
   action.default=COMPLETE

   xpath.path1=/root/a
   action.path1=MERGE
```

```
 <root>          <root>          <root>
   <a/>            <a>             <a>
   <c/>             <b/>            <b/>
</root>    +     </a>      =     </a>
                   <c>            <c/>
                    <d/>        </root>
                 </c>
               </root>
```

original          patch          result

```
public void testPropertyXPathConfigurer() throws Exception {

        String[] sources = {
                "<root><a/><c/></root>",
                "<root><a><b/></a><c><d/></c></root>" };
        Properties props = new Properties();
        props.load(getClass().getResourceAsStream("test.properties"));
        Configurer configurer = new PropertyXPathConfigurer(props);
        XmlMerge xmlMerge= new ConfigurableXmlMerge(configurer);

        String result = xmlMerge.merge(sources);

        String expected =
        "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + NL +
        "<root>"+ NL +
        "   <a>"+ NL +
        "     <b />"+ NL +
        "   </a>"+ NL +
        "   <c />"+ NL +
        "</root>";

        assertEquals(expected.trim(), result.trim());
}
```

## Configuring with Inline Attributes in Patch Document

Another way, to avoid using external `Properties` and show explicitely the merge behavior in the patch document, is to use the **AttributeMergeConfigurer**.

You simply add attributes with a special namespace in the patch elements describing the operations to apply. Example:

```
<root>
  <a>
    <b/>
  </a>
  <d/>
  <e
id='1'/>
  <e
id='2'/>
</root>
```

original

+

```
<root
xmlns:merge='http://xmlmerge.el4j.elca.ch'>
  <a merge:action='replace'>hello</a>
  <c/>
  <d merge:action='delete'/>
  <e id='2' newAttr='3'
  merge:matcher='ID'/>
</root>
```

patch

=

```
<root>
<a>hello</a>
  <c />
  <e id="1"
  />
  <e id="2"
newAttr="3"
  />
</root>
```

result

```

public void testAttributeMerge() throws Exception {
        String[] sources = {

                "<root>                           " +
                " <a>                             " +
                "  <b/>                            " +
                " </a>                            " +
                " <d/>                            " +
                " <e id='1'/>                           " +
                " <e id='2'/>                           " +
                "</root>                            ",

                "<root xmlns:merge='http://xmlmerge.el4j.elca.ch'>
" +
                " <a merge:action='replace'>hello</a>
" +
                " <c/>
" +
                " <d merge:action='delete'/>                           " +
                " <e id='2' newAttr='3' merge:matcher='ID'/>
" +
                "</root>
"
        };


        String result = new ConfigurableXmlMerge(new
AttributeMergeConfigurer()).merge(sources);

        String expected =
        "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + NL +
        "<root>"+ NL +
        "  <a>hello</a>"+ NL +
        "  <c />"+ NL +
        "  <e id=\"1\" />" + NL +
        "  <e id=\"2\" newAttr=\"3\" />" + NL +
        "</root>";

        assertEquals(expected.trim(), result.trim());

}
```

## Writing your own Operations

It is easy to customize and extend the behavior of the XmlMerge module by writing new operations.

For example, one may want to merge `web.xml` files. To add a new parameter to an existing servlet, we must match the right servlet entry, thus match using the tag <servlet-name>. See below an example of a new **Matcher** implementation, the **ServletNameMatcher**.

```
    <web-app>
     <servlet>
    <servlet-name>
         hello
   </servlet-name>
   <servlet-class>

test.HelloServlet
      </servlet-
      class>
     </servlet>

     <servlet>
    <servlet-name>
         bye
   </servlet-name>
   <servlet-class>

 test.ByeServlet
      </servlet-
      class>
     </servlet>

 <servlet-mapping>
    <servlet-name>
         hello
   </servlet-name>
    <url-pattern>
        /hello
    </url-pattern>
     </servlet-
     mapping>
```

+

```
    <web-app>
     <servlet>
    <servlet-name>
          bye
   </servlet-name>
    <init-param>
       <param-name>
          message
        </param-
       name>
          <param-
      value>
          Bye bye!
        </param-
      value>
    </init-param>
    </servlet>
   </web-app>
```

=

```
    <web-app>
     <servlet>
    <servlet-name>
         hello
   </servlet-name>
   <servlet-class>

test.HelloServlet
      </servlet-
      class>
     </servlet>

     <servlet>
    <servlet-name>
         bye
   </servlet-name>
   <servlet-class>

 test.ByeServlet
      </servlet-
      class>
     <init-param>
       <param-name>
          message
        </param-
       name>
          <param-
      value>
          Bye bye!
        </param-
      value>
    </init-param>
```

```
<servlet-mapping>
  <servlet-name>
        bye
  </servlet-name>
   <url-pattern>
        /bye
  </url-pattern>
   </servlet-
   mapping>
  </web-app>
```

original

```
```

patch

```
   </servlet>

<servlet-mapping>
  <servlet-name>
       hello
  </servlet-name>
   <url-pattern>
       /hello
  </url-pattern>
   </servlet-
   mapping>

<servlet-mapping>
  <servlet-name>
        bye
  </servlet-name>
   <url-pattern>
        /bye
  </url-pattern>
   </servlet-
   mapping>
  </web-app>
```

result

Ensure your **ServletMatcherClass** is in the classpath and configure it in the XPath properties:

```
xpath.path1=/web-app/servlet
matcher.path1=com.mycompany.ServletNameMatcher

# Do not touch the existing name
xpath.path2=/web-app/servlet/servlet-name
action.path2=PRESERVE

# Do not touch existing init-params
xpath.path3=/web-app/servlet/init-param
action.path3=INSERT
```

**ServletNameMatcher** implementation:

```
package com.mycompany;

public class ServletNameMatcher implements Matcher {
        public boolean matches(Element originalElement, Element
patchElement) {
                String originalServletName =
originalElement.getChildText("servlet-name");
                String patchServletName    =
patchElement.getChildText("servlet-name");

                return patchServletName != null && originalServletName !=
null &&
                originalServletName.trim().equals(patchServletName.trim());
        }
}
```

## How to use

This section shows the different possibilities how this module can be used.

### Command-line Tool

The module includes a tool to merge XML files from the command-line.

To be able to use the command-line tool, you have to execute the following steps:

- Go to `EL4J_HOME/framework`
- Recursively compile all required targets files: `ant jars.rec.module.module-xml_merge`
- Create an executable distribution of the xml_merge module: `create.distribution.module.eu.module-xml_merge.console`
- The executable distribution can be found in the `module-xml_merge-default` folder under `EL4J_HOME/framework/dist/distribution`. You can copy this folder to any location you want.
- To be able to execute the command-line tool from your desired location, you have to add the location containing the executable distribution your `PATH` environment variable:

- **Windows**: add `YOUR_LOCATION\module-xml_merge-default` to the right end of your PATH environment variable, where `YOUR_LOCATION` denotes the folder into which you have copied the module-xml_merge-default folder.
- **Unix**: launch the following command to set the `PATH` environment variable, where `YOUR_LOCATION` denotes the folder into which you have copied the `module-xml_merge-default` folder: `export PATH=$PATH:"YOUR_LOCATION/module-xml_merge-default"`

The previous steps have to be executed only once. You are now ready to launch the command-line tool from any location by launching the xmlmerge script:

```
xmlmerge [-config <config-file>] file1 file2 [file3 ...]
```

In this command, config-file denotes an optional XPath property file and file1, file2, file3 etc are the xml files to merge. The result is outputted on the standard output.

## Ant Task

The module also includes an Ant task for merging XML files from ant scripts.

Here is an example which shows the usage of this Ant task in a `build.xml` file:

```
   <target name="test-task">
       <taskdef name="xmlmerge"
classname="ch.elca.el4j.xmlmerge.anttask.XmlMergeTask"
           classpath="module-
xml_merge.jar;jdom.jar;jaxen.jar;saxpath.jar"/>

       <xmlmerge dest="out.xml" conf="test.properties">
           <fileset dir="test">
             <include name="source*.xml"/>
          </fileset>
       </xmlmerge>
   </target>
```

In this task, `dest` denotes the output merged file, and `conf` denotes an optional XPath property file. The indicated fileset selects the files to merge (in this example, the files in the `test` directory whose name begins with `source` will be merged).

The jar files which are needed on the classpath to execute this task are `module-xml_merge.jar`, `jdom.jar`, `jaxen.jar` and `saxpath.jar`. The `module-xml_merge.jar` file can be found in the `EL4J_HOME/framework/dist/lib` folder, and the three other ones can be found in the `EL4J_HOME/framework/lib` folder. If you have created an executable distribution of the xml_merge module (see command-line tool), you can also find these libraries in the `lib` folder of the executable distribution.

## Spring Resource

You can also use this module to create an XML Spring Resource on-the-fly by merging XML documents read from other resources. Here is a configuration example:

```
   <bean name="merged"
class="ch.elca.el4j.xmlmerge.springframework.XmlMergeResource">
        <property name="resources">
            <list>
                <bean
class="org.springframework.core.io.ClassPathResource">
                    <constructor-arg>
                        <value>ch/elca/el4j/xmlmerge/r1.xml</value>
                    </constructor-arg>
                </bean>
                <bean
class="org.springframework.core.io.ClassPathResource">
                    <constructor-arg>
                        <value>ch/elca/el4j/xmlmerge/r2.xml</value>
                    </constructor-arg>
                </bean>
            </list>
        </property>
        <property name="properties">
```

```
            <map>
                <entry key="action.default" value="COMPLETE"/>
                <entry key="xpath.path1" value="/root/a"/>
                <entry key="action.path1" value="MERGE"/>
            </map>
        </property>
    </bean>
```

This configuration example is also part of the module and can be found in the `conf/template/xmlmerge-config.xml` file.

## Web demo

The module also contains a web application to demonstrate how XML documents can be merged.

To be able to launch the web application, you have to execute the following steps:

- Go to `EL4J_HOME/framework`
- Recursively compile all required targets files: `ant jars.rec.module.module-xml_merge`
- Deploy the demo application into Tomcat: `ant deploy.war.module.eu.module-xml_merge.web`
- Open in http://localhost:8080/xmlmerge/demo a browser.

## Debug output

To set up some logging facility (to show what is going on in case of problems): Add the following command line switch: `-Dxmlmerge.debug=true`.

# References

- Analysis about general merging of XML (shows that the "perfect XML merge is highly complex and that a pragmatic approach seems reasonable): http://www.cs.hut.fi/~ctl/3dm/thesis.pdf

- JavaWorld article of Laurent Bovet: http://www.javaworld.com/javaworld/jw-07-2007/jw-07-xmlmerge.html