EL4J Reference Documentation Version 1.2

Incremental Improvements for Spring

Rapport	Version	Date	Author(s)	State	Visa
123456	1.46	29 Nov 2007 - 14:47	ABU, MZE, POS, RBO, AMA	Valid	POS

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1 Introduction

EL4J (http://el4j.sourceforge.net/), the Extension Library for the J2EE, adds incremental improvements to the Spring Java framework (http://www.springframework.org/). Among the improvements are:

- The ability to split applications in modules that each can provide their own code and configuration, with transitive dependencies between modules
- Simplified POJO remoting with implicit context passing, including support for SOAP and EJB
- A light daemon manager service for long-running daemons
- Support to see the active beans and their configuration in JMX
- A light exception handling framework that implements a safety facade
- Improvements for Spring RCP

Used libraries and tools

- Most libraries that are included in the spring framework
- Maven 2

For another short introduction to EL4J we refer to the EL4J datasheet available on our company webpage.

EL4J is a package for Java developers - ready to start working. It is an explicit goal of EL4J that you should not loose time and be able to get working right away. From version 1.1 it is published under the LGPL (http://www.gnu.org/licenses/lgpl.txt) at sourceforge. Please contact info@elca.ch for other licensing.

EL4J is already in use in 16+ projects within ELCA (http://www.elca.ch).

This documentation was auto-generated from content of our twiki. Some of the URL-links are undefined (due to the way we created it) and some content is still emerging.

2 Unique Features of EL4J

This document lists the distinctive features of EL4J. A frequent question about EL4J is what it provides additionally to the frameworks it includes. One benefit of EL4J is certainly the selection, integration, and pre-configuration of leading components. More benefit comes from the *new* features that EL4J provides.

The following list shows the distinctive features of EL4J (this list is not exhaustive, please check also the module documentation and the javadoc):

- Application templates to get quickly started: for GUIs and Web UIs. The goal is to have a running sample application within 10 minutes! In this running application you have a proven structure and sample solutions for typical development issues.
- Support for modules with code, default configuration and dependencies. This feature is based on the build system (Maven 2), the basic spring abstractions, some EL4J support and conventions.
    - ♦ More flexible and robust loading of configuration resources
        - ◊ Inclusion and exclusion list to include/ exclude configuration files
        - ◊ Store the list of configuration resources to load in the jar-file manifest
        - ◊ Merging of spring configuration: adding more parameters to an existing list of parameters
    - ♦ Each EL4J module packages functionality with samples, documentation and default configuration.
- Improved remoting
    - ♦ Easier switching between remoting protocols (unification of remoting protocols)
    - ♦ Remote POJOs via SOAP (simpler than with basic Spring), support for JAXB
    - ♦ Auto-generation of RMI-wrappers for POJOs (via Interface Enrichment)
    - ♦ Provide light load-balancing via the more flexible remoting layer
    - ♦ Implicit context passing over process boundaries
    - ♦ Automatically deploy POJOs as EJB 2.1 beans (currently frozen)
- EL4J cockpit
    - ♦ Auto-publication of the list of spring beans with their configuration values, interceptors and other useful info
    - ♦ Get a simple overview of the running threads
    - ♦ Change the log4j configuration dynamically
- Exception handling
    - ♦ Exception handling guidelines
    - ♦ Safety facade
    - ♦ More exception mappings for database accesses (additionally: duplicate values, out of bound values)
- Convenient Maven 2.0 setup
    - ♦ Well thought-through use of Maven. Hierarchical split of configurations. Use of fine-grained projects.
    - ♦ Bugfixes for maven and related tools (we have submitted about 20 patches, some of which are already included in maven)
    - ♦ Own plugin to extend maven: copy tool for combined report generation.
    - ♦ Presentation about how to migrate to mvn and many detailed information and hints
    - ♦ Maven cheat sheet
- GUI: Light Swing framework featuring: Binding of POJOs to Swing components, Event Bus, Docking and MDI support, Exception handling, i18n and resource management, user preference management, simple way to define Actions and selectively enable them, convenience code to simplify the design of forms, ...
- Daemon manager
- License manager
- XML Merger
- Extended file support (fast file observation, directory size information, easier file search capabilities)
- Generic DAO implementation (reduce coding, improve homogenization)
- Easier support for annotation to interceptor mappings (no coding required for basic cases)

- Ajax demo
- TCP forwarder to automatically test TCP connection failures
- Tracking the invocation graph (potentially over process boundaries), measuring performance and generating a sequence diagram for it
- Auto-idempotency interceptor (makes your service calls idempotent)
- Better documentation
    - Architecture discussions
    - EL4J Datasheet
    - Annotation cheat sheets
    - FAQ & infos on how to solve common problems
    - Documentation of each feature
    - Tracing stack document: hints on how to get more information from the layers of your application

The following external components are integrated in EL4J (this list is not exhaustive, please check also the list of included jar-files):

- Spring 2.5 framework
- Maven 2.0, JUnit
- Commons logging, log4j
- Hibernate
- Ibatis
- Acegi security framework
- Swing application framework (from Sun)
- JWebUnit and HtmlUnit
- Eclipse BIRT
- CGLib
- XFire
- Axis
- Caucho remoting: Hessian & Burlap
- DWR
- Struts
- JaMon
- Quartz

3 Maven 2 and installing modules

3.1 Maven 2

EL4J uses Maven 2 as its build system. For more information about maven we refer to its website and our introductory Maven presentation. The specific EL4J Maven plugins are documented via the standard plugin documentation support of Maven (available on our website).

3.2 Installing modules

EL4J (the framework) and applications using it are split in modules. One needs to install only the needed modules and dependencies of modules are automatically taken into account. This section introduces how one can download modules. For more details on the module abstraction, please consult the corresponding section in the core module documentation.

For further information on getting started with the el4j modules we refer to the convenience zip (downloadable from el4j.sf.net). Follow the steps there to set up your environment and have a look at the demo applications.

4 Documentation for module core

4.1 Purpose

The core module of EL4J contains support to split applications into separate modules. Each module can contain code, configuration and dependencies on jar files as well as on other modules. Dependencies are transitive. In addition, the core module contains helpers classes for annotations, implicit context passing and others.

4.2 Support for Maven 2 modules on the level of Spring

The *module* support of the core module is provided in combination with the Maven 2 build system. Maven defines the module abstraction and the core module of EL4J makes use of it and supports it on the level of Spring.

Rationale for the module support:

- Modularity: be able to split your work in smaller sub-parts in order to reduce complexity, to simplify separate development, to reduce size of cody by using only what is needed.
- Provide default configuration for modules: with spring, configuration can sometimes become complicated. We provide support for default spring configurations to modules.
- Dependency management (1): each module lists its requirements (other modules and jar files). These dependencies are then automatically managed (downloaded if needed, added to the classpath, added to deployment packages such as WAR, EAR or zip files)
- Dependency management (2): from each module only the resources of the dependent modules are visible (you can e.g. make certain server-side jar files invisible during the compilation of client-side code, in order to statically ensure they are not used)

The module support is based on the following:

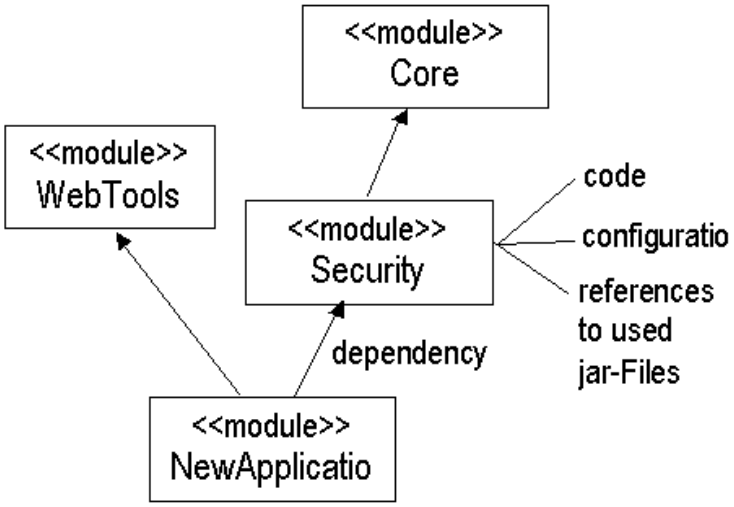
- the module abstraction of Maven 2
- the ModuleApplicationContext (a wrapper for the standard Spring application context)
- a convention on how to organize configuration information within each module

These three parts are described in the next sections.

4.2.1 Module abstraction of Maven 2

Maven 2 (http://maven.apache.org/) is a build system that gives you higher-level build abstractions than Ant. With Maven 2 you can split your application or framework into *modules.* A module can contain code and configuration. Modules can define dependencies on jars and other modules. Dependencies on other modules are transitive (e.g. if A requires B and B requires C, A has implicitly also C available). Maven can package your module into a jar file.

The following picture illustrates 4 modules with dependencies:



For more detail on how to setup modules and for more module features, we refer to the documentation of Maven 2.

4.2.2 ModuleApplicationContext

The ModuleApplicationContext is similar to the existing application contexts of Spring (i.e. `ClasspathXmlApplicationContext`). It is a light wrapper around the existing Spring application contexts.

The use of the ModuleApplicationContext is optional. We recommend it due to its following features:

- it finds all configuration files present in the modules, even if some J2EE-container present them differently (e.g. WLS)
- it solves issues with the order of loading configuration files in some J2EE-containers
- it complements the rest of the configuration support (e.g. via the configuration file exclusion list)
- it allows publishing all its Spring beans with their configuration (publication is possible e.g. to JMX).

The first two features are provided in collaboration with the module support of Maven. (A Maven plugin lists the configuration files contained in each module into the Manifest file of modules. The ModuleApplicationContext then uses this information.)

The reference documentation of the ModuleApplicationContext is located under the web module.

4.2.3 Convention on how to organize configuration

Our convention to organize config files helps to indicate what configuration should be automatically loaded when a module is active. One can also define different configuration scenarios among which one needs to choose one. A sample scenario is the choice of whether we run in a client or a server (e.g. for remoting or security) or what data access technology to use (e.g. ibatis or hibernate). NB: There is an easy way not to load mandatory configuration information.

The configuration files of a module are saved under a folder '/resources'. This '/resources' folder is divided into different subfolders:

- '/mandatory': Here are all the xml and the property files which are always loaded into the ApplicationContext when the module is active.
- '/scenarios': This is the parent folder for different scenarios. It does not contain any file, only subfolders. (e.g. a type of scenario would be 'authentication' and the scenarios of this type would be stateless or stateful). Exactly one scenario of each type must be chosen. All possible combinations of

the scenarios have to work. The testcases of a test module are also placed in the scenarios folder.
    - '/subfolder': For each type of scenarios (see below), there is a subfolder with a context-dependent name. One scenario of each subfolder must be chosen in order to execute the module. Note: Such a subfolder could contain further subfolders.
- '/optional': Here are optional xml and property files which are loaded if requested.
- '/etc': This folder contains varios files that do not suit to another configuration folder, e.g. templates one can provide which can be helpful to efficiently develop applications or to understand the module or images used by the web modules.

By loading all files in '//mandatory' and one scenario of each type into the ApplicationContext, the module has to be executable. This constraint reduces the complexity for developers using this module.

4.2.3.1 Examples

Two examples are provided in order to illustrate the ideas of the above structure.

4.2.3.1.1 Example 1

The first example illustrates how the configuration structuring of the ModuleSecurity (old version) looks like:

- ch.elca.el4j.core.services.security:
    - '/resources/mandatory/':
        - `security-attributes.xml`
    - '/resources/scenarios/':
        - 'authentication/':
            - `stateless-authentication.xml`
            - `stateful-authentication.xml`
        - 'logincontext/':
            - `db-logincontext.xml`
            - `nt-logincontext.xml`
        - 'securityscope/':
            - `local-securityscope.xml`
            - 'distributedsecurityscope/':
                - `client-distributedsecurityscope.xml`
                - `server-distributedsecurityscope.xml`
            - `web-securityscope.xml`
    - '/resources/optional/':
    - '/resources/etc/templates/':

Explanation: In `security-attributes.xml`, the attributes for the authorization interceptor is defined. Since it is always needed, it is put into the '/mandatory/' folder. There are 3 types of scenarios which the developer can choose from. Regarding the authentication there's the choice between a stateless and a stateful authentication. As a next thing it has to be defined which login context is chosen. Last, the security scope has to be defined, i.e. if the environment is set up locally, if it is distributed or if it is web based. In case the environment is distributed, we define a subfolder since there is more than one xml file defining these beans.

Important: in case of a distributed environment, the security module needs a remote protocol which has to be specified. Since in the distributed environment, the security module needs the ModuleRemoting module, the remote protocol is defined in that scope.

4.2.3.1.2 Example 2

A second example illustrates the Remoting And Interface Enrichment module (the current module is slightly different):

- ch.elca.el4j.core.services.remoting:

- '/resources/mandatory/':
- '/resources/scenarios/':
  - 'scope/':
    - `client-config.xml`
    - `server-config.xml`
  - 'protocol/':
    - `rmi-protocol-config.xml`
    - `hessian-protocol-config.xml`
    - `burlap-protocol-config.xml`
- '/resources/optional/':
- '/resources/etc/templates/':
  - `service-exporter-config.xml`
  - `service-importer-config.xml`

Explanation: The developer has to choose exactly one possibility of both of the two scenarios. On the one hand, the scope has to be defined, i.e. if the ApplicationContext is loaded on a client or on a server. Then, the protocol has to be chosen, either rmi, burlap or hessian. Obviously, the remote protocol and its properties has to be the same, on the client and the server. Finally the exporter and the importer are stored under '/resources/etc/templates/' since the content of these xml files highly depends on the specific implementation. Therefore, commented templates are provided.

Remark: it is still possible to load both the client and the server configs in case one would require to have both roles.

4.2.3.1.3 Example 3

This example illustrates the ModuleJmx:

- ch.elca.el4j.services.monitoring.jmx:
  - '/resources/mandatory/':
    - `jmx.xml`
    - `htmlAdapter.xml`
  - '/resources/scenarios/':
  - '/resources/optional/':
    - `jmxConnector.xml`
  - '/resources/etc/templates/':

Explanation: Although the HTML adapter is just one option to access JMX data, it is considered to be the most used. Putting its configuration file in the mandatory folder loads it whenever the module is added as dependency. Users still can use the JMX connector and remove the HTML adapter using the ModuleApplicationContext with its ability to exclude configuration files explicitly.

4.2.3.1.4 Example 4

Configuration of the statistics module (it provides convenience to use the JAMon interceptor):

- ch.elca.el4j.services.performance.jamon:
  - '/resources/mandatory/':
    - `jamon.xml`
    - `jamon-jmx.xml`
  - '/resources/scenarios/':
  - '/resources/optional/':
  - '/resources/etc/templates/':

TBD: is the following still correct as we use no longer the EL4Ant execution units?

Explanation: The module JAMon can be used together with the JMX module or stand-alone. While the former has a dependency on the JMX module and a JMX proxy configured in the `jamon-jmx.xml` file, the latter needs a web application container to display measurements. The dependency and the action to exclude the `jamon-jmx.xml` configuration file are defined in the module's specification in form of two different execution units.

4.2.3.2 Usage of configuration using this convention

This section presents how the security module (as defined above) could be used in an application. Note that Maven adds the `conf` folder of each module automatically to the active classpath:

```
String[] configurationFiles = {"classpath*:mandatory/*.xml",
"scenarios/authentication/stateless-authentication.xml",
"scenarios/logincontext/db-logincontext.xml",
"scenarios/securityscope/local-securityscope.xml"};

ApplicationContext m_ac = new ModuleApplicationContext(configurationFiles, new
String[]{}, false);
```

This code loads the files from the mandatory directory of all modules the current module depends on (as EL4Ant puts these modules automatically in the CLASSPATH, the expression `"classpath*:mandatory/*.xml"` finds all those files). In addition, it selects the appropriate scenarios from the security module. It excludes the `jmx-appender.xml` configuration file from the configuration. In a web context, in `web.xml` this could look like:

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>
        classpath*:mandatory/*.xml,
        classpath*:mandatory/keyword/*.xml,
        classpath*:scenarios/db/raw/*.xml,
        classpath*:scenarios/dataaccess/hibernate/*.xml,
        classpath*:scenarios/dataaccess/hibernate/keyword/*.xml,
        classpath*:optional/interception/transactionJava5Annotations.xml
   </param-value>
</context-param>

...

<listener>
    <listener-class>
        ch.elca.el4j.web.context.ModuleContextLoaderListener
    </listener-class>
</listener>
```

In this case it is the ModuleWebApplicationContext that has the same role as the ModuleApplicationContext.

4.3 Java 5 annotations for Transactions

By declaring annotations (see Java language specification) on methods the application context of Spring is able to detect which method to intercept and what kind of transaction to start or not. There are two annotations specially made for transaction declaration.

- **org.springframework.transaction.annotation.Transactional**

- Javadoc: http://static.springframework.org/spring/docs/2.0.x/api/org/springframework/transaction/annotation/Tra
- Annotation `Transactional` should be used only on methods in implementation classes.

- **ch.elca.el4j.core.transaction.annotations.RollbackConstraint**
  - Contains only some elements from annotation `Transactional`. Used to declare rollback behavior if exception is thrown on method where this annotation is declared. Needs to be declared in combination with a `Transactional` annotation to enable transactional behavior.

Here an example of a declaration on an interface:

```
public interface KeywordDao {
    @RollbackConstraint(rollbackFor = { DataAccessException.class,
            DataIntegrityViolationException.class,
            OptimisticLockingFailureException.class })
    Keyword saveOrUpdate(Keyword keyword) throws DataAccessException,
        DataIntegrityViolationException, OptimisticLockingFailureException;
}
```

And here on an implementation class:

```
public class KeywordDaoImpl implements KeywordDao {
    @Transactional(propagation = Propagation.REQUIRED)
    Keyword saveOrUpdate(Keyword keyword) throws DataAccessException,
        DataIntegrityViolationException, OptimisticLockingFailureException {

        ...

        return xy;
    }
}
```

If the impl class above is now defined as bean in Spring application context you just have to add the predefined Spring config file `classpath*:optional/interception/transactionJava5Annotations.xml` in application context's config locations (view the config file).

If classes `java.lang.RuntimeException` and `java.lang.Error` are not defined in **no-rollback** elements of annotations they will be automatically added to element `rollbackFor`.

Rollback behavior can be defined on annotation `Transactional` too but be aware that only the most specific annotation will be taken. It is not possible to merge multiple `Transactional` annotations. The same matches to annotation `RollbackConstraint`. All rollback contraints defined in annotation `RollbackConstraint` will be automatically added to annotation `Transactional`.

4.3.1 Transaction propagation behaviors

Here an overview of propagation behaviors:

- **required:** execute within a current transaction, create a new transaction if none exists.
- **requires new:** create a new transaction, suspending the current transaction if one exists.
- **supports:** execute within a current transaction, execute nontransactionally if none exists.
- **not supported:** execute nontransactionally, suspending the current transaction if one exists.
- **mandatory:** execute within a current transaction, throw an exception if none exists.
- **never:** execute nontransactionally, throw an exception if a transaction exists.

The default is **required**, which is typically the most appropriate. For more documentation, please refer to the spring or the EJB documentation.

4.3.1.1 Programmatical transaction demarcation (start transaction, commit, rollback in code)

First, do not use this if it is not really necessary. Mostly you can separate your code in methods and use transaction attributes.

You can get the bean `transactionManager` that is defined in file `scenarios/db/rawDatabase.xml` of **module-core** and cast it to `org.springframework.transaction.PlatformTransactionManager`. On this class, you can call methods directly. Please use in addition the attributes, which are defined in **module-core** (attrib.transaction.*).

4.3.1.2 setRollbackOnly is not equals to setReadOnly

In this module there are attributes which name ends with **ReadOnly**. If this kind of attributes are used it is **still possible** to commit changes. This **property** will be only set in the JDBC properties to help intelligently implemented JDBC drivers to optimize connection creation. This means that a JDBC driver can, but must not read this property.

The **setRollbackOnly** method of class **org.springframework.transaction.TransactionStatus** is used to garantee that the current transaction is rolled back. This property of class **TransactionStatus** can be set in code and the current **TransactionStatus** can be retrieved by invoking the static method **org.springframework.transaction.interceptor.TransactionAspectSupport.currentTransactionStatus()**.

4.4 Old support support with attributes (pre JDK 5, now deprecated)

DeprecatedConvenienceAttributesForTransactions

4.5 Annotation/ metadata convenience

Remark: The documentation of this module was not reviewed. The implementation of the feature may be a bit dated. Please refer also the new Spring features for annotation configuration convenience (Chapter 7.9.2. Using metadata-driven auto-proxying of the Spring reference manual).

The annotation convenience support is the successor of the *Attribute Convenience* feature. Annotations are used to describe classes, methods or types; for example in Java annotations can define that a method is deprecated via the *@Deprecated* annotation.

With EL4J you can use annotations to enable AOP aspects.

Benefits of using this module:

- Java Annotations can enable interceptors.
- Spring AOP supports the use of Java Annotations only on methods. EL4J supports also the use of annotations on classes.
- Inheritance of annotation is supported.
- In spring, adding support for a new attribute to spring requires to implement a few new classes. The new classes are typically quite redundant (and they are often implemented via cut and paste). It is the goal of the module to alleviate this.

4.5.1 Linking annotations to interceptors

One configures this via the `GenericMetaDataAdvisor`.

Here's a sample configuration file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

    <!-- Defines the Autoproxy bean which looks for each advisor in this context -->
    <bean id="autoproxy"
        class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>


    <!-- Define the Advisor bean. -->
    <bean id="genericMetaDataAdvisor"
        class="ch.elca.el4j.util.metadata.GenericMetaDataAdvisor">
        <property name="interceptor">
            <ref local="exampleInterceptor"/>
        </property>
        <property name="interceptingMetaData">
            <list>
                <value>ch.elca.el4j.tests.util.metadata.annotations.helper.ExampleAnnotationOne</value>
            </list>
        </property>
    </bean>

    <!-- Define the interceptor to be used by the above defined advisor. -->
    <bean id="exampleInterceptor"
        class="ch.elca.el4j.tests.util.metadata.annotations.helper.ExampleInterceptor">
    </bean>

    <!-- Define the bean which owns a method that should be intercepted. -->
    <bean id="foo"
        class="ch.elca.el4j.tests.util.metadata.attributes.helper.FooImpl"/>

</beans>
```

- The `autoproxy` bean looks for `Advisors`.
- The `genericMetaDataAdvisor` bean extends the `org.springframework.aop.support.DefaultPointcutAdvisor`.
    - The Advice, e.g. a MethodInterceptor can be injected via the property `interceptor`. It is necessary to define one, otherwise, an exception is thrown.
    - The property `interceptingMetaData` takes a list of meta data. The defined interceptor will be invoked if one of these meta data is defined at a method/class. If the parameter is not set, all meta data defined at a method/class are collected.
- The `exampleInterceptor` bean extends `org.aopalliance.intercept.MethodInterceptor`. It also implements `ch.elca.el4j.util.metadata.MetaDataCollectorAware` which sets the `metaDataCollector` of this Interceptor since the Interceptor needs to access the meta data.
- The `foo` bean is a bean having a method `test(int)` where an ExampleAttributeOne is declared. Therefore, a call to `foo.test(int)` will invoke this Interceptor.

DeprecatedAttributeConvenienceSupport

4.5.2 Implementation of an interceptor

To have access to the meta data of the annotation collector, the interceptor (specified in the configuration) has to implement the interface `ch.elca.el4j.util.metadata.MetaDataCollectorAware`. Please refer to its javadoc.

These methods returns a *Collection* of the found meta data or *null* if no meta data was found.

```
/**
 * If a method containing the meta data ExampleMetaData and the parameters
 * are from typ int, check that there value is not higher as defined in ExampleMetaData.
```

```
 * If the argument/s is/are higher, the method will proceeded with the value defined in
 * ExampleMetaData.
 */
public Object invoke(MethodInvocation methodInvocation) throws Throwable {

    int[] param = null;

    // Get meta data from the interceted method
    Collection metaData = m_metaDataCollector.getMethodOperatingMetaData(methodInvocation);

    // Proceed meta data for the method specified (cf. Javadoc)
    if (metaData != null && metaData.size() > 0) {

        // Set the new arguments of the intercepted methods if
        // they are of type int.
        try {
              param = methodInvocation.getArguments();

              for (Iterator iter = collection.iterator(); iter.hasNext();) {
                    Object element = (Object) iter.next();

                if (element instanceof ExampleMetaData) {
                     int value = element.value();
                     for (int i=0; i <param.length; i++) {
                        if (param[i] > value) { param[i] = value; }
                     }
                }

              }
        } catch (Exception e) {
              //Do nothing with the arguments; just proceed the method
        }

    }

    // Proceed intercepted method and return its result
    Object retVal = null;
    try {
       // Execute the intercepted method
       retVal = methodInvocation.proceed();
    } catch (Throwable ex) {
       throw ex;
    }

    return retVal;
}

/**
  * {@inheritDoc}
  */
public void setMetaDataSource(GenericMetaDataCollector metaDataSource) {
    m_metaDataCollector = metaDataSource;
}
```

4.5.3 Semantics of the inheritance

Sometimes it is useful to inherit a meta data to child classes or implementations of interfaces. In other cases, inheritance is not desired because the clearance of the code decreases. Therefore in el4j it is configurable if, and how deep meta data will inherited to the children. The inheritance can be configured in the following steps.

includePackages meta data on packages will be inherited to all classes, interfaces and its methods in the corresponding package and all its subpackages. à Not yet implemented

includeInterfaces = true; meta data on interfaces will be inherited to all classes which implements the

interface. The inheritance goes on to all subclasses of these classes. Example: Class A implements Interface One. Class B extends Class A. So inherit Class B the meta data from Interface One.

includeSuperclasses = false; the superclasses inherit its meta data to all its childs and its methods.

includeClass = true; the class inherits its meta data to its methods.

If nothing will be configured, the following default configuration is used: includePackages = false; includeSuperclasses = false; includeInterfaces = true; includeClass = true;

**Note:* All inheritance will only be made, if the meta data type allows it (e.g. java annnotations can be specific to use only on specific targets, for example only on methods).*

**Hint:* If inheritance is used, document it clearly! Otherwise the clearance of the code can decrease strongly.*

Overwriding

Child meta data overwrites parent meta data.

Example:

A class uses the annotation @ExampleAnnotationOne("Class) and one of its method uses @ExampleAnnotationOne("Method"). In this case a method interceptor got the value Method to proceed.

If all options of the inheritance are used, the following points have to be mentioned:

· Interface meta data are stronger than superclass meta data (cf. ExampleAnnotationTwo? in the example).. · Method meta data are stronger than class, interface and superclass meta data. Also when the class, interface or superclass is in the hierarchie nearer than the meta data definition on the method (cf. ExampleAnnotationTen? in the example).

Example

à Full configuration (everything true)

```
@ExampleAnnotationOne()
public interface Base {

    @ExampleAnnotationTen()
    public void inheritFromMethod(int input);

}

@ExampleAnnotationTwo()
public interface Foo extends Base{

    public void inheritFromClass(int input);

}

@ExampleAnnotationThree()
@ExampleAnnotationTen( Not stronger than ExampleAnnotationTen
on inheritFromMethod(int) in interface Base )
public interface FooBase {

    @ExampleAnnotationTwelve()
    public void overwrideAnnotations(int input);

}

@ExampleAnnotationSix()
```

```
@ExampleAnnotationTwo( Not stronger than ExampleAnnotationTwo
on interface Foo )
public abstract class AbstractFoo implements Foo {

@ExampleAnnotationFourteen()
public abstract void inheritFromMethod(int input);

}


@ExampleAnnotationEight()
public class FooImpl extends AbstractFoo implements FooBase {

@ExampleAnnotationSixteen()
public void inheritFromMethod(int input) { }

public void inheritFromClass(int input) { }


@ExampleAnnotationEight( Overwritten )
@ExampleAnnotationTwelve( Overwritten )
public void overwrideAnnotations(int input) { }


}
```

method public void inheritFromClass(int input) inherit following annotations: - @ExampleAnnotationEight() - @ExampleAnnotationThree() - @ExampleAnnotationTen(Not stronger than ExampleAnnotationTen? on inheritFromMethod(int) in interface Base) - @ExampleAnnotationTwo() Same annotation type on Superclass AbstractFoo? is not inherited because the definition on the interface Foo is stronger. The definition is stronger, also if superclass AbstractFoo? is nearer in the hierarchie. - @ExampleAnnotationOne() - @ExampleAnnotationSix()

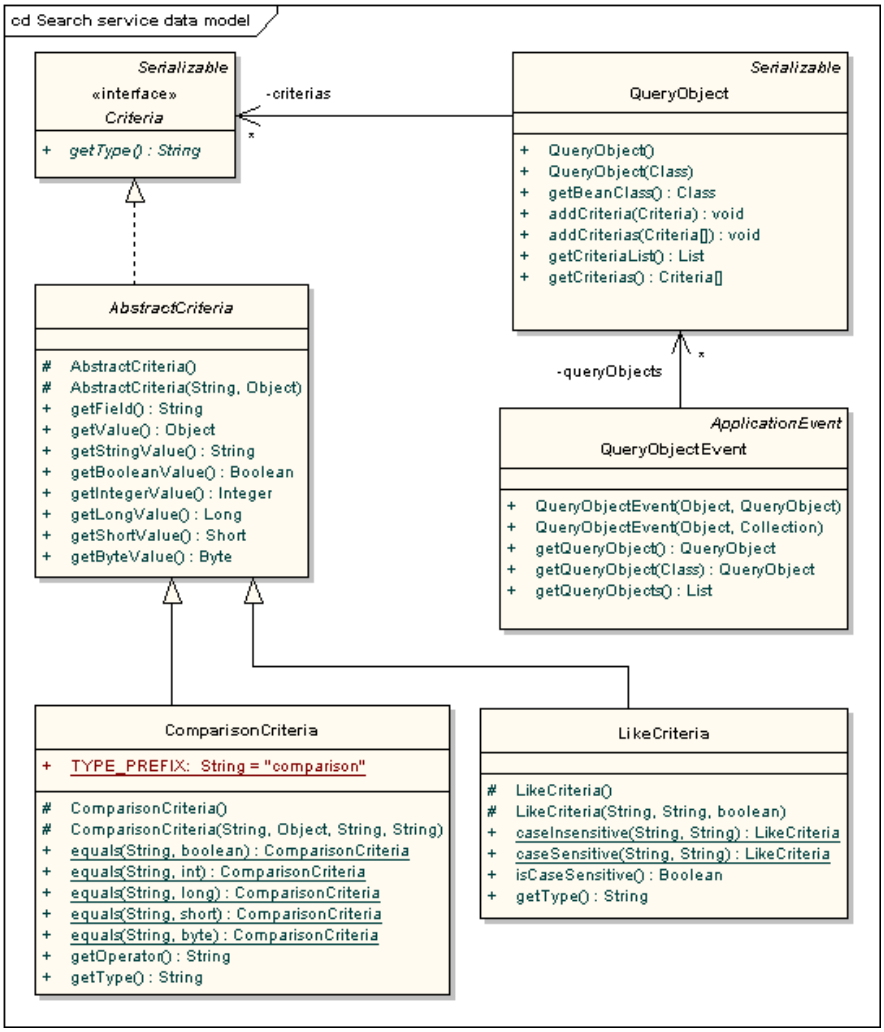
method public void inheritFromMethod(int input) inherit following annotations: - @ExampleAnnotationSixteen() - @ExampleAnnotationTen() Same annotation type on Interface FooBase? is not inherited because the definition on the method in interface Base is stronger. The definition is stronger, also if interface FooBase? is nearer in the hierarchie. - @ExampleAnnotationEight() - @ExampleAnnotationThree() - @ExampleAnnotationTen(Not stronger than ExampleAnnotationTen? on inheritFromMethod(int) in interface Base) - @ExampleAnnotationTwo() Same annotation type on Interface Superclass AbstractFoo? is not inherited because the definition on the interface Foo is stronger. The definition is stronger, also if superclass AbstractFoo? is nearer in the hierarchie. - @ExampleAnnotationOne() - @ExampleAnnotationSix()

method public void overwrideAnnotations(int input) inherit the same as inheritFromClass(int input) except: - @ExampleAnnotationEight(Overwritten) Annotation on Class is overwritten by the method. - @ExampleAnnotationTwelve(Overwritten) Annotation defined by the interface FooBase? is overwritten by the method in class FooImpl?.

4.6 Search service

In the search service we have implemented the **Query Object** pattern of Martin Fowler. See http://www.martinfowler.com/eaaCatalog/queryObject.html for a short introduction.

The idea is to create a query object in the presentation layer (potentially on the client-side) and send this query object trough to the DAO layer. There should be no need to change the query object in between these layers. With this approach you can add search conditions on client-side without modifying service interfaces or depending on underlying data access technology.



In the center we have the query object class. A query object normally belongs to to one java bean, where the java bean is a dto like the reference dto of Reference-Database-Application (see here). In this dto we have nearby other properties property `name`, `description` and `incomplete`. Properties `name` and `description` are strings and property `incomplete` is a boolean.

A query object can have multiple criterias. Currenly we have three criteria classes. The like criteria is made to do searches on strings with the `SQL like` syntax. The second criteria is the comparison criteria, used to compare values. Currently only `equals` compares are implemented. The third criteria is the include criteria, which is used to test if a given value is included in a given set.

```
ReferenceService service = ...
QueryObject query = new QueryObject(ReferenceDto.class);
query.addCriteria(LikeCriteria.caseInsensitive("name", "%JAVA%"));
query.addCriteria(LikeCriteria.caseInsensitive("description", "%WEB%"));
query.addCriteria(ComparisonCriteria.equals("incomplete", true));
query.addCriteria(new IncludeCriteria("keywords", kJava.getKeyAsObject()));
List list = service.searchReferences(query);
...
```

The code above shows the use of these three criteria objects combined with the reference dto. In this code we execute a search on reference dto's fields `name`, `description`, `incomplete` and `keywords`. The expected result is to receive all reference dtos with string `java` (case-insensitive) somewhere in property `name`, with string `web` (case-insensitive) somewhere in property `description`, where property `incomplete` is set to `true` and where the `kjava` keyword is included in the reference dto's `keywords` set. To get all references we could send an empty query object (without any criterias) to the reference service.

To see how the query object can be handled with Hibernate, you can e.g. have a look at the dao class of the Reference-Database-Application and the automatic CriteriaTransformer class of the Hibernate module.

One can also implement this pattern on top of ibatis. However, its a much more manual task.

How the query object could be handled with IBatis you can have a look at dao classes and IBatis config files of the Reference-Database-Application.

4.6.1 Query Object Events

The query object event is used to wrap query objects. This event can be used with Spring's application event publisher. Most application contexts are such an application event publisher. Each *singleton* Sring bean that implements the interface `org.springframework.context.ApplicationListener` will receive these events. Prototype beans must be handeled separately.

For an example you can have a look at the (now deprecated) handling of views (prototype beans) in **module-springrcp**.

4.7 Additional Features

4.7.1 Configuration merging via property files

The class `ch.elca.el4j.core.config.ListPropertyMergeConfigurer` can be used to add items to a list on an existing configuration. Here an example.

xml-config-file.xml:

```
<beans>
    <bean id="configurationTest"
        class="ch.elca.el4j.core.config.ListPropertyMergeConfigurer">
        <property name="location">
            <value>myconfig/mergeable-config-file.properties</value>
        </property>
    </bean>

    <bean id="listTest" class="ch.elca.el4j.tests.core.config.ListClass">
        <property name="abcList">
            <list>
                <value>item 0</value>
            </list>
        </property>
    </bean>
</beans>
```

mergeable-config-file.properties:

```
listTest.abcList=item 2, item 3
```

If the `xml-config-file.xml` is loaded in an application context the property `abcList` of bean `listTest` contains items 0, 2 and 3.

For more information have first a look at the javadoc of the spring class `org.springframework.beans.factory.config.PropertyOverrideConfigurer` and then have a look at
http://el4j.sourceforge.net/framework-modules/apidocs/ch/elca/el4j/core/config/ListPropertyMergeConfigurer.html

Further the list property merge configurer has the possability to add the new values before or after the existing values. By default the new values (from property file) will be appended. To prepend the new values you have to set following property in configurer bean:

```
<property name="insertNewItemsBefore" value="true"/>
```

4.7.2 Bean locator

The class `ch.elca.el4j.core.beans.BeanLocator` can be used to get all beans in an application context, which are an instance of specific type (interface or class) or have a specific bean name. It is also possible to exclude beans. For more information have a look at http://el4j.sourceforge.net/framework-modules/apidocs/ch/elca/el4j/core/beans/BeanLocator.html

4.7.3 Bean type auto proxy creator

The class `ch.elca.el4j.core.aop.BeanTypeAutoProxyCreator` allows autoproxying beans by their type. It helps e.g. to use marker interfaces (such as ServiceInterface[?]/ DAO) that are then used more consistently than can bean naming conventions.

Using a pointcut with a class filter would solve the problem too. It requires writing a new static advisor that configures a `RootClassFilter` and that accepts a list of interceptors. Finally, a `DefaultAdvisorAutoProxyCreator` is required to proxy all classes. Using the `BeanTypeAutoProxyCreator` is much easier.

4.7.4 Exclusive bean name auto proxy creator

This auto proxy creator extends Spring's `BeanNameAutoProxyCreator`. It allows setting a list of name patterns of beans to exclude. The pattern can reference a distinct bean, a prefix or a bean name's suffix. If you don't declare an include pattern (i.e. using the `beanNames` property), all beans will be proxied, except the ones matching the exclude patterns. **Note** Exclusion patterns have higher priority.

Configuration Example

```
<bean id="exclusiveNameAutoProxy"
  class="ch.elca.el4j.core.aop.ExclusiveBeanNameAutoProxyCreator">
  <property name="exclusiveBeanNames"><value>foo*</value></property>
  <property name="interceptorNames">
    <list>
       <value>shortcutInterceptor</value>
    </list>
  </property>
</bean>
```

4.7.5 Abstract parent classes for the typesafe Enumerations Pattern (consider using the new JDK 5 enums)

An Enummeration is a type that can hold one value from a set of well defined values. We provide 2 super classes for the immutable and typesafe enumeration pattern: one `java.lang.Comparable` and the other one not comparable. For an example, please have a look at the javadoc:

- http://el4j.sourceforge.net/framework-modules/apidocs/ch/elca/el4j/util/codingsupport/AbstractDefaultEnum.htm
- http://el4j.sourceforge.net/framework-modules/apidocs/ch/elca/el4j/util/codingsupport/AbstractComparableEnu

4.7.6 Reject (Precondition checking)

As described in the ExceptionHandlingGuidelines, we use the class `ch.elca.el4j.util.codingsupport.Reject` for precondition checking of a method. Have a look at the javadoc for an example: http://el4j.sourceforge.net/framework-modules/apidocs/ch/elca/el4j/util/codingsupport/Reject.html

4.7.7 JNDI Property Configurers

The JNDI property configurers get their values form a JNDI context. Default is `java:comp/env`. This can be overridden by setting the appropriate value in a `JndiConfigurationHelper`, which is injected into a JNDI property configurer.

For a `JndiPropertyPlaceholderConfigurer`, the values are queried one after another. There's no magic there. However, a `JndiPropertyOverrideConfigurer` needs to get the whole list of properties to override. The default strategy is to use a prefix. Default is `springConfig.` (notice the separating point at the end). Another possibility is to put override properties into a distinct context that allows you neglecting the prefixes (however you need to inject a configured `JndiConfigurationHelper` and you have to set the prefix to `null`).

4.7.8 Generic repository

The `ch.elca.el4j.services.persistence.generic.dao.GenericRepository` interface serves as generic access to storage repositories. It is the interface for the DDD-Book's Repository pattern. The repository pattern is similar to the DAO pattern, but a bit more generic. This interface can be implemented in a generic way and can be extended in case a user needs more specific methods. It is based on an idea from the Hibernate website. A more detailed description, illustrating how this interface can be used, can be found here.

4.7.9 DTO helpers

This package supports optimistic locking on the DTO level. Available is an abstract DTO that holds a primary key generator to realize the optimistic locking and an extended version of the abstract DTO that contains in addition the primary key named as `key` in form of a string. To have the primary key generator set for every DTO it is necessary to create DTOs by using the DTO factory, which can be found in this package too.

4.7.10 Primary key

This package contains an interface, which defines an PrimaryKeyGenerator with a method to generate a primary key as a string. Implemented is a UuidPrimaryKeyGenerator that always returns string primary keys with 32 characters [0-9a-z].

4.7.11 SQL exception translation

This package contains exceptions (subclasses of Spring's DataAccessExceptions). These exceptions complement the exception hierarchy of spring for duplicated values and too big values. When to throw which exception and for which database these contigurations are vaild can be found in this module's conf folder in file `sql-error-codes.xml`.

4.8 Packages that implement the core module

- ch.elca.el4j.core.**
- ch.elca.el4j.services.persistence.generic.**
- ch.elca.el4j.services.monitoring.notification.CoreNotificationHelper
- ch.elca.el4j.services.search.**
- ch.elca.el4j.util.**
- attrib.**

Notes:

- `**` means all files from the current package and all sub packages.
- The full package structure of EL4J can be viewed here.

5 Documentation for module remoting

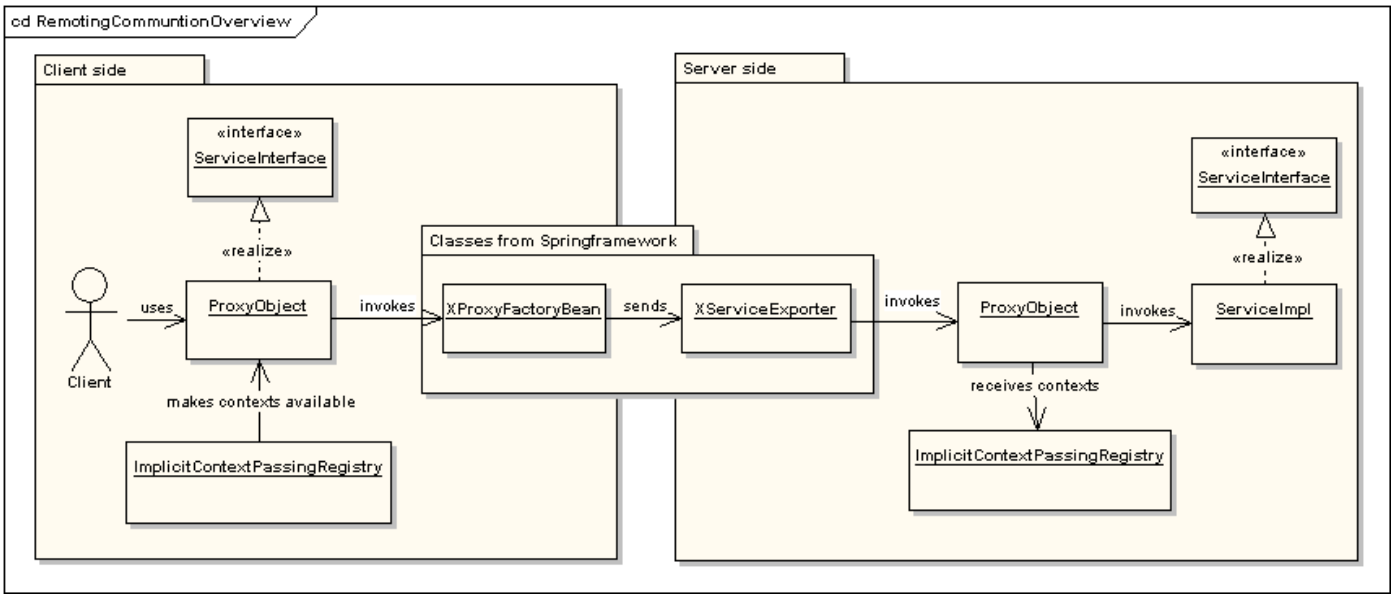
5.1 Purpose

Convenience module for spring POJO remoting: (1) allows **centralized protocol configuration**, (2) simplifies protocol switching (currently between **RMI**, **HttpInvoker**, **Hessian**, **Burlap**, **Soap** and **EJB**), and (3) transparently enriches interfaces for **automatic implicit context passing**.

5.2 Introduction

The Spring framework offers an easy way to distribute POJOs. Available protocols are `Rmi`, `Hessian` and `Burlap`. This module provides in addition implicit context passing. In addition, attention was payed to be able to distribute hundreds of services with a minimum of configuration.

The general idea is to internally use Spring's implementations and offer a proxy object to the outside. This is made on the client and on the server side (see picture).



This module can also be used if you only develop the server or client side!

5.3 How to use

5.3.1 Remoting modules

Currently there are six modules for remoting:

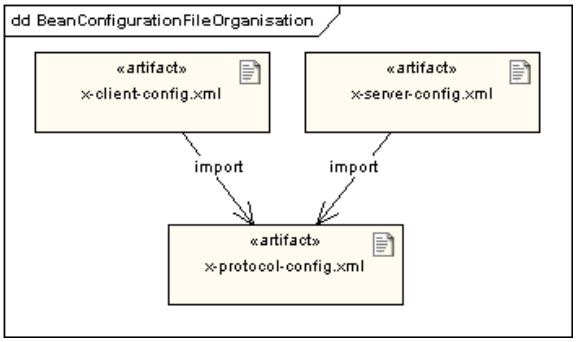
- The core remoting module with name **module-remoting_core** contains only protocol `RMI`.
- For `Hessian` and `Burlap` you have to use **module-remoting_caucho**.
- Our web service stack based on `XFire` can be found in **module-remoting_xfire**.
- Our web service stack based on `JAX-WS` can be found in **module-remoting_jaxws**.
- The old web service stack (based on Axis 1) can be found in **module-remoting_soap** (deprecated).
- For the `EJB` remoting protocol you have to use **module-remoting_ejb** (currently not working).

In addition, there exists a composite protocol which supports load balancing:

- The `load balancing` protocol can be found in **module-remoting_core**.

5.3.2 Configuration

5.3.2.1 Recommended configuration file organisation



Typically we have three configuration files. One for the server, one for the client and one which is shared between server and client. We present first the file that is shared between the server and the client, the `x-protocol-config.xml`. The `x` stands for the protocol such as `rmi`, `hessian` or `burlap`.

5.3.2.1.1 x-protocol-config.xml

This file contains the following for the protocol `rmi`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="remoteProtocol" class="ch.elca.el4j.services.remoting.protocol.Rmi">
        <property name="serviceHost">
            <value>localhost</value>
        </property>
        <property name="servicePort">
            <value>1099</value>
        </property>
        <property name="implicitContextPassingRegistry">
            <ref local="implicitContextPassingRegistry" />
        </property>
    </bean>
    <bean id="implicitContextPassingRegistry" class="ch.elca.el4j.tests.remoting.service.TestImplicitCont
</beans>
```

In this configuration file, we have only two beans defined. One bean for the remoting protocol and one for implicit context passing registry. Each bean that defines a remote protocol needs protocol-specific properties. In addition a reference to a class, which implements the interface `ImplicitContextPassingRegistry` is necessary, if you want to use the implicit context passing feature.

It is possible to have many beans that define a remoting protocol. In the example above it is the `rmi` remoting protocol. This requires the `serviceHost`, where the service is running and it also needs to know the `servicePort`. For the remoting protocol `rmi`, these two properties are mandatory. The other two predefined protocols (`hessian` and `burlap`) need additionally the property `contextPath` that defines in which webserver context the service is running.

5.3.2.1.2 x-client-config.xml

This file contains the following for the protocol `rmi` when we want to get access to the remote calculator bean.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <import resource="rmi-protocol-config.xml"/>

    <bean id="calculator" class="ch.elca.el4j.services.remoting.RemotingProxyFactoryBean">
        <property name="remoteProtocol">
            <ref bean="remoteProtocol" />
        </property>
        <property name="serviceInterface">
            <value>ch.elca.el4j.tests.remoting.service.Calculator</value>
        </property>
    </bean>
</beans>
```

The first element imports the previous discussed **x-protocol-config.xml** file. In this way, we can set the property `remoteProtocol` to a bean that is defined in the file **x-protocol-config.xml**. The second property `serviceInterface` has to be the business interface. These two properties are mandatory.

5.3.2.1.3 x-server-config.xml

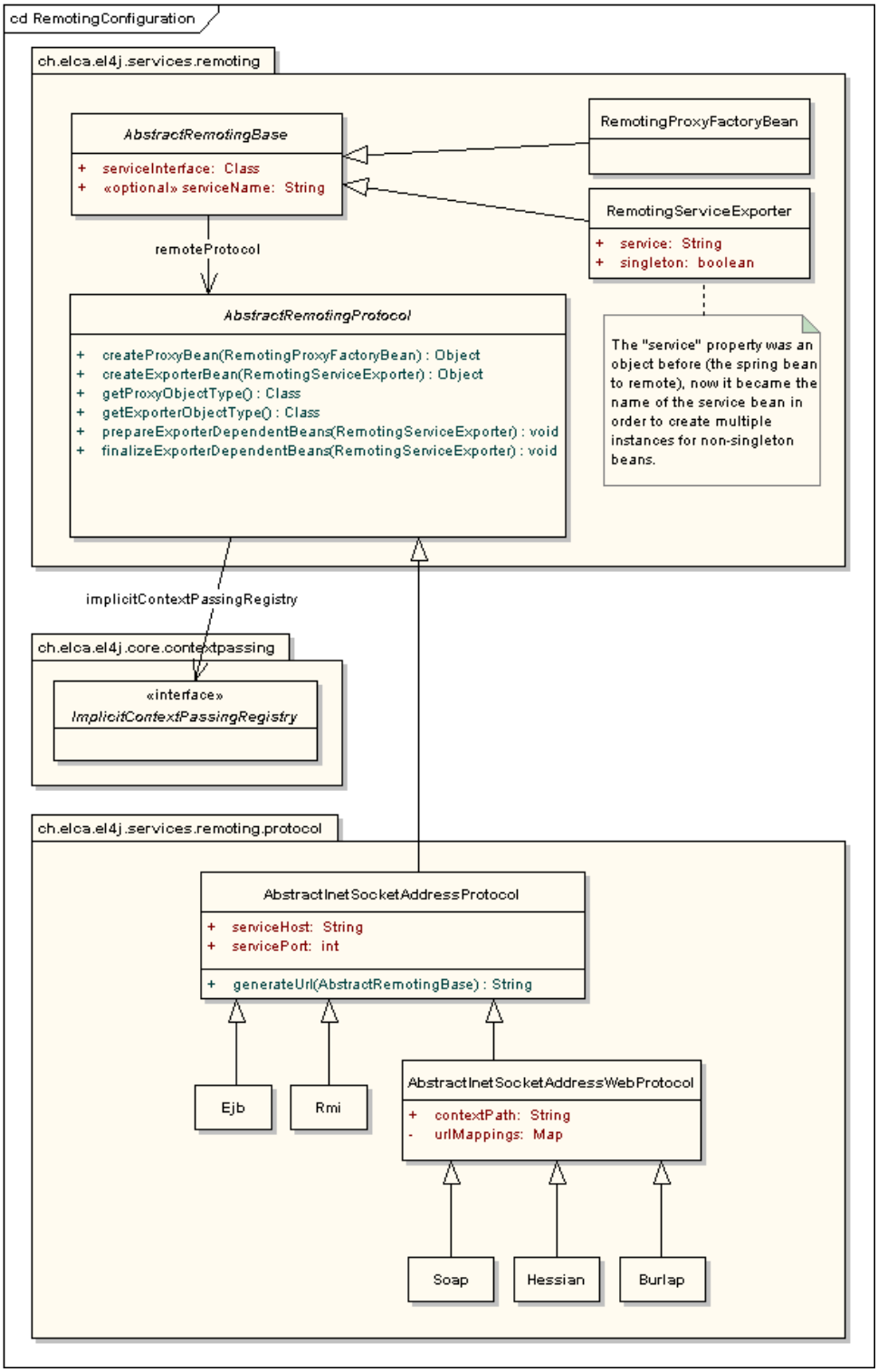
This file contains the following for the protocol `rmi`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <import resource="rmi-protocol-config.xml"/>

    <bean id="calculatorExporter" class="ch.elca.el4j.services.remoting.RemotingServiceExporter">
        <property name="remoteProtocol">
            <ref bean="remoteProtocol" />
        </property>
        <property name="serviceInterface">
            <value>ch.elca.el4j.tests.remoting.service.Calculator</value>
        </property>
        <property name="service">
            <idref bean="calculatorImpl" />
        </property>
    </bean>
    <bean id="calculatorImpl" class="ch.elca.el4j.tests.remoting.service.impl.CalculatorImpl" />
</beans>
```

The first element imports also the **x-protocol-config.xml** file, like the client config does. The second property is also the `serviceInterface`. The difference to the client configuration is that the server configuration needs a reference to the service implementation. The bean for this implementation can be found as second bean definition in this configuration file. These three properties are mandatory.

5.3.2.2 Configuration summary



This picture describs the configuration information needed. On top you can find the base class **`AbstractRemotingBase`** that shares the common part between client (**`RemotingProxyFactoryBean`**) and server side (**`RemotingServiceExporter`**). This base class always needs to know the service interface and it also needs a reference to a class that extends `AbstractRemotingProtocol` such as `Rmi`, `Hessian` or `Burlap`.

While the class **`RemotingProxyFactoryBean`** does not need something more, the class **`RemotingServiceExporter`** needs additionally to the properties from the extended class a reference to the implemented `service`. The `service` must naturally implement the `serviceInterface`.

The property `serviceName` of the base class is optional. It only must be set manually, if the given `serviceInterface` is used twice or more on the same server. If the property `serviceName` is not set, what is normally the case, it will be generated out of the name of the `serviceInterface` and the suffix `.remoteservice`. The suffix `.remoteservice` is needed in webservers to be able to know which requests have to be redirected to the `DispatcherServlet` from Spring. More details follows below.

The class **`AbstractRemotingProtocol`** can have a reference to a class that implements the interface **`ImplicitContextPassingRegistry`**. If such a reference exist, the implicit context passing will be enabled.

The abstract class **`AbstractInetSocketAddressProtocol`** has two required properties. The first is the `serviceHost` which must be the host and the second is the `servicePort` which is the port, where the service is running. **`Rmi`** directly extends this class.

Protocols that are running in a webserver must additionally know in which `contextPath` they are running. This is solved by the abstract class **`AbstractInetSocketAddressWebProtocol`**. This property `contextPath` is mandatory. Inside the webserver, the mapping of services is done automatically by this abstract class. There are two classes that directly extend this abstract class, the **`Hessian`** and the **`Burlap`** protocol.

5.3.2.3 How to use the `Rmi` protocol

The introduction the remoting module in the previous section of the document was made with RMI. So please refer there for general information about remoting with RMI. For additional constraints and implementation details about the RMI remoting, please refer to the last subchapter of this section.

Important points:

- If on host `serviceHost` no rmi registry is running on port `servicePort`, Spring will automatically start a rmi registry.
- The server side must naturally be started before the client side.

5.3.2.4 How to use the `Hessian` protocol

The usage of the `Hessian` protocol on the client side is the same as the `Rmi` protocol.

The server side must be started in a webserver. To realize this, take the following steps.

5.3.2.4.1 Create a web-deployable module

With Maven you can create a module that can be deployed on a webserver such as tomcat. First you have to have the plugin for tomcat installed. This could look like the following snipet:

TBD: adapt the following to Maven

```
<plugin name="j2ee-web-tomcat">
    <attribute name="j2ee-web.container" value="tomcat"/>
    <attribute name="j2ee-web.mode" value="directory"/>
    <attribute name="j2ee-web.home" value="../../external-tools/tomcat"/>
    <attribute name="j2ee-web.port" value="8080"/>
    <attribute name="j2ee-web.manager.username" value="admin"/>
    <attribute name="j2ee-web.manager.password" value="password"/>

    <attribute name="j2ee-war.unpacked" value="true"/>
</plugin>
```

It is highly recommended to define the attribute `j2ee-web.home` relativly to your EL4J project to have the file in your CVS/SVN operating system independent.

After you have added this plugin you can define your `module` with the following lines:

```
    <module name="mymodulename" path="here/is/my/module">
        ...
        <attribute name="runtime.runnable" value="true"/>
        <attribute name="j2ee.war.application"/>
        <attribute name="runtime.command.creator" value=" runtime.command.creator.web"/>
        ...
    </module>
```

Of course you have to add at least a dependency to the `module-remoting_caucho` in this `module`.

Now you can deploy the module and start tomcat via the corresponding ant task, generated by Maven.

5.3.2.4.2 Register Spring's `DispatcherServlet`

To register a servlet you have to create a folder ***webapp*** in your newly created module and in this folder a folder with name ***WEB-INF***. Now you have to create a file with name ***web.xml*** and the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

    <servlet>
        <servlet-name>remote</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>remote</servlet-name>
        <url-pattern>*.remoteservice</url-pattern>
    </servlet-mapping>
</web-app>
```

If you already have a **web.xml** file, just add the two elements `servlet` and `servlet-mapping`. If you have got already a servlet with name `remote` you have to change this name in your newly added two elements `servlet` and `servlet-mapping`.

Declarations:

- The element `load-on-startup` tells the webserver in which order he has to load the servlets. The servlet with the lowest number will be loaded as first and so on. In our example we have only one servlet, so it does not matter which number it has.
- The element `url-pattern` tells the webserver that every request, whose request path ends with `.remoteservice`, should be sent to the servlet with name `remote`.

5.3.2.4.3 Loading Spring configuration file(s)

Internally, the `DispatcherServlet` is looking for the xml file that is in the `WEB-INF` folder and whose name begins with the name of the servlet and ends with `-servlet.xml`. If you have not changed the name of the servlet, the `DispatcherServlet` will look for the file `remote-servlet.xml`. We create now such a file. The content could look like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <import resource="hessian-protocol-config.xml"/>

    <bean id="calculatorExporter" class="ch.elca.el4j.services.remoting.RemotingServiceExporter">
        <property name="remoteProtocol">
            <ref bean="remoteProtocol" />
        </property>
        <property name="serviceInterface">
            <value>ch.elca.el4j.tests.remoting.service.Calculator</value>
        </property>
        <property name="service">
            <ref local="calculatorImpl" />
        </property>
    </bean>
    <bean id="calculatorImpl" class="ch.elca.el4j.tests.remoting.service.impl.CalculatorImpl" />
</beans>
```

The content of this file is exactly the same as for the `Rmi` protocol except that the `import` points to another file. This similarity is by choice, it makes it trivial to switch between different protocols.

Now we have to copy the file **hessian-protocol-config.xml** that is already configured by the client into the folder **WEB-INF**. The content of file `hessian-protocol-config.xml` could look like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="remoteProtocol" class="ch.elca.el4j.services.remoting.protocol.Hessian">
        <property name="serviceHost">
            <value>yourserver</value>
        </property>
        <property name="servicePort">
            <value>8080</value>
        </property>
        <property name="contextPath">
            <value>yourcontextpath</value>
        </property>
        <property name="implicitContextPassingRegistry">
            <ref local="implicitContextPassingRegistry" />
        </property>
    </bean>
    <bean id="implicitContextPassingRegistry" class="ch.elca.el4j.core.contextpassing.DefaultImplicitCont
</beans>
```

Declarations:

- The name of the bean that defines the remoting protocol does not have to be `remoteProtocol`. But when the name of it will be changed, all xml files that import this xml file have to be adapted.

5.3.2.4.4 Needed module classes and libraries

All needed module classes and libraries will be deployed if you execute the deploy ant target of the created module. If you execute this target a second time, the module will be redeployed.

5.3.2.4.5 Reloading context

Normally the reloading of the context will be automatically done, if you are executing the ant target of the module. But sometimes it can be helpful (e.g. if you want to test something) to reload the context manually. If you are using Tomcat, you can reload your context by using the **Tomcat Manager** (http://serviceHost:servicePort/manager/html). You have to login with your account you had created during the installation of Tomcat. By default this is `admin` for the username and `password` for the password. Now you can click on the corresponding link of your context to reload it.

5.3.2.4.6 Test your service and find logging information

Now we are ready to test the service. Open a web browser and enter the address, where the service should be.

Example:

Property	Value
serviceHost	myserver
servicePort	8080
contextPath	remotetest
serviceInterface	ch.elca.el4j.tests.remoting.service.Calculator

For the values above the address would be the following:
http://myserver:8080/remotetest/ch.elca.el4j.tests.remoting.service.Calculator.remoteservice

The result of this GET request should not be a `The requested resource is not available` (HTTP status 404). You should receive an `Internal error` (HTTP status 500). If you can see a stack trace, you should see that there is a message like `HessianServiceExporter only supports POST requests`. If you receive something like that, your service might be running correctly.

Whether it runs correctly or not you can have a look at the console output of your webserver. If you are using Tomcat normally you will find the `stdout.log` in folder `logs` of your Tomcat installation. The file `stdout.log` will be deleted on each restart of Tomcat.

5.3.2.5 How to use the `Burlap` protocol

The usage of the `Burlap` protocol is exactly the same as for the `Hessian` protocol. Just read the last subchapter and replace the word `Hessian` with `Burlap`.

5.3.2.6 How to use the `HttpInvoker` protocol

The usage of the `HttpInvoker` protocol is exactly the same as for the `Hessian` and `Burlap` protocols. Just read the `Hessian` subchapter and replace the word `Hessian` with `HttpInvoker`. One additional change is required: Replace the full qualified path name of the HttpInvokerServiceExporter to "org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter".

5.3.2.7 How to use the `Soap` protocol (XFire)

The usage of the `XFire` protocol is similar to the `Hessian` protocol. The `XFire` protocol requires two additional properties to be set. These are the properties "xfire" and "serviceFactory". The property "xfire" is always assigned a reference to the bean "xfire". By setting the "serviceFactory" property one can choose, which service factory should be used for creating the service.

5.3.2.7.1 Configurations

When using the XFire protocol to publish a webservice there are mainly tree possible combinations. XFire can be used together with Aegis or Jaxb bindings and for Jaxb, the POJO to publish can be annotated using JSR-181 annotations or not.

Component	XFire Default	XFire Jaxb w/o Webservice Annotations	XFire Jaxb with Webservice Annotations
Protocol	XFire	XFire or XFireSoapHeaderContextPassing	
ServiceFactory	ObjectServiceFactory	JaxbServiceFactoryWithoutAnnotations	JaxbServiceFacto

If you are using a Jaxb ServiceFactory and special types (everything that is not built-in) are passed through the interface (as parameter or return value) a JaxbContext has to be passed on the ServiceFactory constructor. This JaxbContext should be initialized with all custom types Jaxb has to serialize. These types should also be annotated using JSR-173 annotations particularly at least a @XMLRootElement annotation should be present.

The XFire protocol also has an optional property wsdlDocumentUrl, which allows to set the URL of the WSDL document. The XFireSoapHeaderContextPassing has the same properties as the XFire protocol and additionally gives the possibility to pass the implicit context within the header of the SOAP messages. To do so, one has to set the protocolSpecificContextPassing property to true and provide another JaxbContext which is initialized with all custom types that are passed within the context, if there are any. This context is then given to the protocol using the property contextPassingContext.

5.3.2.7.2 Example

An common configuration of the protocol could look as follows:

```
<beans>
    <import resource="classpath:org/codehaus/xfire/spring/xfire.xml"/>
    <bean id="xFireProtocol" class="ch.elca.el4j.services.remoting.protocol.XFire">
        <property name="serviceHost">
            <value>${j2ee-web.host}</value>
        </property>
        <property name="servicePort">
            <value>${j2ee-web.port}</value>
        </property>
        <property name="contextPath">
            <value>module-remoting-demos-web</value>
        </property>
        <property name="implicitContextPassingRegistry">
            <ref local="implicitContextPassingRegistry" />
        </property>
      <property name="xfire">
            <ref bean="xfire"/>
        </property>
        <property name="serviceFactory">
           <ref bean="xfire.serviceFactory"/>
        </property>
    </bean>
</beans>
```

On the server side, a web.xml and xfire-servlet.xml file should be setup. Example web.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd http:
version="2.4">
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
         classpath:org/codehaus/xfire/spring/xfire.xml
      </param-value>
   </context-param>
   <listener>
      <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
   </listener>
   <servlet>
      <servlet-name>xfire</servlet-name>
```

```
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>xfire</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Example xfire-servlet.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <import resource="classpath*:mandatory/*.xml"/>
   <import resource="classpath:common/protocols-config.xml"/>
   <import resource="classpath:server/web/benchmark-xfire-server-config.xml"/>
</beans>
```

Example of a setup using context passing in the SOAP header. For the full example and some additional ones please see the tests distributed with the EL4J framework found at `framework\tests\remoting\web\jar`.

```
<!-- XFire Jaxb without annotations -->
<bean id="xFireProtocolJaxb" class="ch.elca.el4j.services.remoting.protocol.XFireSoapHeaderContextPassing
  <property name="serviceHost">
    <value>${jee-web.host}</value>
  </property>
  <property name="servicePort">
    <value>${jee-web.port}</value>
  </property>
  <property name="contextPath">
    <value>module-remoting-tests-web</value>
  </property>
  <property name="implicitContextPassingRegistry">
    <ref local="xfireImplicitContextPassingRegistryJaxb" />
  </property>
  <property name="xfire">
    <ref bean="xfire" />
  </property>
  <property name="serviceFactory">
    <ref bean="xfireJaxbServiceFactoryWithoutAnnotations" />
  </property>
  <property name="protocolSpecificContextPassing">
    <value>true</value>
  </property>
  <property name="contextPassingContext">
    <ref bean="xfireContextPassingContext" />
  </property>
  <property name="serviceProperties">
    <map>
      <entry key="xfire.stax.input.factory">
        <value type="java.lang.String">com.ctc.wstx.stax.WstxInputFactory</value>
      </entry>
      <entry key="xfire.stax.output.factory">
        <value type="java.lang.String">com.ctc.wstx.stax.WstxOutputFactory</value>
      </entry>
    </map>
  </property>
</bean>

<bean id="xfireImplicitContextPassingRegistryJaxb" class="ch.elca.el4j.tests.remoting.service.TestImplici

<bean id="xfireJaxbServiceFactoryWithoutAnnotations"
  class="ch.elca.el4j.services.remoting.protocol.xfire.JaxbServiceFactoryWithoutWebAnnotations"
  singleton="true">
  <constructor-arg index="0">
    <ref bean="xfire.transportManager" />
  </constructor-arg>
```

```
  <constructor-arg index="1">
    <ref bean="xfireJaxbContextPath" />
  </constructor-arg>
</bean>

<bean id="xfireJaxbContextPath" class="javax.xml.bind.JAXBContext"
  factory-method="newInstance">
  <constructor-arg index="0">
    <list>
      <value>
        ch.elca.el4j.tests.remoting.service.CalculatorValueObject
      </value>
    </list>
  </constructor-arg>
</bean>

<!-- JAXBContext used by the XFireJaxb protocol to marshall the implicit context -->
<bean id="xfireContextPassingContext"
  class="javax.xml.bind.JAXBContext" factory-method="newInstance">
  <constructor-arg index="0">
    <list>
      <value>
        ch.elca.el4j.tests.remoting.service.TestXFireContextPassingValue
      </value>
    </list>
  </constructor-arg>
</bean>
```

Remarks:

- The current xfire-remoting implementation cannot transport exceptions from the server side to the client properly (they are converted into XFireFaults).
- When using XFire in combination with Weblogic 9.x one has to make sure that Woodstox is used as Stax implementation as others are known to have errors, especially the weblogic Stax implementation. To define an implmementation a XFire property (`xfire.stax.input.factory` for the input factory) can be set.

Additional links:

- An unoffical JAXB Guide This has also some hints on how to map Java interfaces to XML.
- JSR 173 Specification
- JSR 181 Specification

5.3.2.8 How to use the web service protocol based on JAX-WS 2.0

5.3.2.8.1 JAX-WS

JAX-WS (Java API for XML Web Services) is the successor of the JAX-RPC API. Its reference implementation is part of the Glassfish project and can be found at https://jax-ws.dev.java.net/ .

5.3.2.8.2 How to read this manual

While reading the following sections it's a good practice to have a look at the unit tests to see how it's done on a concrete example. The tests can be found at `external/framework/tests/remoting_jaxws`. The web service is located in the class `ch.elca.el4j.tests.remoting.service.impl.CalculatorImplJaxws` in the `jar` folder, the server configuration (`web.xml`) is inside the `war` directory and the client usage is shown in the test classes inside the `functional-tests` directory. But first of all, let's begin with the server setup.

5.3.2.8.3 Server side setup

5.3.2.8.3.1 web.xml

As in the other SOAP protocols we have to use a `web.xml` file. This could look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
    <context-param>
        <param-name>inclusiveLocations</param-name>
        <param-value>
            classpath*:mandatory/*.xml,
            classpath*:scenarios/server/web/jaxws-server-config.xml
        </param-value>
    </context-param>
        <context-param>
        <param-name>overrideBeanDefinitions</param-name>
        <param-value>false</param-value>
    </context-param>
        <context-param>
        <param-name>mergeResources</param-name>
        <param-value>false</param-value>
    </context-param>

    <servlet>
        <servlet-name>module-context-loader</servlet-name>
        <servlet-class>ch.elca.el4j.web.context.ModuleContextLoaderServlet</servlet-class>
        <load-on-startup>100</load-on-startup>
    </servlet>
    <servlet>
        <servlet-name>jaxws-servlet-spring</servlet-name>
        <servlet-class>ch.elca.el4j.services.remoting.servlet.WSSpringServlet</servlet-class>
        <load-on-startup>101</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>jaxws-servlet-spring</servlet-name>
        <url-pattern>*.Remotingtests</url-pattern>
    </servlet-mapping>
</web-app>
```

This is a minimal working `web.xml`, all these lines have to be included in it. The `inclusiveLocations` specifies which Spring XML files have to be processed by the `WSSpringServlet`. This servlet provides the JAX-WS webservices. The included XML files are explained in the following sections.

5.3.2.8.3.2 jaxws-protocol-config.xml

Let us first have a look at the protocol configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema

    <!-- JAX-WS Soap Protocol -->
    <bean id="jaxwsProtocol" class="ch.elca.el4j.services.remoting.protocol.JaxwsSoapHeaderContextPassing
        <property name="serviceHost">
            <value>yourserver</value>
        </property>
        <property name="servicePort">
            <value>8080</value>
        </property>
        <property name="contextPath">
```

```
            <value>yourcontextpath</value>
        </property>
        <property name="implicitContextPassingRegistry">
            <ref local="jaxwsImplicitContextPassingRegistry" />
        </property>
        <property name="contextPassingContext">
            <ref bean="jaxwsContextPassingContext" />
        </property>
    </bean>

    <bean id="jaxwsImplicitContextPassingRegistry"
        class="ch.elca.el4j.core.contextpassing.DefaultImplicitContextPassingRegistry" />

    <!-- JAXBContext used by the JaxwsJaxb protocol to marshall the implicit context -->
    <bean id="jaxwsContextPassingContext" class="javax.xml.bind.JAXBContext" factory-method="newInstance"
        <constructor-arg index="0">
            <list>
                <value>yourContextPassingValue</value>
            </list>
        </constructor-arg>
    </bean>
</beans>
```

The main bean of this file is the `JaxwsProtocol`. This bean has one other important property beside the well known properties `serviceHost`, `servicePort`, `contextPath` and `implicitContextPassingRegistry`: the `jaxwsContextPassingContext` used for the implicit context passing. Every SOAP message that gets transmitted automatically (therefore implicitly) contains this value. Like this you can share a context over client and server.

5.3.2.8.3.3 jaxws-server-config.xml

The server-side configuration file could look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema

    <import resource="jaxws-protocol-config.xml" />

    <bean id="jaxwsCalculatorExporter"
        class="ch.elca.el4j.services.remoting.RemotingServiceExporter">
        <property name="remoteProtocol">
            <ref bean="jaxwsProtocol" />
        </property>
        <property name="serviceInterface">
            <value>ch.elca.el4j.tests.remoting.service.Calculator</value>
        </property>
        <property name="serviceName">
            <value>Calculator.Jaxws.Remotingtests</value>
        </property>
        <property name="service">
            <idref bean="jaxwsCalculatorImpl"/>
        </property>
    </bean>

    <bean id="jaxwsCalculatorImpl"
        class="ch.elca.el4j.tests.remoting.service.impl.CalculatorImplJaxws" />
</beans>
```

There is nothing special compared to other protocols.

5.3.2.8.4 Client side setup

The client side looks like other remote protocols too. Here an example:

5.3.2.8.4.1 jaxws-client-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema

    <import resource="jaxws-protocol-config.xml" />

    <!-- Default JAX-WS Setup -->
    <bean id="calculator"
        class="ch.elca.el4j.services.remoting.RemotingProxyFactoryBean">
        <property name="remoteProtocol">
            <ref bean="jaxwsProtocol" />
        </property>
        <property name="serviceInterface">
        <!-- use ...service.gen.CalculatorWS if no proxies should be generated -->
            <value>
                ch.elca.el4j.tests.remoting.service.Calculator
            </value>
        </property>
        <property name="serviceName">
            <value>Calculator.Jaxws.Remotingtests</value>
        </property>
    </bean>
</beans>
```

Note: It is very important to import the same `jaxws-protocol-config.xml` as used on server side.

If `serviceInterface` is equal on the client and server side, dynamic proxies are generated to make the client stubs implement the same interface as on the server (reason: JAX-WS generates independent stubs). If this is not whished, the generated class can be used. The name is deduced from the annotations inside the webservice class (see following section).

5.3.2.8.5 Implementation constraints

JAX-WS does not make it easy to integrate it into the EL4J framework. The development of JAX-WS webservices implies the use of special tools that generate client stubs class files. Unfortunately, these stubs don't implement the server's service interface. It is therefore necessary, to write different code on the server (that uses the specified interfaces) and code for the client (that uses the generated classes). The el4j framework tries to mitigate this by providing automatically created dynamic proxies. These allow interacting with the client stubs using the original interface. But this is only possible if some rules are strictly applied:

- Annotate the webservice class as `@WebService`, set the following properties
  - The `name` property must be the name of the implemented core interface + "WS"
  - The `serviceName` property must be the name of the implemented core interface + "WSService"
  - The `targetNamespace` property must be "http://gen." + package name of implemented core interface
- Optionally annotate all methods selected to export with `@WebMethod` (otherwise all methods are exported)
- Pay attention when using `@XML...` annotations. Do not rename properties, otherwise the dynamic proxy cannot perform the translation of the properties.
- Maps aren't supported. So use List<!SomeKeyAndValueType> objects instead.
- Neither multi-dimensional arrays nor nested collections like List<List<...>> are supported out-of-the-box. They need a type adapter (`@XmlJavaTypeAdapter`). An example can be found in the unit tests where an adapter to `int[][]` is shown.

If webservice methods generate exceptions, mind the following:

- Internally, exceptions have to be reconstructed on the client side.
- Therefore only properties of an exception that can be accessed via getter/setter are preserved. Rely only on these properties!

The @WebService annotation must be of the following form:

```
@WebService(name = "XyzWS",
    serviceName = "XyzWSService",
    targetNamespace = "http://gen.package-name-of-Xyz/")
public class XyzImpl implements Xyz {
    @WebMethod
    public void doSomething(){...}
}
```

If you want to annotate the interface instead of the implementation add `@WebService(... endpointInterface=Xyz ... )` to the implementation class.

5.3.2.8.6 Development

The workflow for developing JAX-WS webservice *without* the help of the EL4J framework looks like the following.

The server:

- Write the webservice class and annotate it (@WebService, @WebMethod ...)
- Generate helper classes (needed by the WS-Servlet) using the `wsgen` tool
- Configure the WS-Servlet to load the generated classes.

The client

- Generate a WSDL file from the webservice class using the `wsgen` tool (parameter: -wsdl)
- Replace in the generated WSDL file the string `REPLACE_WITH_ACTUAL_URL` with the actual webservice URL
- Run the `wsimport` tool to generate the client stubs
- Use these classes to communicate with the webservice.

The EL4J framework simplifies this process by integrating the `wsgen` and `wsimport` tool into the maven build process. The additions in the `pom.xml` file look like this:

```
    <build>
        <plugins>
            <plugin>
                <groupId>ch.elca.el4j.maven.plugins</groupId>
                <artifactId>maven-jaxws-plugin</artifactId>
                <executions>
                    <execution>
                        <goals>
                            <goal>wsimport</goal>
                            <goal>wsgen</goal>
                        </goals>
                        <configuration>
                            <hostURL>http://${jee-web.host}:${jee-web.port}</hostURL>
                            <contextURL>${jee-web.context}</contextURL>
                            <serviceURL>*.Jaxws.Remotingtests</serviceURL>
                            <sei>*</sei>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
```

```
    </build>
```

In the `sei` tag, either a (list of) fully qualified class names or a `*` is allowed (this is the default value). The latter goes through all the class files and picks the ones having a `@WebService` annotation (see also MavenJaxwsPlugin).

In this configuration WSDL files are generated from the class files by searching for `@WebService` annotations. Then the plugin generates from all the WSDL files in the `<wsdlDirectory>` (which is by default the output directory of the previous step) client stubs. The client will try to connect the service at the URL composed of the three parameters `<hostURL>`, `<contextURL>` and `<serviceURL>`.

However, it is also possible to modify the generate WSDL files. Specify another `<wsdlDirectory>` and copy the modifies files to this folder. The next `mvn install` will then create the client stubs from these WSDL files.

So the development using EL4J looks like this: The server:

- Write the webservice class and annotate it (@WebService, @WebMethod ...)
- Add and configure the `jaxws-maven-plugin` in your `pom.xml`
- Configure all xml files described above (`web.xml`, `remote-servlet.xml`, `jaxws-server-config.xml`, `jaxws-client-config.xml`)

The client:

- Get the client stub (e.g. using `getBean("Calculator")`). You will then either get access directly to the generated classes or EL4J will create proxies for you (depending on the chosen `serviceInterface` in `jaxws-client-config.xml`)

5.3.2.9 How to use the `Soap` protocol (Axis 1 - this protocol is now deprecated)

The `Soap` protocol must also be used in an webserver like the `Hessian` and `Burlap` protocols. Please read first the **How to use the `Hessian` protocol** before beginning with `Soap`.

5.3.2.9.1 JAX-RPC

`JAX-RPC` (API for XML-based Remote Procedure Call - http://java.sun.com/xml/jaxrpc) is the standard given by java to build RPC-illusion type web services. It uses the `Soap` protocol (http://www.w3.org/TR/soap/). `JAX-RPC` is only the definition of web services. One implementation of `JAX-RPC` is `Axis` (http://ws.apache.org/axis/). In **module-remoting_soap** we use `Axis`.

5.3.2.9.2 Axis (implementation of JAX-RPC)

Axis is used to connect the XML and the java world. Therefore we need serialization and deserialization. Primitive types like int, long, double, String are no problem. They can be covered to XML schema type definitions (`xsd`). More complex types like a `java.util.HashMap` or java beans must be handled via serializers and deserializers. In package `org.apache.axis.encoding.ser` there are several type handlers for frequently used types. Custom serializer and deserializer will be explained later.

5.3.2.9.3 Soap style and usage

The used soap style is **wrapped** (aka **rpc illusion**) and the usage is **literal**. For more information please contact MartinZeltner.

5.3.2.9.4 Server side setup

5.3.2.9.4.1 web.xml

As in `Hessian` and `Burlap` protocols we have to use a `web.xml` file. This could look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

    <!-- Used by all protocols -->
    <servlet>
        <servlet-name>remote</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>100</load-on-startup>
    </servlet>

    <!-- Used by Soap protocol -->
    <listener>
        <listener-class>org.apache.axis.transport.http.AxisHTTPSessionListener</listener-class>
    </listener>
    <servlet>
        <servlet-name>AxisServlet</servlet-name>
        <servlet-class>org.apache.axis.transport.http.AxisServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <!-- Used to see all deployed services with its methods as list -->
        <servlet-name>AxisServlet</servlet-name>
        <url-pattern>/servlet/AxisServlet</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <!-- Url to deployed services -->
        <servlet-name>AxisServlet</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
    <session-config>
        <!-- Default to 5 minute session timeouts -->
        <session-timeout>5</session-timeout>
    </session-config>
</web-app>
```

The `DispatcherServlet` is already known. What stands out is that this servlet has the startup number `100`. Servlets with numbers less than `100` will be started before. One such servlet is the `AxisServlet`. With `Soap` the `AxisServlet` processes each request, not the `DispatcherServlet`. All servlet mappings points to the `AxisServlet` and not to the `DispatcherServlet`. An important servlet mapping beside the `/services/*` is the `/servlet/AxisServlet`. With the help of this, it is possible to get an overview of all deloyed services. The services itself will be deployed in `/services/NameOfTheDeployedService`. The listener `AxisHTTPSessionListener` and the session timeout config are used for session management.

5.3.2.9.4.2 remote-servlet.xml

As `Hessian` and `Burlap` protocols you have to have a file with the name `remote-servlet.xml` in the same directory as the `web.xml` is. The name of this file depends on the `DispatcherServlet` name in `web.xml` such as with `Hessian` or `Burlap`. Such a file could look like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <import resource="soap-protocol-config.xml"/>
    <import resource="soap-server-config.xml"/>
</beans>
```

This file only imports two config files: One for the protocol and one for the server configuration.

5.3.2.9.4.3 soap-protocol-config.xml

Let us first have a look at the protocol configuration file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <bean id="soapProtocol" class="ch.elca.el4j.services.remoting.protocol.Soap">
        <property name="serviceHost">
            <value>yourserver</value>
        </property>
        <property name="servicePort">
            <value>8080</value>
        </property>
        <property name="contextPath">
            <value>yourcontextpath</value>
        </property>
        <property name="protocolSpecificConfiguration">
            <ref local="soapProtocolSpecificConfiguration"/>
        </property>
        <property name="exceptionTranslationEnabled">
            <value>true</value>
        </property>
        <property name="exceptionManager">
            <ref local="soapExceptionManager"/>
        </property>
        <property name="implicitContextPassingRegistry">
            <ref local="implicitContextPassingRegistry" />
        </property>
    </bean>

    <bean id="soapProtocolSpecificConfiguration"
          class="ch.elca.el4j.services.remoting.protocol.soap.SoapSpecificConfiguration">
        <property name="typeMappings">
            <list>
                <bean class="ch.elca.el4j.services.remoting.protocol.soap.axis.encoding.BeanTypeMapping">
                    <property name="types">
                        <list>
                            <value>my.package.MyValueObject</value>
                        </list>
                    </property>
                </bean>
            </list>
        </property>
    </bean>

    <bean id="soapExceptionManager"
          class="ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.SoapExceptionManager">
        <property name="defaultHandler">
            <bean class="ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.DefaultHandler">
                <property name="sendServerSideStackTraceActive">
                    <value>true</value>
                </property>
            </bean>
        </property>
        <property name="soapExceptionHandlers">
            <map>
                <entry key="my.package.MyExceptionWithStringArgumentConstructor">
                    <bean class="ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.StringAr
                        <property name="sendServerSideStackTraceActive">
                            <value>true</value>
                        </property>
                    </bean>
                </entry>
            </map>
        </property>
```

```
        <property name="allowedTranslations">
            <set>
                <value>ch.elca.el4j.core.exceptions.BaseException</value>
                <value>ch.elca.el4j.core.exceptions.BaseRTException</value>
            </set>
        </property>
    </bean>

    <bean id="implicitContextPassingRegistry" class="ch.elca.el4j.core.contextpassing.DefaultImplicitCont
</beans>
```

This main bean of this file is the **ch.elca.el4j.services.remoting.protocol.Soap**. This bean has three other important properties beside the well known properties `serviceHost`, `servicePort`, `contextPath` and `implicitContextPassingRegistry`.

Protocol-specific configuration

One of these properties is the **protocolSpecificConfiguration** which expects a bean of type `ch.elca.el4j.services.remoting.protocol.soap.SoapSpecificConfiguration`. This is a configuration object that contains configuration data that is only needed in special cases. It contains the following three properties:

- **namespaceUri**
  This property is used to set the namespace uri manually. By default this is the service URL.

- **allowedMethods**
  This property is used to define which methods of services should be exported. This must be a comma separated list of method names. Per default all methods will be exported (property value *).

- **typeMappings**
  This property receives a list of beans of class `ch.elca.el4j.services.remoting.protocol.soap.axis.encoding.TypeMapping`. Type mappings are used to serialize and deserialize types that are not standard. This class has the following three properties:
  - **types**
    Contains a list of `java.lang.Class` classes, which must be mapped. It is a list because the same type mapping can be used for several types.
  - **serializerFactory**
    Contains a serializer factory that implements the interface `javax.xml.rpc.encoding.SerializerFactory`. A lot of serializer factories can be found in package `org.apache.axis.encoding.ser`. For example there is the serializer factory `MapSerializerFactory` which can be used for types like `java.util.HashMap`.
  - **deserializerFactory**
    This is the counterpart of the `serializerFactory`. This factory must implement the interface `javax.xml.rpc.encoding.DeserializerFactory`. A lot of deserializer factories can also be found in package `org.apache.axis.encoding.ser`. For example there can be found the deserializer factory `MapDeserializerFactory` which must be used for types like `java.util.HashMap`.

For the most frequently used type mappings, the mapping of java beans, there is the class `ch.elca.el4j.services.remoting.protocol.soap.axis.encoding.BeanTypeMapping` which extends the class `ch.elca.el4j.services.remoting.protocol.soap.axis.encoding.TypeMapping`. The `BeanTypeMapping` differs from the `TypeMapping` in that it has preset serializer (`org.apache.axis.encoding.ser.BaseSerializerFactory`) and deserializer (`org.apache.axis.encoding.ser.BaseDeserializerFactory`) factory.

Pay attention to the fact that at the end every complex type must be matched to several primitive xsd (xml

schema type definition) types such as mentioned above.

Exception translations

One thing that can not be disregarded is the exception handling. The property **exceptionTranslationEnabled** is set by default to `true`.

- **exceptionTranslationEnabled** = `true`
  Thrown exceptions can be left as they are. If exceptions must **NOT** be instantiated with a non-argument-constructor, the next property `soapExceptionManager` must be adapted. Exception translation is only recommended if you would like to communicate between java JVMs, otherwise you should turn the exception translation off.
- **exceptionTranslationEnabled** = `false`
  In this case the next explained property `exceptionManager` is not used! Thrown exceptions must respect the following rules:
  - Each exception must extend `java.rmi.RemoteException`. This is defined in jaxrpc specification (see http://java.sun.com/xml/downloads/jaxrpc.html).
  - Exceptions must be serializable (and deserializable). The most apropriate way is to add getter and setter methods to each exception, so they can be serialized and deserialized as beans. See the **typeMappings** above. Exceptions can also have getter methods and an apropriate constructor for them, but this is not the recommended way. If a constructor can be found, where the parameter types matches, the constructor will be preferred over the setter methods. So, they recommended way is to implement the default constructor and write getter **and** setter methods.
  - You need to setup an appropriate exception manager (see below).

Exception manager

The property **exceptionManager** must be set to an instance of class `ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.SoapExceptionManager`. The properties of this class must be set to classes that implement the interface `ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.SoapExceptionHandler`. `ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.DefaultHandler` is such a class. This class can handle exceptions that **must** be instantiated via the default constructor. The handler `ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.StringArgumentHandler` extends the `DefaultHandler` and can handle exceptions which must be instantiated by using a constructor with several `java.lang.String` attributes. But to know how to get the information out of an exception on server side and be able to instantiate it on client side is very exceptionspecific. The `StringArgumentHandler` can only handle exceptions of type `ch.elca.el4j.core.exceptions.BaseException`, because it knows it can get the information from method `getFormatParameters`, but they must be strings. Now back to the properties of `SoapExceptionManager`.

- **defaultHandler**
  Needs a class which implements the `SoapExceptionHandler` interface. The default is class `ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.DefaultHandler`. This handler will be used if no specific handler can be found.
- **soapExceptionHandlers**
  Needs a map of classes that implements the `SoapExceptionHandler` interface. The key of each entry represents the class name of the exception, which must be translated with the given handler. By default the map is empty.
- **allowedTranslations**
  Allowed translation contains a set of class names, which are allowed to be translated. If a thrown exception is not in this list, it will never be translated, even if there is a handler defined for it.

The `DefaultHandler` class has further the possibility to send the server side stack trace with translated exception. If the property **sendServerSideStackTraceActive** is set to true, you can get the server side stack trace on client side as a string via the method `ch.elca.el4j.services.remoting.protocol.soap.SoapHelper.getLastServerSideStackTrace()`. This method returns always the last server side stack trace of the current thread. By default the property **sendServerSideStackTraceActive** is set to `true`.

5.3.2.9.4.4 soap-server-config.xml

The server-side configuration file could look like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <import resource="soap-protocol-config.xml"/>

    <bean id="soapCalculatorExporter" class="ch.elca.el4j.services.remoting.RemotingServiceExporter">
        <property name="remoteProtocol">
            <ref bean="soapProtocol" />
        </property>
        <property name="serviceInterface">
            <value>ch.elca.el4j.tests.remoting.service.Calculator</value>
        </property>
        <property name="service">
            <ref local="soapCalculatorImpl" />
        </property>
    </bean>

    <bean id="soapCalculatorImpl" class="ch.elca.el4j.tests.remoting.service.impl.CalculatorImpl" />
</beans>
```

As you can see, the server side configuration looks like with other remoting protocols. There is no difference except the choice of the soap protocol of course.

5.3.2.9.5 Client side setup

The client side looks like other remote protocols too. Here an example:

5.3.2.9.5.1 soap-client-config.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <import resource="soap-protocol-config.xml"/>

    <bean id="calculator" class="ch.elca.el4j.services.remoting.RemotingProxyFactoryBean">
        <property name="remoteProtocol">
            <ref bean="soapProtocol" />
        </property>
        <property name="serviceInterface">
            <value>ch.elca.el4j.tests.remoting.service.Calculator</value>
        </property>
    </bean>
</beans>
```

Note: It is very important to import the same `soap-protocol-config.xml` as used on server side.

5.3.2.9.6 Test the Soap service

After you have deployed your Soap service, you should be able to do a call on it. As a first step you have to get the **WSDL** (Web Service Description Language) file. The url to your service is built like the following:

http://*serverhost:serverport*/*contextpath*/services/*serviceName*

- *serverhost* is the host where your web server is running.
- *serverport* is the port where your web server is running.
- *contextpath* is the name of the web context, where the Soap service is running.
- *serviceName* is the name of the service, defined in homonymous property of `RemotingServiceExporter` and `RemotingProxyFactoryBean`. If this property is not defined, the *serviceName* is the name of the service interface plus ".remoteservice" appended at the end.

For the example above the service url would be the following:
http://yourserver:8080/yourcontextpath/services/ch.elca.el4j.tests.remoting.service.Calculator.remoteservice

To get now the wsdl file, you just have to append **?wsdl** to your service url.
http://yourserver:8080/yourcontextpath/services/ch.elca.el4j.tests.remoting.service.Calculator.remoteservice?wsdl

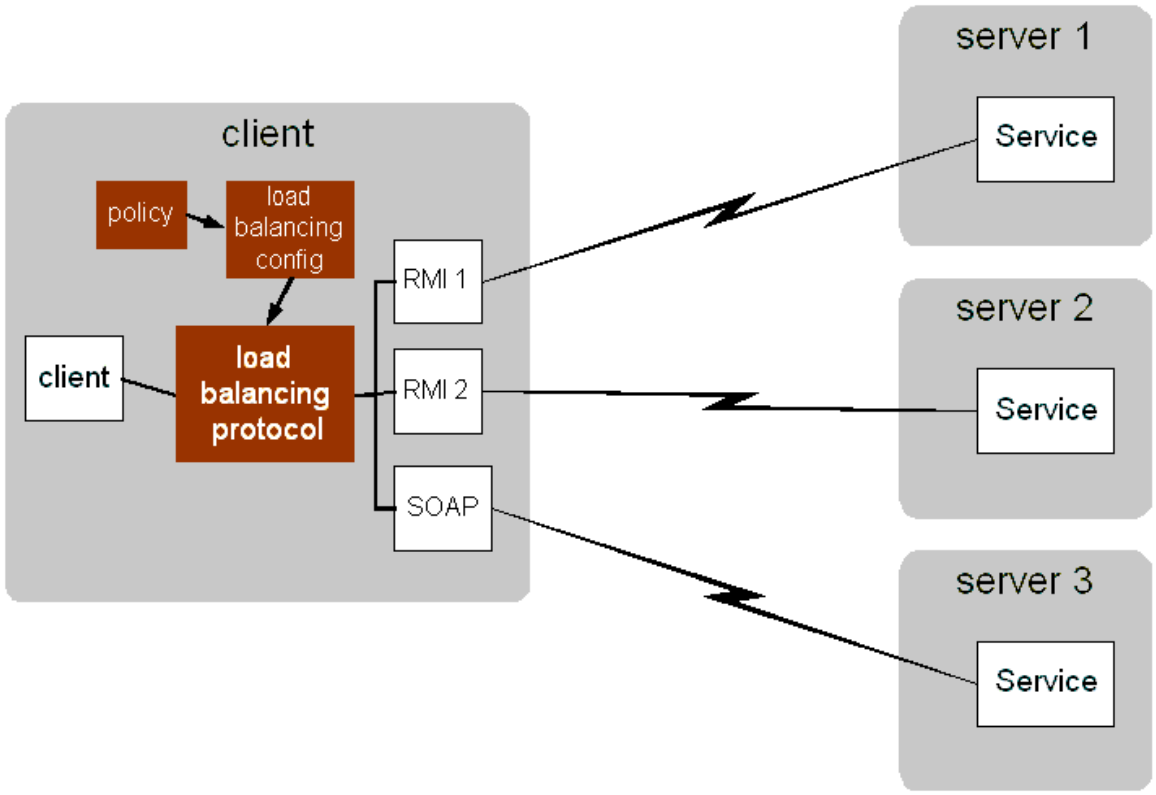
5.3.2.10 How to use the `EJB` protocol

The EJB protocol support is available in the ModuleRemotingEjb.

5.3.2.11 How to use the `Load Balancing` composite protocol

The load balancing protocol is a so-called composite protocol. It applies the Composite Design Pattern and thus allows the user to compose several of the atomic protocols (i.e., a non-composite protocol such as RMI) into this composite protocol. To the outside, it behaves like an atomic protocol. Note that the load balancing protocol is only used on the client side of a (remote) invocation and requires no modifications to existing remoting protocols.

The following figure shows an overview of the load balancing protocol usage. In this example, load balancing composes three (atomic) protocols, however, any number of protocols are supported. Components in red (or in dark color) are part of load balancing, the others are part of other remoting protocols or business objects.



The bean class *LoadBalancingConfiguration* groups the configuration parameters that are supported by load balancing. As an example, consider the following configuration, which defines the client side of the invocation.

It balances load between 3 RMI-servers.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
   <bean id="businessObj"
      class="ch.elca.el4j.services.remoting.RemotingProxyFactoryBean">
      <property name="remoteProtocol">
         <ref bean="loadBalancingProtocol" />
      </property>
      <property name="serviceInterface">
         <value>
             myServiceInterface
         </value>
      </property>
   </bean>

   <bean id="loadBalancingProtocol"  class="ch.elca.el4j.services.remoting.protocol.loadbalancing.protoco
      <property name="protocolSpecificConfiguration">
         <ref bean="loadBalancingProtocolConfiguration" />
      </property>
   </bean>

   <bean id="loadBalancingProtocolConfiguration"
      class="ch.elca.el4j.services.remoting.protocol.loadbalancing.protocol.LoadBalancingProtocolConfigur
      <property name="protocols">
         <list>
            <ref bean="rmiProtocol1"/>
            <ref bean="rmiProtocol2"/>
            <ref bean="rmiProtocol3"/>
         </list>
      </property>
      <property name="policy">
         <ref bean="randomPolicy" />
      </property>
   </bean>

    <bean id="rmiProtocol1" class="ch.elca.el4j.services.remoting.protocol.Rmi">
        <property name="serviceHost">
            <value>localhost</value>
        </property>
        <property name="servicePort">
            <value>8092</value>
        </property>
   </bean>

    <bean id="rmiProtocol2" class="ch.elca.el4j.services.remoting.protocol.Rmi">
        <property name="serviceHost">
            <value>localhost</value>
        </property>
        <property name="servicePort">
            <value>8094</value>
        </property>
    </bean>

    <bean id="rmiProtocol3" class="ch.elca.el4j.services.remoting.protocol.Rmi">
        <property name="serviceHost">
            <value>localhost</value>
        </property>
        <property name="servicePort">
            <value>8099</value>
        </property>
   </bean>
</beans>
```

Although nested load balancing protocols are possible, their usage is discouraged.

5.3.2.11.1 Handling Connection Failures

The load balancing protocol attempts to establish an initial connection to a particular server. If this connection attempt fails, it will ask for the next server from the policy bean and attempt to connect to this server. It repeats this behavior until it succeeds to connect, or no more servers are available. In the latter case, it throws a `ch.elca.el4j.services.remoting.protocol.loadbalancing.NoProtocolAvailableRTException`.

Once a protocol has been initialized and a connection established, it behaves as it would without load balancing. Thus, connection failures are notified to the user.

Retries after a failure: this load-balancing meta-protocol does *not* do automatic retries after a connection failure. In case you would like to have retries, have a look e.g. at the auto-idempotency module that has a retry-interceptor for this purpose. The semantics of the load-balancing meta-protocol is similar to the one of the jboss clustering support.

5.3.2.11.2 Policies

The load balancing protocol comes with a set of predefined policies. These policies govern the sequence in which protocol instances are invoked. Before every method invocation, the load balancing protocol retrieves the next protocol instance to invoke from the installed policy instance.

To minimize overhead, the load balancing protocol caches protocol instances and reuses these cached instances rather than recreating them every time.

Assume that p_i denotes policy instance p_i and that load balancing composes the protocol set {p_1, p_2, p_3}. For instance, p_i could denote the protocol RMI connecting to server running on xyz.elca.ch:7000. The following policies are currently supported:

- *random*: each new call goes to a randomly found protocol instance
- *roundrobin*: p_1 -> p_2 -> p_3 -> p_1 -> p_2 -> ...
- *redirectuponfailure*: p_1 -> p_1 -> p_1 ---"p1 fails"---> p_2 -> p_2 -> ...

The *random* policy removes protocols when a connection failure occurs. The *roundrobin* and *redirectuponfailure* policies do not exclude protocols that cause a connection failure, but switch to the next protocol. This behavior is well suited to handle transient network failures. With a transient failure, the server is still up and running and there is no reason not to reconnect to this server again at a later point in time. With the *random* policy, such servers are excluded. Indeed, with random policy, the load balancing protocol may (with low probability) repeatedly try to connect to the same, temporarily unavailable, server. Thus, these "failed" servers need to be excluded. Consequently, an unstable network may lead to the case in which servers are no longer considered although they may be up and running. It is the application developers responsibility to pick a policy suitable to his/her application, or to plugin his/her own policy.

The installation of an appropriate policy can be done using attribute `policy`. The default policy is *random*.

Defining a customary policy

If the need arises applications can install their own policies to work with load balancing. All policies must extend class `ch.elca.el4j.services.remoting.protocol.loadbalancing.protocol.policy.AbstractPolicy`. The policy implementation receives a notification every time a failure occurs with a particular atomic protocol.

5.3.2.11.3 Limitations

Currently, the load balancing plugin has only been tested with the RMI protocol. Although the tests with other protocols have not yet been performed, there is no reason it should not work with other protocols. Indeed, the load balancing protocol makes no assumption on the protocols other than the ones used also by the instantiating factory.

5.3.2.11.4 Further reading

Please see the load balancing test cases in module-remoting-tests-apps for further examples of how to use the load balancing protocol. Also, please refer to the documentation of the corresponding atomic protocols to learn how to use these.

5.3.2.12 Introduction to implicit context passing

The implicit context allows passing context data along with normal method calls. The term `implicit context` refers to any kind of object that should be included in a call. It is included in a service call in the calling direction, not in the response. Therefore changes made on a server do not affect the client's implicit context.

In practice, there are different ways to implement implicit context passing. The easiest way is if the used communication protocol supports it: one can simply add the implicit context to the remote invocations. However, in the Java context, many protocols do not directly support implicit context passing. Our solution is to add the implicit context in the form of a Map as the last argument of methods. Behind the existing interface, we add transparently a shadow interface that has the additional parameter added. Please refer to the internal design section for more details on this.

Implicit context passing is entirely optional, it can be enabled by defining a context passing registry on the level of the protocol definition.

A service that wants to have some `implicit context` passed, must implement the interface `ch.elca.el4j.remoting.contextpassing.ImplicitContextPasser`. This passer has two responsibilities: to get the data to pass along with the call on the client side, and to push the received data to the service before the real invocation on the server side. One instance of this context passer has to be registered to an `ch.elca.el4j.remoting.contextpassing.ImplicitContextPassingRegistry` on the client side and a second to the registry on the server side. Before a method call is made, the implicit context to include in that call is assembled by the client's registry. Every registered `AbstractImplicitContextPasser` is called to deliver its data. The same thing happens on the server side when the remote call is received, every passer is called by the registry to push its data to the service. This is done completely transparent for the service and the client, if the configuration is properly set up.

On server **and** client side the configuration could look like the following (only a part from the bean configuration file):

```
<bean id="implicitContextPassingRegistry"
        class="ch.elca.el4j.core.contextpassing.DefaultImplicitContextPassingRegistry"/>

<bean id="authenticationServiceContextPasser"
        class="ch.elca.myproject.MyImplicitContextPasserOne">
    <property name="implicitContextPassingRegistry">
        <ref local="implicitContextPassingRegistry"/>
    </property>
</bean>

<bean id="authenticationServiceContextPasser"
        class="ch.elca.myproject.MyImplicitContextPasserTwo">
    <property name="implicitContextPassingRegistry">
        <ref local="implicitContextPassingRegistry"/>
    </property>
</bean>
```

In this example we have two classes that extend the class `AbstractImplicitContextPasser`. On the client side, the `DefaultImplicitContextPassingRegistry` gets the `Serializable` object from both `AbstractImplicitContextPasser` and on server side the `DefaultImplicitContextPassingRegistry` puts the `Serializable` object to the `AbstractImplicitContextPasser` where it has been received the object.

5.3.3 Benchmark

The module **module-remoting-demos** contains a benchmark for the protocols `Rmi`, `Hessian` and `Burlap`. The benchmark compares each protocol with and without context passing. With context passing the `RemotingProxyFactoryBean` and the `RemotingServiceExporter` from this module will be used. Without context passing the classes from Spring will be used directly. By the way, these Spring classes are used behind the scene of this module, so the results of benchmarks without context information should be faster than benchmarks with context information.

The following is the console output of the benchmark was running on a Pentium IV with a 3.0 GHz processor and 1 GB of memory:

```
Please wait, benchmarks are running...
Benchmark 1 of 9 with name 'rmiWithoutContextCalculator' is running... done.
Benchmark 2 of 9 with name 'rmiWithContextCalculator' is running... done.
Benchmark 3 of 9 with name 'hessianWithoutContextCalculator' is running... done.
Benchmark 4 of 9 with name 'hessianWithContextCalculator' is running... done.
Benchmark 5 of 9 with name 'burlapWithoutContextCalculator' is running... done.
Benchmark 6 of 9 with name 'burlapWithContextCalculator' is running... done.
Benchmark 7 of 9 with name 'soapWithContextCalculator' is running... done.
Benchmark 8 of 9 with name 'httpInvokerWithoutContextCalculator' is running... done.
Benchmark 9 of 9 with name 'httpInvokerWithContextCalculator' is running... done.
-----------------------------------------------------------------------------------------
| *Name of test*                      | *Method 1 [ms]* | *Method 2 [ms]* | *Method 3 [ms]* |
-----------------------------------------------------------------------------------------
| rmiWithoutContextCalculator         | 1.57            | 1.72            | 5.94            |
-----------------------------------------------------------------------------------------
| rmiWithContextCalculator            | 1.1             | 1.87            | 3.91            |
-----------------------------------------------------------------------------------------
| hessianWithoutContextCalculator     | 0.63            | 11.57           | 18.78           |
-----------------------------------------------------------------------------------------
| hessianWithContextCalculator        | 0.63            | 13.14           | 18.3            |
-----------------------------------------------------------------------------------------
| burlapWithoutContextCalculator      | 0.62            | 1.1             | 17.68           |
-----------------------------------------------------------------------------------------
| burlapWithContextCalculator         | 0.78            | 1.09            | 17.52           |
-----------------------------------------------------------------------------------------
| soapWithContextCalculator           | 9.7             | 17.2            | 27.54           |
-----------------------------------------------------------------------------------------
| httpInvokerWithoutContextCalculator | 2.66            | 3.13            | 4.69            |
-----------------------------------------------------------------------------------------
| httpInvokerWithContextCalculator    | 2.04            | 2.66            | 5.63            |
-----------------------------------------------------------------------------------------

Legend:        Method 1: double getArea(double a, double b)
               Method 2: void throwMeAnException() throws CalculatorException
               Method 3: int countNumberOfUppercaseLetters(String textOfSize60kB)
```

To **execute the benchmark on your machine** you can run the demo yourself.

5.3.4 Remoting semantics/ Quality of service of the remoting

5.3.4.1 Cardinality between client using the remoting and servants providing implementations

This section discusses how the clients and client requests are mapped to servant objects and how servant objects need to be implemented. The *servant object* is the object that runs on the server-side of the remoting

and implements the real functionality. Basically we allow either a many to 1 mapping of clients to servant objects (Singleton in table below) and a 1 to 1 mapping (Client-activated in table below). A servant object remoted as a *Singleton* can optionally be pooled on the server side (this could then be extended to something similar to the stateless session bean semantics of EJB). In order to set this up, please refer to the spring reference manual. The semantics of the EJB remoting is slightly different. It is required that you understand what you are doing when switching between EJB and other remoting protocols.

Singleton objects that are not pooled need either be reentrant or be properly synchronized (some use the term "reentrant" in a way that these 2 things are equivalent (as a properly synchronized class is naively reentrant with this signification of reentrant)).

The following table summarizes this. On the left hand side, it shows the *desired semantics* and how the servant POJOs are implemented, the right hand side indicates how this semantics is realized with each protocol:

<table>
<tr><th colspan="3">Desired semantics /implementation</th><th>Rmi</th><th>Hessian</th><th>Burlap</th><th>Soap</th><th>EJB</th></tr>
<tr><td rowspan="3">Singleton</td><td colspan="2">POJO is reentrant</td><td colspan="4">Standard use</td><td rowspan="2">N/A</td></tr>
<tr><td rowspan="2">POJO is not reentrant</td><td>synchronized</td><td colspan="4">By using an interceptor in or synchronizing in code</td></tr>
<tr><td>pooled</td><td colspan="4">By using spring's pooling target source (see spring doc)</td><td>Stateless</td></tr>
<tr><td colspan="3">Client-activated</td><td colspan="4">TODO: Is currently not implemented.</td><td>Statefull</td></tr>
</table>

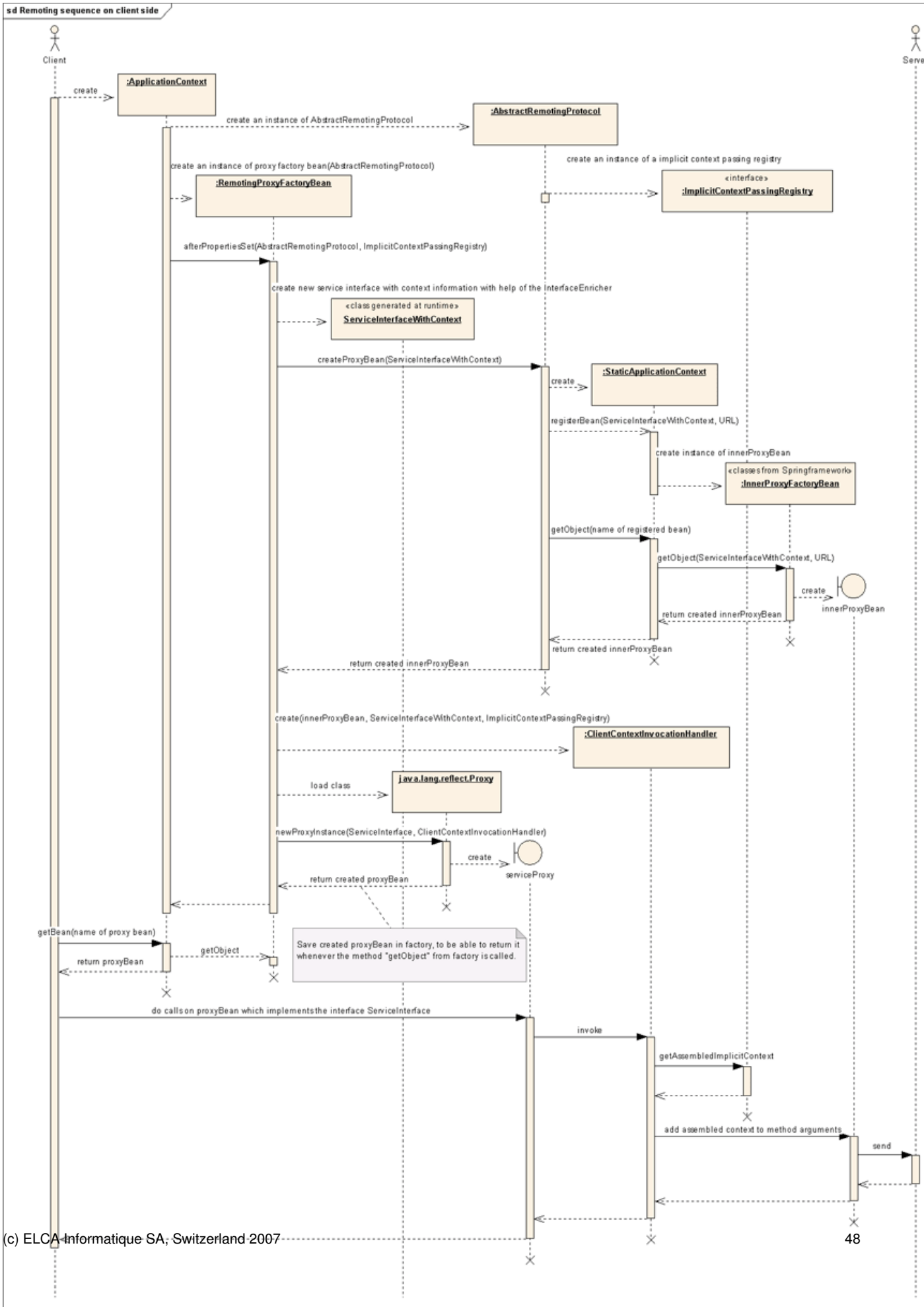
5.3.4.2 What happens when there is a timeout or another problem during remoting

The following document describes what happens in more details. It has been contributed by MSM from the Orchestra project. To understand their context: they use this EL4J remoting to communicate between processes and other projects. They run their code within the ModuleDaemonManager (this explains some of their behavior). The exceptions shown in section 2.5 are thrown because creating 1200 tickets takes about 20 minutes (and 20 minutes is bigger than the timeout value). Thank you, Marc!
RemoteServiceBehaviour_10.doc.

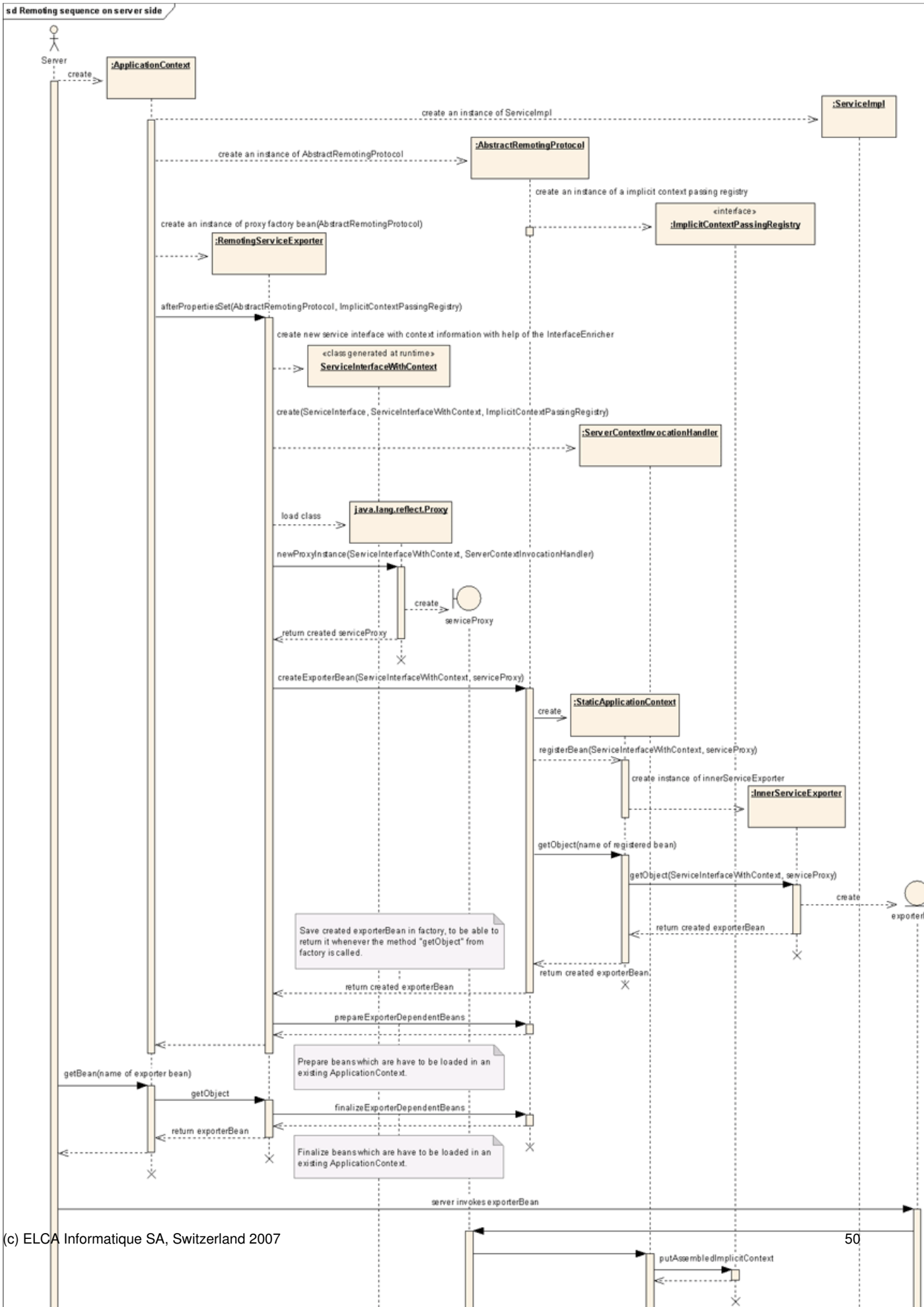
5.4 Internal design

5.4.1 Sequences

5.4.1.1 Sequence diagramm from client side


sd Remoting sequence on client side
Client
Serve
create
:ApplicationContext
create an instance of AbstractRemotingProtocol
:AbstractRemotingProtocol
create an instance of proxy factory bean(AbstractRemotingProtocol)
:RemotingProxyFactoryBean
create an instance of a implicit context passing registry
«interface»
:ImplicitContextPassingRegistry
afterPropertiesSet(AbstractRemotingProtocol, ImplicitContextPassingRegistry)
create new service interface with context information with help of the InterfaceEnricher
«class generated at runtime»
ServiceInterfaceWithContext
createProxyBean(ServiceInterfaceWithContext)
create
:StaticApplicationContext
registerBean(ServiceInterfaceWithContext, URL)
create instance of innerProxyBean
«classes from Springframework»
:InnerProxyFactoryBean
getObject(name of registered bean)
getObject(ServiceInterfaceWithContext, URL)
create
innerProxyBean
return created innerProxyBean
return created innerProxyBean
return created innerProxyBean
create(innerProxyBean, ServiceInterfaceWithContext, ImplicitContextPassingRegistry)
:ClientContextInvocationHandler
load class
java.lang.reflect.Proxy
newProxyInstance(ServiceInterface, ClientContextInvocationHandler)
create
serviceProxy
return created proxyBean
getBean(name of proxy bean)
getObject
return proxyBean
Save created proxyBean in factory, to be able to return it whenever the method "getObject" from factory is called.
do calls on proxyBean which implements the interface ServiceInterface
invoke
getAssembledImplicitContext
add assembled context to method arguments
send

5.4.1.2 Sequence diagramm from server side


sd Remoting sequence on server side
Server
create
:ApplicationContext
create an instance of ServiceImpl
:ServiceImpl
create an instance of AbstractRemotingProtocol
:AbstractRemotingProtocol
create an instance of a implicit context passing registry
«interface»
:ImplicitContextPassingRegistry
create an instance of proxy factory bean(AbstractRemotingProtocol)
:RemotingServiceExporter
afterPropertiesSet(AbstractRemotingProtocol, ImplicitContextPassingRegistry)
create new service interface with context information with help of the InterfaceEnricher
«class generated at runtime»
ServiceInterfaceWithContext
create(ServiceInterface, ServiceInterfaceWithContext, ImplicitContextPassingRegistry)
:ServerContextInvocationHandler
load class
java.lang.reflect.Proxy
newProxyInstance(ServiceInterfaceWithContext, ServerContextInvocationHandler)
create
serviceProxy
return created serviceProxy
createExporterBean(ServiceInterfaceWithContext, serviceProxy)
create
:StaticApplicationContext
registerBean(ServiceInterfaceWithContext, serviceProxy)
create instance of innerServiceExporter
:InnerServiceExporter
getObject(name of registered bean)
getObject(ServiceInterfaceWithContext, serviceProxy)
create
exporterB
return created exporterBean
Save created exporterBean in factory, to be able to return it whenever the method "getObject" from factory is called.
return created exporterBean
return created exporterBean
prepareExporterDependentBeans
Prepare beans which are have to be loaded in an existing ApplicationContext.
getBean(name of exporter bean)
getObject
finalizeExporterDependentBeans
return exporterBean
Finalize beans which are have to be loaded in an existing ApplicationContext.
server invokes exporterBean
putAssembledImplicitContext

5.4.2 Creating a new interface during runtime

In this module, the implicit context can **optionally** be passed from client to server without changing the existing code. This is done by creating a new interface during runtime that slightly changes, **decorates** the service existing interface. But it is important that the created interface has no dependency to the service interface and vice versa. This enrichment is done with the help of the BCEL (Byte Code Engineering Library).

All classes for the interface enrichment are in package `ch.elca.el4j.util.interfaceenrichment` in the **module-core**. The class `InterfaceEnricher` offers methods to create such a new interface. One method is the `createShadowInterfaceAndLoadItDirectly` with parameters `serviceInterface`, `interfaceEnricher` and `classLoader`. The `serviceInterface` is the interface which has to be enriched, the `interfaceEnricher` is a class which implements the interface `EnrichmentDecorator` and the `classLoader` is the `ClassLoader` where the new class has to be loaded. The usage of this classes is explained in its javadoc.

By default, we do the interface enrichment during runtime, if possible. This uses the same mechanism as the CGLIB. The advantage of this is that it can be made transparent in most cases. In contexts where runtime enrichment is not applicable, the interface enrichment also supports interface enrichment during build time.

5.4.3 Internal handling of the `RMI` protocol (in spring and EL4J)

Perhaps you have recognized that the business interface does not extend the class `java.rmi.Remote` and the methods do not have to throw a `java.rmi.RemoteException`. This is normally mandatory to be able to use the `RMI` protocol. Additionally, **prior** to Java 1.5 you have to run the **RMIC (RMI-Compiler)** during build time for the service that implements the service interface.

If you have a service interface that fulfills the RMI requirements and you are using these classes in combination with the `RmiProxyFactoryBean` and `RmiServiceExporter`, the **real** RMI service will be exported. That means that everybody can access the service, whether it uses springs or EL4J's remoting facility or not.

If you have got a service interface that does not extend `java.rmi.Remote` and does not throw a `java.rmi.RemoteException` on each method, Spring will not publish the service directly via `RMI`. Spring uses Java's reflection to send calls through a generic invoke method. In the framework it has a `RMI invoker`, which tunnels every request through the method `invoke`. The `RMI invoker` extends `java.rmi.Remote` and the method `invoke` throws a `java.rmi.RemoteException`. The Stub and skeleton are already prebuild for this `RMI invoker`. (This is the default spring semantics.)

EL4J adds some more flexibility: via the interface decoration, it can wrap a non-RMI-conformant interface with a conformant interface. This support is again transparent for the user. This work similarly in the case of EJB. In the current implementation, this generates a double-indirection of interfaces. The last interface is visible to RMI, the first interface is visible to the user.

Business Interface --> Shadow Interface 1 (RMI-conformant) --> Shadow interface 2 (RMI-conformant and with implicit context passing)

5.4.4 To be done

The module should be able to export a rmi service with its service interface, but the service interface should not have any dependencies to `RMI`. To solve this problem we could create an ant task to call the interface enricher and let him generate and save the generated interface to disk. The interface enricher can already do that. So we could be able to wrap the service implementation with a class, which implements the generated service interface and could redirect method invocations. At the end we could also use the `rmic` to create stub and skeleton for Java 1.4 and below.

5.5 Related frameworks

5.5.1 extrmi

A further framework which also pass the context transparently is the `extrmi`. It can be found at sourceforge http://sourceforge.net/projects/wenbozhu/

- In this solution the implicit context is passed by using `java.lang.reflect`. So this is the same way like this module does it in the worst case. Why worst case? If the remoting is done by reflection the server side published interface is always the same. In a first way this sounds very good, but if you would like to access the server without using the given client stub you have not got any chance.
- Another negative point is that this framework does not help you to simplify switching between remoting protocols. You always have to adapt the business classes to the needs of the used remoting protocol.

Article reference: http://www.javaworld.com/javaworld/jw-04-2005/jw-0404-rmi_p.html

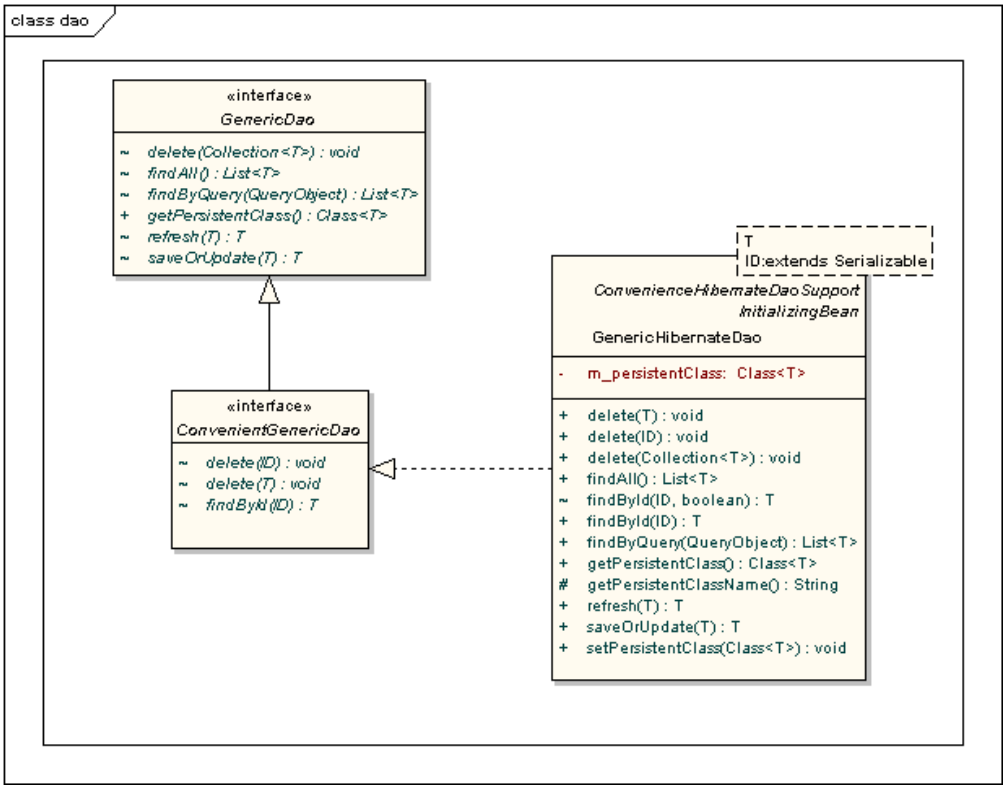
5.5.2 Javaworld 2005 idea

http://www.javaworld.com/javaworld/jw-03-2005/jw-0314-usersession_p.html

6 Generic DAOs in EL4J

6.1 Basic introduction

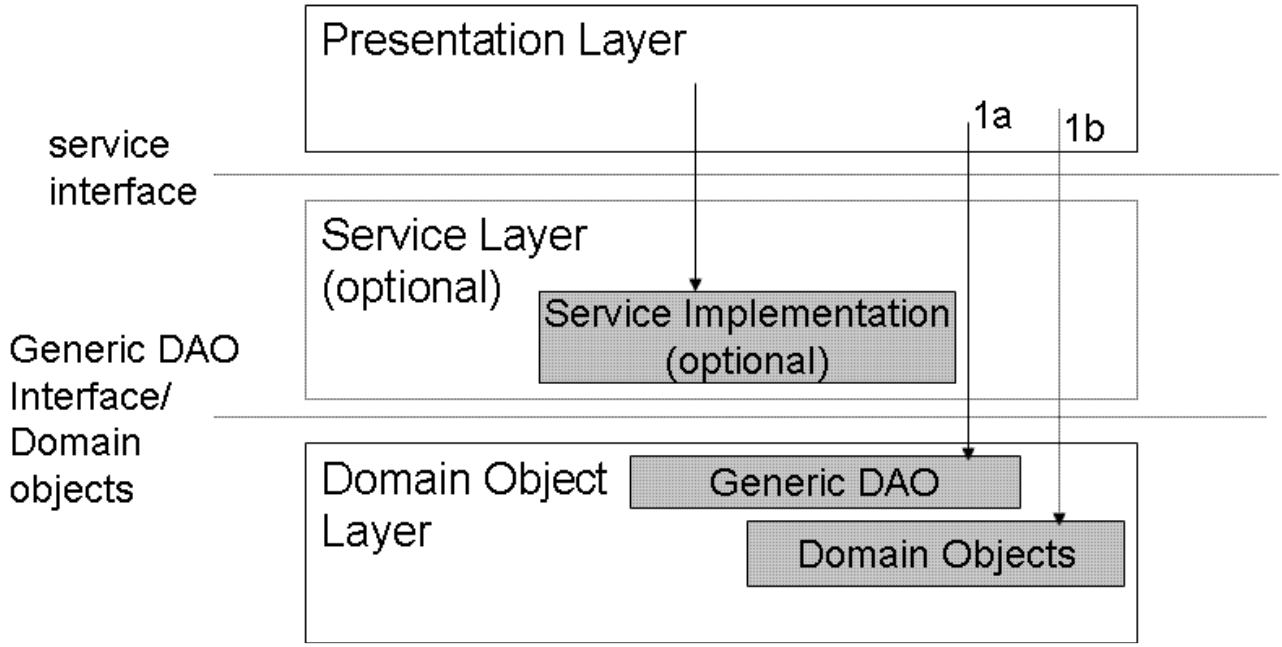
When writing DAOs (Data Access Objects), the basic CRUD (Create, Read, Update, Delete) operations tend to be more or less similar. The goal of the GenericDAO of EL4J is to eliminate these repetitive steps. The following class diagram illustrate this:



A user can use the ConvenienceGenericDao interface to access a database. It already provides the CRUD operations. The class GenericHibernateDao implements this interface for Hibernate (there is also an implementation for iBatis). When custom DAO-methods are required, they can be added in a subclass of the ConvenienceGenericDao (these custom methods are then implemented by hand). (It is also possible to use the canonical GenericDao, but we provide it mainly for internal use.)

6.2 Sometimes we can omit or bypass the service layer

The approach has the following reference architecture (the horizontal, dashed lines indicate the interfaces between layers):



We use the three "traditional" layers one uses typically. But there are some changes:

1. The Service Layer is *optional*. One can bypass the service layer in case the functionality is data-centered (CRUD-operations on a domain object model). The service layer may be useful for complex business functionality or for operations that involve many domain objects. It is also possible that one mainly works on the domain object model and only uses the service layer for a small part of more complex processings.
2. Instead of traditional DAO interfaces to find and store domain objects we use a *GenericDao* object (there is (at the moment) one GenericDao class per Entity). The standard find and store interface is generic (no hand-coding involved): There is no need to write the basic interface by hand. The latter also simplifies to bind (=attach) normal data access into the presentation layer. For particular functionality, one can *extend* the generic DAO interface.
3. The domain object model (normal POJOs) is annotated (with JDK 1.5 annotations) in order to add constraints/ additional data about the model only once (on the model). Sample contraints include: design by contract rules, validation information, mapping to database (JPA) or XML (JAXB), ... These annotations are then consumed by various tools that work with the domain object model (UI-validation, O/R-mapping, O/X-mapping, UI-rendering, ...). The domain object model is made up of *Entities* in the DDD terminology. The Entities in the domain object model are not just a "dumb" holder of data: it contains methods for "real" processings of the domain. When working on the domain layer, one either I) finds Entities via the GenericDao first and then works on these Entities (1a and 1b) and stores them again or II) one creates Entities (via new), works on them and stores them via the GenericDao.
   Please refer also to the Annotation Cheat Sheets.

6.3 Benefits of the approach

- Less duplication & cleaner code: you concentrate on the essential
- No code generation needed
- For the benefits of the DDD, we refer to the referenced book
- For data-centric applications you avoid to have a mostly delegating service layer
- Providing information such as how to validate objects on the domain model and having the domain model everywhere available helps to avoid duplication of such information.

6.4 References

- Where we have the original idea from http://www.hibernate.org/328.html
- A description of a similar implementation http://www-128.ibm.com/developerworks/java/library/j-genericdao.html
- Domain driven design (DDD) book: summary http://www.domaindrivendesign.org/

7 Documentation for module Swing

7.1 Purpose

The Swing module improves the Swing programming model with various little helper classes and a very light application framework.

7.2 Introduction

Creating GUIs is a task with a high amount of repetitive work. In addition to that, Swing itself can be tricky when trying to modify the GUI from a thread other than the EDT. Therefore, a collection of light frameworks, each addressing a different aspect of GUI programming, has been selected and integrated.

7.3 Features of the EL4J GUI framework

- Binding (connect POJO and GUI-Element to automatically synchronize content)
    - The binding is done by name (connect elements that are named the same way), by annotation or by explicit programmatic binding.
    - Change events are automatically tracked on the POJO (via a (optional) Mixin (see PropertyChangeListenerMixin))
    - By default, hibernate validations annotations on the POJO are evaluated to signal the user directly when he violates a validation constraint. This avoids that the validation constraints need to be implemented multiple times (in the GUI *and* on the POJO). You can also easily validate a POJO via the hibernate validator API.
    - Instead of using binding, the standard Swing model-approach can be used as well. Use whatever fits best.
    - In this framework, editing of gui-elements directly change the underlying POJOs. This leads to the question what to do if changes should only be applied if the user clicks "OK"? That's the way to do it:
        - Before letting the user edit the data, wrap the model object with the SaveRestoreMixin and call save(). If the user discards the changes call restore().
        - Keep in mind that only publicly accessible java bean properties get saved and restored! There is no deep copy.
        - Background info: This approach has been chosen to provide a simple but homogeneous way to store various kinds of model object. The different way of binding tables (using TableModels) compared to normal GUI elements like TextFields (which hold their state in the `text` property) made it impossible to implement this features by simply modifying the binding parameters. For TextFields, for example, one could have unbound the model during user interaction to prevent the model from getting updated. But that would have made validation much more difficult.
- Event bus (for simple implicit interconnection besides the normal binding)
    - We recommend to use the binding framework for normal GUI-Element POJO bindings. The event bus would then be for more "high-level" events. Each part of your application should document in the javadoc what events (=what classes) it sends out and what their semantic is.
    - Typically there are different classes that send and receive events via the event bus. In order to have the overview, we recommend to set up a global document (e.g. in an Excel-sheet or a HTML table) that describes all possible events (their classes, their senders and receivers, their semantics, conditions how they need to be handled (EDT or not), ...). For complex cases, even a global sequence diagram could clarify the situation.
    - Rationale: such documentation is rather light but its helps for debugging and newcomers.
- Exception handler framework (see Exceptions)
- MDI-support (see MDIApplication)
- Docking framework (see DockingApplication)
- I18n and resources (names, GIFs) defined external to the application (they are then injected into the application)

    - See NameOfClass.properties and resource bundles
- GUI session management: This automatically manages the user layout so that next time he launches the application this layout is restored (this includes table columns sizes and windows sizes and positions): how many Swing applications do that currently, and how many users are fed up of recreating their preferred layout every time? This can also be extended (to save other aspects of specific Components).
- Flexible @Action annotation to convert a method in an action
    - The basic functionality of this can be found in Sun's application framework
    - EL4J adds some mini-patterns to avoid that all Actions need to be defined in one class: see `addActionMappingInstance(Object)` and `Action getAction(String)` of class GUIApplication. In short, you split your Actions on n classes. In the `startup()` method of your central GUI class (the one you launch via `GUIApplication.launch`) you create instances of the other classes that also handle Actions and you add them to the list of instances to search for actions via `super.addActionMappingInstance(myNewActionImplementer)`.
    - How you could e.g. set up your application: in the central GUI class (the one extending GUIApplication, MDIApplication or DockingApplication) you just draw the top-level elements and put the general Actions (help, about, set global properties, ...). Then for each important concept X that your application treats, you create a new class called XActions. So e.g. if you have an application working on users, you could have a UserActions class that manages the currently active user and whenever one invokes an action (via a menu/ a hotkey or a button) the action on this class determines the current user and invokes the action. We recommend that UserAction extends AbstractBean in order for it to notify property changes (for the activation of Actions, see next point).
- Convenient activation of Actions in function of application state. This works via the `enabledProperty` field of the @Action annotation. Example: `@Action(enabledProperty = "admin") public void editPermissions() {...`
    - One can use the following pattern to make it simple to implement and use this. Lets say we have a boolean flag `admin` that we want to use to enable or disable certain actions. We could then write a method `boolean isAdmin()` and a method `void setAdmin(boolean newValue)` to update the value of `admin`. In the method `setAdmin` we update the value and fire a property change (via `firePropertyChange("admin", oldAdmin, newAdmin);`).
    - As a convenience, you can put the 2nd parameter of invocations to `firePropertyChange` to null. This avoids that you need to remember the old value of the property.
    - When you split Actions into multiple classes, the `isX()` method linked to in `enabledProperty="X"` must be in the same class as the Action method. (This is also why we recommend that your XAction extends AbstractBean.)
- Handle the GUI startup already according to the best practices (e.g. draw the GUI in the EDT)
- Helper class for easier setting up mnemonic texts (see MnemonicText class of Sun's application framework)
- Hints for GUI programming
    - To create forms easily there is a light LayoutManager called DesignGridLayout. We propose this layout manager to speed up the creation of dialogs and to have layouts with a professional touch. Under DesignGridLayout samples (starting from Simple Examples) you find a number of sample layouts to illustrate its simplicity. Please refer also to the demos for an example.
    - Convenience method to create menus rapidly (see `GUIApplication.createMenu`)
    - Access to the spring application context (not to confound with Sun's GUI frameworks own `org.jdesktop.applicationApplicationContext` abstraction!), already during application startup
    - Abbot can be used to test GUIs automatically.
- Support for long-running Tasks (see Task and TaskService classes) Now any time-consuming task can be started on a specific thread and feedback progress to the GUI without having to care about EDT). Task management has some interesting extension points that allow you to monitor pending tasks (e.g. in the status bar), to block the GUI while some tasks are going on.
- Exit handler with veto-able exit
- Reusable components

    - About Dialog
    - Splash Screen
    - IntegerField
    - A list of additional widgets and GUI hints: http://www.tutego.com/java/additional-java-swing-components.htm (but don't use JXTable for sorting as this doesn't work together with beans binding. See Master/Detail Demo how this can be done)
- Bugfixes for the integrated application framework
- Demo application that demonstrates most of these features

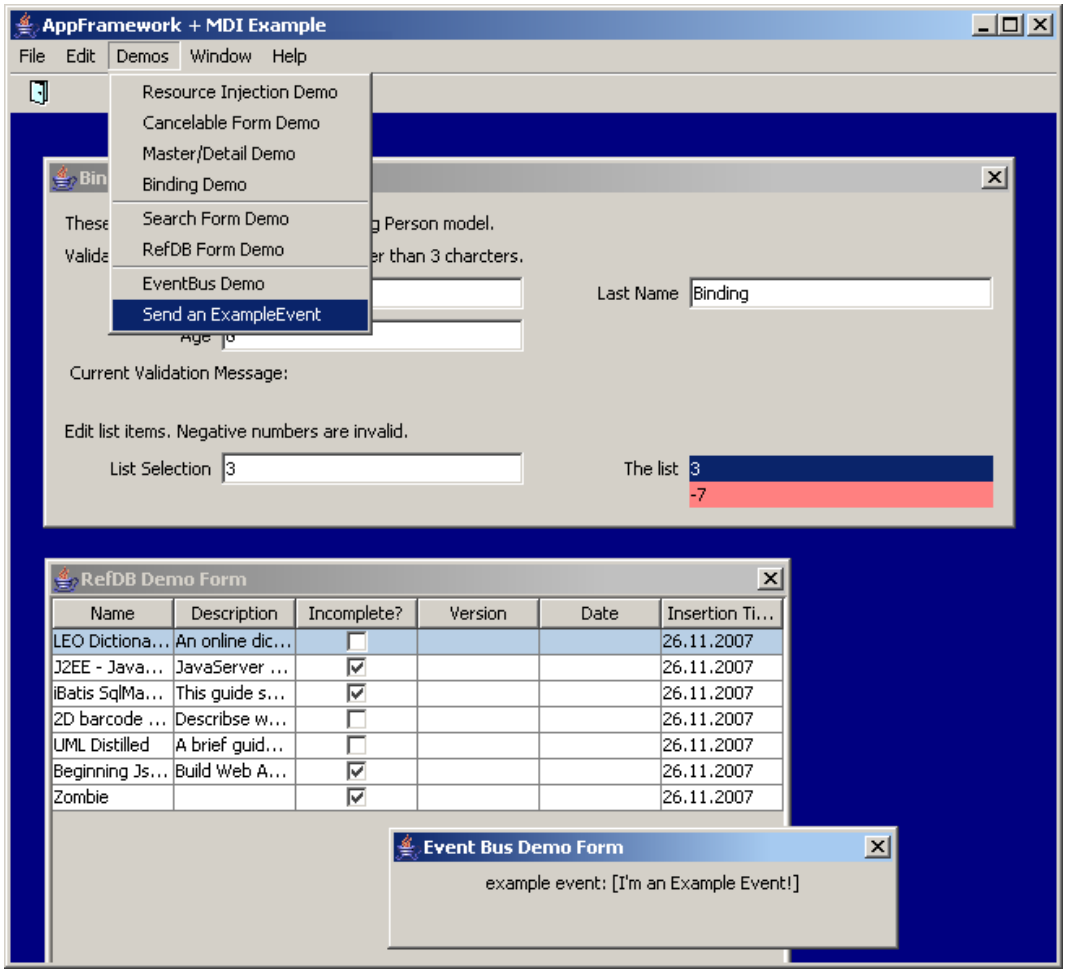
For more information on these features: (1) look through the integrated parts below, (2) check out the demo, and (3) ask us (SWI, POS).

7.4 How to get started with our demo application

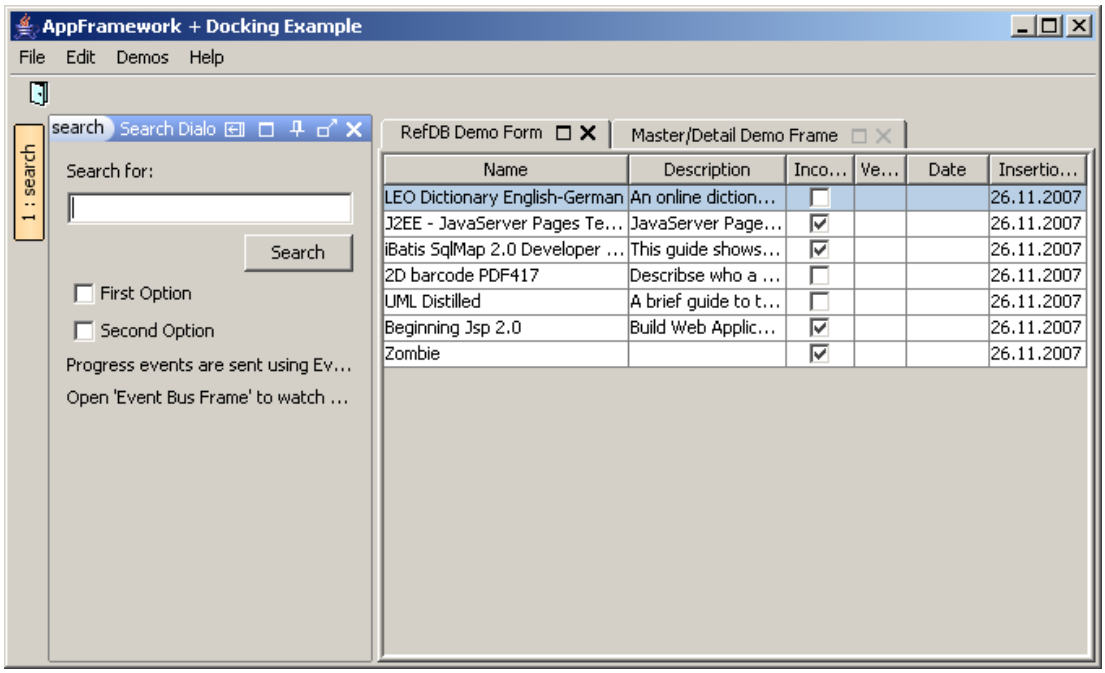
- Download el4j version 1.2 (or higher) or check out the latest el4j version from the svn repository (we recommend to download the el4j convenience zip and in d:/el4j do a `svn co https://el4j.svn.sourceforge.net/svnroot/el4j/trunk/el4j external`) (Alternatively, snapshot-versions/ releases of the components were also uploaded to the EL4J mvn repository.)
- The swing module can be found in `external/external/framework/modules/swing`
- The demo for the swing module can be found under `external/applications/templates/gui`. Follow the instructions in the contained `README.txt` file.
- Start eclipse, import the maven modules you are interested in, study the code and run it (the main program is in MainFormMDI for an MDI demo and MainFormDocking for a "docking" demo)

7.4.1 Demos

- Demo container using MDI (MainFormMDI)



- Demo container using docking framework (MainFormDocking)



- Resource Injection Demo (ResourceInjectionDemoForm)
  - The text of the main label is injected from the properties file. Few code, very easy...

- Cancelable Form Demo (CancelableDemoForm)
  - Shows the use of a special mixin that allows to add save/restore functionality to any bean. This enables Apply and Cancel buttons.
- Master/Detail Demo (MasterDetailDemoForm)
  - A table which is bound to a model. Furthermore the currently selected item can be edited either in the table or in the fields above the table. Theses field are manually bound to the table (see `masterDetail` variable). An other feature is that the table entries can be sorted.
- Binding Demo (BindingDemoForm)
  - Shows various types of bindings (`IntegerFields`, lists, custom validation)
- Search Form Demo (SearchDialog)
  - An example search dialog that uses eventbus (use EventBus Demo to make events visible)
- RefDB Form Demo (RefDBDemoForm and ReferenceEditorForm)
  - Shows how a generic service such as refDB can be integrated (including optimistic locking)
- EventBus Demo (EventBusDemoForm)
  - This frame show what events are generated
- Miscellaneous
  - In the MDI demo, if you click with the right mouse button on the background then a pop-up menu consisting of some menu items is shown
  - In the Help menu there is an about dialog that generally has to be configured only by the .properties file.
  - The menu item Help for Admins is disabled at startup. This is controlled by the `@Action(enabledProperty = "admin")` annotation. You can get admin by clicking on "Toggle admin rights".

7.5 Technologies used internally in the framework

Please refer to the references for more details on these technologies.

- AppFramework (https://appframework.dev.java.net)
  - A framework from sun to simplify Swing threading issues, long-running/non-blockable tasks and provide resource injection through properties files. This framework may be included in JDK 7.
- BeansBinding (https://beansbinding.dev.java.net)
  - A binding framework to simplify automatic binding of bean properties. We applied a patch to enable validation on tables.
- A modified version of Hitch Binding (http://hitch.silvermindsoftware.com) that uses BeansBinding
  - This enables to specify by annotations which form component should be bound to which bean property of the model. Like this much less BeansBinding code has to be written.
  - Its main part (which has not been modified) concerns the annotations.
- Hibernate Validation (http://validator.hibernate.org)
  - Using this validation framework, the annotated model can not only be used for validating the model while writing it to the database, but also to check during editing on the GUI and inform the user.
- eventBus (https://eventbus.dev.java.net)
  - A framework to send events, without having sender and subscriber to know each other.
- MyDoggy (http://mydoggy.sourceforge.net)
  - A Java docking framework to be used in cross-platform Swing applications.

7.6 Some input of JPO on GUI architecture

[We should define] a strategy to manage data: could be MVC, which I don't evangelize because it is always heavy, I rather prefer the notion of "Event Bus" to which any code can push events and any code can subscribe (I am biased: I have developed a library named "HiveEvents" that does just that and I use it heavily for my GUI apps). This generally simplifies coding. However, maintenance can be harder due to events passing around without always knowing where they come from and where they go (the only way to prevent maintenance problems is to document correctly events kinds, events sources and events consumers).

[The] application [framework should] ... avoid bad practices like: subclassing JDialog and add widgets to it (exclusively use your own JPanels subclasses instead), putting business inside the view...

7.7 TODOs

- Fast validation of single property (not the whole model)
- A extended table with Excel-like behavior
- Specialized Widgets: DoubleField, LimitedTextField (max ... chars), 3-state CheckBox...
- Binding support for trees (we wait until beans binding has tree support)
- Swing components having invalid values get a red (X), instead of just a red background

7.8 References

- AppFramework: https://appframework.dev.java.net
  - https://appframework.dev.java.net/intro/index.html
  - Pages 11 to 47 in http://conferences.oreillynet.com/presentations/os2007/os_haase.pdf
  - Another article http://java.sun.com/developer/technicalArticles/javase/swingappfr/
  - Older presentations can be found on the homepage

- Beans Binding: https://beansbinding.dev.java.net
  - At the moment only the blog shows version-1.0-examples.
  - A presentation on a previous version can be nevertheless interesting to get an impression: pages 49 to 74 in http://conferences.oreillynet.com/presentations/os2007/os_haase.pdf

- Hitch Binding: http://hitch.silvermindsoftware.com
  - The Tutorial is only valid up to Section "Make binder calls"
  - Binding is instead done using `BindingGroup group = binder.getAutoBinding(this); group.bind();`
  - The annotation reference can be found in chapter 3 of the manual

- Hibernate Validation: http://validator.hibernate.org
  - http://www.hibernate.org/hib_docs/validator/reference/en/html_single/

- eventBus: https://eventbus.dev.java.net
  - Study the simple example on the home page: https://eventbus.dev.java.net/)
  - The presentation takes a deeper look into it: https://eventbus.dev.java.net/HopOnTheEventBus-Web.ppt

- Standard Swing components: http://java.sun.com/docs/books/tutorial/ui/features/compWin.html
- FAQ on swing: http://www.chka.de/swing/
- GUI test framework (JPO has had positive experience with it): http://abbot.sourceforge.net/doc/overview.shtml
- EDT = event dispatching thread
- Nicer look and feels (laf): nimbus or substance
- A comparison of different layout managers: http://wiki.java.net/bin/view/Javadesktop/LayoutManagerShowdown?TWIKISID=b9e82416ed1c5adaee26bbfbc
- Java desktop community page: http://community.java.net/javadesktop/

8 Documentation for module web

8.1 Purpose

Web Module of EL4J. It includes struts, servlet-api, some commons libraries and a few own classes.

8.2 Features

The following features are included in this module:

- The ModuleWebApplicationContext. It decouples configuration location pattern interpretation form the current classloader.
- Implementation of the SynchronizerToken. This is useful for preventing duplicate form submissions. Further information under http://www.javaworld.com/javaworld/javatips/jw-javatip136.html. You can see an example of the Synchronizer Token in the OldWebApplicationTemplate.
- Xml Tag Transformer. Escapes xml tags in order to display them properly on web pages.

8.3 How to use

8.3.1 General configuration of the web module

The module web application context is used like its non-web counterpart (ModuleApplicationContext). For a sample usage of the other features, please refer to the web application template (the demo application).

8.4 Reference documentation for the Module-aware application contexts

The ModuleWebApplicationContext resolves issues that arise with web container class loaders. In contrast to standalone applications, web applications can't provide their classpath through a command line parameter or through environment parameters. The Servlet specification replaces the missing parameter with `Class-Path` entries in the `MANIFEST.MF`. Unfortunately, they're not respected by every servlet container.

The very same classloader issues appear also in environments other than web containers. The ModuleApplicationContext resolves absolutely the same problems using the same mechanisms. The following description applies to both application contexts.

8.4.1 Concept

Each jar from an EL4J module contains a manifest file with its module's name, its dependencies to other modules and a list of all configuration files it contains. The ModuleWebApplicationContext searches for all manifest files that are in the classpath, extracts their information and builds the complete module hierarchy. Then it creates a list of all provided configuration files, preserving the modules' hierarchy. The ordered list is used to fulfill any resource look-up queries.

In general, this resolves any problems with wildcard notation (e.g. `classpath*:mandatory/*.xml"`: it's guaranteed, that all mandatory files of a module A are loaded before them from module B, if B depends on A). Further, some classloaders have problems recognizing jar files as jars and instead show them as zip files. Spring's pattern resolvers work with jars only, running into troubles if a jar is wrongly taken for a zip. Since the pattern resolver used together with the ModuleWebApplicationContext works on the internal module structure only, there's no dependency on the current environment's classloader.

The ModuleWebApplicationContext can resolve only files that are added to the corresponding attribute in the manifest file. In general, this is just a subset of resources that are loaded during the application's lifecycle. Hence our custom pattern resolver that works on the internal module representation delegates each

unsatisfied request to the standard Spring resource loading mechanism.

So, this solution uses the same infrastructure as the one defined in the servlet specification. However, the processing is done by EL4J, and hence doesn't depend on any servlet container and their specific behavior.

8.4.1.1 ModuleDispatcherServlet

To simplify the usage of the ModuleWebApplicationContext, there's the ModuleDispatcherServlet that configures a Spring DispatcherServlet. It behaves absolutely the same as the one of Spring. Additionally, it allows defining two lists of configuration files which are included and excluded respectively.

Note: You don't have to use any of them. Spring's DispatcherServlet configuration style is still available.

Example configuration making use of the include / exclude feature:

```
    <servlet>
        <servlet-name>remotingtests</servlet-name>
        <servlet-class>
            ch.elca.el4j.web.context.ModuleDispatcherServlet
        </servlet-class>
        <load-on-startup>100</load-on-startup>
        <init-param>
            <param-name>inclusiveLocations</param-name>
            <param-value>WEB-INF/remotingtests-servlet.xml</param-value>
        </init-param>
        <init-param>
            <param-name>exclusiveLocations</param-name>
            <param-value>foobar.xml</param-value>
        </init-param>
    </servlet>
```

Without the need of the exclusive list (the standard DispatcherServlet's naming convention is used, hence the `benchmark-servlet.xml` gets loaded in this context):

```
    <servlet>
        <servlet-name>benchmark</servlet-name>
        <servlet-class>
            ch.elca.el4j.web.context.ModuleDispatcherServlet
        </servlet-class>
        <load-on-startup>100</load-on-startup>
    </servlet>
```

8.4.1.2 ModuleContextLoader

There is also the possibility to declaratively configure and start up the ModuleWebApplicationContext via servlet context parameters in the `web.xml` file. The `ch.elca.el4j.web.context.ModuleContextLoader` class provides this capability. This context loader is created by the `ch.elca.el4j.web.context.ModuleContextLoaderListener` class.

You have to configure the `ModuleContextLoaderListener` in the `web.xml` file as follows:

```
<listener>
    <listener-class>
        ch.elca.el4j.web.context.ModuleContextLoaderListener
    </listener-class>
</listener>
```

When the web application starts up, this listener will execute the `ModuleContextLoader`, which initializes and starts a `ModuleWebApplicationContext` based on one or more servlet context parameters that are merged. You can specify the following context parameters:

- inclusiveLocations (mandatory): specifies the configuration locations which will be included in the application context
- exclusiveLocations (optional, defaults to null): specifies the configuration locations which will be excluded from the application context
- overrideBeanDefinitions (optional, defaults to false): indicates whether bean definition overriding is allowed in the application context
- mergeResources (optional, defaults to true): indicates whether the resources retrieved by the configuration files section of the manifest files should be merged with resources found by searching in the file system

A typical configuration example could look like this:

```
<context-param>
    <param-name>inclusiveLocations</param-name>
    <param-value>
       classpath*:mandatory/*.xml,
        classpath*:scenarios/db/raw/*.xml,
        classpath*:scenarios/dataaccess/hibernate/*.xml,
        classpath*:optional/interception/transactionJava5Annotations.xml
    </param-value>
</context-param>

<context-param>
    <param-name>exclusiveLocations</param-name>
    <param-value>
       classpath*:scenarios/dataaccess/hibernate/keyword-core-repository-hibernate-config.xml
    </param-value>
</context-param>
```

8.4.2 Build system integration

We provide a hook task for the Maven build system that gathers all the needed configuration information automatically and writes them into the manifest file. In the default mode, it simply collects all files that are controlled by the resources plugin.

TBD: correct the above link to the resource plugin to the new maven concept for this. Check also if the content below is still valid.

8.4.2.1 Adding files manually

While adding files automatically to the manifest file is most of the time comfortable, it's sometimes necessary to specify the list of files manually.

TBD: how is this done with the Maven plugin?

Please check the javadoc of the ModuleApplicationContexts: there are some new features to control the loading of classpath resources.

8.4.3 Limitations

- Our custom resource pattern resolver handles wildcard classpath resources only, i.e. location patterns starting with `classpath*:` (with or without a wildcard pattern in the part following). Any other location pattern is resolved through delegation.
- Potentially, every response to a given request is incomplete, if answered by the custom module pattern resolver. The pattern resolver delegates unsatisfied requests only. Requests for which at least one resource is found are not handled by Spring's pattern resolver. Specifying configuration files manually may resolve the problem. See the earlier section on adding files manually for details.

8.4.4 `MANIFEST.MF` configuration section format

Of course, the manifest file can also be written by hand. Here is the format of the configuration section:

Add to the manifest of each module

1. the module's dependencies (`Dependencies`)
2. the list of all the configuration files it defines (`Files`)
3. the name of the module (actually the name of the jar file) (`Module`)

Example:

- 3 Modules: A,B,C
- B depend on A
- C depends on A

B contains b1.xml, b2.xml

C contains c.xml

A contains a.xml

The manifest of A contains

```
Name: config
Module: A
Files: a.xml
Dependencies:
```

The manifest of B contains

```
Name: config
Module: B
Files: b1.xml b2.xml
Dependencies: A
```

The manifest of C contains

```
Name: config
Module: C
Files: c.xml
Dependencies: A
```

8.4.5 Implementation Alternative: Idea

The resource pattern resolver delegates single-resource requests to one of Spring's pattern resolvers. That's because the configuration file list contained in a manifest file provides classpath-relative paths only. This paths could be made absolute using the manifest file's path as prefix. This would resolve problems with equally named resources. **Note**: loading all resources from the classpath using the `classpath*:` prefix requires top-down processing of the module hierarchy whereas loading of a single resource (i.e. using the `classpath:` prefix) bottom-up. Not sure if it works in all environments.

Example Manifest file location:

```
file:/C:/el4j/framework/lib/module-core_1.0.jar!/META-INF/MANIFEST.MF
```

Configuration files' prefix:

```
file:/C:/el4j/framework/lib/module-core_1.0.jar!/
```

8.4.6 Resources

- ModuleWebApplicationContextToDo specifies the problem more extensively
- ModuleWebApplicationContextToDoSpecification solution specification

9 Documentation for module Hibernate

9.1 Purpose

Convenience module for Hibernate.

9.2 How to use

Just include **classpath:scenarios/db/hibernateDatabase.xml** and create a new Spring config leaned on template for session factory bean.

You can then access Hibernate via the ConvenienceHibernateTemplate class. See keyword dao of Reference-Database-Application for example usage.

9.2.1 Criteria transformation

This module includes a CriteriaTransformer class which allows the transformation of EL4J criteria (described in ModuleCore) to hibernate criteria. This is useful if you want to use Hibernate to perform search queries which are based on a QueryObject object. See keyword dao of Reference-Database-Application for example usage.

9.2.2 Generic Hibernate repository

This module also contains the `ch.elca.el4j.services.persistence.dao.GenericHibernateRepository` class, a Hibernate-specific implementation of the `ch.elca.el4j.services.persistence.generic.dao.GenericRepository` interface.

9.2.3 Hibernate validation support

This module also contains support for bean validation. Bean validation is performed by specifying invariant constraints on the domain object model using the validation annotations defined by the Hibernate Validator, which is part of the Hibernate Annotations project. This is the way we recommend implementing validations on objects. The goal is to define the invariant constraints **only once** (on the domain object model), and to reuse them wherever the object is used.

The Hibernate Annotations reference documentation describes how validation constraints are implemented on the domain object model and how domain objects are validated.

Single bean properties can be validated using the pre-defined validation annotations of the Hibernate Annotations project, which check that bean properties respect different constraints (for example the `@NotNull` or the `@Range(min, max)` annotations). In addition to applying these built-in validation constraints, it is possible to perform *custom* bean validation. This can be achieved by adding a custom validation method to the domain class which will be validated and by annotating this method with the `@AssertTrue` annotation. It is the responsibility of this validation method to specify the custom validation constraints for the domain class in which it is defined. In such a method, it is for example possible to verify that different properties of a domain object are consistent between each other. The `@AssertTrue` annotation checks that this validation method evaluates to true, which should only be the case if all the custom constraints defined by that method evaluate to true.

The following example shows the usage of Hibernate's validation support and illustrates how to perform custom bean validation:

```
public class Reference {
```

```
    private Date m_documentDate;
    private Date m_whenInserted;

    @NotNull
    public Date getDocumentDate() {
        return m_documentDate;
    }

    @NotNull
    public Date getWhenInserted() {
        return m_whenInserted;
    }

    /**
     * Checks whether the reference is valid. Should always be true.
     * @return true if the reference is valid, false otherwise
     */
    @AssertTrue
    public boolean invariant() {
        return (getDocumentDate().getTime() <= getWhenInserted().getTime();
    }
}
```

In this example, we define validation constraints on the `Reference` domain object. We use the built-in `@NotNull` annotation to express that both the `documentDate` and the `whenInserted` properties must not be null. Custom validation, which can be used in conjunction with the other validation annotations, is implemented in the bean's `invariant()` method, which ensures that the reference's creation date is an earlier date than its insertion date.

The *validation* of a bean (i.e. the verification of the constraints) is performed as described in section 4.2 of the Hibernate Annotations reference documentation: verification can either take place at the database schema level, or via the Hibernate Validator's built-in Hibernate event listeners, or at the application level. This applies for both the built-in validation annotations and the custom validations defined in a validation method.

The `RefDB` demo application also contains an example illustrating bean validation. The `ch.elca.el4j.apps.refdb.dto.ReferenceDto` domain object is annotated in a similar way as the `Reference` bean in this documentation, and the `ch.elca.el4j.tests.refdb.ReferenceValidationTest` shows how to perform validation at the application level.

10 Documentation for module IBatis

10.1 Purpose

Convenience module for IBatis.

10.2 How to use

10.2.1 Dao layer

Just include location **classpath:scenarios/dataaccess/ibatisSqlMaps.xml** in spring config location and you have bean **convenienceSqlMapClientTemplate** available to have access to IBatis. See keyword dao of Reference-Database-Application for example usage.

10.2.2 Type handler callbacks

Type handler callbacks are used to convert database types to java types and vice versa.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE sqlMap
    PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="refdb-core">
    <typeAlias
        type="ch.elca.el4j.services.persistence.ibatis.callback.BlobToObjectTypeHandlerCallback"
        alias="blobHandler" />

    <resultMap id="file" class="ch.elca.el4j.apps.refdb.dto.FileDto">
        <result property="key" column="KEYID"/>
        <result property="keyToReference" column="KEYTOREFERENCE"/>
        <result property="name" column="NAME"/>
        <result property="mimeType" column="MIMETYPE"/>
        <result property="size" column="CONTENTSIZE"/>
        <result property="content" column="CONTENT" typeHandler="blobHandler"/>
        <result property="optimisticLockingVersion"
            column="OPTIMISTICLOCKINGVERSION"/>
    </resultMap>

    <statement id="getFileByKey" parameterClass="int" resultMap="file">
        select KEYID, KEYTOREFERENCE, NAME, MIMETYPE, CONTENTSIZE, CONTENT,
            OPTIMISTICLOCKINGVERSION from FILES where KEYID=#value#
    </statement>

    <update id="updateFile">
        update FILES set KEYTOREFERENCE=#keyToReference#, NAME=#name#,
            MIMETYPE=#mimeType#, CONTENTSIZE=#size#,
            CONTENT=#content,handler=blobHandler#,
            OPTIMISTICLOCKINGVERSION=OPTIMISTICLOCKINGVERSION+1
            where KEYID=#key#
                and OPTIMISTICLOCKINGVERSION=#optimisticLockingVersion#
    </update>
</sqlMap>
```

In example above we save a serializable java object in a blob and read/deserialize it as/to a java object.

Here the callback type classes:

java type	database type	type callback handler class

java.lang.String	CLOB	com.ibatis.sqlmap.engine.type.ClobTypeHandlerCallback
byte[]	BLOB	com.ibatis.sqlmap.engine.type.BlobTypeHandlerCallback
java.io.Serializable	BLOB	ch.elca.el4j.services.persistence.ibatis.callback.BlobToObjectTypeHandlerCallback

11 Documentation for module security

11.1 Purpose

The security module provides authentication and authorization for EL4J applications. The core part of this module is the Acegi Security System for Spring. Attribute-enabled interceptors are used to perform access controls.

11.2 Features

Besides the Acegi Security System library, this module contains an AuthenticationServiceContextPasser and an AuthenticationService which allows the user to transparently log in to a server and transparently invoke the server's methods. For a demonstration of this feature, please consult the module-security-tests.

11.3 How to use

Please refer to the demo application for now.

12 Documentation for module exception handling

12.1 Purpose

This module provides configurable exception handlers that allow separating the exception handling from the actual business logic. There are two exception handlers: a safety facade that handles technical exceptions for collections of POJOs and a context exception handler that allows handling exceptions in function of what context is active. These exceptions handlers complement the EL4J exception handling guidelines.

12.2 Important concepts

EL4J supports two frameworks to handle exceptions, the **Safety Facade** and the **Context Exception Handler**. Both handle exceptions of several beans and both use exactly the same core framework. The former is intended to be used nearby a service to handle implementation-specific and technical exceptions. Instead of handling such *abnormal* exceptions in the business code, the handling of abnormal exceptions is delegated to the safety facade. This simplifies the use of the wrapped service, as one can concentrate on its core functionality. In addition, it separates the concerns of its core business functionality and abnormal cases. The latter context exception handler is used to handle exceptions in different ways, depending on the current context (e.g. show errors in message boxes if in a gui context or log them into a file if in server context). Both exception handler frameworks can be used to build complex exception handler hierarchies consisting of different risk communities, as described in the ExceptionHandlingGuidelines.

Both exception handling frameworks are added to a project whenever they are needed. The handlers are configured in Spring configuration files, where you just change the names of the original beans and where you add new proxied versions of them. This still allows accessing the bare beans, without going through an exception handling facade, which is needed to build risk communities.

As already mentioned, the context exception handler handles exceptions according to the current context. It is set through a static method and is valid for the thread's whole life or until it's set to another value. It's considered a mistake if the context has not been set before the context exception handler treats an exception, hence a `MissingContextException` (unchecked) is thrown.

12.3 How to use

12.3.1 Configuration

Both the safety facade and the context exception handler are configured with a list of exception configurations. Each exception configuration associates a set of exceptions with its exception handler (see below for more details on exception handlers). The `ExceptionConfiguration` interface contains two methods, one for checking whether the configuration is able to handle the given situation, the other returns the configuration's exception handler. There are two default `ExceptionConfiguration` implementations:

- **ClassExceptionConfiguration** just checks the caught exception's type.
- **MethodNameExceptionConfiguration** checks the caught exception's type as well as the name of the method that threw it.

To configure a safety facade, one configures a list of exception configurations (please refer to the example below).

A context exception handler is configured with a map: The key of the map represents the context, the value the context's list of exception configurations (the format of the list of exception configurations (for each context) is the same as above).

12.3.1.1 Exception handlers

There are a number of exception handlers covering the most common cases:

- **RethrowExceptionHandler** forwards the exception to the caller.
- **SimpleLogExceptionHandler** logs the exception and the invocation description that raised it on trace level.
- **SimpleExceptionTransformerExceptionHandler** transforms the caught exception into the one specified by an exception class. The handler tries to fill it with the original exception's message, cause and stack trace.
- **SequenceExceptionHandler** allows declaring a list of exception handlers which are invoked one after another until one succeeds (=does not throw another exception). If all fail it returns the last caught exception.
- **RetryExceptionHandler** retries the very same invocation several times. The last caught exceptoin is rethrown if all retries didn't succeed.
- **RoundRobinSwappableTargetExceptionHandler** retries the invocation on different targets, that is exchanged each time the current one fails. The handler modifies the proxy too, let it use the exchanged target for new invocations. This allows automatically reconfiguring the system at runtime (e.g. change the data source if the current one isn't reachable anymore).

All of them extend the `AbstractExceptionHandler` that provides a logging abstraction which allows users to set whether proxy generated log messages are registered as if they're coming form the exception handler (default) or whether they are reported as if they're coming from the proxied class.

Additionally, there are a number of abstract handlers making it easier to build custom strategies. The **AbstractReconfigureExceptionHandler** for example helps to reconfigure a bean (e.g. making it use another collaborator).

12.3.1.2 Example 1: Safety Facade for one Bean

This is the simplest form of a safety facade: The actual bean is renamed, an exception configuration is provided and the safty facade is created. There are no changes in the Java code, as long as no unsafe bean access is needed.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="unsafeA" class="ch.elca.el4j.tests.services.exceptionhandler.A"/>

    <bean id="A" class="ch.elca.el4j.services.exceptionhandler.SafetyFacadeFactoryBean">
        <property name="target"><ref local="unsafeA"/></property>
        <property name="exceptionConfigurations">
            <list>
                <bean class="ch.elca.el4j.services.exceptionhandler.ClassExceptionConfiguration">
                    <property name="exceptionTypes">
                        <list>
                            <value>java.lang.ArithmeticException</value>
                        </list>
                    </property>
                    <property name="exceptionHandler">
                        <bean class="ch.elca.el4j.services.exceptionhandler.handler.SimpleLogExceptionHan
                            <property name="useDynamicLogger"><value>true</value></property>
                        </bean>
                    </property>
                </bean>
            </list>
        </property>
    </bean>
</beans>
```

12.3.1.3 Example 2: Context Exception Handler

The context exception handler is initialized the same way as the safety facade. However, there is an additional indirection (the map) to setup different policies for each context.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="unsafeA" class="ch.elca.el4j.tests.services.exceptionhandler.AImpl"/>

    <bean id="A" class="ch.elca.el4j.services.exceptionhandler.ContextExceptionHandlerFactoryBean">
        <property name="target"><ref local="unsafeA"/></property>
        <property name="policies">
            <map>
                <entry key="gui">
                    <list>
                        <bean class="ch.elca.el4j.services.exceptionhandler.ClassExceptionConfiguration">
                            <property name="exceptionTypes">
                                <list>
                                    <value>java.lang.ArithmeticException</value>
                                </list>
                            </property>
                            <property name="exceptionHandler">
                                <bean class="ch.elca.el4j.tests.services.exceptionhandler.MessageBoxExcep
                            </property>
                        </bean>
                    </list>
                </entry>

                <entry key="batch">
                    <list>
                        <bean class="ch.elca.el4j.services.exceptionhandler.ClassExceptionConfiguration">
                            <property name="exceptionTypes">
                                <list>
                                    <value>java.lang.ArithmeticException</value>
                                </list>
                            </property>
                            <property name="exceptionHandler">
                                <bean class="ch.elca.el4j.tests.services.exceptionhandler.LogExceptonHand
                                    <property name="useDynamicLogger"><value>true</value></property>
                                </bean>
                            </property>
                        </bean>
                    </list>
                </entry>
            </map>
        </property>
    </bean>
</beans>
```

Corresponding Java snippet (**Note**: set the context to a valid value. Otherwise, a `MissingContextException` (unchecked) is thrown.):

```
A m_a = getA();
ContextExceptionHandlerInterceptor.setContext("gui"); // set the current thread's context
m_a.div(1, 0); // handles any exceptions using the gui policy
ContextExceptionHandlerInterceptor.setContext("batch"); // set the current thread's context
m_a.div(1, 0); // handles any exceptions using the batch policy
m_a.div(1, 0); // handles any exceptions using the batch policy
```

12.3.1.4 Example 3: RoundRobinSwappableTargetExceptionHandler

This example shows the round robin swappable target exception handler. **Note** the handler requires a `HotSwappableTargetSource` in order to reconfigure the proxy.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans">
    <bean id="unsafeA" class="ch.elca.el4j.tests.services.exceptionhandler.A"/>
    <bean id="B" class="ch.elca.el4j.tests.services.exceptionhandler.B"/>

    <bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
        <constructor-arg><ref local="unsafeA"/></constructor-arg>
    </bean>

    <bean id="A" class="ch.elca.el4j.services.exceptionhandler.SafetyFacadeFactoryBean">
        <property name="target"><ref local="swapper"/></property>
        <property name="exceptionConfigurations">
            <list>
                <bean class="ch.elca.el4j.services.exceptionhandler.MethodNameExceptionConfiguration">
                    <property name="methodNames">
                        <list>
                            <value>concat</value>
                        </list>
                    </property>
                    <property name="exceptionTypes">
                        <list>
                            <value>java.lang.UnsupportedOperationException</value>
                        </list>
                    </property>
                    <property name="exceptionHandler">
                        <bean class="ch.elca.el4j.services.exceptionhandler.handler.RoundRobinSwappableTa
                            <property name="swapper"><ref local="swapper"/></property>
                            <property name="targets">
                                <list>
                                    <ref local="unsafeA"/>
                                    <ref local="B"/>
                                </list>
                            </property>
                        </bean>
                    </property>
                </bean>
            </list>
        </property>
    </bean>
</beans>
```

Using a `ProxyFactoryBean` and an explicit interceptor to do the same work as the `SafetyFacadeFactoryBean` above would look like this

```
<bean name="safetyFacade" class="ch.elca.el4j.services.exceptionhandler.SafetyFacadeInterceptor">
    <property name="exceptionConfigurations">
        <list>
            <ref local="roundRobinSwappableTargetExceptionConfiguration"/>
        </list>
    </property>
</bean>

<bean id="A" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="targetSource"><ref local="swapper"/></property>
    <property name="interceptorNames">
        <list>
            <idref bean="safetyFacade"/>
        </list>
    </property>
</bean>
```

12.3.1.5 Example 4: Using several exception handlers, each configured by a separate exception configuration

Several exception handlers are configured by multiple exception configurations.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="unsafeA" class="ch.elca.el4j.tests.services.exceptionhandler.AImpl"/>

    <bean id="A" class="ch.elca.el4j.services.exceptionhandler.SafetyFacadeFactoryBean">
        <property name="target"><ref local="unsafeA"/></property>
        <property name="exceptionConfigurations">
            <list>
                <bean class="ch.elca.el4j.services.exceptionhandler.ClassExceptionConfiguration">
                    <property name="exceptionTypes">
                        <list>
                            <value>java.lang.ArithmeticException</value>
                        </list>
                    </property>
                    <property name="exceptionHandler">
                        <bean class="ch.elca.el4j.services.exceptionhandler.handler.SequenceExceptionHand
                            <property name="exceptionHandlers">
                                <list>
                                    <bean class="ch.elca.el4j.services.exceptionhandler.handler.SimpleLog
                                        <property name="useDynamicLogger"><value>true</value></property>
                                    </bean>

                                    <bean class="ch.elca.el4j.services.exceptionhandler.handler.RetryExce
                                        <property name="retries"><value>5</value></property>
                                        <property name="sleepMillis"><value>0</value></property>
                                        <property name="useDynamicLogger"><value>true</value></property>
                                    </bean>

                                    <bean class="ch.elca.el4j.services.exceptionhandler.handler.SimpleExc
                                        <property name="transformedExceptionClass">
                                            <value>java.lang.RuntimeException</value>
                                        </property>
                                    </bean>
                                </list>
                            </property>
                        </bean>
                    </property>
                </bean>

                <bean class="ch.elca.el4j.services.exceptionhandler.ClassExceptionConfiguration">
                    <property name="exceptionTypes">
                        <list>
                            <value>java.lang.UnsupportedOperationException</value>
                        </list>
                    </property>
                    <property name="exceptionHandler">
                        <bean class="ch.elca.el4j.tests.services.exceptionhandler.ReconfigureExceptionHan
                            <property name="c"><ref local="C"/></property>
                        </bean>
                    </property>
                </bean>
            </list>
        </property>
    </bean>
</beans>
```

12.4 References

1. Moderne Softwarearchitektur -- Umsichtig planen, robust bauen mit Quasar, Johannes Siedersleben, dpunkt.verlag, 2004, ISBN 3-89864-292-5

12.5 Internal design

Each secured bean is hidden behind a proxy that wraps each invocation into a try-catch block. If the invocation that is delegated to the bare bean throws an exception, the facade looks up an appropriate exception handler and delegates the handling to it. The interceptors are instantiated with one of the two convenience factories, the `ch.elca.el4j.services.exceptionhandler.SafetyFacadeFactoryBean` and the `ch.elca.el4j.services.exceptionhandler.ContextExceptionHandlerFactoryBean`. These two factories create the interceptor transparently. They extend Spring's `AdvisedSupport` and simply add the exception handling interceptor. Using the `ProxyFactoryBean` provides access to the interceptor and allows adding additional interceptors (however this is not recommended since the exception handler interceptor should wrap the complete unsafe bean).

Important Although it's possible to use Spring's auto proxy features, it's not recommended because it hides the unsafe bean, making it impossible to build risk communities.

There are three common properties, independent of whether you use a safety facade or a context exception handler and independent from the way you create the proxies (convenience factory or ProxyFactoryBean):

Property	Description	Default value	
		Safety Facade	Context Exception Handler
defaultBehaviourConsume	`true` consumes any exceptions that are **not** handled by an exception handler. `false` rethrows unhandled exceptions to the caller.	`true`	`true`
forwardSignatureExceptions	`true` forwards any exceptions which are defined in the invoked method's signature. `false` forces to handle these exceptions by the handlers too.	`true`	`true`
handleRTSignatureExceptions	Declares whether unchecked exceptions that are listed in a method's signature should go through an exception handler or whether they are forwarded to the caller. `true` for handle, `false` for forward.	`true`	`true`

12.5.1 Context Exception Handler

The `ContextExceptionHandlerInterceptor` uses a `ThreadLocal` to store the current context. There's no mechanism that resets the context transparently, preventing pooled threads to use a wrong context. **Setting the appropriate context is the programmer's responsibility.**

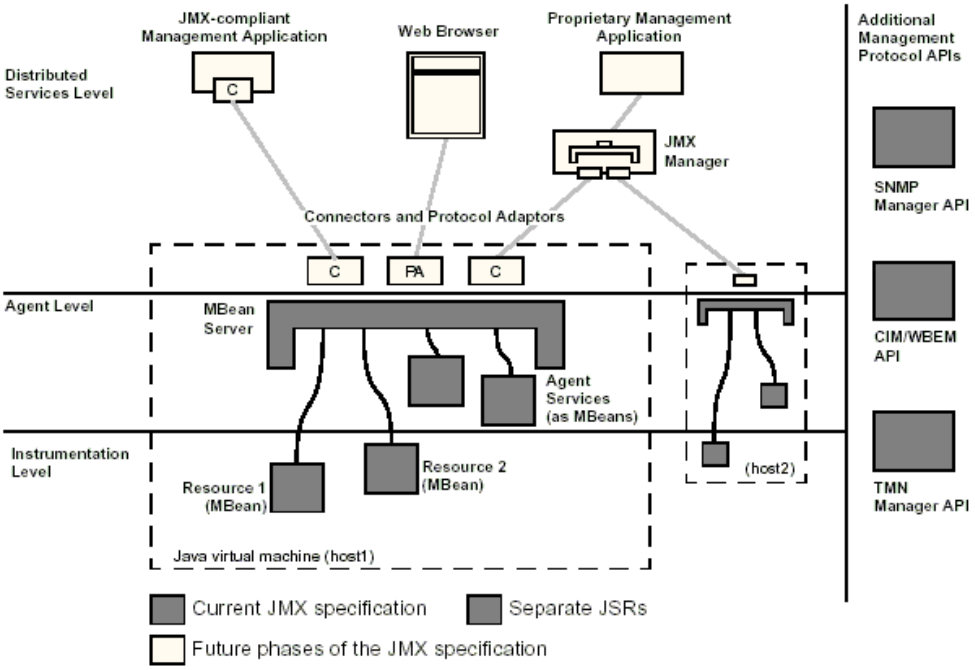
13 Documentation for module JMX

13.1 Purpose

The module **jmx** supports developpers in understanding spring applications by providing an automatic (implicit) view of the currently loaded spring beans and their configurations. This becomes even more interesting as Spring (and EL4J) allow splitting configuration information in many files, making it sometimes hard to figure out what config applies. For the impatient: JmxModuleForTheImpatient

13.2 Introduction to Java management eXtensions (JMX)

The follwing picture shows the components of JMX:



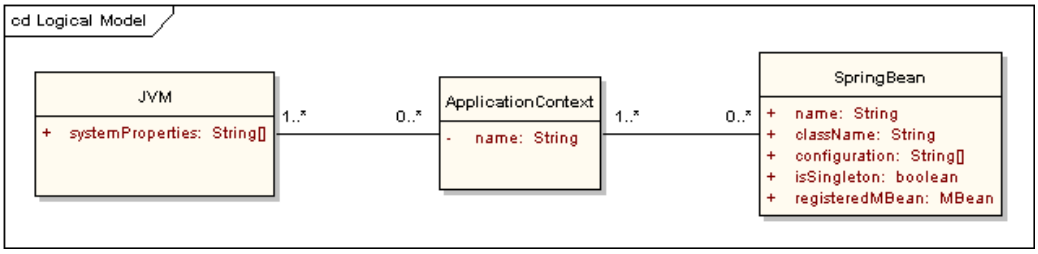
The central part of JMX is the *MBean Server*. *Managed Beans*, special Java objects that a developper wants to have controlled during runtime, are registered at the MBean Server. These Manged Beans or Mbeans are typically proxies for other components in the JVM one wants to monitor. These MBeans can be manipulated via JMX at runtime, i.e. their attributes can be read and edited and their methods can be invoked. Finally there are connectors that allows to access MBeans from remote.

13.3 Feature overview

We provide 2 ways to publish Spring Mbeans to JMX:

- Implicit publishing: publish all spring beans and their config automatically (we use the ModuleApplicationContext for this)
- Explicit publishing (as Spring provides it normally)

The following class diagram illustrates the mbeans we publish implicitly:



Besides all the spring beans, the **jmx** package also creates a JVM proxy in order to display the system properties etc. Furthermore, each ApplicationContext will be mirrored by a proxy that also provides links to all the loaded beans in it.

13.4 Usage

13.4.1 Spring/JDK versioning issue

The usage of the module depends on the used Spring and JDK versions. By default the module works with Spring 1.2 and JDK 1.4.2.

13.4.1.1 Spring versions 1.1 <-> 1.2

Spring supports JMX since version 1.2. If you are using Spring 1.1, you have to include a library with the missing files. This can be done by adding the following dependency in the `module.xml` file of the JMX module:

```
<dependency jar="spring-jmx-1.1.4.jar"/>
```

Difference in module-jmx:

Refactoring of `org.springframework.jmx.JmxMBeanAdapter` (Spring 1.1 extension) into `org.springframework.jmx.export.MBeanExporter` (Spring 1.2). Therefore you have to adapt the `beans.xml` as is described at Editing an MBean by replacing the corresponding class name. Everything else remains unchanged.

13.4.1.2 JDK versions 1.4.2 <-> 1.5

Since JDK 1.5, JMX is supported. If you are using JDK 1.5, you have to exclude the following 4 libraries in the `module.xml`, i.e. deleting these lines.

```
<dependency jar="jmxremote-1.4.2.jar"/>
<dependency jar="jmxremote_optional-1.4.2.jar"/>
<dependency jar="jmxri-1.4.2.jar"/>
<dependency jar="jmxtools-1.4.2.jar"/>
```

There is no difference in using the JMX module.

13.4.2 Basic Configuration (implict publication)

The **JMX** package has to be included in the build path of your project which can be achieved by setting a dependency in your project to the module-jmx. First of all you need the `jmx.xml` which you can find at `mandatory/jmx.xml`. If you load the Application Context with one of your config locations equal to `classpath*:mandatory/*.xml`, then `jmx.xml` is loaded.

Here is a possible configuration file `jmx.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
```

```
<beans>
   <bean id="mBeanServer" class="ch.elca.el4j.services.monitoring.jmx.MBeanServerFactoryBean">
      <property name="defaultDomain">
         <value>foobar1</value>
      </property>
   </bean>
   <bean id="jmxLoader" class="ch.elca.el4j.services.monitoring.jmx.Loader">
      <property name="server">
         <ref bean="mBeanServer"/>
      </property>
   </bean>
</beans>
```

- The bean `mBeanServer` creates a MBeanServer on the defined `defaultDomain`. Since the MBeanServerFactoryBean ensures that there is only one MBeanServer per domain, you can register as many ApplicationContexts at the same MBeanServer as you want, or easily assign each ApplicationContext a different MBeanServer by defining an unique domain for each MBeanServer. If you do not define this property, the MBeanServer on the domain "defaultDomain" will be taken.
- The bean `jmxLoader` defines the loader which is responsible for setting up the whole JMX world.

13.4.3 Connector

If you want to use **JMX** in a project, then you have to define what kind of connector you want to set up. At the moment, EL4J provides a HtmlAdapter and a JMXConnector.

13.4.3.1 HtmlAdapter

The bean `htmlAdapter` is a HTTP connector that allows observing the MBean Server of the property `mbeanServer`. **This adapter is installed by default**. The page can be called with `http://localhost:9092`. If no `port` is defined, the default one is 9092. The Html Adapter is defined as follows:

```
<bean id="htmlAdapter" class="ch.elca.el4j.jmx.HtmlAdapterFactoryBean">
   <property name="mbeanServer">
      <ref bean="mBeanServer"/>
   </property>
   <property name="port">
      <value>9092</value>
   </property>
</bean>
```

13.4.3.2 JmxConnector

The bean `jmxConnector` is a JMX connector based on the JSR-160 jmxmp protocol. Any client tool able to handle this protocol can be used to work with this MBeans. One such tool is MC4J. The bean definition provided is the following:

```
<bean id="jmxConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean">
   <property name="server">
      <ref bean="mBeanServer"/>
   </property>
   <!-- This is the default URL anyway -->
   <property name="serviceUrl">
      <value>service:jmx:jmxmp://localhost:9876</value>
   </property>
</bean>
```

Note: This class is only available as of Spring 1.2. This connector is optional (is it in the optional conf directory).

13.4.4 Example with one ApplicationContext

Here is a possible Test Class that uses **jmx**:

```
public class TestClass {

    public static void main(String[] args) {

        ApplicationContext ac = new ClassPathXmlApplicationContext(new String[] {"classpath*:mandatory/*.

        System.out.println("Waiting forever...");
        try {
            Thread.sleep(Long.MAX_VALUE);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

13.4.5 Configuration (explicit publication)

If you want to edit fields or invoke operations of a spring bean, e.g. on the bean `Foo1`, then you have to explicitly register it via `mBeanExporter`. The automatically created proxies for Spring beans do not allow editing their fields. The `beans.xml` configuration file could look like this:

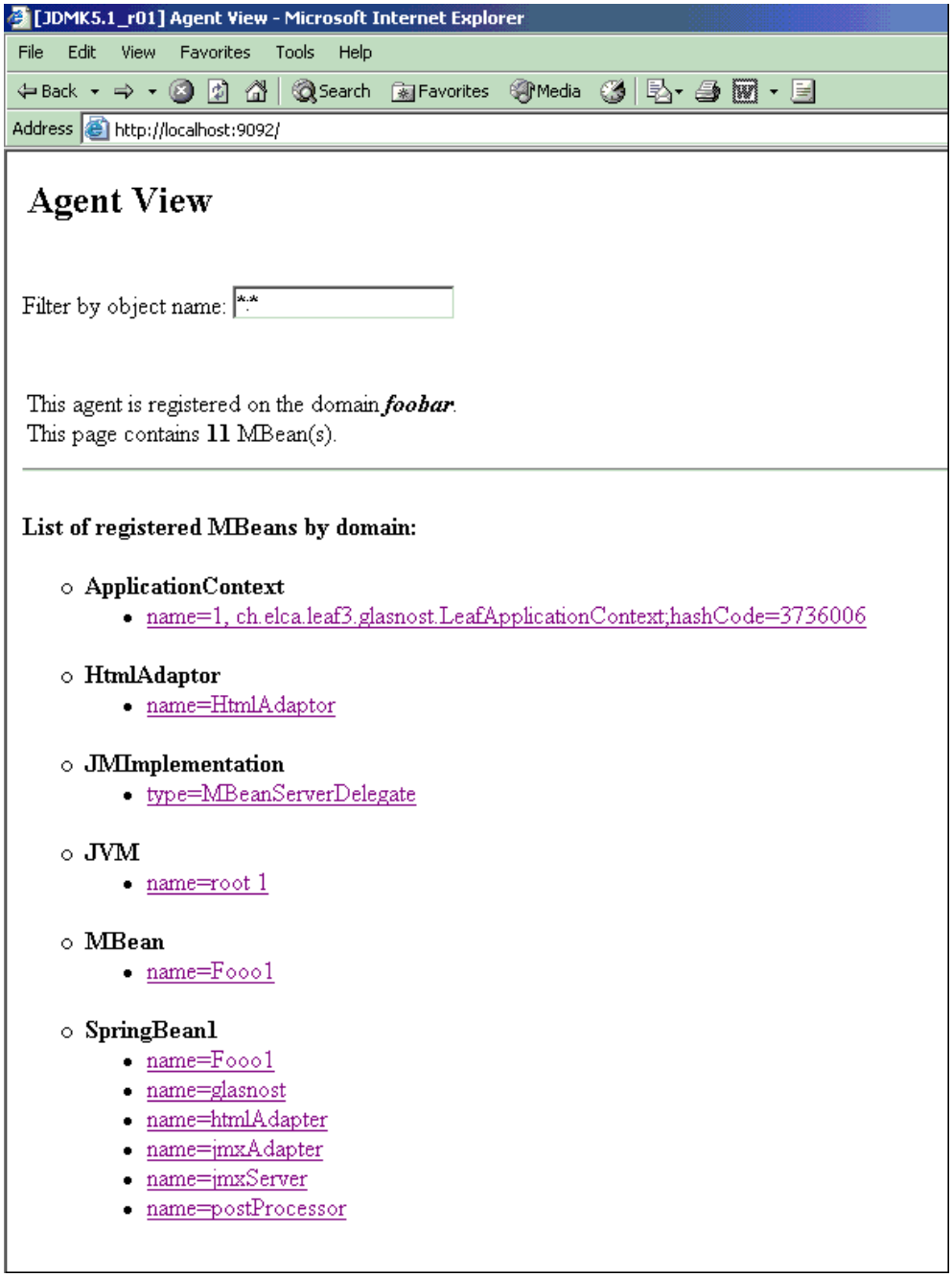
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
   <import resource="classpath:optional/htmlAdapter.xml"/>
   <bean id="mBeanExporter" class="org.springframework.jmx.export.MBeanExporter"
      depends-on="mBeanServer">
      <property name="beans">
         <map>
            <entry key="MBean:name=Foo1">
               <ref bean="Foo1"/>
            </entry>
         </map>
      </property>
      <property name="server">
         <ref bean="mBeanServer"/>
      </property>
   </bean>
   <bean id="Foo1" class="ch.elca.el4j.test.Foo1">
      <property name="fullName">
         <value>foo</value>
      </property>
   </bean>
</beans>
```

In the bean `mBeanExporter` you can register which beans you want to expose as MBeans, i.e. you want to be able to modify. In this example, bean `Foo1` will be exposed. The property `server` of `mBeanExporter` has to be set to the `mBeanServer` bean (which is loaded via `classpath*:mandatory/jmx.xml`). You can define as many `mBeanExporters` as you want, but do not forget to give each `mBeanExporter` bean in the same ApplicationContext a different name.

Important: The following Naming Convention has to be preserved regarding the property `beans`: The key entry has to be "MBean:name="+ `beanName`. Each SpringBean contains a link to its MBean if there is one.

By directing your browser to `http://localhost:9092`, you get the following view:



As you can see, the **module-jmx** created a proxy bean called "SpringBeanX:name=beanName" where X is a static counter. In the domain "MBean", you can find all the MBeans that you have registered via `jmxAdapter`, which is now called `mBeanExporter`.

13.4.6 Example with more than one ApplicationContext

If more than one ApplicationContext is loaded, then you have two possibilities:

- If you want to register another ApplicationContext at the same `mBeanServer`, then you have to choose the same `defaultDomain` as the other Application Context since the domain of the MBean Server actually defines the MBean Server.
- If you want to register another ApplicationContext at a different `mBeanServer`, then you have to follow these two steps:

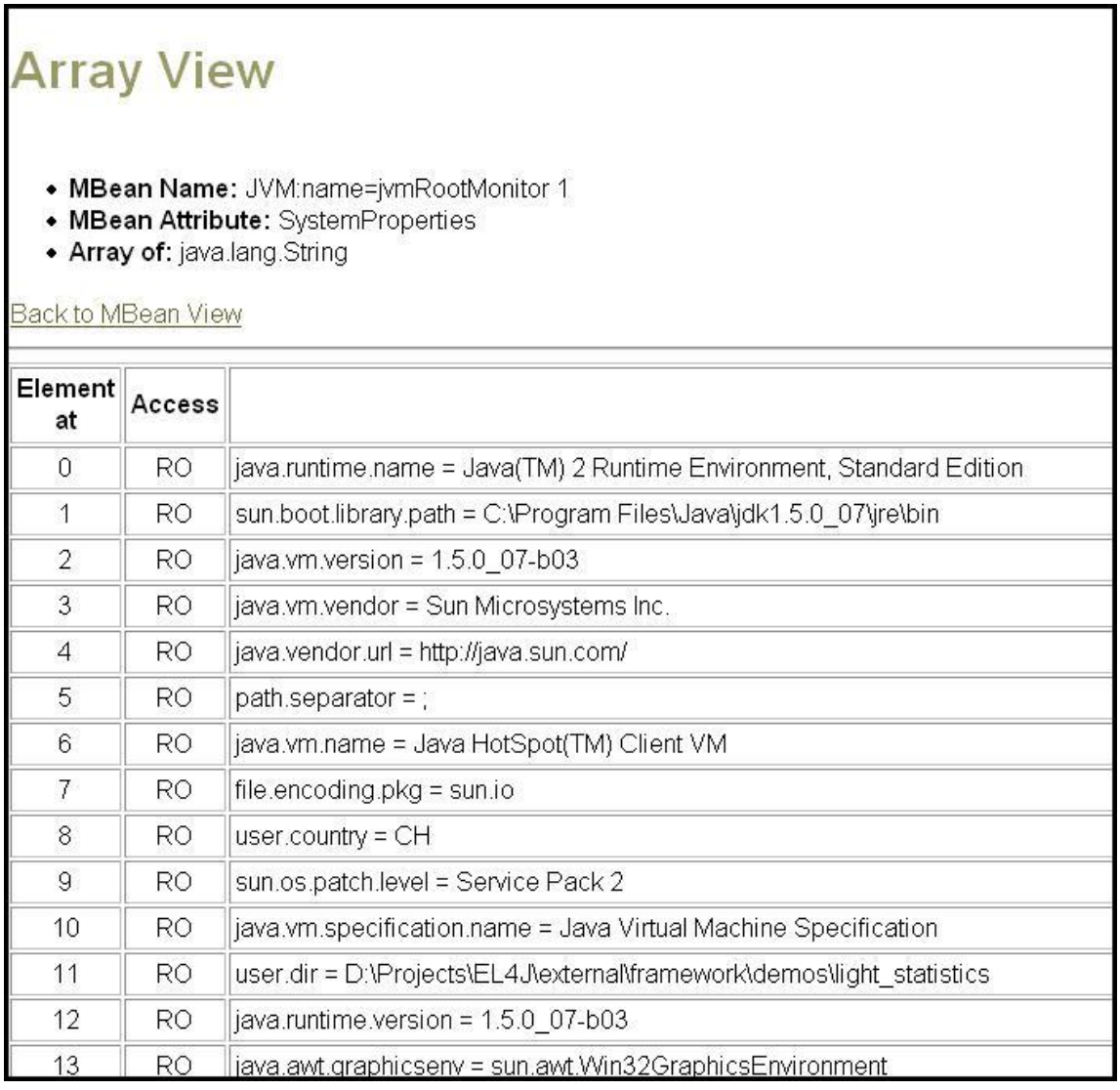
- Override the `defaultDomain` property of the `mBeanServer` bean by the `org.springframework.beans.factory.config.PropertyOverrideConfigurer` for example with the entry `mBeanServer.defaultDomain=foobar2`.
- The connector tho this MBean Server has to use a non-used port, e.g. `9093`.

13.5 Implemented Features

There are a lot of MBeans already implemended and published. Here only a few examples are shown. The best way to find out more about the implemented beans is to browse yourself through them!

13.5.1 JVM-Monitor

The JVM-Monitor MBean is published under the domain 'JVM'. It contains important information about the current JVM, such as which application context is loaded, values of the system properties and properties of the currently running threads (see screen shots below).

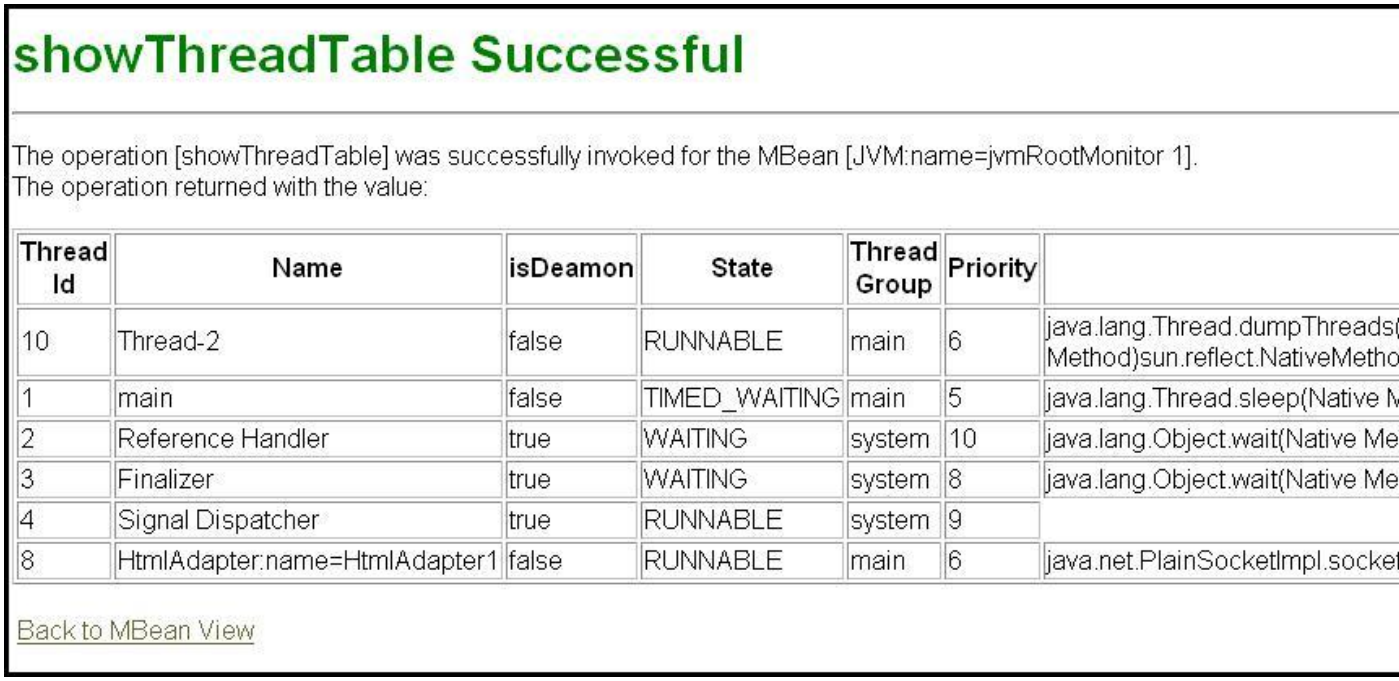


Array View

- **MBean Name:** JVM:name=jvmRootMonitor 1
- **MBean Attribute:** SystemProperties
- **Array of:** java.lang.String

Back to MBean View

Element at	Access	
0	RO	java.runtime.name = Java(TM) 2 Runtime Environment, Standard Edition
1	RO	sun.boot.library.path = C:\Program Files\Java\jdk1.5.0_07\jre\bin
2	RO	java.vm.version = 1.5.0_07-b03
3	RO	java.vm.vendor = Sun Microsystems Inc.
4	RO	java.vendor.url = http://java.sun.com/
5	RO	path.separator = ;
6	RO	java.vm.name = Java HotSpot(TM) Client VM
7	RO	file.encoding.pkg = sun.io
8	RO	user.country = CH
9	RO	sun.os.patch.level = Service Pack 2
10	RO	java.vm.specification.name = Java Virtual Machine Specification
11	RO	user.dir = D:\Projects\EL4J\external\framework\demos\light_statistics
12	RO	java.runtime.version = 1.5.0_07-b03
13	RO	java.awt.graphicsenv = sun.awt.Win32GraphicsEnvironment



showThreadTable Successful

The operation [showThreadTable] was successfully invoked for the MBean [JVM:name=jvmRootMonitor 1].
The operation returned with the value:

Thread Id	Name	isDeamon	State	Thread Group	Priority	
10	Thread-2	false	RUNNABLE	main	6	java.lang.Thread.dumpThreads(Method)sun.reflect.NativeMetho
1	main	false	TIMED_WAITING	main	5	java.lang.Thread.sleep(Native M
2	Reference Handler	true	WAITING	system	10	java.lang.Object.wait(Native Me
3	Finalizer	true	WAITING	system	8	java.lang.Object.wait(Native Me
4	Signal Dispatcher	true	RUNNABLE	system	9	
8	HtmlAdapter:name=HtmlAdapter1	false	RUNNABLE	main	6	java.net.PlainSocketImpl.socket

Back to MBean View

13.5.2 Log4jConfig

The Log4jConfig MBean is published under the domain 'JVM'. It shows information about the loaded loggers. Furthermore it allows change of the level of loggers. Furthermore it can also generate XML configuration code of logger level changes made during a session (see screen shots below).

List of MBean attributes:

Name	Type	Access	
Name	java.lang.String	RO	log4j
RootLoggerLevel	java.lang.String	RW	WARN

Apply

List of MBean operations:

Description of showLogLevelCache
java.lang.String showLogLevelCache

Description of showAppenders
[Lorg.apache.log4j.Appender; showAppenders (java.lang.String)p1

Description of showLogLevel
org.apache.log4j.Level showLogLevel (java.lang.String)p1

Description of changeLogLevel
void changeLogLevel (java.lang.String)p1
(java.lang.String)p2

Description of showAvailableAppendersList
[Ljava.lang.String; showAvailableAppendersList

showLogLevelCache Successful

The operation [showLogLevelCache] was successfully invoked for the MBean [JVM:name=log4jConfig].
The operation returned with the value:

```
<logger name="ch.elca.el4j.demos.statistics.light">
        <level value="DEBUG"/>
</logger>

<root>
        <level value="WARN"/>
</root>
```

Back to MBean View

Some available features:

- The 'changeLogLevel(category , level)' method allows to change the log level of a certain category logger. To see the log level of a category, the method 'showLogLevel(category)' can be used.
- The 'RootLoggerLevel' property allows to change the level of the root logger.
- The 'showLogLevelCache' method returns an XML string, which represents all the logger level changes made through the methods 'changeLogLevel' or property 'RootLoggerLevel' to the logger levels. The output string is suitable for copy-pasting into a Log4j.xml configuration file (see output in screen shot above).
- Normally appenders to loggers are specified in the Log4j.xml file. But sometimes its quite handy to be able to add/remove appenders for certain loggers, without having to shutdown the application. To enable this functionality, four methods are implemented: The 'AvailableAppendersList' property shows a list of appenders (appenderName and reference to appenderObject), which are available for attachment to a logger. The 'addAppender(category , appenderName)' method allows to attach an appender (which is listed in the 'AvailableAppendersList' property), to a logger category. The 'removeAppender(category , appenderName)' disattaches an appender from a logger category. For seeing, which appenders are connected to a logger the method 'showAppenders(category)' can be used. The appenders which are available ('AvailableAppendersList' property), are loaded from a bean with the name 'log4jJmxLoader' during the initialization of the application. The 'log4jJmxLoader' bean and appenders can be instanciated as follows:

```
<bean id="log4jJmxLoader" class="ch.elca.el4j.services.monitoring.jmx.Log4jJmxLoader">

   <property name="appenders">
      <map>
         <entry>
            <key>
               <value>nullAppender</value>
            </key>
            <ref bean="nullApp" />
         </entry>
      </map>
   </property>
</bean>

<bean id="nullApp" class="org.apache.log4j.varia.NullAppender" />
```

The 'Log4jJmxLoader' class has a hashmap property 'appenders'. The key of this hashmap is the name of the appender bean (the appenderName as shown in the 'AvailableAppendersList' property). The hashmap value is an appender bean.

13.5.3 Spring Beans

Spring beans are published under the domain 'SpringBean'. Among other properties, it can be found out if the spring bean is proxied, which interceptors it has, the application context, etc.

13.5.4 JDK 1.5 Standard MBeans

If JMX is running under a JRE verion 1.5 or higher, automatically the JDK 1.5 MMBeans are published under the domain 'java.lang'. These beans give information about garbadge collection, memory management, threads, operating system, etc.

13.6 Patch

The original 'jmxtools-1.4.2.jar' jar-file from Sun contained code, which instanciated two threads, which were not started as deamon threads. The problem with this approach was, that these two threads remained active, also after the main application thread was finished. Therefore these two threads hindered the JVM from beeing shut down. This 'bug' was localized in the 'com.sun.jdmk.comm' package(classes 'CommunicatorServer' and 'HtmlRequestHandler'). The 'HtmlRequestHandler' class was patch directly, by setting the thread to deamon, before starting it. The 'CommunicatorServer' class was patched in its subclass

'HtmlAdaptorServer', because the source code of 'CommunicatorServer' was not available to us. The patched jar-file was named 'jmxtools-1.4.2_deamon_patch.jar'.

For the time being the pached jmxtools has been removed from the module-jmx as the patch did not initialize the `CommunicatorServer` properly. The created thread wasn't stored and therefore the method stop led to a NPE. The original problem should be solved now as the `HtmlAdapterFactoryBean` implements additionally a destroy method, which causes the `HtmlAdaptorServer` to stop. -- PhilippeJacot - 21 Dec 2006

13.7 References

- Further information regarding JMX can be found under http://java.sun.com/products/JavaManagement/index.jsp.
- JmxModuleForTheImpatient shows how easily you can use this in your applications.

14 Documentation for module Web Test

14.1 Purpose

The module Web Test includes all necessary libraries to test Web Applications using JWebUnit or HtmlUnit.

14.2 Overview of our webtests

Please refer to the short presentation about web tests in the following ppt presentation:
http://leaffy.elca.ch/java/el4j/Marketing_Mirror/MonthlyNews/2006/NewEL4JFeaturesJune06.ppt

14.3 Example

```
// Import
import net.sourceforge.jwebunit.TestingEngineRegistry;
import net.sourceforge.jwebunit.WebTestCase;

// Test Class
public class JWebUnitExampleTest extends WebTestCase {

    // set up, choose test engine and base URL
    public void setUp() {
        setTestingEngineKey(TestingEngineRegistry.TESTING_ENGINE_HTMLUNIT);
        getTestContext().setBaseUrl("http://www.elca.ch");
    }

    // describe tests
    public void testElcaURL() throws Exception {
        beginAt("/");

        // check title
        assertTitleEquals("ELCA: Technology-Consulting-Innovation: " +
                "job opportunities: careers: Informatiker: software engineer: " +
                "Programmierer: jobs: software design: business strategy: crm: " +
                "web design: data warehouse: edm");

        // check if text is present
        assertTextPresent("ELCA is a leading Swiss IT solutions provider");

        // check if link is present
        assertLinkPresentWithText("Newsletter");
    }
}
```

15 Documentation for module TcpForwarder

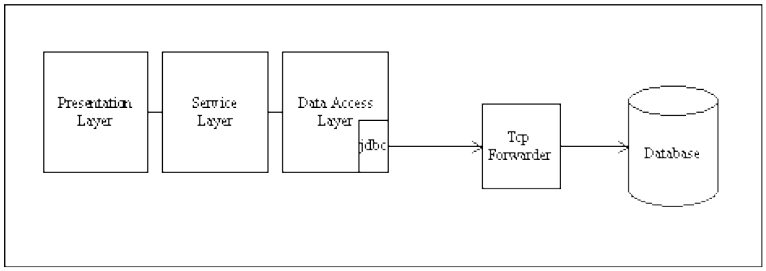
15.1 Purpose

The TcpForwarder module helps to test network failures or connection problems at runtime by controlling TCP connections that can be switched on / off between the source and the destination.

15.2 Important concepts

The TCP forwarder represents a service intended to forward TCP traffic directed to a specific port. The TCP forwarder controls connections between a source and a destination and is able to switch them on / off on demand (either using a simple user interface or programmatically). Therefore, the original application has to switch its destination port, e.g. the port connected to a database, to the input port of the TCP forwarder.

The following picture shows how TCP traffic is forwarded from an application's data access layer to a database:



15.3 How to use

15.3.1 Command line user interface to switch TCP connections on or off

EL4J provides a simple user interface to switch on/off TCP connections called *TCPForwarderTool*.

15.3.1.1 Parameters (tbd in code)

- Input Port: The TCP forwarder's input port - your application has to switch its destination port to this port to be able to use the TCP forwarder
- Destination Port: The TCP forwarder's target port, which has to be your application's original destination port
- Destination URL (optional)

After startup, the input and destination Ports are connected: the TCP forwarder listens on the input port and forwards all traffic to the destination port.

15.3.1.2 Commands

Once started, you can control the TCP forwarder using console commands:

- '1' to unplug the connection between input port and destination port
- '2' to restore the connection between input port and destination port
- '3' to exit

15.3.1.3 Notes

- Don't forget to switch the destination port of your application to the input port of your TCP forwarder.

15.3.2 Programmatically halting and resuming network connectivity

It is also possible to programmatically control network connectivity by using the TCP forwarder's elementary functions directly in code. An example is presented in the automated tests (using JUnit or WebTests with JWebUnit) of the *tcp_forwarder-tests* module.

15.3.2.1 Code configuration

15.3.2.1.1 Import libraries:

```
import java.net.Inet4Address;
import java.net.InetSocketAddress;
import java.net.SocketAddress;

import ch.elca.el4j.tcpred.TcpInterruptor;

// only needed for JWebUnit tests
import net.sourceforge.jwebunit.TestingEngineRegistry;
import net.sourceforge.jwebunit.WebTestCase;
```

15.3.2.1.2 Set up TCP forwarder:

- Forwarding from `INPUT_PORT` to `DEST_PORT`:

```
TcpInterruptor ti = new TcpInterruptor(INPUT_PORT, DEST_PORT);
```

- Forwarding from `INPUT_PORT` to `DEST_URL:DEST_PORT`:

```
SocketAddress target = new InetSocketAddress(Inet4Address.getByName(DEST_URL), DEST_PORT);
TcpInterruptor ti = new TcpInterruptor(INPUT_PORT, target);
```

15.3.2.2 Switch on / off connections

- Cut a connection: `ti.unplug();`
- Restore a connection: `ti.plug();`

15.4 Demonstration code

Please refer to the tests for this module here:
http://el4j.svn.sourceforge.net/viewvc/el4j/trunk/el4j/framework/tests/tcp_forwarder/java/ch/elca/el4j/tcpred/tests/TestDE

16 Documentation for module Light Statistics

16.1 Purpose

The module **lightStatistics** allows setting up performance measurements very easily.

16.2 Important concepts

This module uses a simplified version of the Spring performance interceptor to gather execution times.

16.2.1 Monitoring strategies

- **JMX**: Allows querying performance measurements via JMX
  - **HTML adapter**: Provides a simple web based JMX interface available by default at http://localhost:9092.
  - **JMX connector**: activates the JMX connector to allow JMX conformant viewer to query data.
- **JAMon admin jsp**: The JAMon admin jsp is deployed along with the web application (in fact, the jsp file is always provided but only usable within a web application server).

16.3 How to use

16.3.1 Configuration

Using either the JAMon admin jsp within a web application container or the JMX HTML adapter, you don't have to do anything except adding the dependency to this module to your project definition. By default, all beans are advised by the measurement interceptor. The set can be limited to a particular name pattern using Spring's configuration override facilities.

16.3.2 Demo

A demo module named **module-light_statistics-demos** is provided. It shows how to use the JMX HTML adapter in a stand-alone application.

16.3.3 How to set up the module-light_statistics for the ref-db sample application

This example shows how to use the performance monitor in the red-db sample application.

TBD: the following needs to be adapted to how this is done with maven:

16.3.3.1 binary-modules.xml

Add the following two lines to the `binary-modules.xml` file that is in the refdb's root directory.

```
<attribute name="binrelease.version.module-light_statistics" value="1.0"/>
<attribute name="binrelease.version.module-jmx" value="1.0"/>
```

16.3.3.2 project.xml

Add the following dependency to the `refdb-web` module definition:

```
<dependency module="module-light_statistics">
    <mapping target="jmx"/>
</dependency>
```

If you just want to use the admin jsp file without the JMX support, then replace the mapping target `jmx` with `web` (the jsp file is always provided, but runs in a web application environment only). Doing so, the lines you have to insert look like this:

```
<dependency module="module-light_statistics">
    <mapping target="web"/>
</dependency>
```

16.3.3.3 Limit the set of interecepted beans

Spring allows overriding configurations in properties files. Limiting the set of intercepted beans makes use of this feature. The key in the properties file is `lightStatisticsMonitorProxy.beanNames`. More details about how to use spring's configuration features can be found under the PropertyConfiguration topic.

Important: In order to get the ref-db web application run with this module you have to limit the set of advised beans (there are some beans that are not advisable)! e.g. use this in your `override.properties` file:

```
lightStatisticsMonitorProxy.beanNames=reference*
```

and use the following bean definition:

```
<bean id="propsOverride" class="org.springframework.beans.factory.config.PropertyOverrideConfigurer">
    <property name="locations">
        <value>classpath:mandatory/override.properties</value>
    </property>
</bean>
```

Notice: both files, the `override.properties` and the bean definition can be added to the ref-db web's `mandatory` folder to be loaded automatically.

16.4 FAQ

- I got exceptions that Spring can not inject some dependencies. Without the module-light_statistics, everything runs nicely.
  - Maybe there are some classes that cannot be advised. Use the `lightStatistics-override.properties` to specify the beans to advise, as described here.

16.5 References

- JAMon web site http://www.jamonapi.com/
- Detailed statistics service of EL4J

17 Documentation for module Detailed Statistics

17.1 Purpose

The detailed statistics module measures the duration of service invocations via interceptors and makes these measures and their call-graph available via a sequence diagram in SVG.

17.2 Important concepts

This package includes: (1) a measure interceptor that measures invocation times that are stored by (2) the measurement collector service. Finally (3) a statistics analyzer service allows analyzing the data collected and dump it e.g. to CSV (Excel) or a sequence diagram picture (png) files. Alternatively it is also possible to measure the times of other events than service invocations by calling the measurement collector service via its API.

17.3 What can be measured?

Any spring service invocation can be measured. Measures can even be made over JVM-boundaries: the ID of the measure is then exchanged via the implicit context passing of EL4J.

Potentially anything can be measured, as one can explicitly call the API of the mesurement collector service manually.

17.4 Description of the attributes of a measurement

To identify each measure/sub-measure corresponding to a same end-to-end measure, the following attributes are defined:

- a **measure ID**, that must be different for every end-to-end measure and which is made of the machine name and the invocation time,
- a **sequence number** which corresponds to the call-level. This means that a same number can be used more than once if a bean calls several other beans (see below for an example),
- the **time** when measure started, which must be used to reorder the measures done on the server.

Other attributes are allowed to identify :

- the measured **component ID**,
- the **type** of the measured component,
- the **duration of the measure** (in milliseconds).

17.5 How to use

17.5.1 Configuration

You need to add a dependency on the detailed statistics module. Then you state what beans you want to measure by either providing them explicitly with a Spring proxy or configuring an auto-proxy, that adds all beans to the measurement if they are not explicitly excluded. (Remark: we are also considering a JDK 1.5 annotation that selects whether the performance of the method shall be tracked.)

17.5.2 How to get the statistics information via JMX

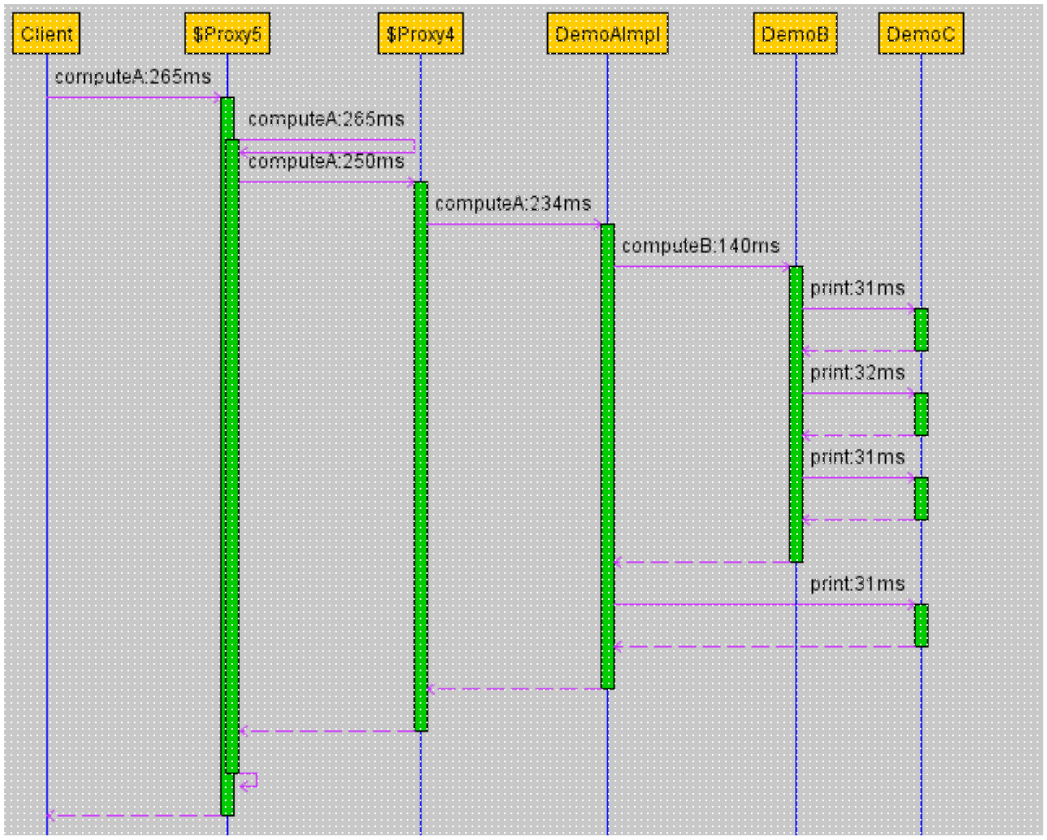
The most convenient access to the detailed statistics information is via the JMX interface. On the MBean of the detailed statistics tool (the detailedStatisticsReporter), you can get various information:

- The list of all recent measure IDs (via the method `showMeasureIDTable`)
- All measures corresponding to one measure ID in a CSV file (comma separated excel file). For this, you use the method `createCSVFile`. The first argument is the name of the file, the second argument is the measure ID. At the moment, this file is written on the machine that runs the JMX HTTP server.
- All measures corresponding to one measure ID as a sequence diagram (png format). For this, you use the method `createDiagramFile`. The first argument is the name of the file, the second argument is the measure ID. At the moment, this file is written on the machine that runs the JMX HTTP server.

17.5.3 Demo

There is a demo for the use of the detailed statistics module. In the demo, all server-side beans are intercepted with help of an auto-proxy whereas the client side bean is intercepted with an explicit Spring proxy.

Please refer to the readme-file of the demo for more information on how to launch the demo:
http://el4j.svn.sourceforge.net/viewcvs.cgi/*checkout*/el4j/trunk/el4j/applications/demos/detailed_statistics/README.tx



18 Documentation for module ShellLauncher

18.1 Purpose

Allows passing a bean shell expression on the JVM-command line that is launched in your application. The purpose of this is to help with debugging or the understanding of your application. One way to use it is to allow a remote login in your JVM.

18.2 How to use

To start using this module, add the following to your project's `pom.xml` file.

```
<dependency>
    <groupId>ch.elca.el4j.modules</groupId>
    <artifactId>module-bshlauncher</artifactId>
    <version>1.0</version>
</dependency>
```

Then you can set a bsh (beanshell) expression that should be launched at startup (this is basically any Java code). You do this via `-Del4j.bsh.launchstr=javaCodeToLaunch`.

We have predefined some scriptslets. You can also set your own scripts in the `resources/bsh_scriptlets/` folder of your modules. Please refer to the sample scriptlets in the bsh_launcher module. One special feature (of the basic bsh) is the scriptlet `server(portNumber)`. It is like a remote login into the JVM of your application. This means you can execute any Java code in your JVM (so this can be a major security risk - remove this dependency in critical deployments!). Here is an introduction on how to use BeanShell http://www.beanshell.org/manual/quickstart.html#Quick_Start .

How to use this "server(portNumber)" scriptlet in short:

- put the following string when you launch the JVM: `-Del4j.bsh.launchstr=server(2000);` (typically you just add this to your MAVEN_OPTS)
- normally launch your application with mvn
- connect to your application via `http://localhost:2000/remote/jconsole.html` (assuming your application runs on localhost)
- (optionally) if you would like to enable cut and paste from/ to your system clipboard, you need to set the following java permission (in your currently active `java.policy` file, refer e.g. to your browsers Java Console and look at the System properties for this). On my machine I added in the file `C:\Program Files\Java\jre1.6.0_03\lib\security\java.policy` the following section:

```
grant codeBase "http://localhost:2000/*" {
   permission java.awt.AWTPermission "accessClipboard";

};
```

- alternative: you can also connect to your application via telnet through the following call: `telnet localhost 2001`

Other ideas to do via bsh scripts: track # of threads used over time, memory usage (e.g. print it every 10 seconds), threadInfo(), ...

19 Documentation for module XmlMerge

19.1 Purpose

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

19.2 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:

```
<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>
```

original patch result

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

19.3 Module contents

The module contains the following stuff:

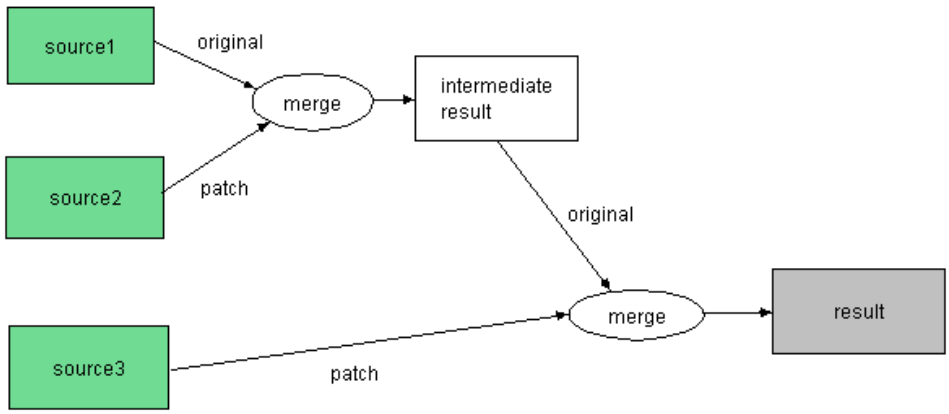
- Interfaces and infrastructure supporting the concepts the module is based on (in fact this forms the API and SPI).
- Default implementations of these interfaces.
- Convenience support for configuring your merge using XPath (outside of the XML documents) or with XML attributes within the XML documents 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.

19.4 Important concepts

19.4.1 Original and Patch

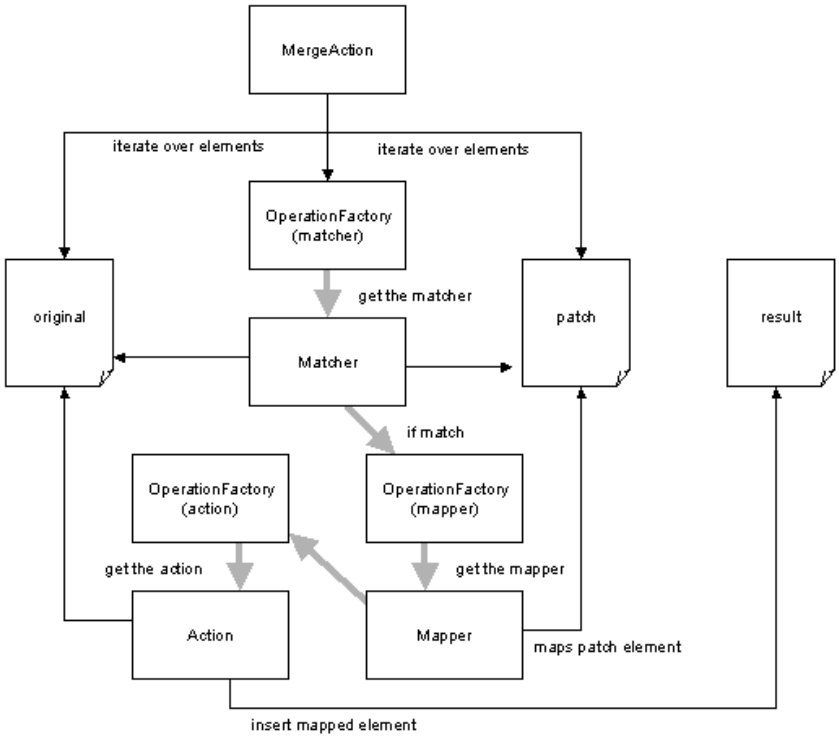
The *sources* are The XML documents (as java.lang.String, java.io.InputStream or org.w3c.dom.Document) that we want to merge.



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.

19.4.2 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.

19.4.3 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.

19.4.3.1 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.

19.4.3.2 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.

19.5 Built-in implementations

19.5.1 Operations

The module provides implementations of operations that are commonly used:

19.5.1.1 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.

19.5.1.2 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.

19.5.1.3 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.

19.5.2 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**.

19.5.3 XmlMerge Implementation

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

19.5.4 Operation Factories

Three implementations of the operation factory are provided:

- **StaticOperationFactory**. Returns the same operation for all element pairs. Used when the same behaviour 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.

19.6 Configuring your Merge

You have currently three ways to configure an XmlMerge instance:

19.6.1 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>
  <a/>
  <c/>
</root>
```

original

+

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

patch

=

```
<root>
  <a>
    <b/>
  </a>
  <c/>
</root>
```

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.

19.6.2 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>
  <a/>
  <c/>
</root>
```

+

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

=

```
<root>
  <a>
    <b/>
  </a>
  <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());
}
```

19.6.3 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>
```

+

```
<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>
```

=

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

original patch 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());

}
```

19.7 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>

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

+

```
<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>

  <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>
```

original patch 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());
        }
}
```

19.8 How to use

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

19.8.1 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.

19.8.2 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.

19.8.3 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.

19.8.4 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.

19.9 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

20 Exception handling guidelines

For an introduction to general rules of exception handling in Java, please refer to chapter 2 of LEAF 2 exception handling guidelines.

20.1 Topics

- When to define what type of exception, normal vs. abnormal results
  - What results are signalled with exceptions?
  - Use checked or unchecked exceptions?
  - When to define own exceptions, when to reuse the existing ones?
- Implementing exceptions
- Where and how to handle exceptions
  - Who handles what exceptions?
  - How to handle exceptions?
  - How to trace exceptions?
  - How to throw an exception as a consequence of another exception?
- Related useful concepts and hints
- Antipatterns
- References

20.2 When to define what type of exceptions? Normal vs. abnormal results

Throwing exceptions is expensive (in some examples up to 800 times slower than returning a "normal" value!). Therefore exceptions should be used for exceptional cases only (i.e., for cases that do not occur frequently).

A method invocation on an interface can have 2 fundamentally different type of results:

- **Normal results**: the result matches the level of abstraction of the interface. Examples: If one tries to make a withraw on a bank account, possible results are: ok, that the account is overdrawn or locked. These are normal and expected events, on the same level of abstraction than the interface. Normal errors that are expected (i.e. a subset of *normal results*) are often also called *business exceptions*.
- **Abnormal results**: these results are not on the level of abstraction of the interface. They reveal implementation details and/or are for very unlikely events. The caller can typically not do much in response to an abnormal result. Such results are typically best handled on a higher level (often global for an application). Abnormal results are also appropriate to signal that the method was used improperly (e.g. when a precondition has been violated). Abnormal results are also used for situations that can't be handled during runtime. Examples: OutOfMemoryError, PreconditionRTException, SQLException indicating that the connection to the database is lost, RemoteException, ...

We will see later that we typically use checked exceptions for normal results and unchecked exceptions for abnormal results.

20.2.1 Further examples

Because this is a very important distinction, here are some more examples. Whether a result is normal or abnormal depends on its *context*:

- Method Account.withdraw()
  - normal results: ok, overdrawn or locked
  - abnormal results: RemoteException (of RMI), SQLException **Rationale:** They have nothing to do with the withdraw method, they are an implementation detail.
- Method DatabaseAccessLayer.connectDb()

    - normal results: ok, not ok (not ok may be the same thing as the SQLException of the previous example: in this context it is normal)
    - abnormal results: RemoteException (of RMI) **Rationale:** RMI has nothing to do with databases accesses, it's an orthogonal issue.
- Consider an order system of an online-shop. Every 1'000'000th customer gets a gift. Such a result is sufficiently rare that we could say it is abnormal. (So something abnormal does not need to be a mistake!) We could therefore throw a runtime exception for this abnormal case.

20.2.2 How to handle normal and abnormal cases

For **normal results** that are expected special cases (including expected errors) we use **checked exceptions** or special **return values**. One should be conservative with checked exceptions. Avoid many newly defined checked exceptions. This leads to many catch blocks in the code (this makes the code longer and harder to read). Try also to avoid having a method throwing too many checked exceptions. Such a method can be very cumbersome to use. (As a bad example, please have a look at the Java API for reflection (package java.lang.reflect)). Signaling special cases via return values is sometimes appropriate when the event occurs often (due to the implied performance overhead of exceptions).

We use **unchecked exceptions** to inform about **abnormal results**. As with checked exceptions, try again to avoid too many new exceptions. Names of unchecked exceptions should have a `RTException` suffix.

Remark: The RemoteException of RMI violates these guideline, as it should be an unchecked exception. (Many people consider this a design mistake of RMI.)

20.3 Implementing exceptions classes

You can use the classes BaseExceptions and BaseRTExceptions as base classes for new exceptions. These classes provide base support for exception internationalization.

Do not use the string message of an exception to differentiate among different exception situations. For example, one should *not* use in a project just one exception class (e.g. the predefined BaseException) with different String messages to differ between situations. This bad practice makes it hard to react differently in function of what happened (as it would require parsing the exception message), it would also not allow adding particular attributes to the exception class, and would not document what type of exceptions can be thrown in a method signature. Finally, it would make exception message internationalization harder (because one would need to parse the exception message first). Sometimes it is desirable not to write one exception class per exception situation (e.g. there may just be too many exception classes). In such cases on can use a common base exception class and use an error code to differentiate between the exceptions.

We recommend not to make a difference between exceptions of EL4J code and exceptions of applications using EL4J. (This means that the same rules apply and that there is no separate exception hierarchy for the two contexts.)

Remember that one should avoid adding too many new exceptions. You can reuse (i.e. use in your method signatures) exceptions of the JDK. Frequently useful candidates are IllegalArgumentException or IndexOutOfBoundsException.

20.4 Handling exceptions

20.4.1 Where to handle exceptions?

Normal results of invocations should be handled by the code making the invocation. Optionally it may make sense to propagate the exception to the caller of the invoking class (in other words: up the calling stack).

Abnormal results (those returned via unchecked exceptions) are typically passed up the calling stack and handled on a higher level (not directly where the invocation was made). Handle an abnormal result only if you

can really do something against the problem or if you are on the top-level of a component that is responsible to handle all abnormal cases. A pattern that separates the handling of abnormal situations in a nice way is the SafetyFacade.

20.4.2 How to trace exceptions?

One should not trace normal results (including exceptions that signal normal results!) of method invocations. (Unless there is some external requirement for this.)

Abnormal situations should be traced where they are caught.

Please refer to TracingInfrastructure for more detail on general tracing. Typically one uses `error` or `fatal` priority levels when tracing abnormal situations.

20.4.3 Rethrowing a new exception as the consequence of a caught exception

Try to avoid making too many such exception translations (i.e. in a catch statement *translate* an exception by throwing another exception for it). If you do it anyway, you should wrap the caught exception in the newly thrown exception (in order not to loose information). The Exception class of JDK 1.4 provides support for this.

20.5 Related useful concepts and hints

20.5.1 Add attributes to the exception class

As Java exceptions are classes, it is possible to add attributes to exception classes. This can be useful e.g. to include information needed to fix the abnormal situation or to provide more information about the exceptional situation.

Such attributes are particularly useful when the exception is treated programmatically (e.g. to do something in function of the value of such attributes). Having these attributes explicitly as attributes and not just embedded in the error message avoids that the error message needs to be parsed. In addition, it helps to internationalize exceptions. See also the example of the `BaseException` class that illustrates how this can be used with JDK MessageFormats. The string message of exceptions should contain all attributes that are useful for someone trying to figure out what went on. (We don't print automatically all attributes of exceptions.)

20.5.2 Mentioning unchecked exceptions in the Javadoc

Sometimes it is useful to mention unchecked exceptions that can be thrown by a method (even though this is not required by Java). This can be made in the code (one has the right to add an unchecked exception to the `throws` definition of a method) or in the Javadoc. This makes the user of the method aware of the unchecked exception that may be potentially be thrown by the method.

20.5.3 Checking for pre-conditions in code

Assumptions a programmer of code makes about how the code is used, are called pre-conditions. Violated pre-conditions are abnormal situations. Therefore one should use unchecked exceptions to indicate pre-conditions. Pre-conditions are checked in the beginning of the body of method implementations. One should keep such checks in the code of publicly available methods even if the code is deployed in a production environment. Such pre-condition checks are particularly useful when a component is used after its creation or in another context. Rationale: such pre-conditions check that the assumptions of the programmer are valid. **Do not use assertions to check the parameters of a public method.**

There is a Reject class (in the core module) that helps to support this usage. This usage is also recommended in the assertion guidelines of sun. (We refer to the text: "By convention, preconditions on public methods are

enforced by explicit checks that throw particular, specified exceptions.") We propose to check such preconditions on public methods via the Reject class.

Sample use:

```
public void saveAccount(Account x) {
  Reject.ifNull(x, IllegalArgumentException.class,  x must not be null");
   // throws an IllegalArgumentException if x == null
```

An alternative to this class is Spring's [[http://leaffy.elca.ch/java/javaTechnologyDoc/extracted/spring-framework-2.0.2/docs/api/org/springframework/util/Asse class. Please refer also the the AssertionUsageGuidelines for more details about assertion usage.

20.5.4 Exception-safe code

Exception-safety is a property of well-implemented code. There is weak and strong exception safety.

For a method m() that is **weakly exception safe**, the following conditions hold when it throws an exception:

1. m() does not complete its operation.
2. m() releases all the resources it allocated before the exception was thrown.
3. If m() changes a data structure, then the structure must remain in a consistent state.

In summary, if a weakly exception safe method m() updates the system state, then the state must remain reasonable.

Strongly exception safe methods additionally verify the following condition:

- If a method m() terminates by propagating an exception, then it has made no change to the state of the program.

Both exception safety properties are desirable. However as the implementation of strongly exception safe methods can be quite tricky, we only require methods in EL4J to be weakly exception safe. Please refer to § 3.5.1 of the LEAF 2 exception handling guidelines for more details.

20.5.5 Handling SQL exceptions

To handle SQL exceptions, we strongly recommend the helper classes of the spring framework. This support is sometimes referred to as Spring's generic DataAccessException hierachy. To profit from this hierachy, use the Spring simplification templates for integration of iBatis or Hibernate. This allows profiting from this hierarchy almost for free. EL4J provides an improvement to this exception mapping, please refer to the file sql-error-codes.xml of the core module and the package ch.elca.el4j.services.persistence.generic.sqlexceptiontranslator.

20.5.6 Exceptions and transactions

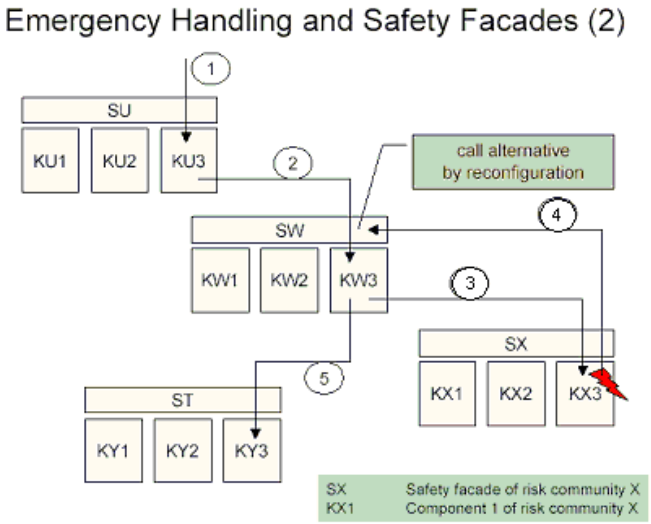
Transactions should often be rolled back after an exception occurs. Please refer to ModuleTransactionAttributes for a description on how to do this automatically.

20.5.7 SafetyFacade pattern

The goal of the safety facade is that it handles all the abnormal situations, such that for a user of a component, the business operation either succeeds or completely fails. The safety facade has made all attempts to fix or retry and has informed the required parties if necessary. A safety facade takes the responsibility of treating abnormal situation away from normal business code. This is particularly interesting as the code can typically not do much against it. The safety facade wraps a group of component implementations

(e.g. via dynamic proxies) and provides a "better" quality of service (i.e. either the components work or they fail completely) for the users of the component implementations.

The following picture illustrates this. Four groups of components ("risk communities") are each assembled with their safety facade. The safety facade treats all abnormal situations. The numbers indicate a sample use of a safety facade: KU3 calls KW3 and KW3 calls KX3. KX3 indicates an abnormal situation (throws an unchecked exception). The safety facade SW therefore reconfigures the system such that KW3 retries once again with the component KY3.



The module ModuleExceptionHandling implements a safety facade. This idea is described in "Moderne Software Architekturen", §5.4 .

20.6 Antipatterns

Exceptions are considered to be an important tool of modern programming languages, but they become a nuisance for programmers. We list some of the typical problems (antipatterns) encountered in projects ranging from 1 to more than 100 man-years:

- There are a large number of different exceptions classes. It is neither clear when exceptions should be thrown nor how they should be handled.
- A huge number of exception classes create undesired dependencies between the caller and the callee.
- The code gets messy because of nested try-catch blocks.
- Many catch blocks are either empty, contain little value-adding code (output to the console, useless mappings of one exception class into another) or at best some logging, but no true exception handling.
- Exceptions are misused to return ordinary values.

Please keep these antipatterns in mind! They are mostly avoided if you use the previous rules and common sense.

20.7 References

- LEAF 2 exception handling guidelines: http://leaffy.elca.ch/leaf/Documentation_Mirror/guidelines/LEAFExceptionHandlingGuidelines.doc
  - ♦ The usage of exceptions has changed in the EL4J. Chapter 2 remains valid.
- Errors and Exceptions Rights and Responsibilities, Johannes Siedersleben, ECCOP 2003, paper: http://www.sdm.de/web4archiv/objects/download/pdf/vonline_siedersleben_ecoop03.pdf, slides:

http://homepages.cs.ncl.ac.uk/alexander.romanovsky/home.formal/Johannes-talk.pdf
- Moderne Software Architekturen, Siedersleben, 2004, Chapter 5 (in German)
- Rules for Developing Robust Programs with Java, Article about exception handling in Java
  http://www.idi.ntnu.no/grupper/su/fordypningsprosjekt-2003/fordypning2003-Nguyen-og-Sveen.pdf
    - Easy to read, many interesting patterns about exception handling.
- EL4J HighLevelExceptionHandlingGuidelines

21 Maven plugins

The standard maven documentation of the EL4J plugins can be found under http://el4j.sourceforge.net/plugins/index.html. This section contains some additional documentation.

21.1 Database

The Database plugin can be used to automate the starting up of a database, creating and droping tables as well as adding and deleting data. The database initialization is typically done via configuration files from the classpath. This allows that each maven project adds its own database initialization. The plugin helps to ensure that this initialization information is applied in the correct order.

The idea that each project should be able to define its own SQL resources comes from the module abstraction of EL4J.

21.1.1 Dependencies to external jars

The plugin requires Spring for path matching and Derby for starting the Derby Network Server.

21.1.2 Goals

21.1.2.1 Goal `start`

This goal can run in two modes:

1. The first one is to start the configured database server through the console command `mvn db:start`. It does only know how to launch Derby (when other database are set, the command does nothing). This can be run on an arbitrary project and does in general not require configuration information. If run outside of EL4J and Derby is enabled (default), it requires the `toolsPath` property. This goal blocks by default the maven execution until the user hits Ctrl-C (this can be changed by setting `db.wait` to `false`).
2. The second mode is to integrate the goal into the build life cycle of a project. This requires the `wait` property to be set to `false`, because the goal would block the execution otherwise.

21.1.2.2 Goal `create`

This goal creates tables in the database. It takes the `sqlSourceDir` property, replaces `{db.name}` with the actual database name (specified via profile), resolves the given class paths to filesystem paths and looks in these directories for `.sql` files of the format `create*.sql`, where * denotes an arbitrary character sequence. (CAVEAT: sqlSourceDir is evaluated in the classpath, not in the normal file system path!) It then takes these `.sql` files, extracts all semi-colon separated SQL statements of them and executes these statements, starting with the statements of the SQL files of the highest dependencies (e.g. if A depends on B and B depends on C, it would execute C's statements first, then B's and finally A's). Moreover, it uses the JDBC driver from `driverPropertiesSource` and the connection properties from `connectionPropertiesSource` to establish a database connection.

21.1.2.3 Goal `update`

This goal works like the `create` goal, but executes update statements on the tables.

21.1.2.4 Goal `delete`

Taking the same properties like `create` and `update`, this goal deletes data in the tables. Note that it executes the SQL statements in reversed (!) order. Taking the example from the `create` goal this would mean that it starts with A's statement, then process B's and finally C's. This order ensures that SQL

constraints are not violated.

21.1.2.5 Goal `drop`

The `drop` goal works like the `delete` goal instead that it drops tables. Note that compared to `silentDrop` it throws an exception in case it encounters an error.

21.1.2.6 Goal `silentDrop`

The `silentDrop` goal works just like the `drop` goal, but doesn't throw an exception in case of an error.

You would use `silentDrop` in the `pre-integration-phase` to ensure that all tables are dropped prior of executing the `create` goal and `drop` in the `post-integration-phase` to clean up after your tests and where you actually want to get exceptions.

21.1.2.7 Goal `stop`

This goal stops the Derby Network Server. This can be used when integrating the plugin to the build lifecycle. If not used, java will clean up and stop the Derby Network Server when the execution of maven is done.

21.1.2.8 Goal `block`

Convenience goal to block until one hits Ctrl-C. Is unconditional (blocks also with a db.wait flag set to false).

21.1.2.9 Goal `prepare`

This goal is a convenience goal for executing `start`, `silentDrop` and `create` in sequence. It simplifies the typical code in the `pom.xml` file to the following:

```
<plugin>
   <groupId>ch.elca.el4j.plugins</groupId>
   <artifactId>maven-database-plugin</artifactId>
   <executions>
      <execution>
        <configuration>
          <connectionPropertiesSource>
             scenarios/db/raw/keyword-tests-override-{db.name}.properties
          </connectionPropertiesSource>
        </configuration>
        <goals>
          <goal>prepare</goal>
        </goals>
        <phase>pre-integration-test</phase>
      </execution>
    </executions>
  </plugin>
```

If you use the default name for the .properties files (*artifactId* -override-{db.name}.properties in directory scenarios/db/raw/), you may even omit these settings.

21.1.2.10 Goal `cleanUp`

This goal is a convenience goal for executing `drop` and `stop` in sequence. It simplifies the typical code in the `pom.xml` file to the following:

```
<plugin>
   <groupId>ch.elca.el4j.plugins</groupId>
   <artifactId>maven-database-plugin</artifactId>
   <executions>
      <execution>
        <configuration>
```

```
            <connectionPropertiesSource>
               scenarios/db/raw/keyword-tests-override-{db.name}.properties
             </connectionPropertiesSource>
          </configuration>
          <goals>
            <goal>cleanUp</goal>
          </goals>
          <phase>post-integration-test</phase>
        </execution>
      </executions>
    </plugin>
```

Here, the same rules for connectionPropertiesSource as in `prepare` exist.

21.1.3 Properties

- `db.wait`: Specifies if the start goal should be blocking or not (default value `true`).
- `db.connectionPropertiesSource`: Path to a property file where connection properties (username, password and url) are set. Sample content:

```
dataSource.url=jdbc:derby:net://localhost:1527/"refdb;create=true;":retrieveMessagesFromServerOnGetMessag
dataSource.username=refdb_user
dataSource.password=**********
```

If you don't provide a value for this parameter, a .properties file is selected automatically. Naming pattern: `artifactId-override-{db.name}.properties` in directory `scenarios/db/raw/`, where *artifactId* is replaced by current ArtifactId[?]. The base directory may be changed using parameter `connectionPropertiesDir` (see below). The naming pattern can be changed, too (see `db.connectionPropertiesSourceTemplate`). If there is no .properties file for the current artifact, all the artifacts in the dependecy tree are checked (from the leafs towards the root), until a .properties file is found.

- `db.connectionPropertiesSourceTemplate`: Template for filenames for .properties files used to read the connection settings of the database. You can use the variables {groupId}, {artifactId}, {version} and {db.name}. E.g. `{artifactId}-override-{db.name}.properties` (this is the default-value)

- `db.connectionPropertiesDir`: Directory where to look for .properties files for DB-connection (default value `scenarios/db/raw/`).

- `db.driverPropertiesSource`: Path to property file that contains the JDBC driver name (default value `scenarios/db/raw/common-database-override-{db.name}.properties` (contained in the EL4J database module)). Sample content:

```
dataSource.driverClassName=com.ibm.db2.jcc.DB2Driver
dataSource.validationQuery=VALUES CURRENT TIMESTAMP
```

- `db.sqlSourceDir`: SQL source directories, i.e. directories where to find the .sql files (default value `/etc/sql/general/, /etc/sql/{db.name}/`). The source directories are separated with the `separator`.
- `separator`: Separator for string lists (default value ",").

21.1.4 Examples

21.1.4.1 Console usage

`mvn db:start` To start the Derby Network Server in blocking mode.

21.1.4.1.1 Integrating plugin into build lifecycle

- Add the following snippets to the section in the part of the `pom.xml` file of your project
  - In the `pre-integration-test` phase:

```
<plugin>
   <groupId>ch.elca.el4j.plugins</groupId>
   <artifactId>maven-database-plugin</artifactId>
   <executions>
      <execution>
        <configuration>
          <connectionPropertiesSource>
             scenarios/db/raw/keyword-core-tests-override-{db.name}.properties
          </connectionPropertiesSource>
          <wait>false</wait>
        </configuration>
        <goals>
          <goal>start</goal>
          <goal>silentDrop</goal>
          <goal>create</goal>
        </goals>
        <phase>pre-integration-test</phase>
      </execution>
    </executions>
  </plugin>
```

- `integration-test` phase: We use a trick to shift the tests from the `test` to the `integration-test` phase, where we can specify plugins to be executed before and after the `integration-test` phase.

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <skip>true</skip>
            </configuration>
              <executions>
                <execution>
                  <id>surefire-it</id>
                  <phase>integration-test</phase>
                    <goals>
                     <goal>test</goal>
                    </goals>
                <configuration>
              <skip>false</skip>
          </configuration>
        </execution>
    </executions>
 </plugin>
```

- `post-integration-test` phase:

```
<plugin>
   <groupId>ch.elca.el4j.plugins</groupId>
   <artifactId>maven-database-plugin</artifactId>
   <executions>
      <execution>
        <configuration>
          <connectionPropertiesSource>
             scenarios/db/raw/keyword-core-tests-override-{db.name}.properties
           </connectionPropertiesSource>
          <wait>false</wait>
        </configuration>
        <goals>
          <goal>drop</goal>
          <goal>stop</goal>
```

```
        </goals>
        <phase>post-integration-test</phase>
      </execution>
    </executions>
  </plugin>
```

- Note that you don't need the `db.` prefix for the properties.
- You can use `{db.name}` in the `connectionPropertiesSource` as well as the `driverPropertiesSource` to keep the configuration generic.
- You can specify more than one SQL source directory by using the separator.
- All the resources are taken from the classpath, which includes the project as well as all dependencies.
- `prepare` and `cleanUp` were earlier named `prepareDB` and `cleanUpDB`.
- Finally, note that we added the goal `siltenDrop` before `create` to be sure that the the tables do not exist when we try to create them. `SilentDrop` is the only goal that won't fail if it encounters an exception.

Please refer also to the MavenCheatSheet for more examples

21.2 DepGraph

The DepGraph plugin can be used to draw a dependency graph from the project, the mojo is executed in. It traverses all dependencies and creates a graph using Graphviz.

21.2.1 External prerequisites

Apart from maven and the plugin itself one has to make sure that Graphviz (see below) is installed and its executables - in particular `dot` - are in the PATH environment variable.

21.2.2 Description

There are two maven goals: depgraph and fullgraph to get a dependency graph just for your project or a graph for all the modules as they are interconnected. Please refer to the next two section for more info on how to use them.

In the full graph, the same logical artefact can exist multiple times (e.g. in different versions). When calculating the classpath for a project, maven will eliminate the unneeded artefact (typically the one with the older version id). Please refer to maven doc for more details. We have also used this plugin to detect duplicate jars in a big project.

Handling of child->parent dependencies: To make the graph output more readable, parent->child dependencies are not considered as dependencies in case no other dependencies exist. This typically leads to "orphan" artifacts that seem to be not connected to the rest of the artifacts. You can use the `depgraph.filterEmptyArtifacts` property in case you want to eliminate them from the graph.

21.2.3 Goal `depgraph`

Creates the graph to the local project.

21.2.3.1 Properties

- `depgraph.outFile`: The file to write to. Default is *name of the project.png* in the current directory. Using another extension than png, one can change the format the output is in
- `depgraph.outDir`: The directory the output files are written to. If no `outDir` is given, the path is relative to the working directory
- `depgraph.artifactFilter`: Only include artifacts that contain the given pattern. Java regexp is used, so any possible regexp can be used
- `depgraph.groupFilter`: Like `artifactFilter`, but filters group

- `depgraph.versionFilter`: Like `versionFilter`, but filters version
- `depgraph.dotFile`: The file to write the dot file to. By default, no dot file is written.
- `depgraph.filterEmptyArtifacts`: Delete all artifacts that none depends on and that depend on none. By default these artifacts are shown.

21.2.4 Goal `fullgraph`

Creates a graph of all projects that are in reachable modules from the current project.

21.2.4.1 Properties

The same properties as for the `depgraph` goal are available.

21.2.5 Links

- Graphviz Homepage
- The Java tutorials: Regex

21.2.6 Open Issues

- Dependency of war-packaged artifacts are not resolved (unless the mojo is invoked on them explicitly).
  - ♦ To solve this issue, take a look at the eclipse plugin, this maybe has the same issue
- We show all dependencies to artifacts. Sometimes maven globally unifies n dependencies to different versions of an artifact (e.g. it only retains the newest artifact). This unification not shown in our graph. One can also see this behavior as a feature (you see the whole picture).

21.2.7 Examples

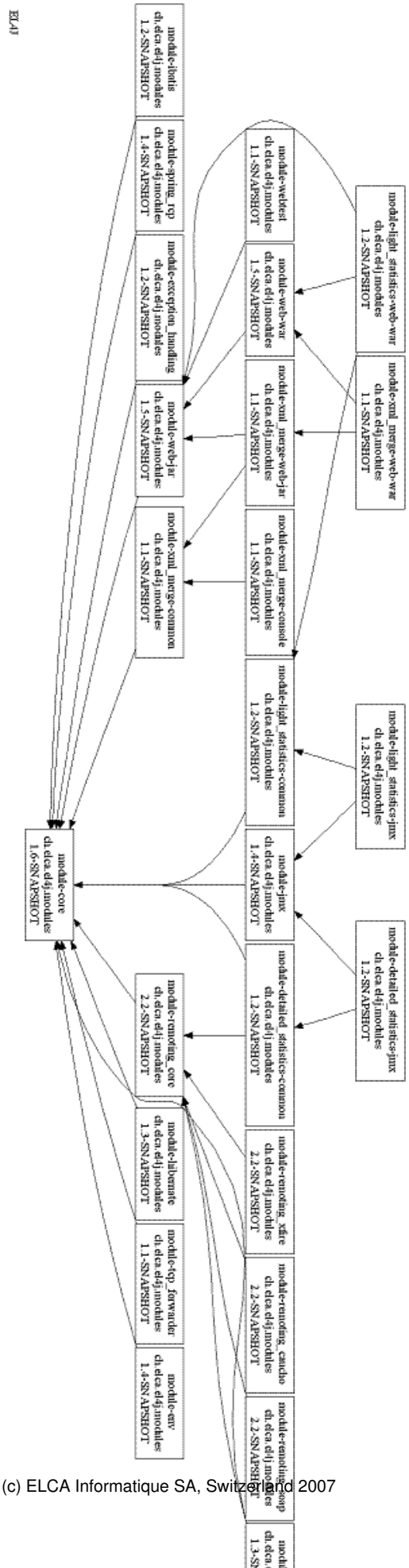
21.2.7.1 Command line

```
mvn depgraph:fullgraph -Ddepgraph.groupFilter="ch.elca"

mvn depgraph:fullgraph -Ddepgraph.groupFilter="(ch.elca.el4j.modules)|(ch.elca.el4j.demos)|(ch.elca.el4
  -Ddepgraph.filterEmptyArtifacts=true -Ddepgraph.dotFile=el4j.dot
```

21.2.7.2 Sample Output graphs

21.2.7.2.1 Small output graph


EL4J
module-ibatis
ch.elca.el4j.modules
1.2-SNAPSHOT
module-spring_rcp
ch.elca.el4j.modules
1.4-SNAPSHOT
module-exception_handling
ch.elca.el4j.modules
1.2-SNAPSHOT
module-web-jar
ch.elca.el4j.modules
1.5-SNAPSHOT
module-xml_merge-common
ch.elca.el4j.modules
1.1-SNAPSHOT
module-core
ch.elca.el4j.modules
1.6-SNAPSHOT
module-webtest
ch.elca.el4j.modules
1.1-SNAPSHOT
module-web-war
ch.elca.el4j.modules
1.5-SNAPSHOT
module-xml_merge-web-jar
ch.elca.el4j.modules
1.1-SNAPSHOT
module-xml_merge-console
ch.elca.el4j.modules
1.1-SNAPSHOT
module-light_statistics-common
ch.elca.el4j.modules
1.2-SNAPSHOT
module-jmx
ch.elca.el4j.modules
1.4-SNAPSHOT
module-detailed_statistics-common
ch.elca.el4j.modules
1.2-SNAPSHOT
module-remoting_core
ch.elca.el4j.modules
2.2-SNAPSHOT
module-hibernate
ch.elca.el4j.modules
1.3-SNAPSHOT
module-tcp_forwarder
ch.elca.el4j.modules
1.1-SNAPSHOT
module-env
ch.elca.el4j.modules
1.4-SNAPSHOT
module-light_statistics-web-war
ch.elca.el4j.modules
1.2-SNAPSHOT
module-xml_merge-web-war
ch.elca.el4j.modules
1.1-SNAPSHOT
module-light_statistics-jmx
ch.elca.el4j.modules
1.2-SNAPSHOT
module-detailed_statistics-jmx
ch.elca.el4j.modules
1.2-SNAPSHOT
module-remoting_xfire
ch.elca.el4j.modules
2.2-SNAPSHOT
module-remoting_caucho
ch.elca.el4j.modules
2.2-SNAPSHOT
module-remoting_soap
ch.elca.el4j.modules
2.2-SNAPSHOT

21.2.7.2.2 All modules, different view

Medium graph

21.2.7.2.3 All modules, test modules and samples of EL4J:

Big graph

Please refer also to the MavenCheatSheet for more examples

21.3 Version Plugin

A plugin to keep track of used dependencies for a given project. It is intended to get a hint which referenced artifacts could be updated to a new version.

21.3.1 Goals

21.3.1.1 Goal `list`

List all dependencies, plugins, managed dependencies and managed plugins and list all atifacts that are not up-to-date, showing the newer available versions.

21.3.1.1.1 Properties

- `listAll` List all artifacts, including those that are up-to-date (*optional, defaults to* `false`)

21.3.1.2 Goal `overview`

Go through all projects in the reactor and list the dependencies and and plugins. Managed plugins and dependencies are not listed as they are inherited and therefore appear typically through the whole reactor.

21.3.1.2.1 Properties

- *The same properties as for the goal* `list` *are available*

21.3.1.3 Goal `version`

List available versions for a given artifact. This goal uses the remote repositories of the project in the current directory, so results may differ if executed at different places.

21.3.1.3.1 Properties

- `artifactid` The ID of the artifact to be searched
- `groupid` The ID of the group of the artifact to be searched
- `scope` The scope the artifact has to be in (*optional, defaults to* `runtime`)
- `type` The type of the searched artifact (*optional, defaults to* `jar`)

21.3.1.4 Known Issues

- When searching for old plugins some plugins, some do not appear as plugins but only in the pluginManagement (It's assumed the ones missing are not explicitly listed in the pom, but called as they're bound to a lifecycle phase. E.g. maven-war-plugin)
- When executing the plugin only those plugins and dependencies are considered that belong to an active profile

21.3.1.5 Example Output

21.3.1.5.1 `version:list`

```
$ mvn version:list -N
...
[INFO] [version:list]
[INFO] Artifact ID: maven-javadoc-plugin
[INFO] Group ID: org.apache.maven.plugins
[INFO] Version: 2.1-20061003.094811-3
[INFO]  Newer Versions:
[INFO]          2.2
[INFO]  Occurences in:
[INFO]          "EL4J" as PluginManagement
[INFO]
[INFO] Artifact ID: junit
[INFO] Group ID: junit
[INFO] Version: 3.8.2
[INFO]  Newer Versions:
[INFO]          4.0
[INFO]          4.1
[INFO]          4.2
[INFO]  Occurences in:
[INFO]          "EL4J" as DependecyManagement
[INFO]
[INFO] Artifact ID: maven-war-plugin
[INFO] Group ID: org.apache.maven.plugins
[INFO] Version: 2.0.2-20060907.100703-1
[INFO]  Newer Versions:
[INFO]          2.0.2
[INFO]  Occurences in:
[INFO]          "EL4J" as PluginManagement
```

21.3.1.5.2 `version:overview`

```
$ mvn version:overview
...
[INFO]    task-segment: [version:overview] (aggregator-style)
...
[INFO] Artifact ID: jamon
[INFO] Group ID: com.jamonapi
[INFO] Version: 1.0
[INFO]  Newer Versions:
[INFO]          2.0
[INFO]  Occurences in:
[INFO]          "EL4J module light statistics common" as Dependency
[INFO]
[INFO] Artifact ID: acegi-security
[INFO] Group ID: org.acegisecurity
[INFO] Version: 1.0.1
[INFO]  Newer Versions:
[INFO]          1.0.2
[INFO]  Occurences in:
[INFO]          "EL4J module security" as Dependency
[INFO]
[INFO] Artifact ID: jaxrpc
[INFO] Group ID: javax.xml
[INFO] Version: 1.1
[INFO]  Newer Versions:
[INFO]          2.0
[INFO]  Occurences in:
[INFO]          "EL4J module remoting soap" as Dependency
[INFO]
[INFO] Artifact ID: plexus-utils
[INFO] Group ID: org.codehaus.plexus
[INFO] Version: 1.2
[INFO]  Newer Versions:
```

```
[INFO]          1.3-SNAPSHOT
[INFO]  Occurences in:
[INFO]          "EL4J plugin helper for maven repositories" as Dependency
[INFO]          "EL4J plugin decorator for manifest files" as Dependency
[INFO]          "EL4J plugin collector for files" as Dependency
[INFO]
[INFO] Artifact ID: exec-maven-plugin
[INFO] Group ID: org.codehaus.mojo
[INFO] Version: 1.0.1
[INFO]  Newer Versions:
[INFO]          1.0.2
[INFO]  Occurences in:
[INFO]          "EL4J website" as Plugin
```

21.3.1.5.3 `version:version`

```
$ mvn version:version -Dversion.artifactid="spring" -Dversion.groupid="org.springframework" -N
...
[INFO] [version:version]
[INFO] Using the currenct projects "EL4J" repositories.
[INFO] Used repositories:
[INFO]  el4jReleaseRepositoryExternalhttp://el4.elca-services.ch/el4j/maven2repository
[INFO]  el4jSnapshotRepositoryExternalhttp://el4.elca-services.ch/el4j/maven2snapshots
[INFO]  el4jReleaseRepositoryInternalhttp://leaffy.elca.ch/java/maven2repository
[INFO]  centralhttp://repo1.maven.org/maven2
[INFO] Artifact ID: spring
[INFO] Group ID: org.springframework
[INFO] Scope: runtime
[INFO] Type: jar
[INFO]  1.0-m4
[INFO]  1.0-rc1
[INFO]  1.0
[INFO]  1.1-rc1
[INFO]  1.1-rc2
[INFO]  1.1
[INFO]  1.1.1
[INFO]  1.1.2
[INFO]  1.1.3
[INFO]  1.1.4
[INFO]  1.1.5
[INFO]  1.2-rc1
[INFO]  1.2-rc2
[INFO]  1.2
[INFO]  1.2.1
[INFO]  1.2.2
[INFO]  1.2.3
[INFO]  1.2.4
[INFO]  1.2.5
[INFO]  1.2.6
[INFO]  1.2.7
[INFO]  1.2.8
[INFO]  2.0-m2
[INFO]  2.0-m4
[INFO]  2.0-rc4-snapshot-patched-el4j-20060830
[INFO]  2.0-rc4-snapshot-patched-el4j-20060831
[INFO]  2.0-rc4
[INFO]  2.0
[INFO]  2.0.2
```

21.4 Environment plugin

Here is the documentation of the EL4J env plugin:
http://el4j.sourceforge.net/plugins/maven-env-support-plugin/index.html

Why is this plugin required? We put here the mail argumentation of MZE (in a rather raw form).

21.4.1 Question 1

> J'ai une petite question sur maven-env-plugin. > > Quel est le but exact de ce plugin, et pourquoi ne pas > utiliser les fonctions de filtrage supportées par maven ?

21.4.2 Answer 1

This plugin was made due to the following reasons:

- The file that would be need to be filtered can be placed in the normal resource directory. If you just use the Maven filtering you must enable filtering for the complete resource directory but this can cause unmeant replacements. The filtered placeholders are under control (Maven have a lot of properties that would be used for filtering).
- The environment can be changed in one file without building the hole project (i.e. easily adapt delivered WAR/EAR at costumer side).
- You can easily get information about the used environment on runtime (see ch.elca.el4j.util.env.EnvPropertiesUtils).

21.4.3 Question 2

For the first point, using maven you can define as many resource folder as you want determining for each folder you want to enable filtering or not.

I don't understand the second point. Since resource are usually bundled with war/ear You generally have to make rebuild Unless you're using JNDI customisation, or you can customize the classpath to force the server to load preferably a property files outside of the war Or can you do it in another way?

21.4.4 Answer 2

1. It's correct that you can define multiple resource folders, but you have to specially move the resources into this folder (further loose history in CVS) if they suddenly need to be filtered. That's ugly.
2. Clear you can rebuild but if you have already multiple environments prepared you can save time. The general point is not to loose the placeholders by replacing them in the resource files with their values.

22 Acknowledgments

There are many persons that have contributed to EL4J (in alphabetical order):

- Christoph Bäni
- Frank Bitzer
- Raphael Boog
- Simon Börlin
- Laurent Bovet
- Andi Bur
- Reto Fankhauser
- Christian Gasser
- Adrian Häfeli
- Jacques-Olivier Haenni
- Claude Humard
- Philippe Jacot
- Marc Lehmann
- Yves Martin
- Alex Mathey
- Martin Meier
- Adrian Moos
- Philipp Oesch
- Philipp H. Oser
- Andreas Pfenninger
- Stefan Pleisch
- Jean-François Poilpret
- Nicola Schiper
- Marc Schmid
- Christoph Schwitter
- David Stefan
- Florian Süss
- Michael Vorburger
- Rachid Waraich
- Stefan Wismer
- Martin Zeltner
- Martin Zingg

Thank you!

23 References

- EL4J Website, http://el4j.sourceforge.net/
- Commercial EL4J Website, http://www.elca.ch/live/3/home/Solutions/EL4J.php
- Professional Java Development with the Spring Framework; Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, Colin Sampaleanu; wrox; 2005