TECHNOLOGY CONSULTING INNOVATION

ELCA

Reference Documentation

EL4J 3.1

Imput.	Report	Version	Date	Author(s)	Status	Visa
6220	EL4J	3.1	23.12.10			
6220	EL4J	1.7	15.12.09	POS, MZE, SWI, DZI, JHN	final	

Table of Contents

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.

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, ...
- JSF framework: an integration based on Seam, Facelets, Ajax4Jsf? , and Richfaces. It does not require EJB3 (is is based on Spring).
- 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 2.5.1 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
- Seam
- JSF
- Struts
- JaMon
- Quartz

Maven 2 and installing modules

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).

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.

Documentation for module core

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.

edit purpose

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:

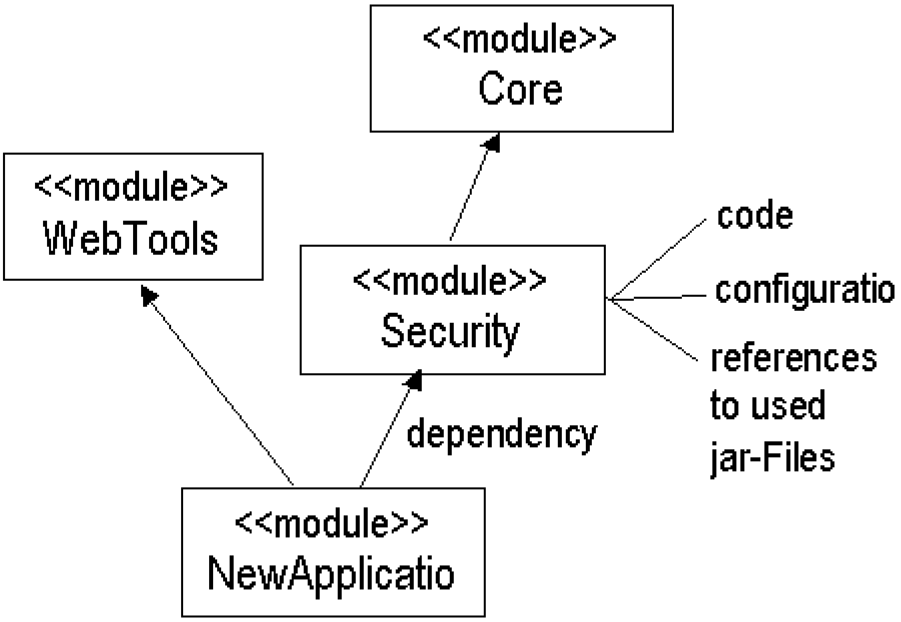
- the module abstraction of Maven 2
- the ModuleApplicationContext (a wrapper for the standard Spring application context)
- a convention on how to organize configuration information within each module

These three parts are described in the next sections.

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.

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.

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. Choose the bean names carefully to avoid conflicts! Ensure that all the beans that you define in here do not disturb when present (it may sometimes be better to use the directory '/mandatory/<module-name>' instead).

- '/mandatory/<module-name>': Contains all xml files that are necessary to use the module as it is (standard configuration). An application that uses this module is free to include this folder or provide a modified version.

- '/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 various 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.

Examples

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

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.

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.

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.

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.

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.

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/Transactional.html
  - 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`.

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.

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.*).

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()**.

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

DeprecatedConvenienceAttributesForTransactions

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.

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.DefaultAdvisorAutoProxyC
reator"/>



    <!-- 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.ExampleAnnotatio
nOne</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.ExampleIntercept
or">
    </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

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;
}
```

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.

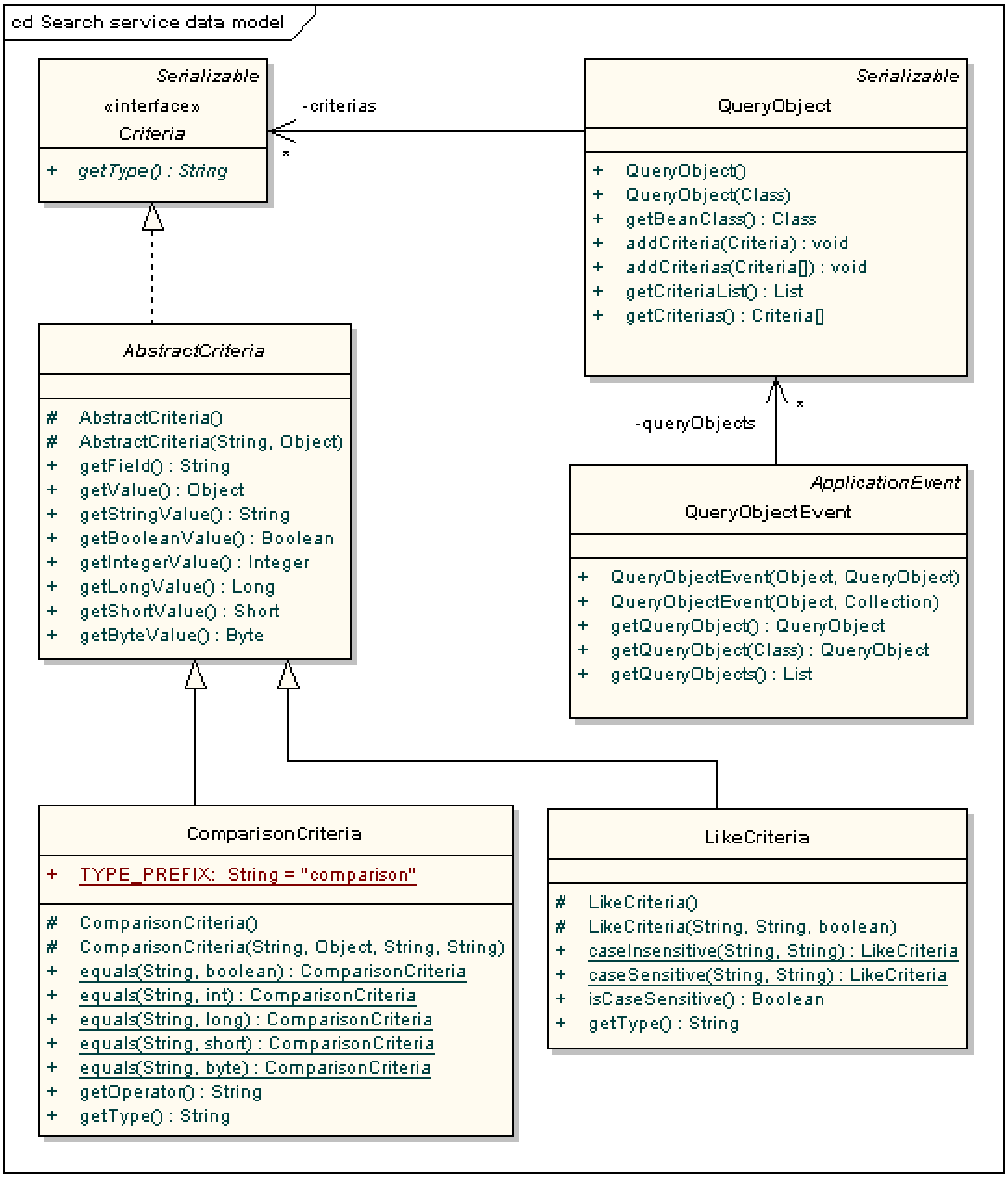
- @ExampleAnnotationTwelve(?Overwritten?)

Annotation defined by the interface FooBase? is overwritten by the method in class FooImpl? .

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.


cd Search service data model
Serializable
«interface»
Criteria
+ getType() : String
-criterias
*
Serializable
QueryObject
+ QueryObject()
+ QueryObject(Class)
+ getBeanClass() : Class
+ addCriteria(Criteria) : void
+ addCriterias(Criteria[]) : void
+ getCriteriaList() : List
+ getCriterias() : Criteria[]
AbstractCriteria
# AbstractCriteria()
# AbstractCriteria(String, Object)
+ getField() : String
+ getValue() : Object
+ getStringValue() : String
+ getBooleanValue() : Boolean
+ getIntegerValue() : Integer
+ getLongValue() : Long
+ getShortValue() : Short
+ getByteValue() : Byte
-queryObjects
*
ApplicationEvent
QueryObjectEvent
+ QueryObjectEvent(Object, QueryObject)
+ QueryObjectEvent(Object, Collection)
+ getQueryObject() : QueryObject
+ getQueryObject(Class) : QueryObject
+ getQueryObjects() : List
ComparisonCriteria
+ TYPE_PREFIX: String = "comparison"
# ComparisonCriteria()
# ComparisonCriteria(String, Object, String, String)
+ equals(String, boolean) : ComparisonCriteria
+ equals(String, int) : ComparisonCriteria
+ equals(String, long) : ComparisonCriteria
+ equals(String, short) : ComparisonCriteria
+ equals(String, byte) : ComparisonCriteria
+ getOperator() : String
+ getType() : String
LikeCriteria
# LikeCriteria()
# LikeCriteria(String, String, boolean)
+ caseInsensitive(String, String) : LikeCriteria
+ caseSensitive(String, String) : LikeCriteria
+ isCaseSensitive() : Boolean
+ getType() : String

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.

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**.

Additional Features

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"/>
```

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

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.

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>
```

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.html
- http://el4j.sourceforge.net/framework-modules/apidocs/ch/elca/el4j/util/codingsupport/AbstractComparableEnum.html

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

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`).

Generic configuration

See ModuleCoreGenericConfig

Pattern and Interface for the implementation of codelists

See CodeListsEnumPattern

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.

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.

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].

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`.

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.

Documentation for module remoting

Purpose

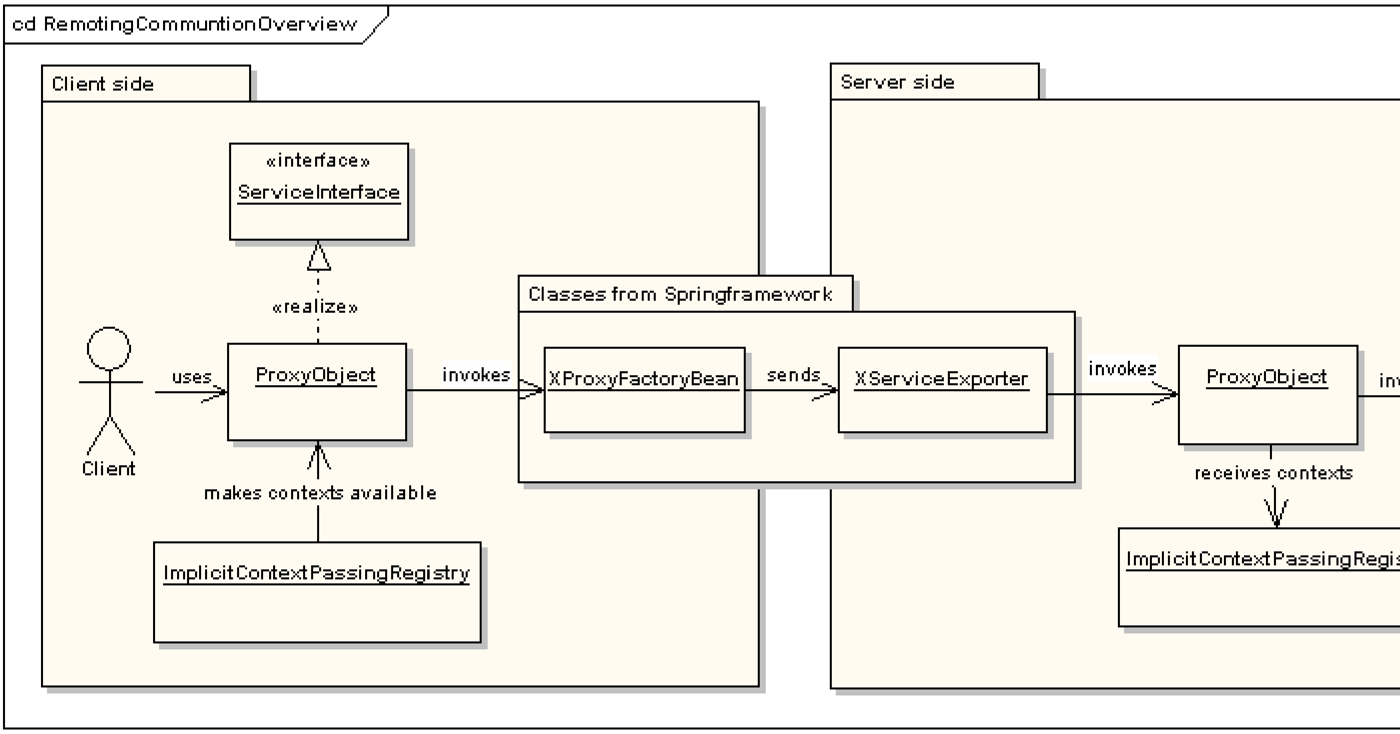
Convenience module for spring POJO remoting: (1) allows **centralized protocol configuration**, (2) simplifies protocol switching (currently between **RMI**, **HttpInvoker**, **Hessian**, **Burlap**, **Soap**, **Jax-WS** and **EJB**), and (3) transparently enriches interfaces for **automatic implicit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

edit purpose

Introduction

The Spring framework offers an easy way to distribute POJOs. Available protocols are `Rmi`, `Hessian` and `JAX-WS`. 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!

Remoting modules

Currently there are six modules for remoting:

- The core remoting module with name `module-remoting_core` contains the protocols **RMI**, **HttpInvoke** (of Spring), and **composite** protocols (see below).
- For **Hessian** you have to use `module-remoting_caucho`.
- Our web service stack based on **JAX-WS** can be found in ModuleRemotingJaxws.
- For the `EJB 2.0` remoting protocol you have to use ModuleRemotingEjb (currently not working as there was little desire and EJB 2.1 is legacy today).
- Our web service stack based on **XFire** can be found in `module-remoting_xfire` (deprecated).

- The old web service stack (based on Axis 1) can be found in=module-remoting_soap= (deprecated).

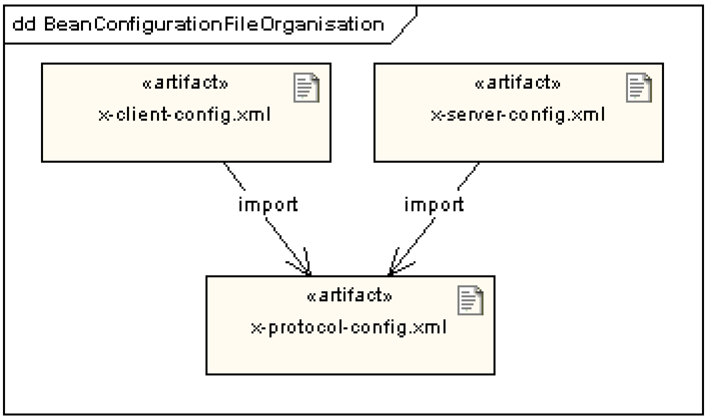
In addition, there exists a composite protocol (it uses the composite pattern) that supports load balancing:

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

How to use

Basic configuration

Recommended configuration file organization



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` or `hessian`.

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.TestImplicitContextPassingRegist
ry" />
</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 predefined protocols (`hessian` and `http-invoker`) need additionally the property `contextPath` that defines in which webserver context the service is running.

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.

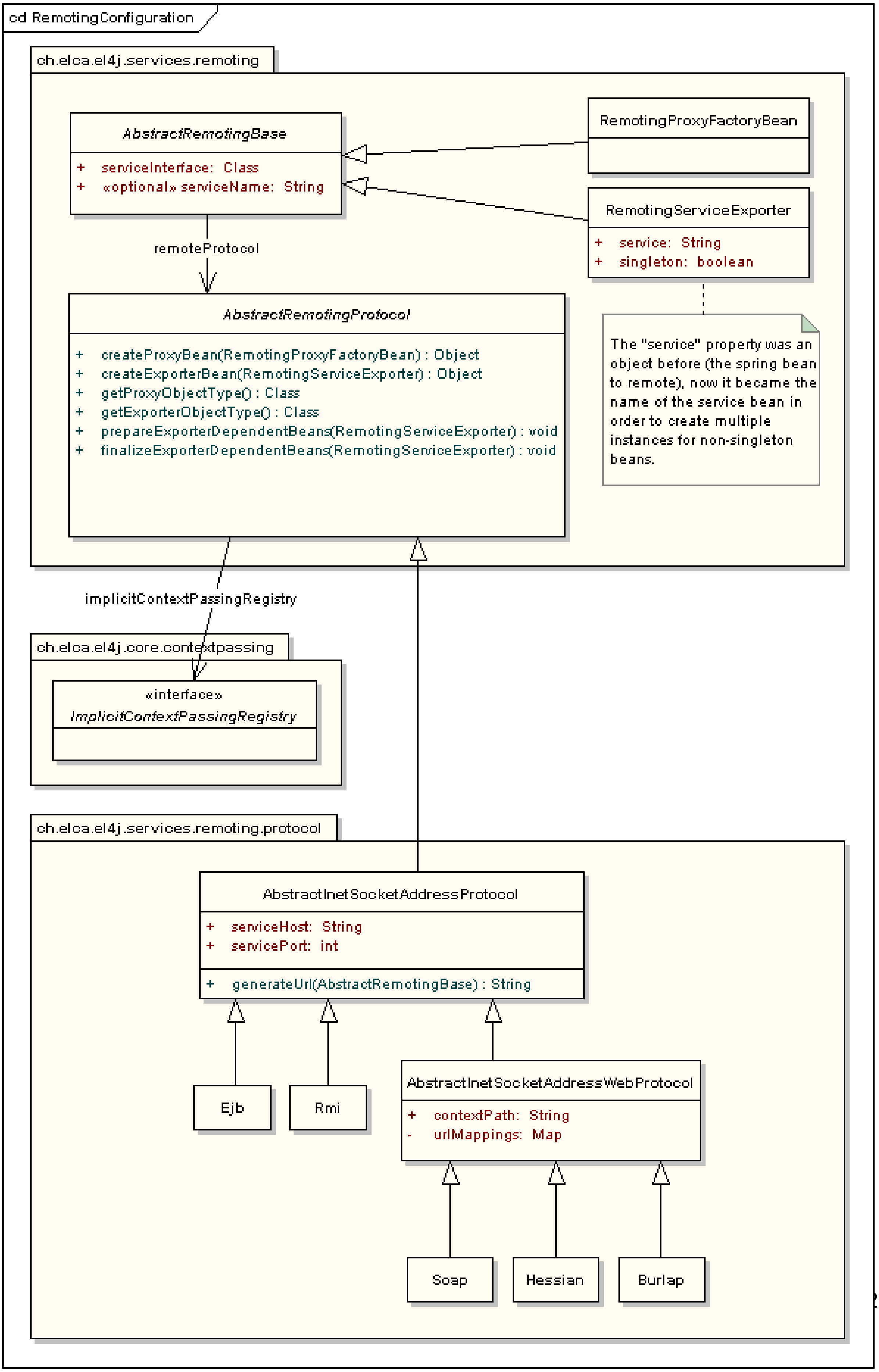
x-server-config.xml

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.

Configuration summary


cd RemotingConfiguration
ch.elca.el4j.services.remoting
AbstractRemotingBase
+ serviceInterface: Class
+ «optional» serviceName: String
RemotingProxyFactoryBean
RemotingServiceExporter
+ service: String
+ singleton: boolean
remoteProtocol
AbstractRemotingProtocol
+ createProxyBean(RemotingProxyFactoryBean) : Object
+ createExporterBean(RemotingServiceExporter) : Object
+ getProxyObjectType() : Class
+ getExporterObjectType() : Class
+ prepareExporterDependentBeans(RemotingServiceExporter) : void
+ finalizeExporterDependentBeans(RemotingServiceExporter) : void
The "service" property was an object before (the spring bean to remote), now it became the name of the service bean in order to create multiple instances for non-singleton beans.
implicitContextPassingRegistry
ch.elca.el4j.core.contextpassing
«interface»
ImplicitContextPassingRegistry
ch.elca.el4j.services.remoting.protocol
AbstractInetSocketAddressProtocol
+ serviceHost: String
+ servicePort: int
+ generateUrl(AbstractRemotingBase) : String
Ejb
Rmi
AbstractInetSocketAddressWebProtocol
+ contextPath: String
- urlMappings: Map
Soap
Hessian
Burlap

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` or `Hessian`.

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 is one class that directly extends this abstract class, the **`Hessian`** protocol.

How to use the `Rmi` protocol

The introduction of the remoting module in the previous section 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.

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.

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.

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`.

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.DefaultImplicitContextPassingRegist
ry" />
</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.

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.

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.

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.

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`.

How to use the web service protocol based on JAX-WS 2.1

The EJB protocol support is available in the ModuleRemotingJaxws.

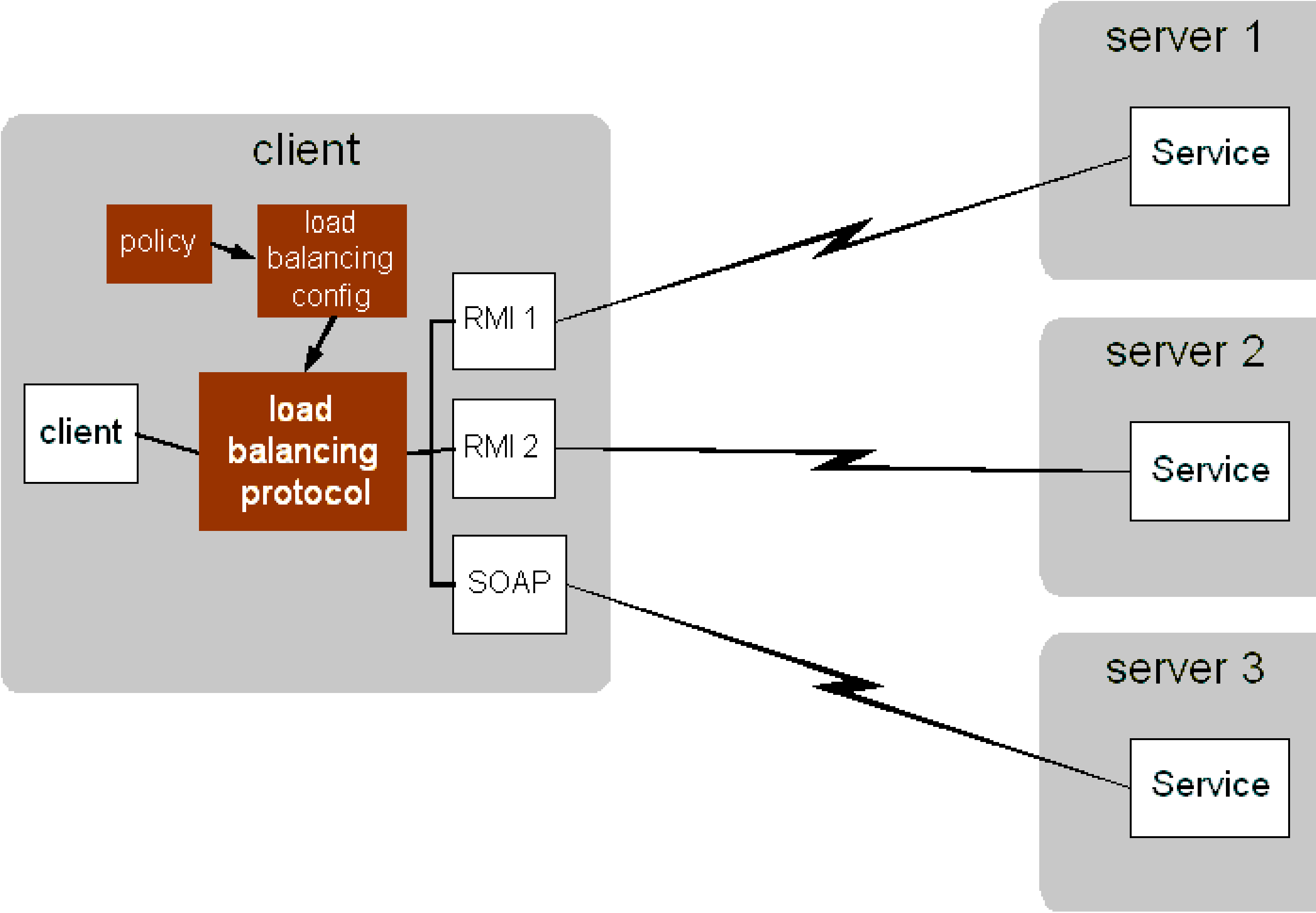
How to use the `EJB` protocol

The EJB protocol support is available in the ModuleRemotingEjb.

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.protocol.LoadB
alancingProtocol">
      <property name="protocolSpecificConfiguration">
         <ref bean="loadBalancingProtocolConfiguration" />
      </property>
   </bean>

   <bean id="loadBalancingProtocolConfiguration"

class="ch.elca.el4j.services.remoting.protocol.loadbalancing.protocol.LoadB
alancingProtocolConfiguration">
      <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.

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.

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*: There is an ordered list of servers through which the load balancer loops, distributing the invocations over all servers. *Example:* p_1 -> p_2 -> p_3 -> p_1 -> p_2 -> ...
- *redirectuponfailure*: This policy is similar to the round robin one. However, the server is only changed if a call fails. Therefore, the same server is used until an error occurs. *Example:* 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.

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.

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.

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.DefaultImplicitContextPassingRegist
ry"/>

<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.

Use of ThreadLocal

It is strictly forbidden to use `ThreadLocal` in combination with implicit context passing! The retry interceptor uses a new child thread for the actual execution of the invoked method. In this child thread, all context defined using a `ThreadLocal` in the parent thread will not be available. Therefore, **implicit context passing does not work when a `ThreadLocal` is used**. With an `InheritableThreadLocal` (which subclasses `ThreadLocal`), however, implicit context passing works as the child thread receives initial values from it's parent thread if `InheritableThreadLocal` is used. Thus, it is strongly recommended to use `InheritableThreadLocal` and not `ThreadLocal`.

Benchmark

The module **module-remoting-demos** contains a benchmark for various remoting protocols. 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 result of the benchmark running on the Laptop of POS, an Intel T2400, Dual core with 1.83 Ghz and 2 GB of memory:

```
 ---------------------------------------------------------------------------
-------------------------------------
 | *Name of test*                       | *Method 1 [ms]* | *Method 2 [ms]*
| *Method 3 [ms]* | *Method 4 [ms]* |
 ---------------------------------------------------------------------------
-------------------------------------
 | rmiWithoutContextCalculator          | 1.084           | 1.622
| 4.991           | 0.419           |
 ---------------------------------------------------------------------------
-------------------------------------
 | rmiWithContextCalculator             | 1.119           | 1.875
| 5.394           | 0.478           |
 ---------------------------------------------------------------------------
-------------------------------------
 | hessianWithoutContextCalculator      | 0.678           | 1.637
| 24.398          | 0.847           |
 ---------------------------------------------------------------------------
-------------------------------------
 | hessianWithContextCalculator         | 1.081           | 2.384
| 25.454          | 1.075           |
 ---------------------------------------------------------------------------
-------------------------------------
```

```
 | httpInvokerWithoutContextCalculator | 1.353           | 3.031
| 6.931           | 1.3             |
 ---------------------------------------------------------------------------
-------------------------------------
 | httpInvokerWithContextCalculator    | 1.381           | 3.294
| 6.65            | 1.35            |
 ---------------------------------------------------------------------------
-------------------------------------

Legend:        Method 1: double getArea(double a, double b)
               Method 2: void throwMeAnException() throws
CalculatorException
               Method 3: int countNumberOfUppercaseLetters(String
textOfSize60kB)
               Method 4: ComplexNumber add(ComplexNumber cn1,ComplexNumber
cn2)
```

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

Remoting semantics/ Quality of service of the remoting

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:

Desired semantics /implementation			**Rmi**	**Hessian**	**Burlap**	**Soap**	**EJB**
Singleton	POJO is reentrant		Standard use				N/A
	POJO is not reentrant	synchronized	By using an interceptor in or synchronizing in code				
		pooled	By using spring's pooling target source (see spring doc)				Stateless
Client-activated			*TODO: Is currently not implemented.*				Statefull

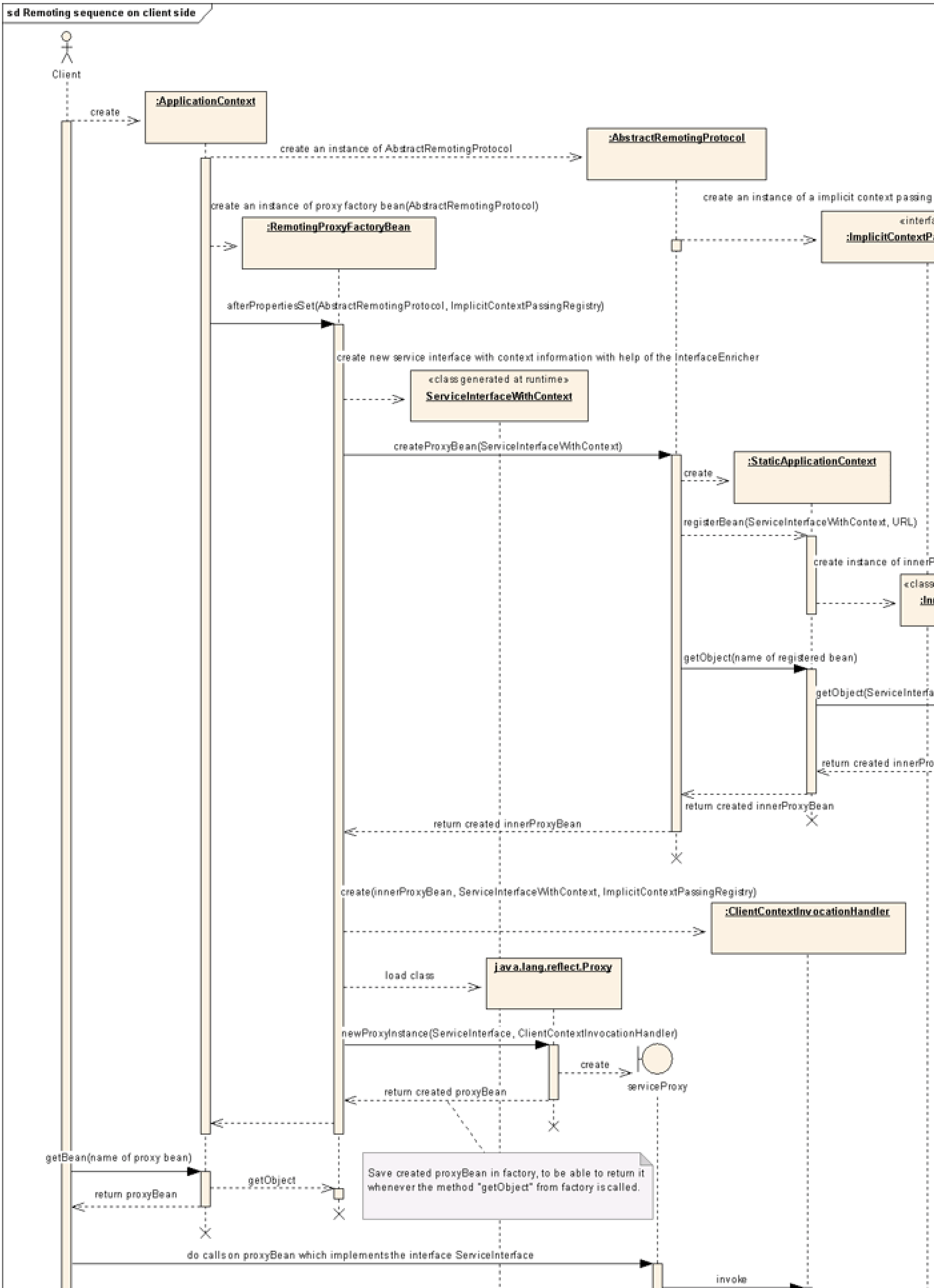
What happens when there is a timeout or another problem during remoting

The following document describes what happens in more details. It has been contributed by VISA{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.

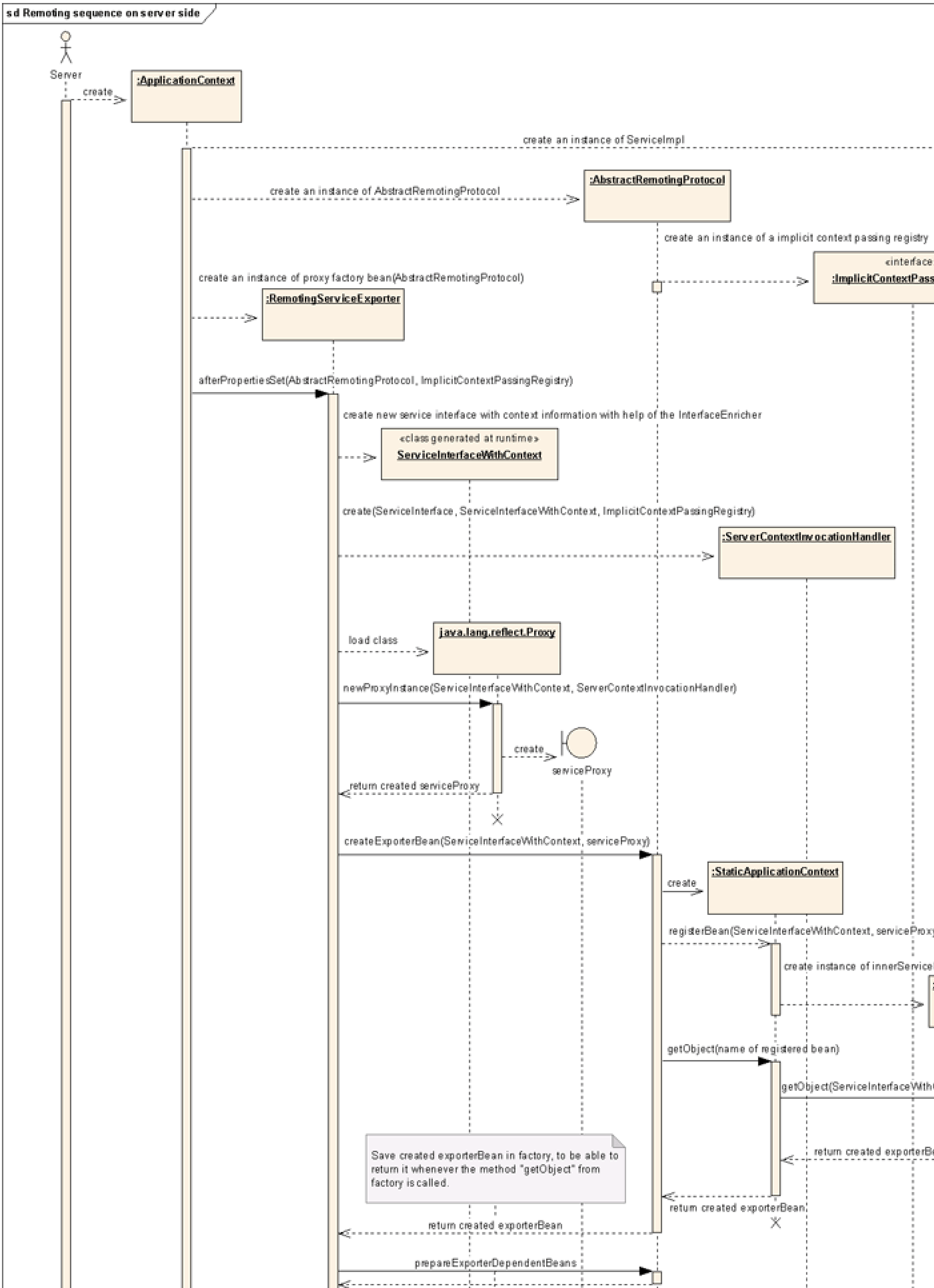
Internal design

Sequences

Sequence diagramm from client side


sd Remoting sequence on client side
Client
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
«interfa
:ImplicitContextP
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 innerP
«class
:In
getObject(name of registered bean)
getObject(ServiceInterfa
return created innerPro
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

Sequence diagramm from server side


sd Remoting sequence on server side
Server
create
:ApplicationContext
create an instance of ServiceImpl
:AbstractRemotingProtocol
create an instance of AbstractRemotingProtocol
create an instance of a implicit context passing registry
«interface
:ImplicitContextPass
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, serviceProx
create instance of innerService
getObject(name of registered bean)
getObject(ServiceInterfaceWith
return created exporterBe
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

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.

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)

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.

Related frameworks

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

Javaworld 2005 idea

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

Documentation for module Jax-WS remoting

JAX-WS (Java API for XML Web Services) is the successor of the JAX-RPC API. It is part of the Java EE 5 API and its reference implementation can be found at https://jax-ws.dev.java.net/ .

Purpose

Convenience module to use and provide Jax-WS web services. This module extends the ModuleRemoting, i.e. supports the same remoting features and allows switching from one of the other protocols to Jax-WS and vice versa.

edit purpose

How to use

Adaptations for Java 6

For the impatient: Download jaxb-api.jar and copy it into `C:\jdk1.6.0_12\jre\lib\endorsed` (adapt path to your JDK).

JAX-WS 2.1 uses JAXB 2.1 which conflicts with the version 2.0 included in JavaSE 6. Migrating JAXB 2.0 to JavaSE 6 section 7.1.2 explains how to solve this problem. **Attention**: If files are copied to lib/endorsed, check that the filenames do not contain version numbers (e.g. there must be jaxb-api.jar, not jaxb-api-2.1.jar).

Notes on JAX-WS server deployment

JaxWsSpringServerDeploymentNotes

Two usage scenarios and their differences

The probably most important to understand this documentation is that one should note that there are basically two different usage scenarios when talking about JAX-WS. Even tough both of them have some similarities in their configuration, the underlying scope is completely different.

Client only (pure consuming of a provided webservice)

In this scenario, the intention is to write a client application for a given, provided webservice which is perhaps not even under our control. For this target webservice, a WSDL must be provided and accessible over http. Using the wsimport tool, we generate client java classes from the WSDL file.

Server and Client (create a new webservice and use id)

In this scenario, the intention is to write a webservice by ourselves – and also use it. To make things a little more (or a little less) understandable, this goal can be achieved in two different ways:

- **Method 1** Write java code (interfaces and implementation) with corresponding JAX-WS annotations (@WebService), then automatically create a WSDL using the wsgen tool and finally generate client java classes from this WSDL as in the *Client only* scenario. This is especially useful if the source code of the server is (or should) not be available to clients.

- **Method 2** Write java code (interfaces and implementation) with corresponding JAX-WS annotations and access the service over the java service interface from the server tier using the server interfaces specified before (remoting method). If you control both the client and the server, you typically don't want to work with generated classes but use the ones that you have written (service interfaces, data structures and so on). In this case we choose the remoting mode that uses the JAX-WS support provided by Spring.

In the following subsections, this icon indicates a reference to code / modules inside the EL4J framework. For a better understanding of JAX-WS, we encourage you to look at the referenced code samples.

But before we can describe the two scenarios in more detail, we have to take a look at the common configuration parts for both potential usage scenarios.

Common configuration files

The protocol configuration file (jaxws-protocol-config.xml)

There are two JAX-WS protocols available (both with and without implicit context passing):

Jaxws and JaxwsSoapHeaderContextPassing

Protocols to be used whenever code is generated using `wsgen` and / or `wsimport` (Scenario Client only or Scenario Server & Client, Method 1).

```
<?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/beans/spring-beans-2.0.xsd">

    <!-- JAX-WS Soap Protocol -->
    <bean id="jaxwsProtocol"
class="ch.elca.el4j.services.remoting.protocol.JaxwsSoapHeaderContextPassing">
        <!--
            Properties "serviceHost" and "servicePort" are ignored.
            They have to be specified in the configuration section of
maven-jaxws-plugin.
        -->
        <property name="serviceHost">
            <value>ignored</value>
        </property>
        <property name="servicePort">
            <value>0</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.DefaultImplicitContextPassingRegist
ry" />

    <!-- 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 `contextPath` and `implicitContextPassingRegistry`: the `jaxwsContextPassingContext` used for the implicit context passing. Every SOAP message that gets transmitted automatically (therefore implicitly) contains this value. This is how you can share implicit context between a client and a server.

JaxwsSpring and JaxwsSpringSoapHeaderContextPassing

Protocols to be used for clients that work with the Java interfaces of the server (Scenario Server & Client, Method 2) using the JAX-WS remoting mode provided by Spring.

```
<?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/beans/spring-beans-2.0.xsd">

    <!-- JAX-WS Soap Protocol for clients that do not need generated code -
->
    <bean id="jaxwsSpringProtocol"

class="ch.elca.el4j.services.remoting.protocol.JaxwsSpringSoapHeaderContext
Passing">
        <property name="serviceHost">
            <value>${jee-web.host}</value>
        </property>
        <property name="servicePort">
            <value>${jee-web.port}</value>
        </property>
        <property name="contextPath">
            <value>${jee-web.context}</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.DefaultImplicitContextPassingRegist
ry" />

    <!-- 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>
```

This requires at least EL4J version 1.6. So you also have to add the following protocol (for the client part). It is not very nice to have different EL4J protocols on client and server side, but the pure Spring JAX-WS server implementations have some limitations documented in SimpleJaxWsServiceExporter and SimpleHttpServerJaxWsServiceExporter. However, if they don't apply to your project, using only the Spring JAX-WS protocol might be an alternative.

For a fully featured example including a configuration for both protocols, look at `src/main/resource/scenarios/common/remotingtests-jaxws-protocol-config.xml` in the `module-remoting-jaxws-test-jar-wsgen` (`framework/tests/remoting_jaxws/jar-wsgen`).

The client configuration file (jaxws-client-config.xml)

As for the protocol configuration file (jaxws-protocol-config.xml), there are two versions for the client configuration depending on the choosen method to access the service:

Configuration when using generated java classes using wsimport (Scenario Client only or Scenario Server & Client, Method 1)

```
<?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/beans/spring-beans-2.0.xsd">

    <import resource="jaxws-protocol-config.xml" />
    <!-- JAX-WS Setup for classes generated by wsimport. -->
    <bean id="calculator"
        class="ch.elca.el4j.services.remoting.RemotingProxyFactoryBean">
        <property name="remoteProtocol">
            <ref bean="jaxwsProtocol" />
        </property>
        <property name="serviceInterface">
            <value>

ch.elca.el4j.tests.remoting.service.service.gen.CalculatorWS
            </value>
        </property>
        <property name="serviceName">
            <value>Calculator.Jaxws.Remotingtests</value>
        </property>
    </bean>
</beans>
```

For a fully featured example configuration, look at `src/main/resource/scenarios/client/remotingtests-jaxws-shakespeare-config.xml` in the `module-remoting-jaxws-test-jar-wsimport` (`framework/tests/remoting_jaxws/jar-wsimport`).

Configuration when not using generated java classes (remoting mode) (Scenario Server & Client, Method 2)

```
<?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/beans/spring-beans-2.0.xsd">

    <import resource="jaxws-protocol-config.xml" />
    <!-- JAX-WS Setup for clients that do not use generated code (using
Spring remoting) -->
    <bean id="calculator"
        class="ch.elca.el4j.services.remoting.RemotingProxyFactoryBean">
        <property name="remoteProtocol">
            <ref bean="jaxwsSpringProtocol" />
        </property>
        <property name="protocolSpecificConfiguration">
            <ref local="jaxwsProtocolSpecificConfiguration" />
        </property>
        <property name="serviceInterface">

<value>ch.elca.el4j.tests.remoting.jaxws.service.Calculator</value>
        </property>
        <property name="serviceName">
            <value>Calculator.Jaxws.Remotingtests</value>
        </property>
    </bean>

    <bean id="jaxwsProtocolSpecificConfiguration"
class="ch.elca.el4j.services.remoting.protocol.JaxwsSpringProtocolConfigura
tion">
        <property name="namespaceUri">

<value>http://gen.service.jaxws.remoting.tests.el4j.elca.ch/</value>
        </property>
        <property name="serviceName">
            <value>CalculatorWSService</value>
        </property>
        <property name="portName">
            <value>CalculatorWSPort</value>
        </property>
    </bean>
</beans>
```

The properties values in the bean 'jaxwsProtocolSpecificConfiguration' have to match the values in the @WebService annotation.

For a fully featured example configuration, look at `src/main/resource/scenarios/client/remotingtests-jaxws-client-config.xml` in the `module-remoting-jaxws-test-jar-wsgen` (`framework/tests/remoting_jaxws/jar-wsgen`).

Client only (pure consuming of a provided webservice)

In this scenario, the intention is to write a client application for a given, provided webservice which is perhaps not even under our control. For this target webservice, a WSDL is provided and accessible over http. In the following example, we use the Shakespeare webservice from xmlme.com as the service provider (Link to the WSDL file of service)

As mentioned before, we use the help of the wsimport tool to generate client java classes from the WSDL file. As described in the development section, the wsimport tool is integrated into the maven build process with the EL4J `maven-jaxws-plugin`. Therefore, the configuration for the java code generation can be inserted directly inside projects pom.xml file in the plugin configuration.

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

```
<wsdlUrl>http://www.xmlme.com/WSShakespeare.asmx?WSDL</wsdlUrl>
               </wsdlUrls>
            </configuration>
         </execution>
      </executions>
   </plugin>
```

When building the project using `mvn clean install`, the WSDL file inside is fetched and the java client code is generated for the project.

The module `module-remoting-jaxws-test-jar-wsimport` (`framework/tests/remoting_jaxws/jar-wsimport`) adopts this technique using the previously mentioned Shakespeare webservice. The generated java classes can be found under `target/jaxws/wsimport/java`.

Server and Client (create a new webservice and use id)

In this scenario, the intention is to write a webservice by ourselves – and also use it.

As we also setup a server now, some additional configuration files must be present.

The server configuration file (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/beans/spring-beans-2.0.xsd">

    <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>
```

For a fully featured example configuration, look at `src/main/resource/scenarios/server/web/remotingtests-jaxws-server-config.xml` in the `module-remoting-jaxws-test-jar-wsgen` (`framework/tests/remoting_jaxws/jar-wsgen`).

Servlet configuration file (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.

For a fully featured example web.xml configuration, look at `src/main/webapp/WEB-INF/web.xml` in the `module-remoting-jaxws-tests-war` (`framework/tests/remoting_jaxws/war`).

Server and Client (Method 1)

After we have written the interfaces and the service implementation with the necessary JAS-WS annotations, we use the wsgen tool to generate the corresponding WSDL file for the service specified by the java interface.

Afterwards we create the client classes from the previously generated WSDL file with the help of the wsimport tool analogical – anolog in the client only scenario.

Even if this approach looks like an unnecessary indirection, it can be handy if the client classes have (or should) not have direct access to the server interface classes.

As described in the development section, the wsgen and wsimport tools are integrated into the maven build process with the EL4J `maven-jaxws-plugin`. Therefore, the configuration for the wsdl and java code generation can be inserted directly inside projects pom.xml file in the plugin configuration.

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

```
<wsdlDirectory>${project.build.directory}/jaxws/wsgen/wsdl</wsdlDirectory>
            </configuration>
         </execution>
      </executions>
   </plugin>
```

When building the project using `mvn clean install`, the WSDL file is generated in directory.

The module `module-remoting-jaxws-test-jar-wsgen` (`framework/tests/remoting_jaxws/jar-wsgen`) adopts this technique. The generated WSDL file can be found under `target/jaxws/wsgen/wsdl`. The generated java classes can be found under `target/jaxws/wsimport/java`.

Server and Client (Method 2)

If the client classes have direct access to the java interfaces defined for the server, we can basically skip the step of wsdl and client stubs generation. The service the is provided dynamically using the remoting mode of JAX-WS provided by Spring.

To achieve this, we basically just need the client configuration and then access the interface defined in the server using the (Spring) `ApplicationContext`.

The module `module-remoting-jaxws-tests-functional_tests` (`framework/tests/remoting_jaxws/functional-tests`) adopts this technique.

Avoid LazyInitializationExceptions while marshaling persisted objects

Detailed description: Persisted objects that have lazily initialized properties/references get wrapped using Hibernate Proxies. When such an object should be sent over the network, JAXB tries to serialize it. During this process it tries to access all the fields that have to included and therefore leads to a LazyInitializationException (because in general, the Hibernate session has already been closed). Simply setting the property to null is not a solution, because the object is therefore modified and Hibernate is willing to store it to the database.

The solution using module-jaxws (based on https://forum.hibernate.org/viewtopic.php?f=1&t=998896 and Do Phuong Hoang):

- Declare the improved AccessorFactory at package level inside the package-info.java:

```
@XmlAccessorFactory(HibernateJAXBAccessorFactory.class)
package ch.elca.your.project.dom;

import com.sun.xml.bind.XmlAccessorFactory;
import
ch.elca.el4j.services.remoting.jaxb.hibernate.HibernateJAXBAccessorFactory;
```

- Annotate the web service implementation with `@UsesJAXBContext(JAXBContextFactoryImpl.class)`

To control which values should be loaded, it is recommended to use the DataExtent feature described in HibernateGuidelines#ControlLoading

Implementation constraints (for EL4J version 1.5.1 and below)

Remark: Starting with EL4J version 1.6 these constraints do `NOT` apply anymore.

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.

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 ...), if you use Spring JAX-WS for the client part also annotate the interface
- 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>
```

```
                            <genWsdl>true</genWsdl>
                            <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 later 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 ...), if you use Spring JAX-WS for the client part also annotate the interface
- 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:

- All required stubs are generated by the maven-jaxws-plugin (see example configuration shown above). If you use Spring JAX-WS, you don't need the wsimport goal.
- Get the client stub (e.g. using `getBean("Calculator")`). You will then either get access directly to the generated classes or (if you use Spring JAX-WS) Spring will create proxies for you (depending on the chosen protocol)

WS security with JBoss

Successful tested tutorial:
http://www.developer.com/java/other/article.php/10936_3802631_1/Securing-Web-Services-in-JBoss-Application-Server-with-WS-Security.htm

Known limitations

- Spring and JAXWS: No mixing of @Webmethod and @Transactional Annotations allowed (see last post for work-around)

Documentation for module EJB remoting

Remark: in the current version of el4j, this module is not kept up to date. We consider the EJB 2-like support not so important any more.

Purpose

Convenience module to expose spring beans as **EJB session beans**. This module extends the ModuleRemoting, i.e. supports the same remoting features and allows switching from one of the other protocols to EJB and vice versa.

edit purpose

Important concepts

Remark: The EJB support is currently not working in EL4J 1.1 (due to the new build system). We will fix it as soon as possible. The documentation still is EL4Ant? -specific.

EL4J provides its own remoting services infrastructure (ModuleRemoting) that simplifies the use of remoting protocols supported by Spring. In addition, it adds support to pass an implicit context between the remote parties.

This module allows deploying Spring beans in EJB compliant containers, wrapping them transparently into session beans. Since most application containers do not allow creating enterprise beans at runtime, they have to be generated at deploy time and packed into an EAR. This is done transparently by the build system, if the needed plugin is activated.

How to use

Configuration

The protocol configuration is analogous to the description in the easy remoting module.

How to use the EJB protocol

Different from the other remoting protocols, where all remoting specific objects are created dynamically at runtime, most EJB containers demand beans to be available at deploy-time. This requires to generate session bean wrappers during the build process. EL4Ant supplies a module using XDoclet which automates this step. Apart form this speciality one can do as much as with the other protocols. However there are some constraints given by the EJB world. Currently, JBoss, WebLogic and WebSphere (not much tested yet) are supported.

Protocol definition

Additionally to the protocol-independent properties, EJB protocol definitions have another one specifying the JNDI environment. This is a Properties object defining the initial JNDI context. There are already such environment definitions for the three supported EJB containers, stored in the plugin's scenarios.

The EJB protocol uses a configuration object in order to specify service-related configurations. It contains a number of properties describing the EJB session bean's lifecycle, a mapping from EJB to service bean methods and a map to supply additional XDoclet tags.

Constraints

- Service beans wrapped in stateful session beans must live a **prototype** lifecycle. Beans wrapped in stateless session beans can be either declared as singletons or prototypes (be aware of what this means, you can create contention points!
- The service bean must implement `java.lang.Serializable` and all members it references (directly or indirectly) too. Note: using `writeObject` and `readObject` may help dealing with complicated situations.
- Exceptions thrown by service beans must be subclasses of `java.lang.Exception` (JBoss allows using `java.lang.Exception`, but WebLogic doesn't. see EJB 2.1 spec 18.1.1).

Method Mappings

EJB method name	property name	additional info / constraints

EJB method name	property name	additional info / constraints
ejbActivate()	activate	A method with `void` as return type and an empty argument list. All checked exceptions are wrapped into an unchecked one that aren't propagated to the client.
ejbPassivate()	passivate	
ejbRemove()	remove	
setSessionContext(SessionContext ctx)	sessionContext	A method that takes a `javax.ejb.SessionContext` as parameter
create(Object[] args)	create	There's only one custom create method available that takes an object array as parameter. Parameters are supplied through the `createArgument` property, a list. Any checked exceptions are wrapped in a `javax.ejb.CreateException`
afterBegin()	afterBegin	A method with `void` as return type and an empty argument list. All checked exceptions are wrapped into an unchecked to keep them on server side.
beforeCompletion()	beforeCompletion	
afterCompletion(boolean commit)	afterCompletion	A method that takes a `boolean` argument. All Exceptions are kept on server side.

Exceptions

All exceptions defined on the service interface are sent to the client. Additionally, all runtime exceptions thrown by the service bean are forwarded to the client too (wrapping and unwrapping is done transparently). We chose to do this for developer convenience. If the exception class does not exist on the client side, the string of the exception message is displayed. All other exceptions stay on the server side. Especially exceptions thrown during activation and passivation are wrapped into runtime exceptions and stay on server-side. The tests (`module-remoting_ejb-tets`) provide some examples.

Adding additional XDoclet tags

The EJB remoting plugin allows adding additional XDoclet tags or to override exisiting ones. They are specified through the configuration object that has to be specified on the RemotingServiceExporter as well as on the RemotingBeanFactory. Additional tags are provided by the `docletTags` property, which is a map and conform the following naming scheme:

- **class** add XDoclet tags on class level
- **null** add XDoclet tags to all methods
- **<methodName>** add XDoclet tags to all methods with the given name
- **<methodName(java.lang.String, int)>** add XDoclet tags to the method with the given signature (Note: consists of the method name and the paremeters' fully qualified types only -- no return type or variable names)

1.1.1.1.1.1 *Example*

```
<property name="docletTags">
    <map>
        <entry key="class">
            <value>@ejb.util generate="logical"</value>
        </entry>
        <entry>
            <key><null/></key>
            <value>@ejb.do-whatever foo="bar"</value>
        </entry>
        <entry key="foo">
            <value>@ejb.dao call="helloWorld"</value>
        </entry>
        <entry key="bar(java.lang.String, org.foo.bar.Foobar, int)">
            <value>@jboss.persistence datasource="foo" read-
only="false"</value>
        </entry>
        <entry key="passivate">
            <list>
                <value>@test arg="doit"</value>
                <value>@ejb.interface-method view-type="both"</value>
            </list>
        </entry>
   </map>
</property>
```

How to use the build system plugin

The EJB build system plugin adds two hooks to the project that generate the needed session beans transparently. In order to get them activated, one has to add the following attributes:

```
<attribute name="runtime.runnable" value="true"/>
<attribute name="j2ee.ear.application"/>
<attribute name="remoting.ejb" value="true"/>
<attribute name="remoting.ejb.inclusiveLocations"
value="classpath*:mandatory/env.xml,classpath*:/gui/server-config.xml"/>
<!-- configurations to include -->
<attribute name="remoting.ejb.exclusiveLocations"
value="classpath*:gui/client-config.xml"/> <!-- configurations to exclude -
->
```

For using the plugin you have to copy the `remoting_ejb.jar` into your build system's lib directory. And it has to be added to the project in order to use it. You can add it to the `plugins.xml` (there are no attributes to set):

```
<plugin name="ejb" file="buildsystem/remoting_ejb/remoting_ejb.xml"/>
```

Known issues

- **Weblogic 8.1** Weblogic is not able to resolve Spring wildcard-locations, where the asterisks is in the second part. You have to enumerate the configuration locations explicitly or use the module application context.
  - valid use of wildcards in Weblogic:
    - `classpath:foo/bar.xml`
    - `classpath*:foo/bar.xml`
  - invalid use of wildcards in Weblogic:
    - `classpath:foo/*.xml`
    - `classpath*:foo/*.xml`

How to use the EJB remoting module without the EL4Ant? build system

There's no need to use the EL4Ant? build system in order to use the EJB remoting module. You need to perform the following steps:

1. compile the source code
2. run the EJB wrapper generator which creates an annotated Java source file.
3. run The XDoclet task for your application server that will use the annotated Java file created before to generate all the needed EJB specific files.
4. create an appropriate application.xml
5. pack the classes and all required libraries into an EAR file

In principle there's no binding to a specific build system at all. But using Ant simplifies the whole process (e.g. XDoclet Ant tasks).

Todo

The EJB wrapper generator's core is contained in the EJB remoting module, whereas the binding to Velocity, which is used to generate the wrapper, is located in the build system plugin (see internal design for details). Providing a separate jar file that contains all the needed libraries (Spring, Velocity, commons-logging, ...) and that supports direct usage through the command line would simplify the above process.

References

- ModuleRemoting
- EJB 2.1 specification
  http://java.sun.com/j2ee http://jcp.org/en/jsr/detail?id=153
- XDoclet
  http://xdoclet.sourceforge.net/
- Velocity
  http://jakarta.apache.org/velocity/

Internal design

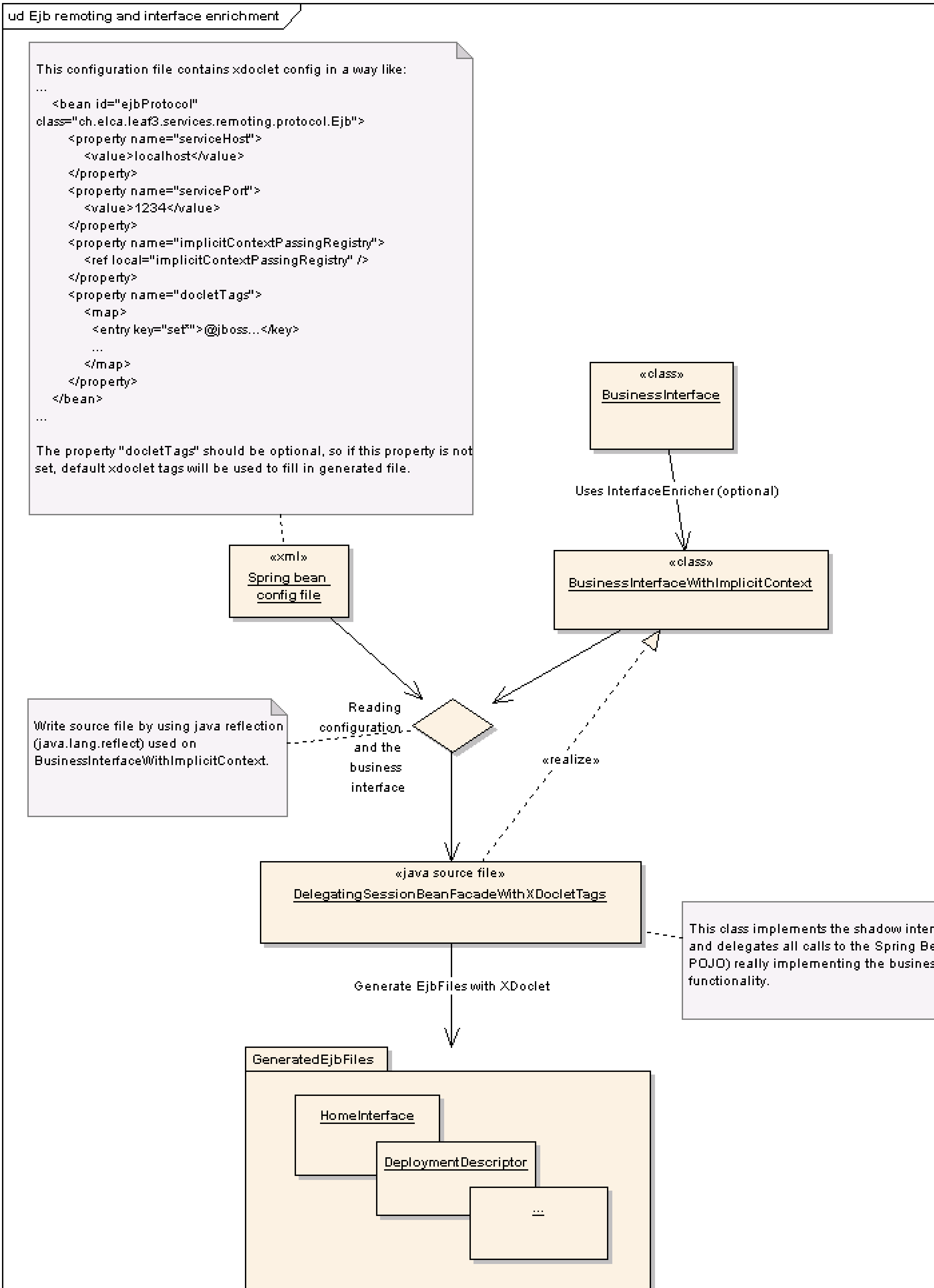
EJB generation

The session beans are generated using a hook of the build system. This hook uses Velocity to create session beans that delegate the invocations to the POJOs that implement the services. The generated Java files contain javadoc comments that are read by XDoclet to create the real EJB-aware classes. Next, the hook compiles the generated classes and puts them in the module's class path. Finally, it generates the deployment descriptor (`application.xml`).

The buildsystem requires to start the complete application to be able to create the templates. Since there's no easy mechanism to filter the configuration files that are needed to start the application in a Spring container, the user has to provide them with the `remoting.ejb.inclusiveLocations` and `remoting.ejb.exclusiveLocations` attributes.

Each application container has it's own deployment descriptor. The remoting EJB plugin gets the container that is currently used from the actual J2EE? -EJB plugin. This plugin also provides all the tasks to manage the server (e.g. starting, stopping, deploying) and to create EARs. Each remoting EJB application creates its EAR file. Of course, this file can contain more than one EJB application using build system dependencies.

Important: To avoid dependencies, we use reflection to get access to classes residing in the framework module. For simplicity, there's a facade to generate the beans that is in the EJB remoting module. Its interfaces are copied to the build system plugin where we just need to get the facade's implementation by reflection. All other operations are performed on the interface.



ud Ejb remoting and interface enrichment
«class»
BusinessInterface
Uses InterfaceEnricher (optional)
«class»
BusinessInterfaceWithImplicitContext
«xml»
Spring bean config file
Reading configuration and the business interface
«realize»
«java source file»
DelegatingSessionBeanFacadeWithXDocletTags
Generate EjbFiles with XDoclet
GeneratedEjbFiles
HomeInterface
DeploymentDescriptor
...

This configuration file contains xdoclet config in a way like:

```
...
   <bean id="ejbProtocol"
class="ch.elca.leaf3.services.remoting.protocol.Ejb">
       <property name="serviceHost">
          <value>localhost</value>
       </property>
       <property name="servicePort">
          <value>1234</value>
       </property>
       <property name="implicitContextPassingRegistry">
          <ref local="implicitContextPassingRegistry" />
       </property>
       <property name="docletTags">
          <map>
            <entry key="set*">@jboss...</key>
            ...
          </map>
       </property>
   </bean>
...
```

The property "docletTags" should be optional, so if this property is not set, default xdoclet tags will be used to fill in generated file.

Write source file by using java reflection (java.lang.reflect) used on BusinessInterfaceWithImplicitContext.

This class implements the shadow inter and delegates all calls to the Spring Be POJO) really implementing the busines functionality.

Adding support for another container

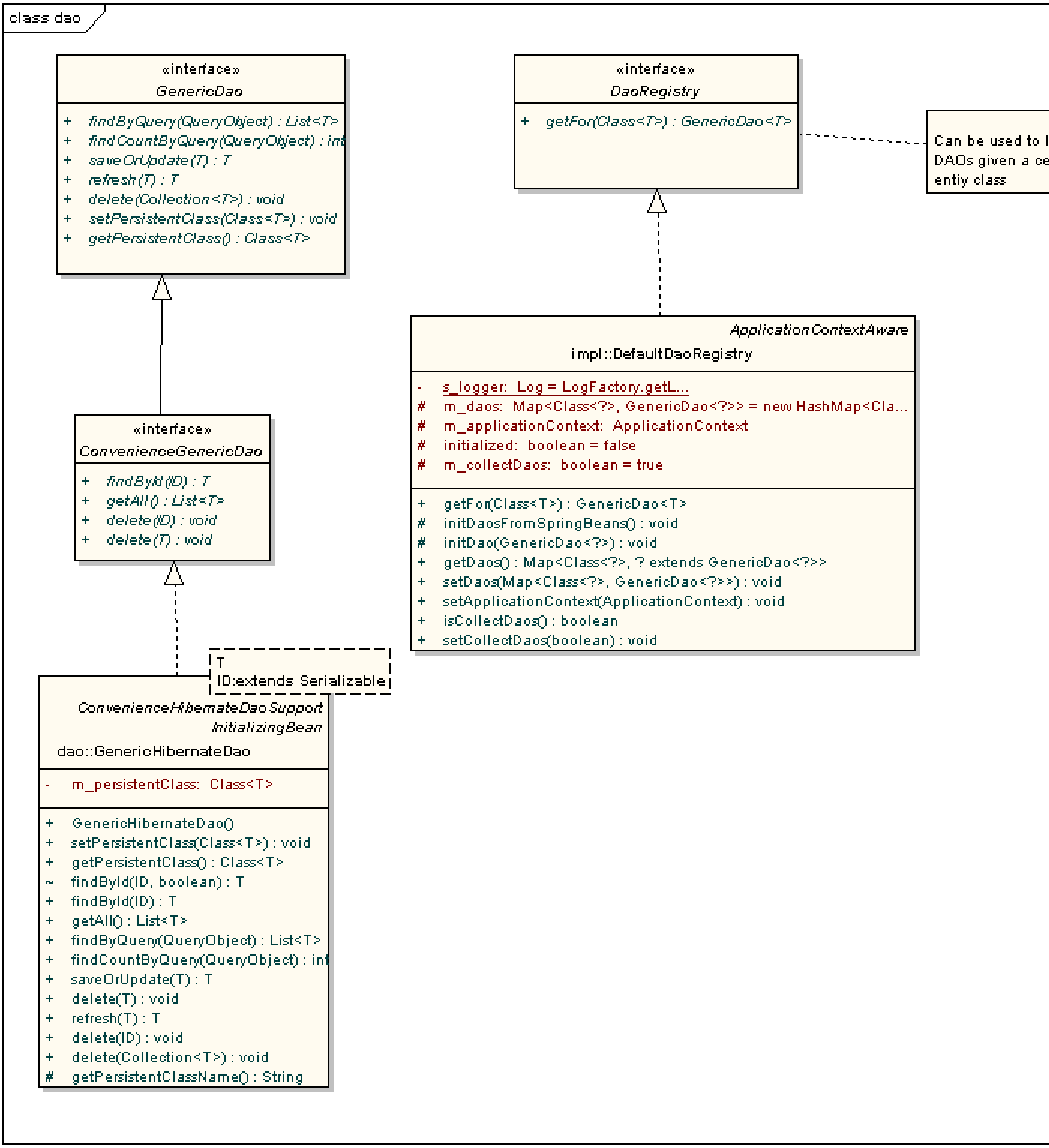
Adding support for a different container is straightforward. There are two things to do:

1. add a new ant file for XDocleting the generated files. You can just copy an existing one, do a search and replace on the container's name and adapt the container specific XDoclet target.
2. add the plugin dependency to the `ejb/ear.xml` file.

Generic DAOs in EL4J

Basic introduction

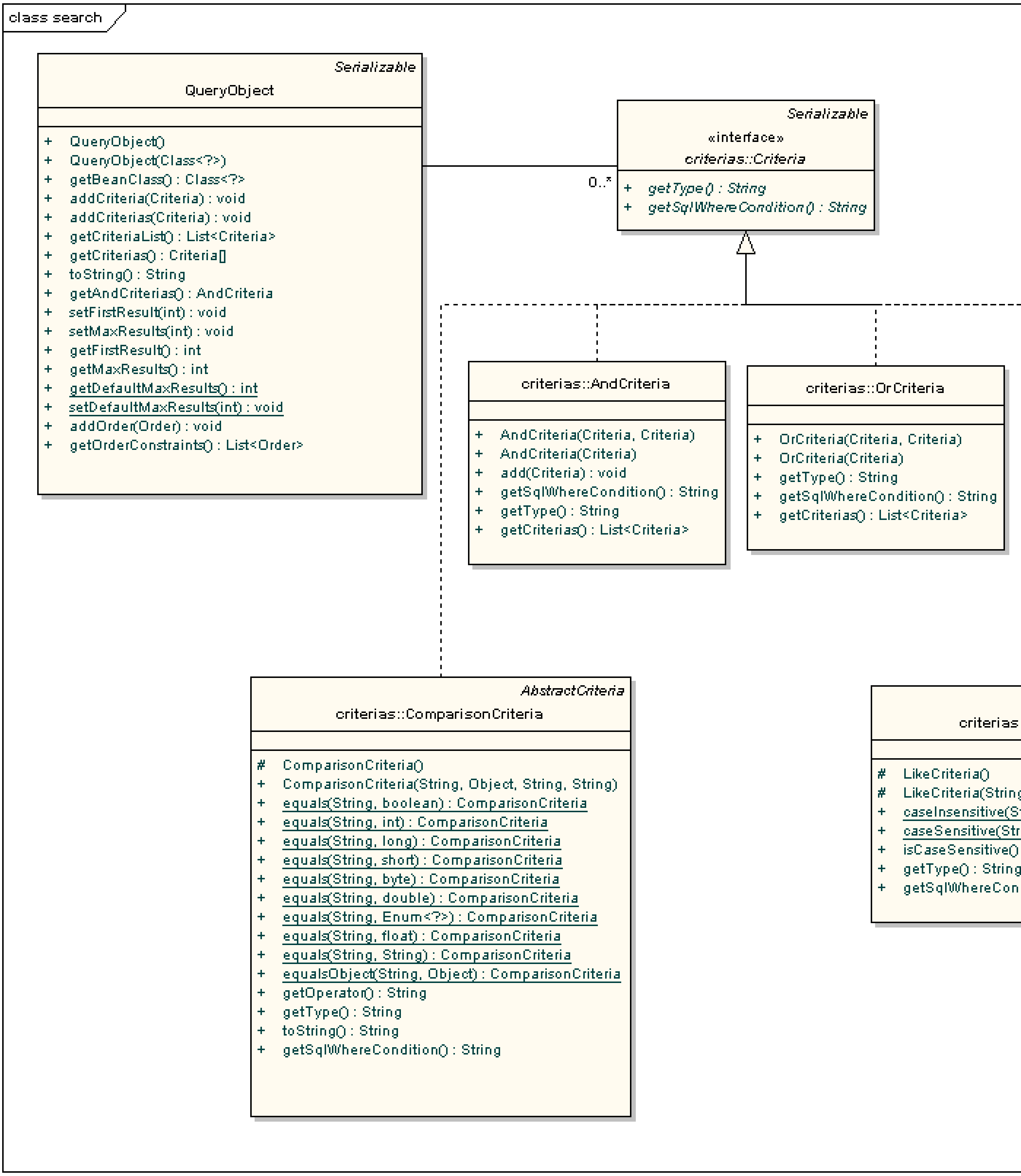
When writing DAOs (Data Access Objects), the basic CRUD (Create, Read, Update, Delete) operations tend to be more or less similar. The goal of the GenericDAO of EL4J is to eliminate these repetitive steps. The following class diagram illustrate this:


class dao
«interface»
GenericDao
+ findByQuery(QueryObject) : List<T>
+ findCountByQuery(QueryObject) : int
+ saveOrUpdate(T) : T
+ refresh(T) : T
+ delete(Collection<T>) : void
+ setPersistentClass(Class<T>) : void
+ getPersistentClass() : Class<T>
«interface»
DaoRegistry
+ getFor(Class<T>) : GenericDao<T>
Can be used to l
DAOs given a ce
entiy class
ApplicationContextAware
impl::DefaultDaoRegistry
- s_logger: Log = LogFactory.getL...
# m_daos: Map<Class<?>, GenericDao<?>> = new HashMap<Cla...
# m_applicationContext: ApplicationContext
# initialized: boolean = false
# m_collectDaos: boolean = true
+ getFor(Class<T>) : GenericDao<T>
# initDaosFromSpringBeans() : void
# initDao(GenericDao<?>) : void
+ getDaos() : Map<Class<?>, ? extends GenericDao<?>>
+ setDaos(Map<Class<?>, GenericDao<?>>) : void
+ setApplicationContext(ApplicationContext) : void
+ isCollectDaos() : boolean
+ setCollectDaos(boolean) : void
«interface»
ConvenienceGenericDao
+ findById(ID) : T
+ getAll() : List<T>
+ delete(ID) : void
+ delete(T) : void
T
ID:extends Serializable
ConvenienceHibernateDaoSupport
InitializingBean
dao::GenericHibernateDao
- m_persistentClass: Class<T>
+ GenericHibernateDao()
+ setPersistentClass(Class<T>) : void
+ getPersistentClass() : Class<T>
~ findById(ID, boolean) : T
+ findById(ID) : T
+ getAll() : List<T>
+ findByQuery(QueryObject) : List<T>
+ findCountByQuery(QueryObject) : int
+ saveOrUpdate(T) : T
+ delete(T) : void
+ refresh(T) : T
+ delete(ID) : void
+ delete(Collection<T>) : void
# getPersistentClassName() : String

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.)

QueryObject

The following UML diagram shows the QueryObject with the possible contained search criteria.


class search
Serializable
QueryObject
+ QueryObject()
+ QueryObject(Class<?>)
+ getBeanClass() : Class<?>
+ addCriteria(Criteria) : void
+ addCriterias(Criteria) : void
+ getCriteriaList() : List<Criteria>
+ getCriterias() : Criteria[]
+ toString() : String
+ getAndCriterias() : AndCriteria
+ setFirstResult(int) : void
+ setMaxResults(int) : void
+ getFirstResult() : int
+ getMaxResults() : int
+ getDefaultMaxResults() : int
+ setDefaultMaxResults(int) : void
+ addOrder(Order) : void
+ getOrderConstraints() : List<Order>
0..*
Serializable
«interface»
criterias::Criteria
+ getType() : String
+ getSqlWhereCondition() : String
criterias::AndCriteria
+ AndCriteria(Criteria, Criteria)
+ AndCriteria(Criteria)
+ add(Criteria) : void
+ getSqlWhereCondition() : String
+ getType() : String
+ getCriterias() : List<Criteria>
criterias::OrCriteria
+ OrCriteria(Criteria, Criteria)
+ OrCriteria(Criteria)
+ getType() : String
+ getSqlWhereCondition() : String
+ getCriterias() : List<Criteria>
AbstractCriteria
criterias::ComparisonCriteria
# ComparisonCriteria()
+ ComparisonCriteria(String, Object, String, String)
+ equals(String, boolean) : ComparisonCriteria
+ equals(String, int) : ComparisonCriteria
+ equals(String, long) : ComparisonCriteria
+ equals(String, short) : ComparisonCriteria
+ equals(String, byte) : ComparisonCriteria
+ equals(String, double) : ComparisonCriteria
+ equals(String, Enum<?>) : ComparisonCriteria
+ equals(String, float) : ComparisonCriteria
+ equals(String, String) : ComparisonCriteria
+ equalsObject(String, Object) : ComparisonCriteria
+ getOperator() : String
+ getType() : String
+ toString() : String
+ getSqlWhereCondition() : String
criterias
# LikeCriteria()
# LikeCriteria(String
+ caseInsensitive(S
+ caseSensitive(Str
+ isCaseSensitive()
+ getType() : String
+ getSqlWhereCon

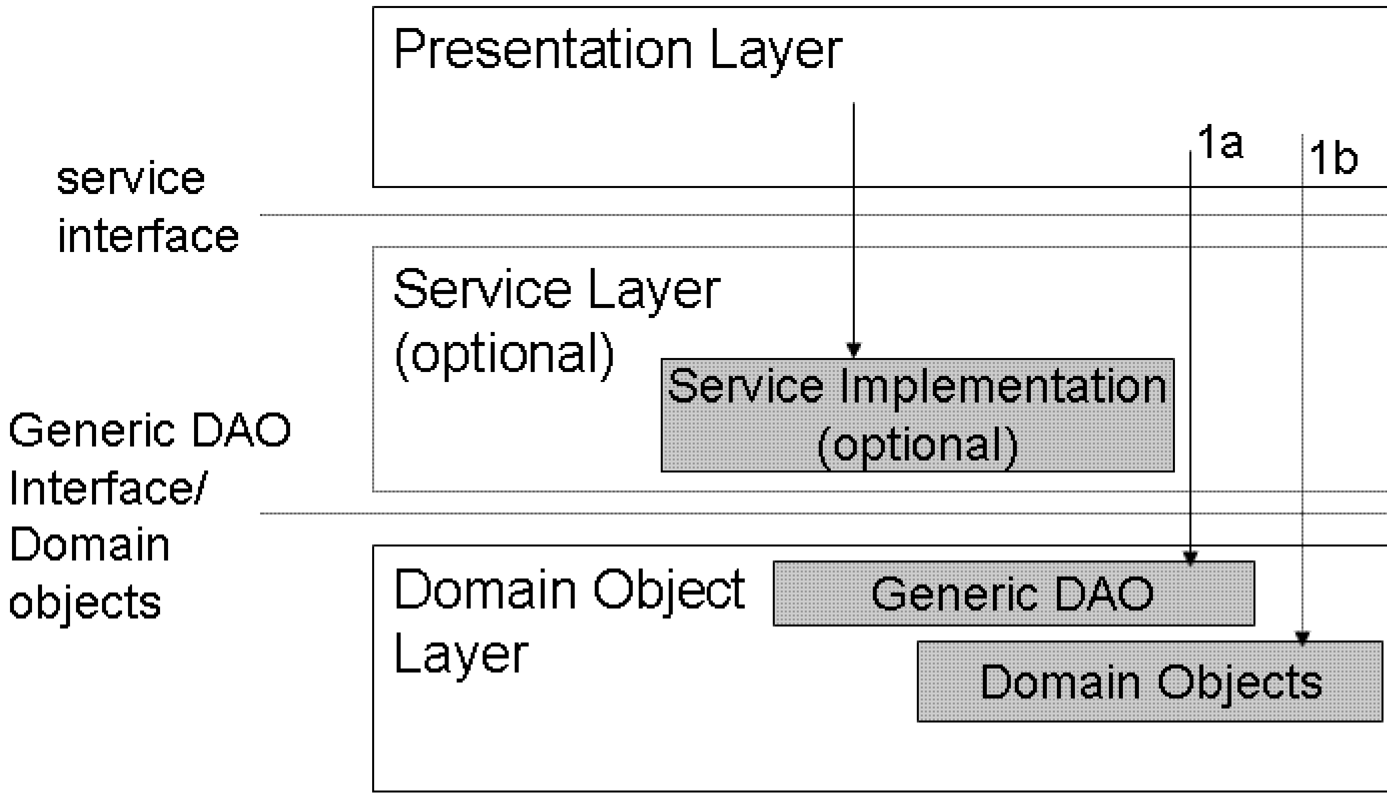
Here is some sample code that shows how to use this:

```
import static ch.elca.el4j.services.search.criterias.CriteriaHelper.*;
..
query = new QueryObject();
query.addCriteria(
  or (and (not(new ComparisonCriteria("name","Ghost","!=","String")),
          (or (not(like("name",  "%host%")),
               like("name", "%host%"))))));

query.setMaxResults(20);     // defaults to 100 elements
query.setFirstResult(40);    // defaults to 0

// we want the name field ordered alphabetically
query.addOrder(Order.desc("name"));
list = dao.findByQuery(query);
```

Remark: the CriteriaHelper class that is imported statically (a feature since Java 5) has convenience methods `or()`, `and()`, and `not()` that create `OrCriteria`, `AndCriteria` and `NotCriteria`, respectively.

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.

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.

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/

- Discussion about the fact that DAOs are no longer needed. We think DAOs are still useful. http://www.adam-bien.com/roller/abien/entry/jpa_ejb3_killed_the_dao

Documentation for module Swing

Purpose

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

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.

This module is based on Suns AppFramework, so it is strongly recommended to read https://appframework.dev.java.net/intro/index.html.

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. By name is the easiest, annotations allow to use different names for the GUI elements (but this is more verbose), and explicit programmatic binding provides full control (which is needed for list and tables).
    - Whenever the user changes the content of a bound GUI component, change events are fired internally. On order to make these events reach the underlying POJO, it needs an event listener infrastructure, which e.g. can be added at runtime by using a special Mixin (see PropertyChangeListenerMixin): `bindingEnabledPOJO = PropertyChangeListenerMixin.addPropertyChangeMixin(ordinaryPOJO)`.

- 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 if this is necessary. Trees for example cannot be handled by Beans Binding yet, so you have to implement a TreeModel here.

- In this framework, editing values in gui-elements directly changes the underlying POJOs. This leads to the question what to do if changes should only be applied if the user clicks "OK"? There are two ways to handler this. You can make a copy of the POJO and connect it to the GUI. If changes are accepted replace the original by the modified. This approach guarantees full control. If the POJO is a Java Bean and doesn't need deep copy there's a simpler way:

  - 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 of primitive or immutable type 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.

  - More details: ModuleSwingBinding

- cookSwing: XML representation of Swing components
  - Each XML tag represents a Swing component, which will be generated at runtime. See section cookSwing below
  - For more information see ModuleSwingXML
- 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. This could be status events, user closed a frame, etc; in general something that is not directly described by a model. 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 `getActionsContext()` of class GUIApplication. In short, you split your Actions on n classes. Overwrite the `m_actionsContext` (if you are in your central GUI class i.e. the one you launch via `GUIApplication.launch`) or create a new instance of `ActionsContext` using `ActionsContext.create(Object... instancesWithActionMappings)`. Then you can search for actions using `m_actionsContext.getAction()`.
    - 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. Although this works in this special case it is better to write correct code. See http://stuffthathappens.com/blog/2007/12/15/javabeans-propertychangesupport-trick/.

    - 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)

- 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 confuse with Sun's GUI frameworks own `org.jdesktop.application.ApplicationContext` 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: see references at the end
- 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).

Some extensions to the integrated frameworks

Properties files (AppFramework)

The properties provided by the framework are of the form: `nameOfTheGUIComponent.property`. Module-Swing adds the following:

- `title`: If the component gets wrapped into a frame, then the frame gets this title
- `name`: If the component gets wrapped into a frame, then the frame gets this name
- `help`: a URL of a website that provide more information about this component/frame

Action context (AppFramework)

By default, the actions (methods annotated with `@Action`) need to be defined in the main application (or a super class). This can lead to "Spaghetti-Code" and work-arounds are rather inconvenient. Therefore module-Swing adds the notion of ActionsContext, which is very similar to Spring's ApplicationContext. The ActionsContext consists of a list of objects that contain annotated methods and an optional parent context. Retrieving an action using `getAction` works as follows:

- Search the action in the list of objects (the order is important, this also allows action overriding). Of course, actions defined in super-classes are considered, too.
- If no matching action could be found, the parent action context is asked.

Bind non standard GUI components

Most of the standard components included in Swing are ready to use, i.e. if you bind a value to a JTextField it is known that the property 'text' of this component is meant actually.

Additional default properties for GUI components can be specified using `BindingFactory.getDefaultProperties().register(widgetInstanceOrClass, propertyToBind)`.

How to get started with our demo application

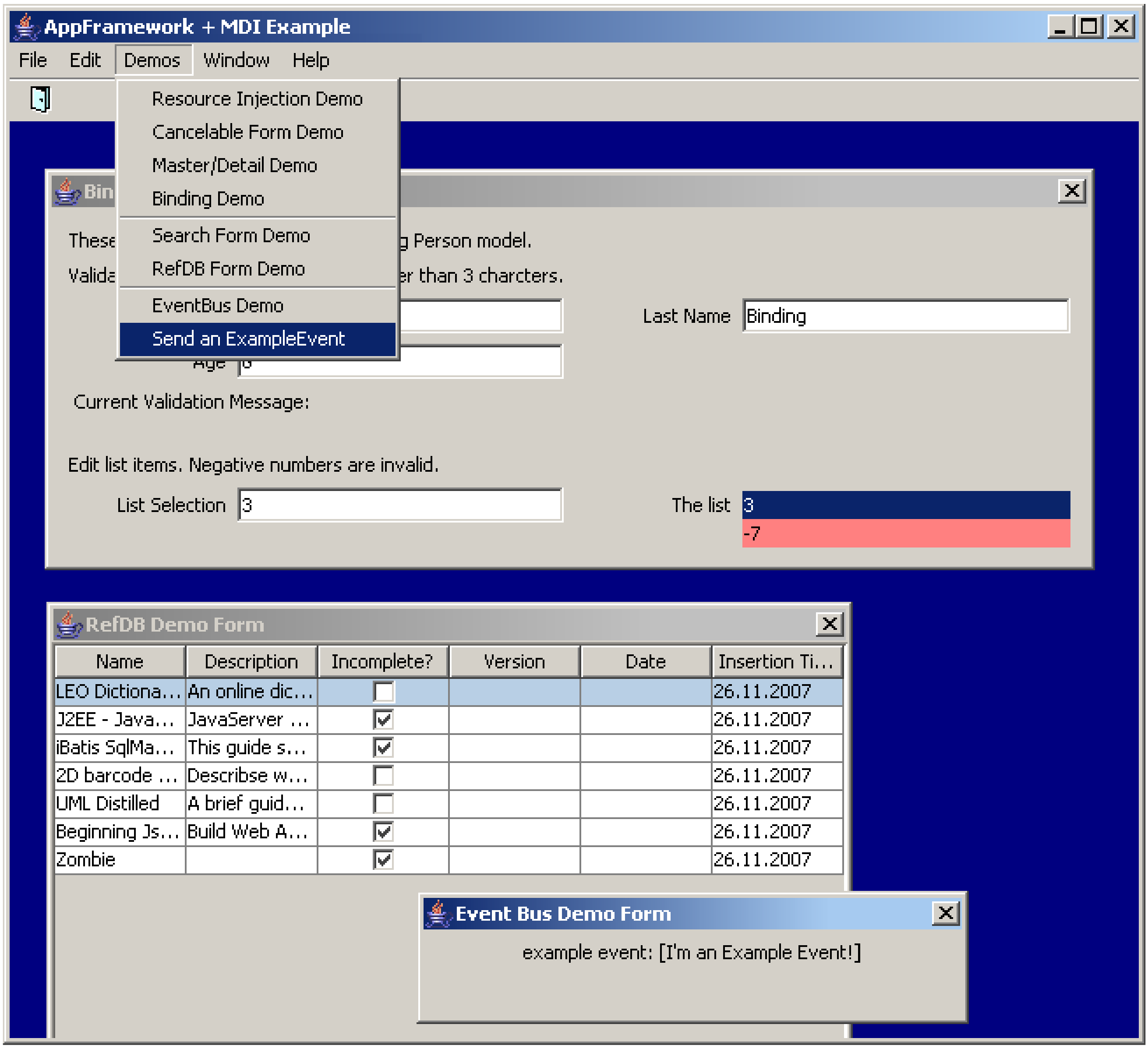
- Download el4j version 1.2 (or higher) or check out the latest el4j version from the svn repository (we recommend to download the el4j convenience zip and in d:/el4j do a `svn co https://el4j.svn.sourceforge.net/svnroot/el4j/trunk/el4j external`) (Alternatively, snapshot-versions/ releases of the components 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)

Demos

- Demo container using MDI (MainFormMDI)



These … g Person model.

Valida … er than 3 charcters.

Last Name: Binding

Age: 6

Current Validation Message:

Edit list items. Negative numbers are invalid.

List Selection: 3

The list:
3
-7

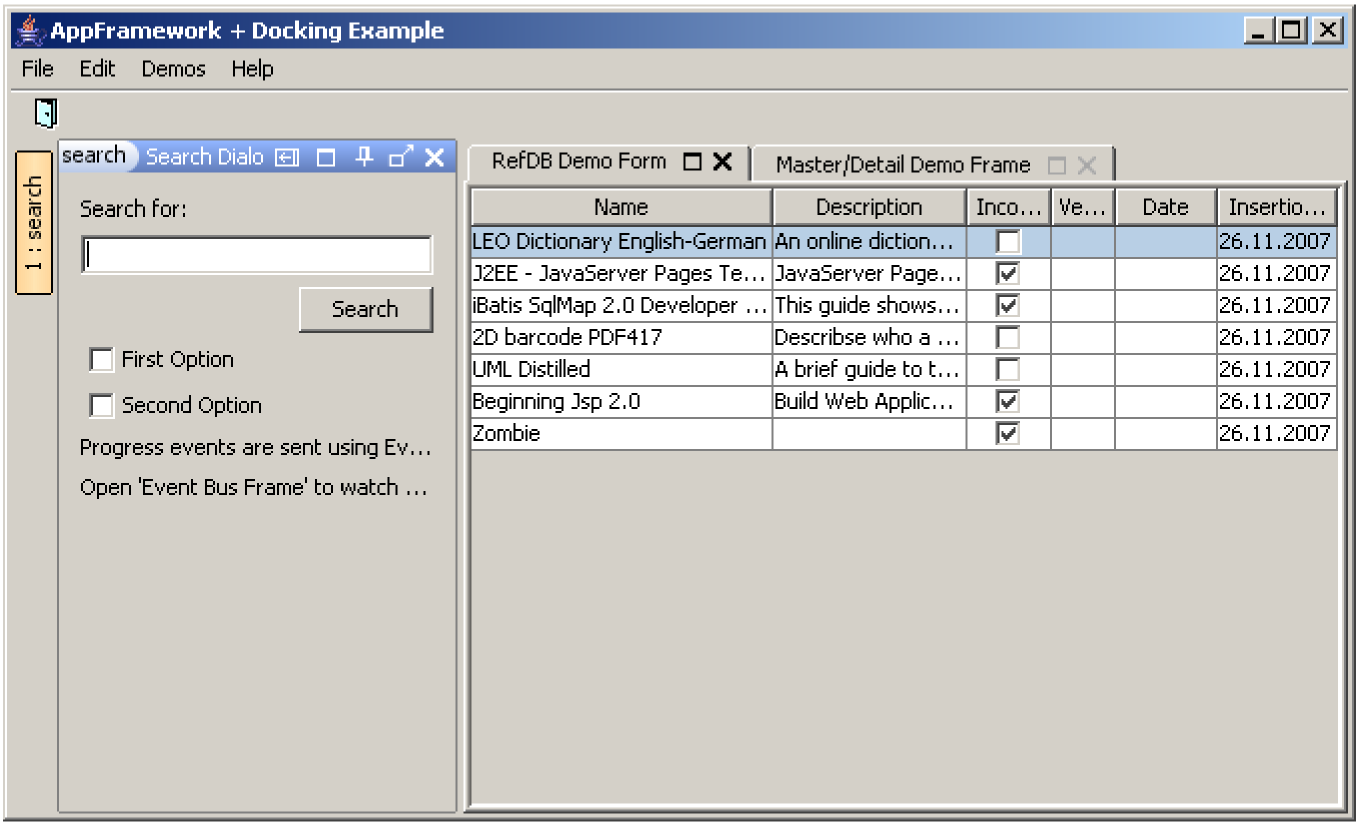
RefDB Demo Form

Name	Description	Incomplete?	Version	Date	Insertion Ti...
LEO Dictiona...	An online dic...	☐			26.11.2007
J2EE - Java...	JavaServer ...	☑			26.11.2007
iBatis SqlMa...	This guide s...	☑			26.11.2007
2D barcode ...	Describe w...	☐			26.11.2007
UML Distilled	A brief guid...	☐			26.11.2007
Beginning Js...	Build Web A...	☑			26.11.2007
Zombie		☑			26.11.2007

Event Bus Demo Form

example event: [I'm an Example Event!]

- Demo container using docking framework (MainFormDocking)



Search for:

☐ First Option

☐ Second Option

Progress events are sent using Ev...

Open 'Event Bus Frame' to watch ...

Name	Description	Inco...	Ve...	Date	Insertio...
LEO Dictionary English-German	An online diction...	☐			26.11.2007
J2EE - JavaServer Pages Te...	JavaServer Page...	☑			26.11.2007
iBatis SqlMap 2.0 Developer ...	This guide shows...	☑			26.11.2007
2D barcode PDF417	Describse who a ...	☐			26.11.2007
UML Distilled	A brief guide to t...	☐			26.11.2007
Beginning Jsp 2.0	Build Web Applic...	☑			26.11.2007
Zombie		☑			26.11.2007

- 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 (SearchForm)
  - An example search dialog that uses eventbus (use EventBus Demo to make events visible)
- Caching Demo (CachingDemoForm)
  - Demonstrates how to configure and use EhCache using Spring.
- 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 shows what events are generated
- cookSwingDemo (XMLDemoForm) (EL4J 1.4 and higher)
  - This frame shows how cookSwing can be used to describe the GUI in XML
- 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".

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 (swing module specific) patch to enable validation on tables.
- A modified version of Hitch Binding (http://hitch.silvermindsoftware.com) that uses BeansBinding. As only the core is used, the modified version is directly integrated into the swing module (see Javadoc, xref).
  - 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.

Configuration

There are some constant configuration parameters that need to be available when using this module. The default values are set in `ch.elca.el4j.services.gui.swing.config.DefaultConfig`. For an example how to override them see Spring bean `overrideConfig` in `applicationGeneral.xml" in =swing-demo-thin-client`. The following entries are mandatory:

- `invalidColor`: the color to mark values as invalid
- `selectedColor`: the color to mark values as selected (mainly used in `JTables`)
- `validationResponder`: the validation responder (a class that knows how to react when the user entered valid/invalid data e.g. to change the text of a validation message label)
- `cellRenderer`: the general cell renderer (e.g. for `JLists`)
- `comboBoxRenderer`: the renderer for `JComboBox` entries
- `tableCellRenderer`: the renderer for table cells
- `tableCellEditor`: the editor for table cells

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

GUI programming hints

- Any DB call is potentially time-consuming, so every DB access should be executed asynchronously (using `Task` from AppFramework)

- For responsive GUIs never execute synchronous methods which run longer than 50-100ms. Also keep in mind that execution time can increase over time.
- If you are working with eventBus events, do not forget to add the following lines to your log4j.xml. Otherwise you will not see the exceptions thrown in your event listeners:

```
<logger name="org.bushe.swing.event">
    <level value="WARN"/>
</logger>
```

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

- cookSwing: http://cookxml.yuanheng.org/cookswing/
  - The site contains the documentation of all tags.

- Standard Swing components: http://java.sun.com/docs/books/tutorial/ui/features/compWin.html

- More Swing components:
  - 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
  - https://swingx.dev.java.net/
  - http://common.l2fprod.com/index.php

binding. See Master/Detail Demo how this can be done)

- An alternative docking framework can be found at https://flexdock.dev.java.net/ but it seems to be sleeping

- FAQ on swing: http://www.chka.de/swing/

- GUI test framework: TestingWithAbbot (a demo GUI test is part of the application template).

- EDT = event dispatching thread. It's important use the EDT correctly in your swing applications (otherwise your application becomes unresponsive or deadlocks). Here is a book chapter about it. Here is a library that has tools to check for incorrect usage of the EDT (refer to the section about testing and debugging). The `Tasks` abstraction of Sun's App Framework helps to solve some EDT issues.

- A comparison of different layout managers: http://wiki.java.net/bin/view/Javadesktop/LayoutManagerShowdown?TWIKISID=b9e82416ed1c5adaee26bbfbcb3e1db7
- Java desktop community page: http://community.java.net/javadesktop/
- Nicer look and feels (laf)
  - substance The downside of substance is that it requires a lot of memory.
  - nimbus
  - EaSynth configuration for Synth Look&Feel We have experience and a license for EaSynth Designer. Contact DMY or YMA.
- Icons
  - Silk: icon set (16x16)
  - IcoFX: a freeware icon editor (e.g. to modify icons)
  - Icon search engine

Documentation for XML GUI representation in module Swing

Introduction

The XML-GUI support in module Swing is based on CookSwing, an extensible library that builds Java Swing GUI from XML documents. Its most important features are:

- All Swing components can be configured.
- All AWT/Swing LayoutManagers, including GridBagLayout and SpringLayout, can be used.

- All Swing Border classes, including complex CompoundBorder can be built.
- Simple custom tags extensions, with many fully functional tags can be added using one line statement. In fact, you can view the CookSwing tag library as an extension to CookXml, the XML decoding engine behind CookSwing.
- You can include other XML documents in a CookSwing XML file.

Let's have a look at a simple example:

```
<panel preferredsize="640,480">
   <borderlayout>
      <constraint location="North">
         <toolbar>
            <button action="quit" />
            <button action="cut" />
            <button action="copy" />
            <button action="paste" />
         </toolbar>
      </constraint>

      <constraint location="Center">
         <desktoppane />
      </constraint>
   </borderlayout>
</panel>
```

As you can image this represents a simple panel with a toolbar on top and a desktoppane filling the rest of the space. I hope that you agree that this is more readable than flat Swing code.

To insert this panel into a frame simply call (in the frame constructor):

```
CookSwing cookSwing = new CookSwing(this);
add(cookSwing.render("example.xml"));      // add panel from XML file to
frame
```

Namespaces

Before we start to explore more features let's have a look at the XML namespace. The module Swing contains XML Schema files (xsd), that validate these XML files and help XML editors to provide completion and assistance to the user. The whole schema is split into four namespaces:

- `http://cookxml.sf.net/` : cookXml core
- `http://cookxml.sf.net/common/` : cookXml common tags
- `http://cookxml.sf.net/cookswing/` : cookSwing tags
- `http://www.elca.ch/el4j/cookSwing` : additional tags provided by module Swing

This leads to clean separation of tags, but also increases the size of xml namespace declaration. The full size header looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<panel xmlns="http://cookxml.sf.net/cookswing/"
   xmlns:el4j="http://www.elca.ch/el4j/cookSwing"
   xmlns:cc="http://cookxml.sf.net/common/"
   xmlns:cx="http://cookxml.sf.net/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://cookxml.sf.net/cookswing/ cookSwing.xsd
      http://www.elca.ch/el4j/cookSwing el4jSwing.xsd
      http://cookxml.sf.net/common/ cookXmlCommon.xsd
      http://cookxml.sf.net/ cookXml.xsd">
```

In the following examples we will skip it.

XML Tags

The XML tags and attributes for swing components usually have the same name as the components they describe. To get an impression how they work, download the examples from the sourceforge page and look at the swing demo.

A complete list of cookSwing tags can be found at http://cookxml.yuanheng.org/cookswing/tagdoc/index.html

As cookSwing is an extension of cookXml, some tags and attributes are documented there. Here's a incomplete list of often used tags and attributes from cookXml:

- `<noadd/>` just create components that are inside, but do no add them to parent.
- `var="m_myComponent"` assign the created component to the variable `m_myComponent` of the object passed to the cookSwing constructor.
- `varref="m_myComponent"` do not create a new component but use the component stored in the variable `m_myComponent`
- `<include/>` include another xml file

AppFramework integration

The integration of the AppFramework is very easy. Use the `name` attribute of a components to be able to modify it's resources from the properties file. Actions can be set using the `action` attribute, `<button action="run" />` for example. When you click on the button the method `run` (which has to be annotated with `@Action`) will be executed.

So

```
<menubar>
   <menu name="fileMenu">
      <menuitem action="quit" />
   </menu>
</menubar>
```

can be used together with

```
fileMenu.text = File

quit.Action.text = Exit
quit.Action.mnemonic = E
quit.Action.icon = icons/exit_16.png
```

BeansBinding integration

BeansBinding support comes with several new tags.

Simple Bindings

A simple binding of a textfield to m_person.lastName looks like this:

```
<textfield>
    <el4j:binding src="m_person" property="lastName" updateStrategy="read
write" />
</textfield>
```

Supported attributes:

- `src` the source object to bind. This can be an expression like `m_person.children`, where all the expression behind the first dot need bean accessors.
- `property` the actual property of the source object to bind (see also General remark on attribute 'properties')
- `validation` should invalid values be rendered differently? (default: false)
- `updateStrategy` one of "read", "read once" or "read write". See javadoc of `org.jdesktop.beansbinding.AutoBinding` for more information (default: read)

List Bindings

Lists can be bound like this:

```
<list>
    <el4j:listbinding src="m_someListOfPersons" property="lastName" />
</list>
```

Supported attributes:

- `src` the source list to bind. This can be an expression like `m_person.children`, where all the expression behind the first dot need bean accessors.
- `property` the actual property of each element in the source list to bind (see also General remark on attribute 'properties')

- `validation` should invalid values be rendered differently? (default: false)
- `rendererBean` a custom `javax.swing.ListCellRenderer` Spring bean to render the list cells (default is `GUIApplication.getInstance().getConfig().get("cellRenderer")`)

Combobox Bindings

Comboboxes work just like lists:

```
<combobox>
    <el4j:comboboxbinding src="m_someListOfPersons" property="lastName" />
</combobox>
```

Supported attributes:

- `src` the source list to bind. This can be an expression like `m_person.children`, where all the expression behind the first dot need bean accessors.
- `property` the actual property of each element in the source list to bind (see also General remark on attribute 'properties')
- `validation` should invalid values be rendered differently? (default: false)
- `rendererBean` a custom `com.silvermindsoftware.hitch.validation.response.ComboBoxRenderer` Spring bean to render the combobox items (default is `GUIApplication.getInstance().getConfig().get("comboBoxRenderer")`)

Table Bindings

Tables need configuration for each column. In the following example, a table is bound to `m_persons`, where the first column shows the `firstName` property, the second `lastName` and the third `age`. The first name is directly editable inside the table cell and as the `updateStrategy` is "read write", the new values are immediately propagated (the other strategies `read` and `read once` have the same effect here, except that no column can be editable. See javadoc of `JTableBinding` for more details). The validation makes the background color red for table cells having invalid values. As known from Swing, columns can have a display class: Boolean values are shown as checkboxes, for example.

```
<table>
    <el4j:tablebinding src="m_persons" updateStrategy="read write"
validation="true">
        <el4j:column label="First Name" property="firstName"
editable="true" />
        <el4j:column label="Last Name" property="lastName" />
        <el4j:column label="Age" property="age" class="java.lang.Integer"
editable="true" />
    </el4j:tablebinding>
</table>
```

Supported attributes of `tablebinding`:

- `src` the source list to bind
- `updateStrategy` one of "read", "read once" or "read write". See javadoc of `org.jdesktop.swingbinding.JTableBinding` for more information (default: read)
- `validation` should invalid values be rendered differently? (default: false)
- `rendererBean` a custom `javax.swing.table.TableCellRenderer` Spring bean to render the table cells (default is `GUIApplication.getInstance().getConfig().get("tableCellRenderer")`)
- `editorBean` a custom `javax.swing.table.TableCellEditor` Spring bean to edit a table cell (default is `GUIApplication.getInstance().getConfig().get("tableCellEditor")`)

Supported attributes of `column`:

- `label` the column label shown in the table header
- `property` the actual property of each element in the source list to bind (see also General remark on attribute 'properties')
- `editable` is this column editable? (default: false)
- `class` the display class (e.g. `java.lang.Boolean` values are shown as checkboxes) (default: `java.lang.String`)

Unfortunately, `column` doesn't allow to set any renderers or editors because of the design of beansbinding. So you have to do it in Java code: After turning the bindings on (`m_binder.bindAll()`) you can access the table columns (e.g. `m_references.getColumnModel().getColumn(columnIndex).setCellRenderer(...)`).

General remark on attribute 'property'

The attribute 'properties' can have one of the following types:

- empty: the source object itself is taken instead of a property of it. Example:
- bean property: the general use case where the property is specified. Example:
- EL property: the advanced way to specify properties: Some EL examples: `"${address.streetNr}"`, `"${address.streetName} ${address.streetNr}"`, `"${age > 65}"`. Note that some of these bindings cannot be synchronized in both directions (Hint: What should the program do if the user sets ${age > 65} from true to false?). See javadoc of `org.jdesktop.beansbinding.ELProperty` for more details.

Additional tags

Flat Toolbar

The following tag is just for convenience: It generates a flat toolbar.

```
<el4j:flattoolbar>
    <button action="quit" />
</el4j:flattoolbar>
```

Designgridlayout

This tag implements the designgridlayout layout manager.

```
<el4j:designgridlayout>
    <el4j:row align="left" height="30">
        <label text="A label in left aligned row that has a height of 30 
pixels" />
    </el4j:row>
    <el4j:row>
```

```
        <cc:null colspan="3"/> <!-- skip three columns -->
        <label name="lblAge" horizontalalignment="RIGHT" />
    </el4j:row>
    <el4j:row align="center">
        <button action="info" />
    </el4j:row>
</el4j:designgridlayout>
```

New attributes for all XML tags:

- `colspan` the number of columns that a component spans

`designgridlayout` has no attributes.

Supported attributes of `row`:

- `height` the height of the row in pixels
- `align` the alignment of the row (`left`, `center`, `right`, `grid`)

Create component structures in Java

If one wants to create a part of the GUI programmatically the `create-component` tag can help. It supports two attributes:

- `create-method` : The method to call to get the component (if this is omitted a `JPanel` is created)
- `finish-method` : The method to call to do some work after all elements inside the `create-component` are created.

All elements inside this tag are created but NOT added to the parent. This can be done by implementing an Adder or manually using the `finish-method`.

Caution: In the following example all components will be created inside the XML files. However, **it is recommended that you instantiate such components in Java code unless you need special tags (e.g. binding) or XML id references**. In this sense, the example should only give you an impression, don't take it one by one.

```
<el4j:create-component finish-method="setGridPanelLayout">
    <textfield cx:var="m_name"/>
    <textfield cx:var="m_description" />
</el4j:create-component>
```

This example creates a simple `JPanel` and two text fields. Afterwards it calls the method "setGridPanelLayout", which adds the components to the panel using a special layout manager. Note that Designgridlayout is already supported to be written in XML (see section above).

```
private void setGridPanelLayout(JPanel formPanel) {
    DesignGridLayout layout = new DesignGridLayout(formPanel);
    formPanel.setLayout(layout);

    layout.row().add("Name").add(m_name);
    layout.row().add("Description").add(m_description);
}
```

Hints

- Unfortunately, eclipse has some problems validating xml files against the cookSwing schemata. The coding support (completion) is not perfect, neither. So keep in mind that not all errors are real ones. NetBeans has a better XML editor, so if you are creating XML-GUIs for a longer time, consider installing it...

Documentation for binding in module Swing

General

Binding in module Swing is based on PropertyChangeLister. All the Swing components support this by default, but your model probably does not. Therefore you should enhance you model using `yourBindableModel = PropertyChangeListenerMixin.addPropertyChangeMixin(yourModel);`.

Binding components using XML

If you are already describing your GUI using XML, this approach is the most comfortable. See ModuleSwingXML

Binding components using Java code and annotations

This abstraction level allows to bind one model object to the GUI. Binding more than one model is possible but a bit more difficult.

To bind a model to a GUI component, the following step are required:

- Annotate the GUI component with `@Form(autoBind = true)`
- Annotate the model with `@ModelObject(isDefault = true)`
- Create a `Binder` and bind it.

In the following example the text field `m_firstName` is bound to the property `m_person.firstName` (the prefix `m_` gets removed automatically if present) and the text field `m_lastName` is bound to `m_person.lastName`:

```
@Form(autoBind = true)
public class BindingDemoForm extends JPanel {
    private JTextField m_firstName;
    private JTextField m_lastName;

    @ModelObject(isDefault = true)
    private Person m_person;

    private final Binder m_binder = BinderManager.getBinder(this);


    ...
    m_binder.bindAll();
    ...
}
```

Binding more than one model can be achieved by specifying a modelId:

```
    @ModelObject
    private Person m_anotherPerson;

    @BoundComponent(modelId = "anotherPerson")
    private JTextField m_firstNameOfAnotherPerson;
```

See also the annotation section in the Hitch manual. But keep in mind that big parts of the API of the binder have completely changed.

An example can be found here: BindingDemoForm

Binding components using BeansBinding directly

This is the most verbose but also the most powerful way to create bindings. It is recommended to first try the higher lever abstractions.

A good introduction into BeansBinding can be found at Shannon Hickey's Blog. There is not too much information about it on the web and one has to pay attention that pre-1.0 version had a slightly different API. But the source code is documented :-).

Documentation for module Hibernate

Purpose

Convenience module for Hibernate.

edit purpose

How to use

Just include `classpath:scenarios/dataaccess/hibernate/*.xml` and create a new Spring config based on template for session factory bean.

Like this the following files are included:

- `hibernateSessionFactory.xml`: The `abstractSessionFactory` bean, which sets some default values for your custom session factory
- `hibernateDataSource.xml`: The `dataSource` bean using C3P0?
- `hibernateDefaultBeans.xml`: Some other default beans like `transactionManager` and `abstractDao` that need no additional configuration in general.

If you want to replace some of these beans, just use the `exclusiveConfigLocations` property of `ModuleApplicationContext` to exclude a file, and define your version of these beans.

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

Criteria transformation

This module includes a CriteriaTransformer? class that allows the transformation of EL4J Query Objects (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.

Remark: The EL4J Query Objects API has been introduced to be independent from the persistence technology (Hibernate or iBatis). Even if you only use Hibernate now, it can be interesting to decouple your application from Hibernate (e.g. in the client).

Generic Hibernate DAO

Refer to GenericDao

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 (also in the GUI/ web user interface).

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;
        }

        /**
        * @return true if the insertion date is after the document's
creation date.
        */
        @AssertTrue
        public boolean areDateCorrect() {
                if (getDocumentDate() != null && getWhenInserted() != null)
{
                        return (getDocumentDate().getTime() <=
getWhenInserted().getTime();
                } else {
                        return true;
                }
        }
}
```

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 `areDateCorrect()` method, which ensures that the reference's creation date is an earlier date than its insertion date.

Make sure that any method where you use the annotation `@AssertTrue` is null-pointer-save! Further do not repeat checks, as here, we just start the check if both dates are not null. Else we let this test pass. Other tests will fail instead and stop the save-process.

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.

JPA Extension

How to use

Include `classpath:scenario/dataaccess/jpa/*.xml` instead of `classpath:scenarios/dataaccess/hibernate/*.xml`.

The configuration files in the above location contain the following:

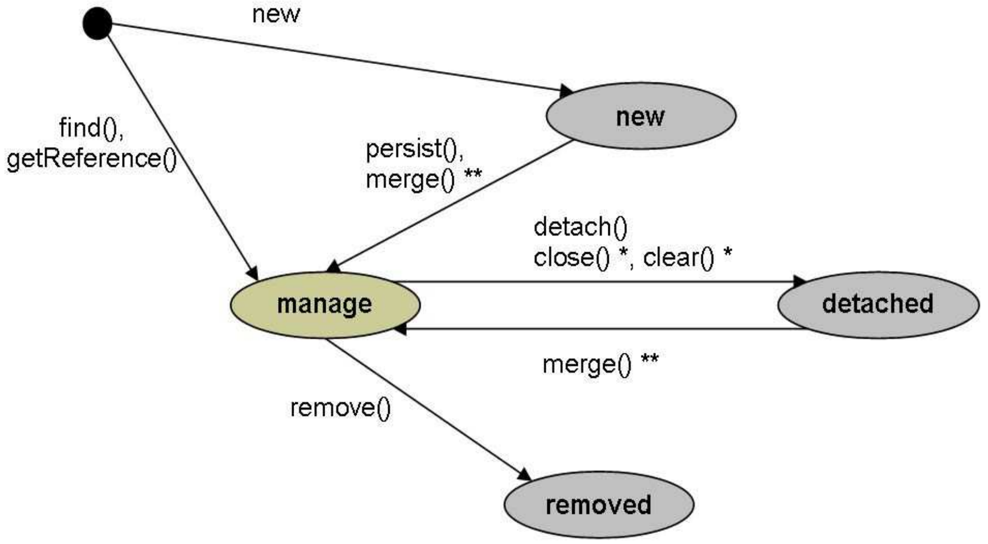
- `jpaEntityManagerFactory.xml`: The `entityManagerFactory` instead of the `SessionFactory`
- `jpaDataSource.xml`: The `dataSource` like with the hibernate configuration

- `jpaDefaultBeans.xml`: The `jpaHelper` and the `transactionManager`

Entity Lifecycle

To persist, detach, merge and remove the entities use the `jpaHelper` or create your own helper class. The states jpa works with are shown in the picture below. The use of jpa differs from the usage of the hibernate dao.

- There is no longer a `saveOrUpdate` function. To add an entity to the persistence context `persist` is used and `merge` to reattach it.
- `persist` or `merge` means not "save now", but put the entity into the persistence context.
- Changes applied to a `managed entity` do not have to be saved explicitly to the database. The entity will be automatically updated in the database after a `flush` or `commit`.
- To force a immediate saving to the database use `flush`. This is usually not necessary.
- The function `merge` returns the object as a `managed` entity. The "old" object is still be `detached`.
- Check, if the entity is already managed or detached, using the function `contains`.



* *Affects all instances in the persistence context*
** *Merging returns a persistent instance, orginial doesn't change state*

Some examples can be found in the jpa tests of the application template called `refdb`.

ConvenienceGenericJpaDao?

There is a `ConvenienceGenericJpaDao` interface implemented by `GenericJpaDao` that works analogously to the `ConvenienceGenericHibernateDao`. The differences between the two are minimal, with the following exceptions:

- the Hibernate DAO provides find-methods for EL4J `QueryObject` and Hibernate `DetachedCriteria`. The JPA DAO uses the `QueryBuilder`.
- the Hibernate DAO is part of the GenericDAO? interface hierarchy, the JPA DAO is not.
- In order to stay close to the vanilla terminology, you should use `persist()` and `merge()` instead of `saveOrUpdate()` in the JPA DAO
- The `saveOrUpdate()` from the JPA DAO corresponds to the `saveOrUpdateCopy()` method in Hibernate, i.e. the object returned is not necessarily the same as the saved one. You thus have to assign it again:

```
entity = dao.saveOrUpdate(entity);
```

References on JPA

- JPA Presentation
- JPA Implementation Patterns

Criteria Queries:

- Typesafe JPA
- Criteria Queries in JPA 2.0 and Hibernate

References

- HibernateGuidelines
- Java Persistence with Hibernate, Christian Bauer & Gavin King, Manning
- http://www.hibernate.org/

Documentation for module web

Purpose

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

edit purpose

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.

How to use

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).

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.

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.

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>
```

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>
```

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.

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.

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.

`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
```

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!/
```

Resources

- ModuleWebApplicationContextToDo specifies the problem more extensively
- ModuleWebApplicationContextToDoSpecification solution specification

Documentation for module security

Purpose

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

edit purpose

SSL certificate creation

For the convenient creation of SSL certificates (for CA, client and server), please have a look inside the `etc/openssl` directory of the framework.

Features

Besides the Spring Security 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.

NT login demo

Refer to `internal/applications/demos/nt_login`. Be sure to read the readme file.

Basic User Admin GUI and components

ModuleBasicUserManagement

A limited security demo is in the JSF template

`internal/applications/templates/jsf`

A GUI demo with security

`external/applications/templates/gui`

Implicit context passing

Please refer to ModuleRemoting#ImplicitContextPassing . This can be used to pass the security context with remote invocations (it's like a thread local that is transported to the server (but not back!)).

NTLM support

NTLM allows to do single-sign-on with a browser application (the browser automatically sends your windows credentials to the server). This works in IE, and can be enabled for Firefox, use the property network.automatic-ntlm-auth.trusted-uris (under URL `about:config`).

MSM / QKP have experience with this. Here is an email that has more info and a helper class: RENtlmloginwithtomcat.msg.

Encrypt passwords that go over the wire

This features encrypts all Spring Security passwords that are sent over the wire using remoting. For encryption a pre-shared symmetric key (AES, without any salt) is used, which therefore does not protect against man-in-the-middle attacks. As users often use the same or similar passwords (although they shouldn't) this approach makes it is very hard just to sniff the network traffic and try the password e.g. for the user's email account.

To enable password encryption do the following:

- Generate a symmetric key:
    - `cd external/framework/modules/security`
    - `mvn exec:java`
    - Copy the generated key (including the trailing ==)
- Replace your default remoting context passer `AuthenticationServiceContextPasser` by `SecureUsernamePasswordAuthenticationServiceContextPasser` and set the property "key". If you don't use implicit context passing yet, see ModuleRemoting#ImplicitContextPassing. Here's an example:

```
<bean id="authenticationServiceContextPasser"
```

```
class="ch.elca.el4j.services.security.authentication.SecureUsernamePassword
AuthenticationServiceContextPasser">
    <property name="implicitContextPassingRegistry">
        <ref bean="implicitContextPassingRegistry"/>
    </property>
    <property name="key" value="HERE_COMES_YOUR_KEY" />
</bean>
```

Block requests from unauthorized IP addresses

To allow requests only from a range of IP addresses add a dependency to module-security and the following snippet into the web.xml:

```
<filter>
    <filter-name>IP Address Filter</filter-name>
    <filter-
class>ch.elca.el4j.services.security.filters.IPAddressFilter</filter-class>
    <init-param>
        <param-name>ipAddresses</param-name>
        <param-value>operation.allowedIPAddresses</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>IP Address Filter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

The list of allowed IP addresses is passed as system property 'operation.allowedIPAddresses' and contains a comma separated list of IP addresses.

Example using jetty: `mvn db:prepare jetty:run -Doperation.allowedIPAddresses=127.0.0.1,142.31.2.26`

Basic HTTP authentication via JNDI (i.e. LDAP)

See ExternalToolTomcat for Tomcat.

How to use

Please refer to the demo application and the Spring Security documentation for now.

References

- SecurityGuidelines

Documentation for module exception handling

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.

edit purpose

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.

How to use

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).

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).

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
dler">
                            <property
name="useDynamicLogger"><value>true</value></property>
                        </bean>
                    </property>
                </bean>
            </list>
        </property>
    </bean>
</beans>
```

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.ContextExceptionHandlerFactor
yBean">
        <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.MessageBoxExceptionHand
ler"/>
                            </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.LogExceptonHandler">
                                    <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
```

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.MethodNameExceptionConfigurat
ion">
                    <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
rgetExceptionHandler">
```

```
                            <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>
```

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
ler">
                            <property name="exceptionHandlers">
                                <list>
                                    <bean
class="ch.elca.el4j.services.exceptionhandler.handler.SimpleLogExceptionHan
dler">
                                        <property
name="useDynamicLogger"><value>true</value></property>
                                    </bean>
                                    <bean
class="ch.elca.el4j.services.exceptionhandler.handler.RetryExceptionHandler
">
                                        <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.SimpleExceptionTransf
ormerExceptionHandler">
                                        <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.ReconfigureExceptionHandler">
                            <property name="c"><ref local="C"/></property>
                        </bean>
                    </property>
                </bean>
            </list>
        </property>
    </bean>
</beans>
```

References

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

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`

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.**

Documentation for module JMX

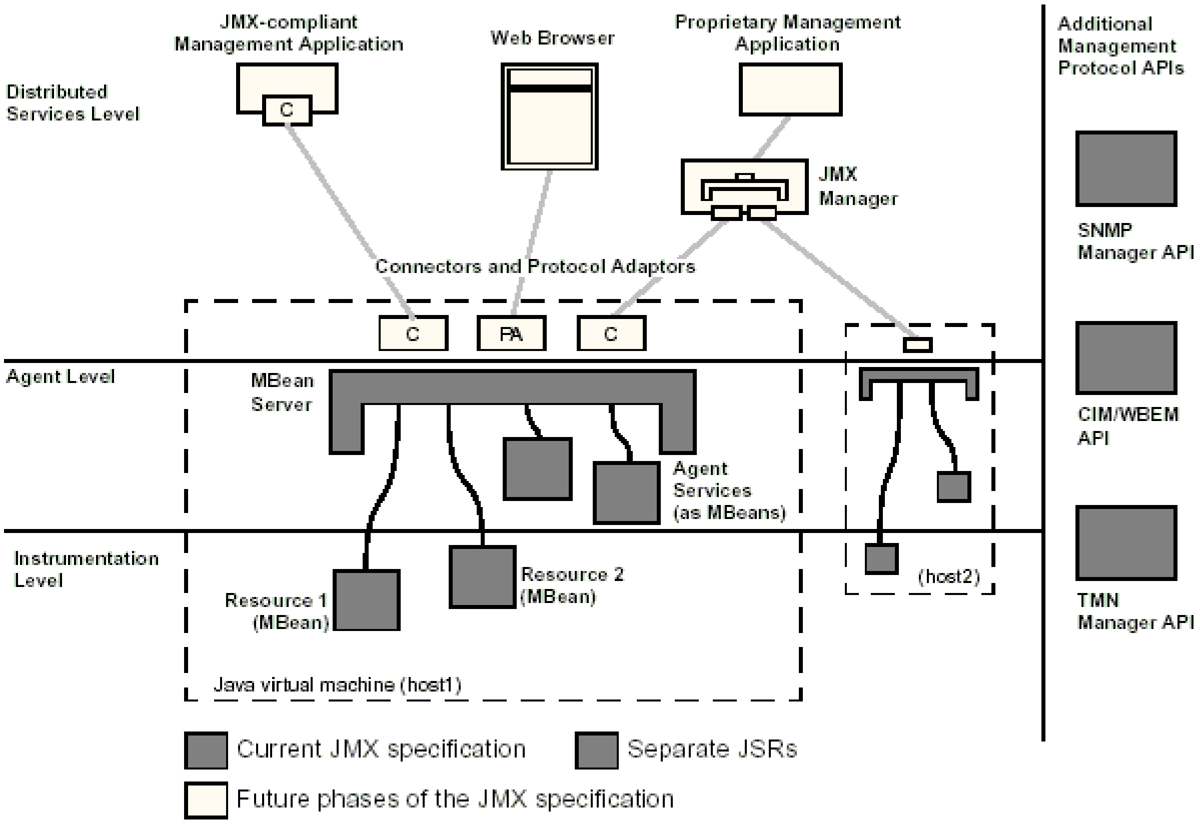
Purpose

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

edit purpose

Introduction to Java management eXtensions (JMX)

The follwing picture shows the components of JMX:


JMX-compliant Management Application
C
Web Browser
Proprietary Management Application
Additional Management Protocol APIs
Distributed Services Level
JMX Manager
SNMP Manager API
Connectors and Protocol Adaptors
C
PA
C
Agent Level
MBean Server
CIM/WBEM API
Agent Services (as MBeans)
Instrumentation Level
Resource 2 (MBean)
Resource 1 (MBean)
(host2)
TMN Manager API
Java virtual machine (host1)
Current JMX specification
Separate JSRs
Future phases of the JMX specification

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.

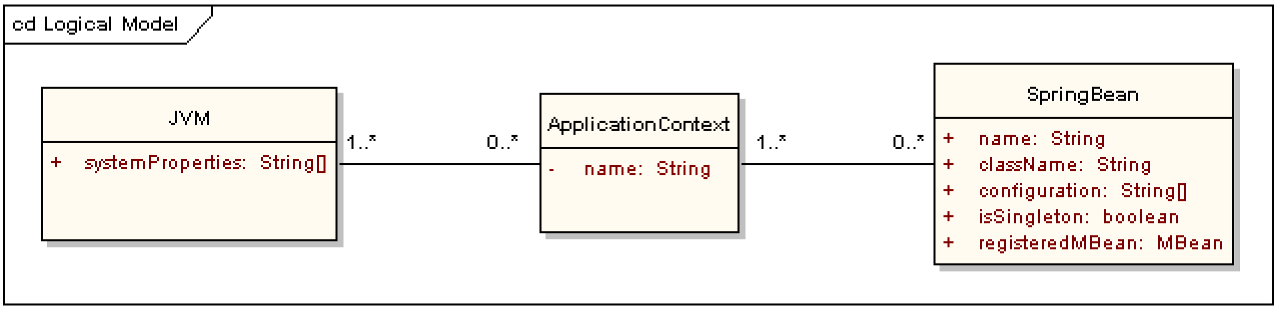
Feature overview

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.

Usage

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.

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.

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.

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.

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.

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>
```

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).

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/*.xml", "classpath:app/beans.xml"});

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

Configuration (explicit publication)

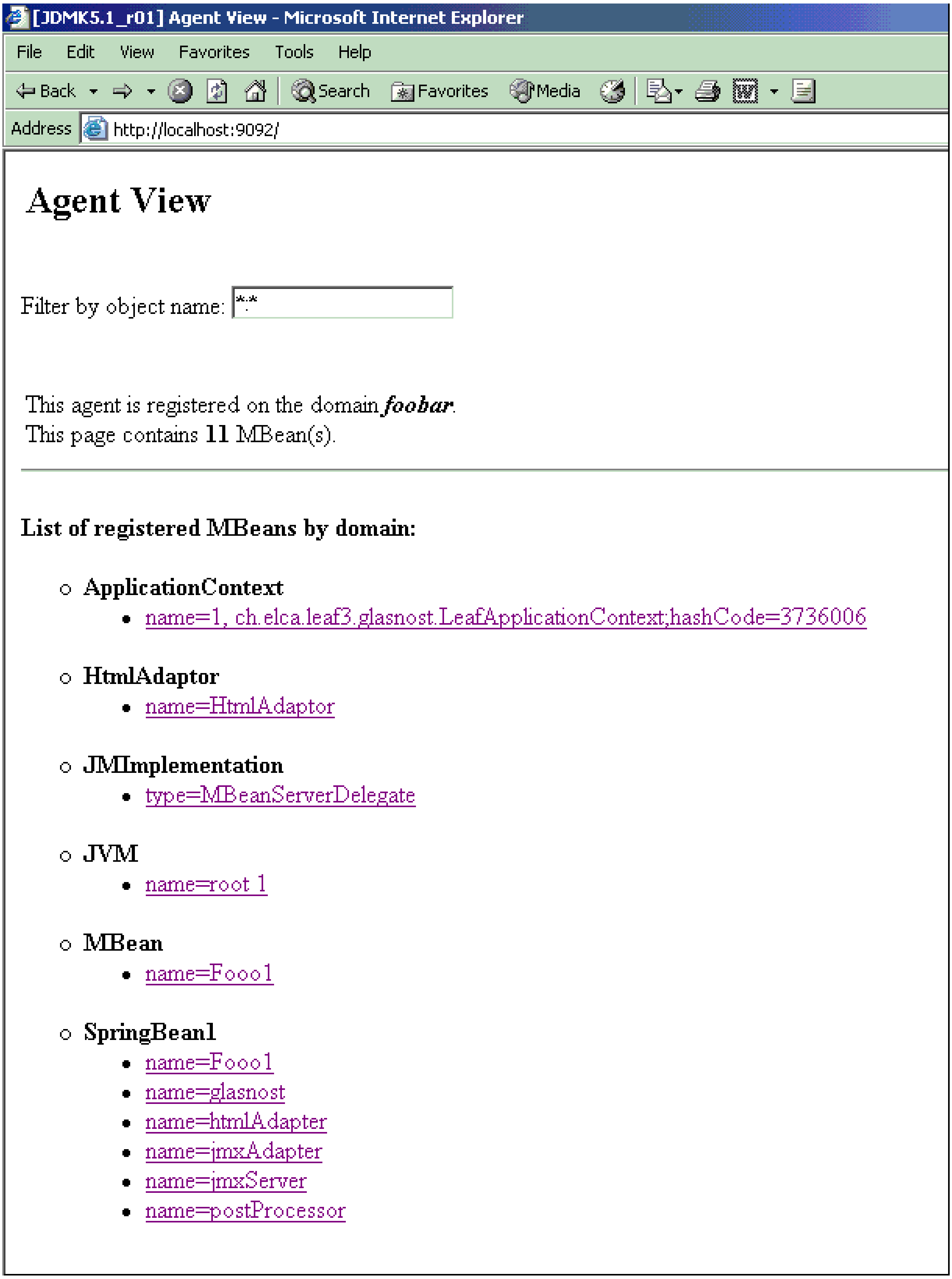
If you want to edit fields or invoke operations of a spring bean, e.g. on the bean `Foo1`, then you have to explicitly register it via `mBeanExporter`. The automatically created proxies for Spring beans do not allow editing their fields. The `beans.xml` configuration file could look like this:

```
<?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:


[JDMK5.1_r01] Agent View - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:9092/

Agent View

Filter by object name: *:*

This agent is registered on the domain ***foobar***.
This page contains **11** MBean(s).

List of registered MBeans by domain:

- **ApplicationContext**
  - name=1, ch.elca.leaf3.glasnost.LeafApplicationContext;hashCode=3736006
- **HtmlAdaptor**
  - name=HtmlAdaptor
- **JMImplementation**
  - type=MBeanServerDelegate
- **JVM**
  - name=root 1
- **MBean**
  - name=Fooo1
- **SpringBean1**
  - name=Fooo1
  - name=glasnost
  - name=htmlAdapter
  - name=jmxAdapter
  - name=jmxServer
  - name=postProcessor

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`.

Example with more than one ApplicationContext

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

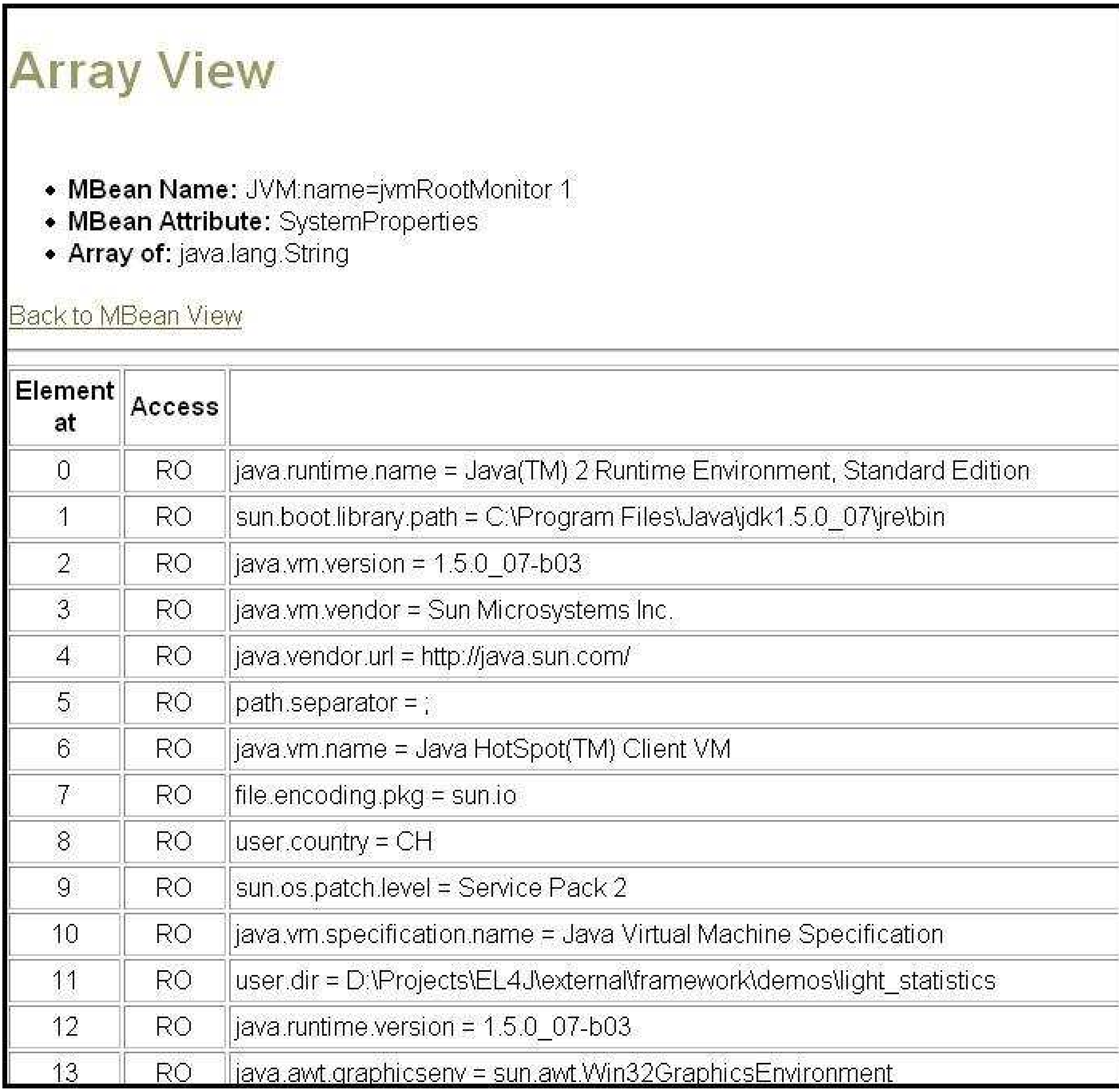
- If you want to register another ApplicationContext at the same `mBeanServer`, then you have to choose the same `defaultDomain` as the other Application Context since the domain of the MBean Server actually defines the MBean Server.
- If you want to register another ApplicationContext at a different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

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!

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).

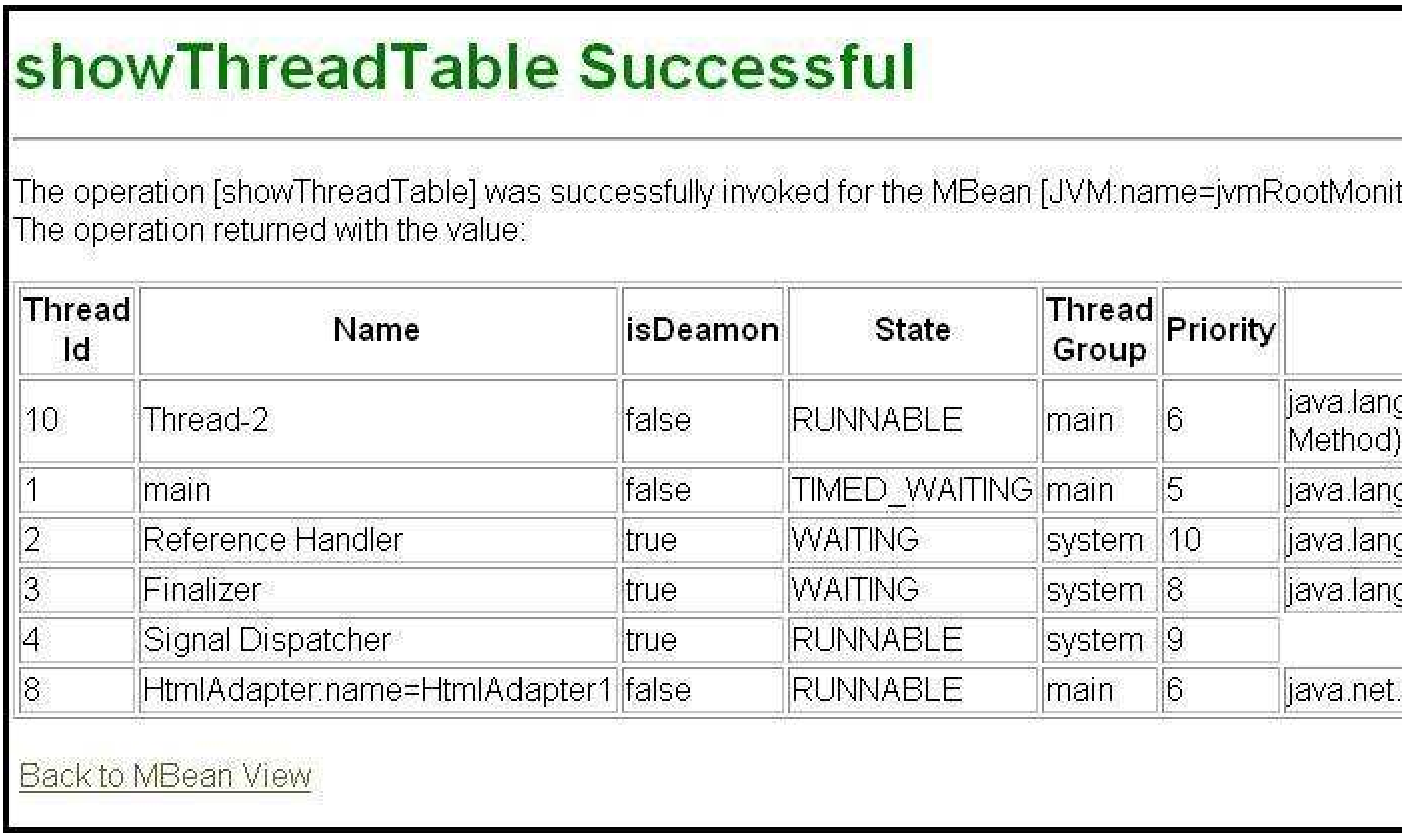


Array View

- **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



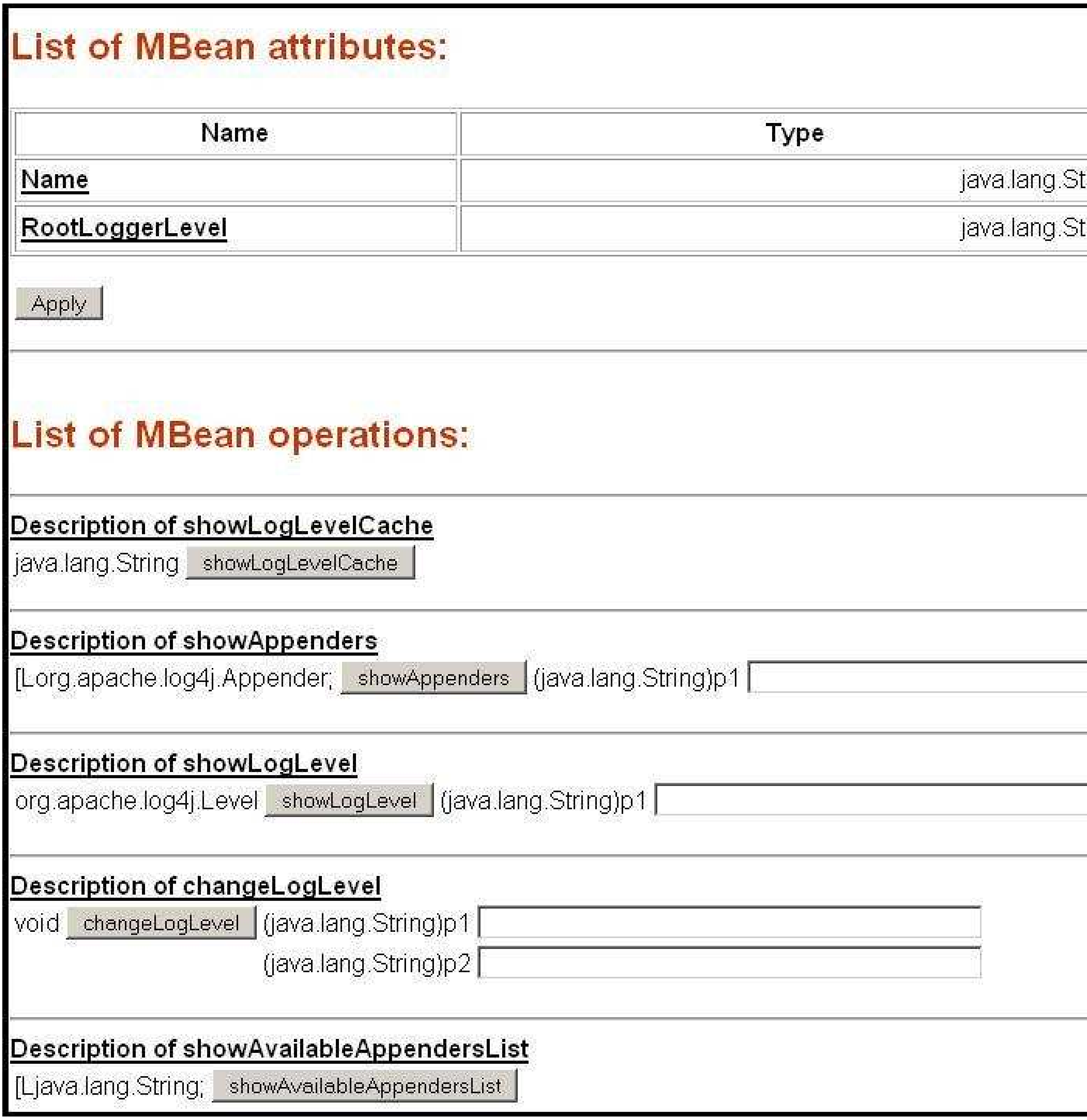
showThreadTable Successful

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

Thread Id	Name	isDeamon	State	Thread Group	Priority	
10	Thread-2	false	RUNNABLE	main	6	java.lang Method)
1	main	false	TIMED_WAITING	main	5	java.lang
2	Reference Handler	true	WAITING	system	10	java.lang
3	Finalizer	true	WAITING	system	8	java.lang
4	Signal Dispatcher	true	RUNNABLE	system	9	
8	HtmlAdapter:name=HtmlAdapter1	false	RUNNABLE	main	6	java.net.

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
Name	java.lang.St
RootLoggerLevel	java.lang.St

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

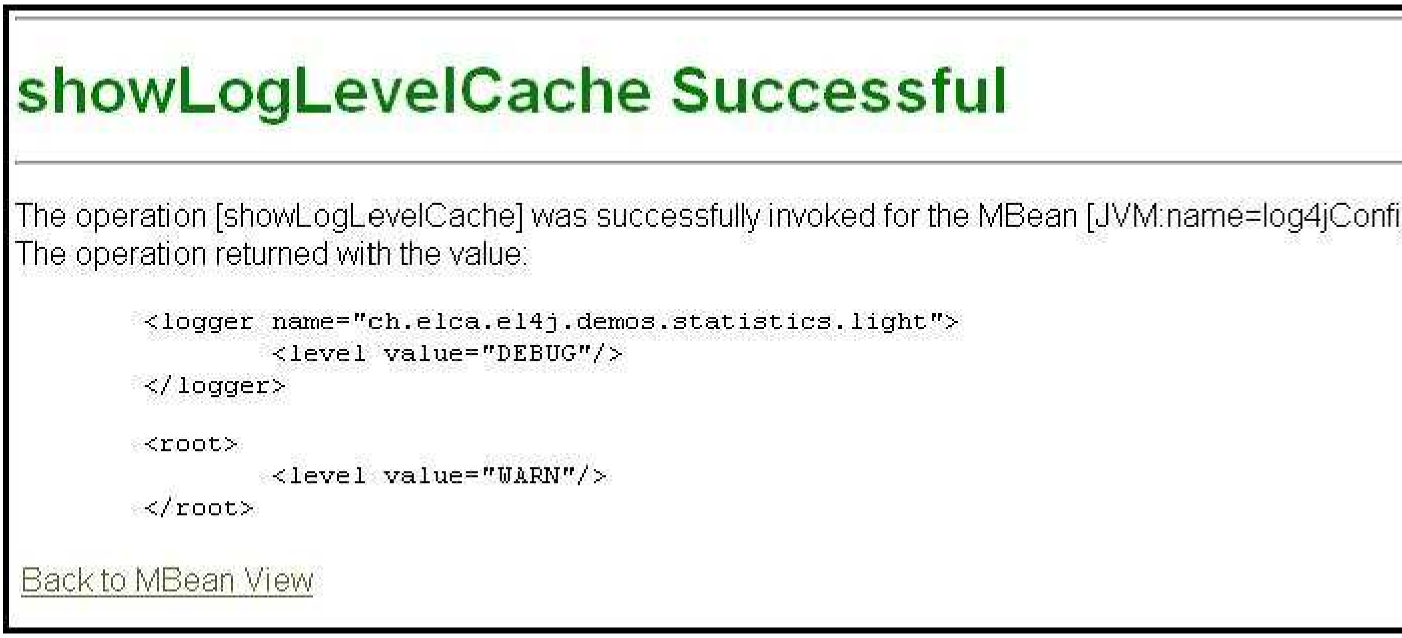
org.apache.log4j.Level showLogLevel (java.lang.String)p1

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=log4jConfi
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.

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.

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.

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

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.

Documentation for module TcpForwarder

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.

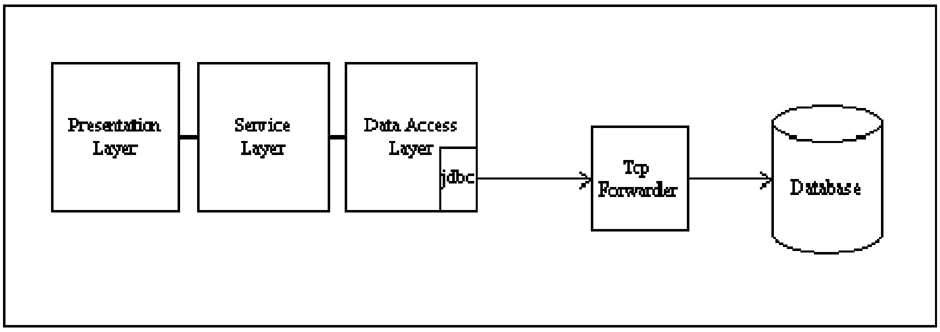
Limitations

The TCP forwarder does not work with oracle and probably other databases that provide load balancing. The problem is that during the first connection to the oracle database server it tells the client which server and port to use. Afterwards, the client connects directly to the given server and therefore bypasses the forwarder.

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:



How to use

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*.

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.

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

Notes

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

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.

Code configuration

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;
```

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);
```

Switch on / off connections

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

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/TestDB.java?revision=628&view=markup

Documentation for module Light Statistics

Purpose

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

edit purpose

Important concepts

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

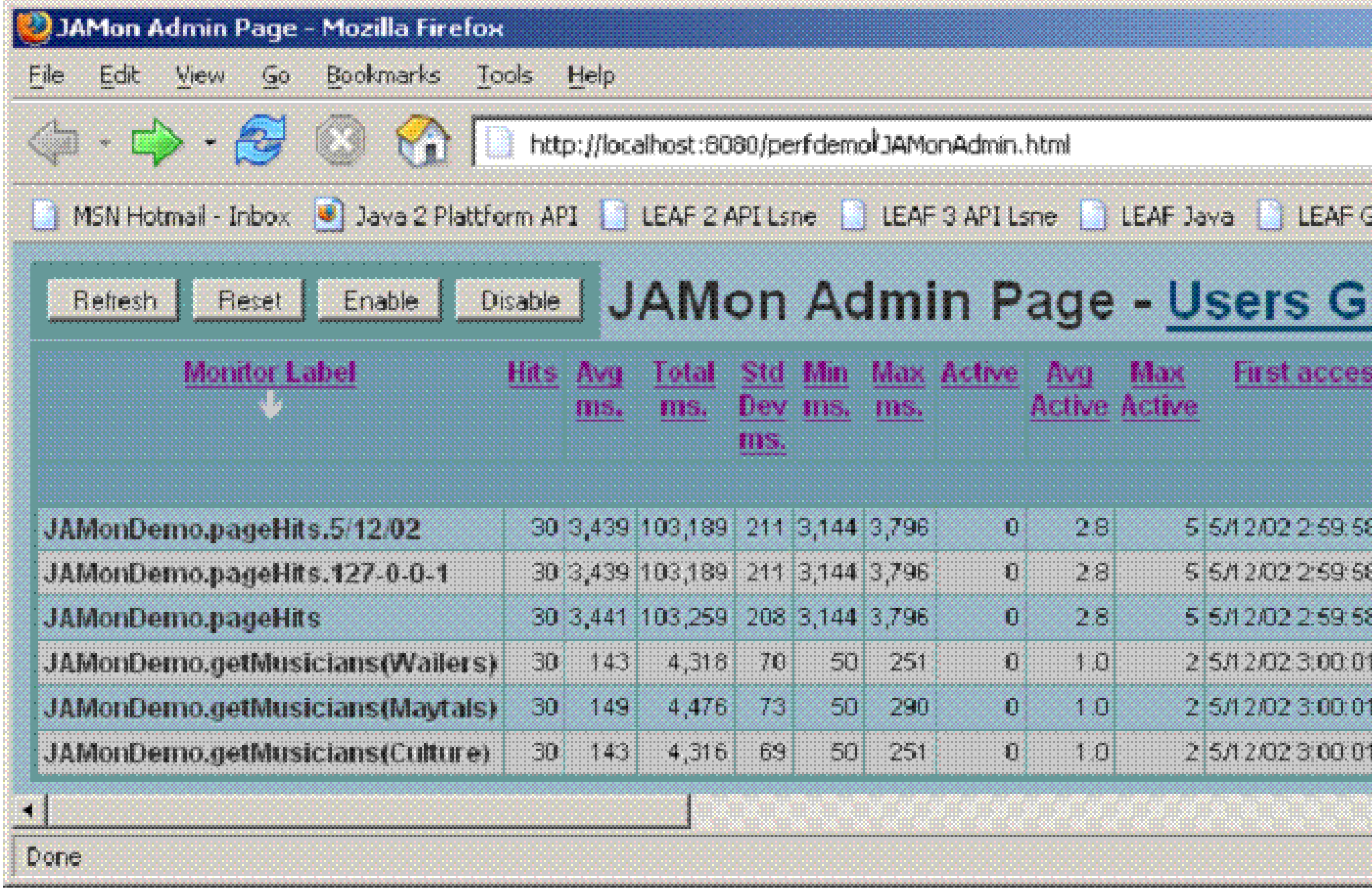
Monitoring strategies

- **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).

How to use

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.


JAMon Admin Page - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/perfdemo/JAMonAdmin.html
MSN Hotmail - Inbox
Java 2 Plattform API
LEAF 2 API Lsne
LEAF 3 API Lsne
LEAF Java
LEAF G
Refresh
Reset
Enable
Disable
Done

JAMon Admin Page - Users G

Monitor Label	Hits	Avg ms.	Total ms.	Std Dev ms.	Min ms.	Max ms.	Active	Avg Active	Max Active	First acces
JAMonDemo.pageHits.5/12/02	30	3,439	103,189	211	3,144	3,796	0	2.8	5	5/12/02 2:59:58
JAMonDemo.pageHits.127-0-0-1	30	3,439	103,189	211	3,144	3,796	0	2.8	5	5/12/02 2:59:58
JAMonDemo.pageHits	30	3,441	103,259	208	3,144	3,796	0	2.8	5	5/12/02 2:59:58
JAMonDemo.getMusicians(Wailers)	30	143	4,316	70	50	251	0	1.0	2	5/12/02 3:00:01
JAMonDemo.getMusicians(Maytals)	30	149	4,476	73	50	290	0	1.0	2	5/12/02 3:00:01
JAMonDemo.getMusicians(Culture)	30	143	4,316	69	50	251	0	1.0	2	5/12/02 3:00:01

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.

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:

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"/>
```

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>
```

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.

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.

References

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

Documentation for module Detailed Statistics

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.

edit purpose

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.

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.

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 composed of machine name and invocation time,

- a **sequence number** which corresponds to the call-level. As beans can call several other beans, the sequence number depth increases for each sub-bean,
- 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).

How to use

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.)

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 (`detailedStatisticsReporter`), you can get various information:

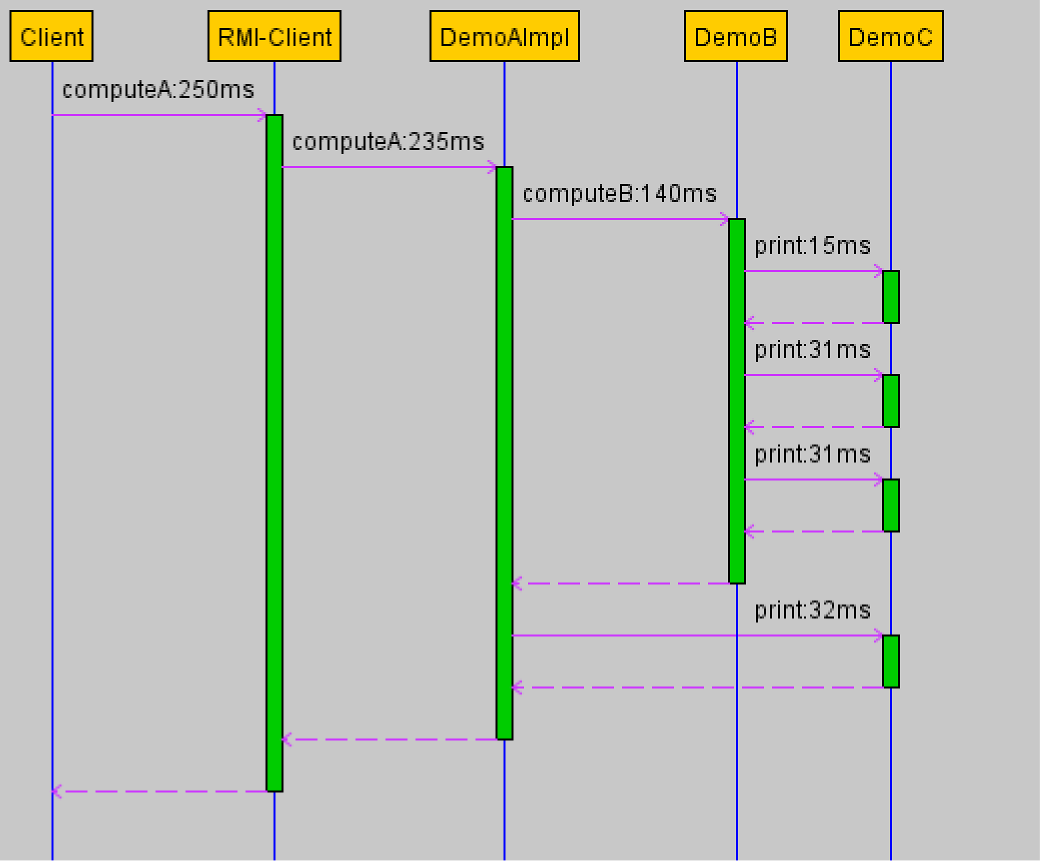
- The list of all recent measure IDs (via the method `showMeasureIDTable`)
- All measures corresponding to one measure ID in a CSV file (comma separated excel file). For this, you use the method `createCSVFile`. The first argument is the name of the file, the second argument is the measure ID. Refer to `showMeasureIDTable` (see line above) for a list of available measure IDs. 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.

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. The demo application runs in two JVMs, which communicate through RMI.

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.txt



Documentation for module ShellLauncher

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.

edit purpose

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(), ...

Documentation for module XmlMerge

Purpose

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

edit purpose

Introduction

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

Here is a merge example:

```
<root>
  <a>
    <b/>
  </a>
<d id="0"/>
<d id="1"/>
</root>
```

original

+

```
<root>
  <a>
    <c/>
  </a>
<d id="1"
newAttr="2"/>
</root>
```

patch

=

```
<root>
  <a>
    <b />
    <c />
  </a>
<d id="0" />
<d id="1"
newAttr="2" />
</root>
```

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

Module contents

The module contains the following stuff:

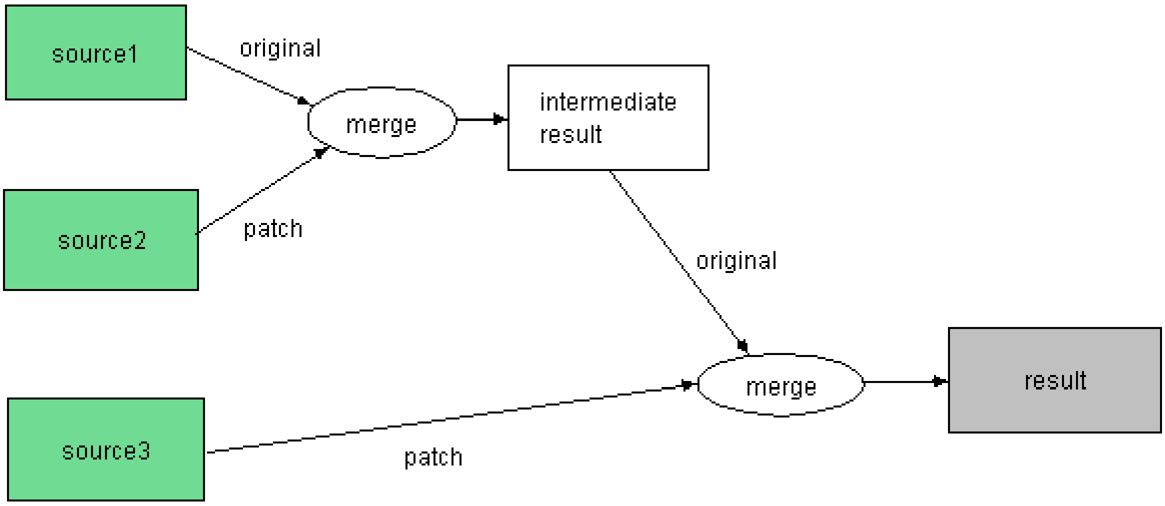
- Interfaces and infrastructure supporting the concepts the module is based on (in fact this forms the API and SPI).
- Default implementations of these interfaces.
- Convenience support for configuring your merge using XPath (outside of the XML documents) or with XML attributes within the XML documents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

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

Important concepts

Original and Patch

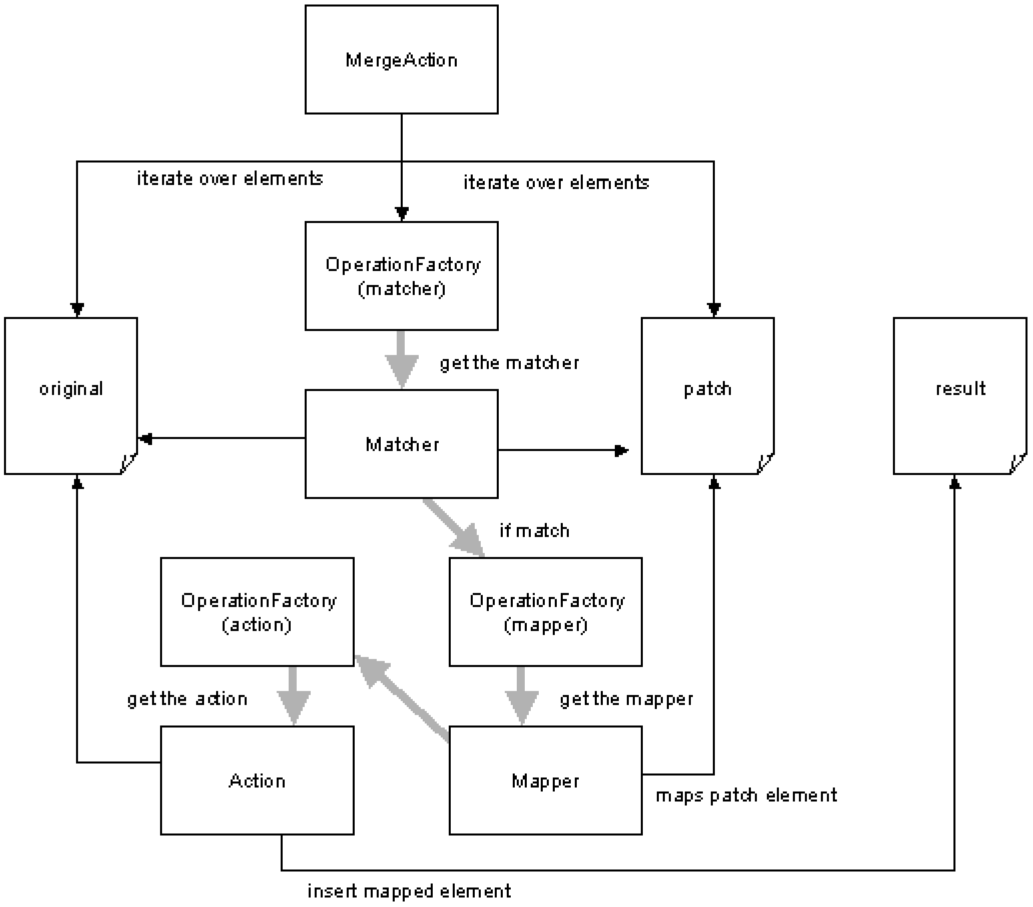
The *sources* are The XML documents (as java.lang.String, java.io.InputStream or org.w3c.dom.Document) that we want to merge.



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

Processing model

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



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

Core Concepts as Java Interfaces

See also the javadocs.

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

Operations

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

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

Configuration with Factories

Used terminology

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

- **XmlMerge**. Entry-point to perform the merge, it provides the `merge` methods. One can configure it by injecting the **MergeAction** and **Mapper** applied to the root element.

- **Configurer**. Interface for convenience classes configuring the root merge action and root mapper of an XmlMerge instance; thus, it can also configure the operation factories. This way, with only a few lines of code, one can use XmlMerge. The **ConfigurableXmlMerge** wrapper class automatically applies a **Configurer** on an XmlMerge instance.

Built-in implementations

Operations

The module provides implementations of operations that are commonly used:

Matchers

- **TagMatcher**. The original and patch elements match if the tag name is the same.

- **IdentityMatcher**. The original and patch elements match if the tag name are the same and the *id* attribute value are the same.

- **SkipMatcher**. The original and patch elements never match. Useful to force inserting the elements.

Mapper

- **IdentityMapper**. "Do nothing" mapper, it returns an exact copy of the element.

- **NamespaceFilterMapper**. Maps by removing all elements and attributes of a given namespace. Useful with the **AttributeOperationFactory** which allow defining the actions to apply as attributes in the patch document.

Actions

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

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

Aliases for Built-In Operations

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

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

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

XmlMerge Implementation

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

Operation Factories

Three implementations of the operation factory are provided:

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

Configuring your Merge

You have currently three ways to configure an XmlMerge instance:

Programming the Configuration

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

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

original patch result

```
public void testXPathOperationFactory() throws Exception {

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

        XmlMerge xmlMerge= new DefaultXmlMerge();

        MergeAction mergeAction = new OrderedMergeAction();

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

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

        factory.setOperationMap(map);

        mergeAction.setActionFactory(factory);

        xmlMerge.setRootMergeAction(mergeAction);

        String result = xmlMerge.merge(sources);

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

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

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

Configuring with XPath and Properties

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

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

```
xpath.pathName=XPath
```

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

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

Example:

`test.properties`:

```
action.default=COMPLETE

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

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

original   patch   result

```
public void testPropertyXPathConfigurer() throws Exception {

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

        String result = xmlMerge.merge(sources);

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

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

Configuring with Inline Attributes in Patch Document

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

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

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

original

+

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

patch

=

```
<root>

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

result

```

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

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

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


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

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

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

}
```

Writing your own Operations

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

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

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

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

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

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

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

+

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

=

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

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

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

  test.ByeServlet
        </servlet-
        class>
      <init-param>
        <param-name>
           message
         </param-
        name>
          <param-
       value>
           Bye bye!
         </param-
       value>
      </init-param>
  <servlet-mapping>
     <servlet-name>
          bye
    </servlet-name>
      <url-pattern>
           /bye
     </url-pattern>
      </servlet-
      mapping>
     </web-app>
```

original

```
```

patch

```
       </servlet>

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

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

result

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

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

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

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

ServletNameMatcher implementation:

```
package com.mycompany;

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

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

How to use

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

Command-line Tool

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

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

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

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

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

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

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

Ant Task

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

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

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

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

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

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

Spring Resource

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

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

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

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

Web demo

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

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

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

Debug output

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

References

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

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

Documentation for module SocketStatistics

Purpose

With the `SocketStatistics` module, you can get statistics, logs and an inside view about the Socket connections of your (any) java application. The information can be accessed directly over an M(X)Bean using jconsole or the java visual vm and is logged using the already present logging facility of the monitored application.

Important concepts

The module is mainly based on an implementation of `SocketImplFactory` which creates (and returns) an alternative implementation of the interface `SocketImpl`: `SocketImplLogger`. Particular methods of this class (`bind`, `connect`, `getInputStream` and `getOutputStream`) do not only delegate the calls to an instance of `java.net.SocksSocketImpl`, but also handle the logging. The forwarding of method calls is handled by the `ReflectiveDelegator` class using reflection.

The traffic logging in its turn is handled by `OutputStreamLogger` and `InputStreamLogger` - both extensions of the `OutputStream` / `InputStream` classes - which are returned to the application when the Java Socket class is using `SocketImplLogger` for the creation of a new Socket.

The gathering of statistics is in the responsibility of the actual `SocketStatistics` and the `ConnectionStatistics` classes. Every Socket has therefore a reference to a `ConnectionStatistics` instance.

Logging

The `SocketStatistics` Module uses the logging facility of the underlying / calling application. Therefore, the produced log entries of `SocketStatistics` can be found inside the applications log. To achieve this, a `GenericLogger` is introduced.

At creation time of a new `GenericLogger` instance, the `GenericLogFactory` checks for the presence of a an already initiated log facility from the running application. The search order is as follows:

1. `SLF4J` (org.slf4j.LoggerFactory)

2. Apache Commons Logging (org.apache.commons.logging)
3. Log4J (org.apache.log4j)

If none of these logging facilities are found, the JDK logging facility (java.util.logging) is used with the default configuration.

How to use

There are two possibilities to use SocketStatistics:

- inside your code (Method1)
- directly by the java vm / runtime (Method2)

💡For use according to Method2, the SocketStatistics jar can be downloaded directly from the Maven 2 EL4J repository

Method 1 - inside your code

If you want to use SocketStatistics within your own code, this can be done by three simple line of java:

```
SocketImplFactory sif = new LoggerSocketFactory();
Socket.setSocketImplFactory(sif);
ServerSocket.setSocketFactory(sif);
```

With these lines, we replace the basic SocketImplFactory for all further created Sockets in the natural "Java way".

Method 2 - directly by the java vm / runtime

The second approach is a bit more hacky - but rather convenient.

Within the SocketStatistics module, there is also a slightly modified version of java.net.Socket. In this modified Socket, the LoggerSocketFactory is used as default SocketFactory. So every created Socket (by the use of the modified java.net.Socket class) has the capability of logging and the generation of statistics.

This version of java.net.Socket is "injected" to the java vm using the Xbootclasspath option.

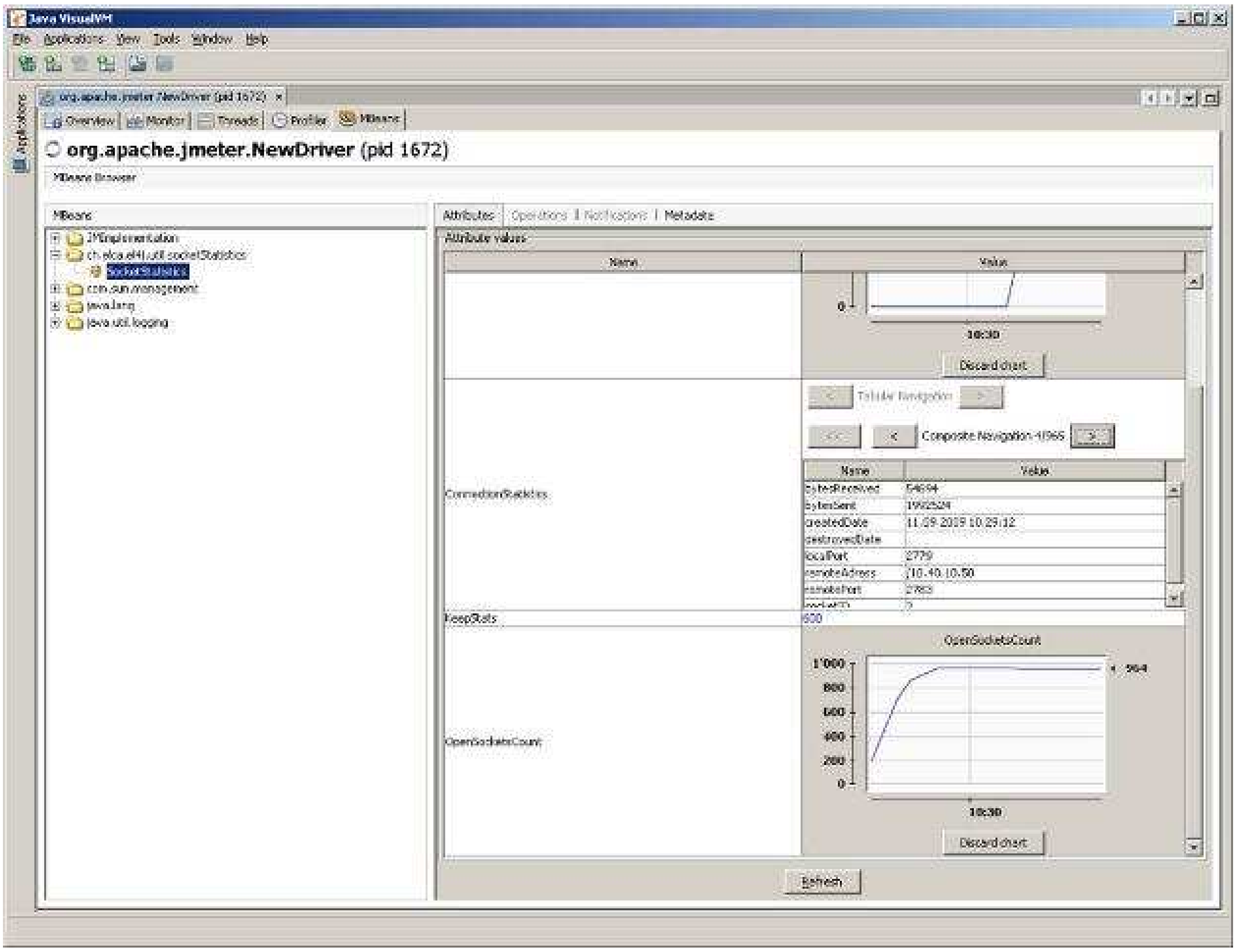
```
java -Xbootclasspath/p:[PATH TO THE SocketStatistics JAR] [REST OF THE
NORMALY USED STATEMENTS]
```

Hint: Using the extra paramter /p, the classes inside the jar are prepend in front of the default bootstrap class path - and do not completely replacing the whole bootstrap class path.

Monitoring using M(X)Bean

On the first call of the method createSocketImpl, the `LoggerSocketFactory` registers an MXBean on the `PlatformMBeanServer`. Therefore, the information about open (and closed) Sockets is also accessible over the ch.elca.el4j.util.SocketStatistics MBean using jconsole or the java visual vm.

💡If the MBean Tab is missing in the java visual vm, it has to be installed as plugin first. Go to Tools -> Plugins -> Available Plugins Tab -> Select VisualVM? - MBeans -> Click on install



org.apache.jmeter.NewDriver (pid 1672)

Name	Value
ConnectionStatistics	
KeepStats	600
OpenSocketsCount	

Name	Value
bytesReceived	54694
bytesSent	1992524
createdDate	11.09.2009 10:29:12
destroyedDate	
localPort	2779
remoteAdress	10.40.10.50
remotePort	2783
socketID	2

The values shown are basically rather self explaining. The **editable** **`KeepStats`** **value** configures how long (in seconds) after a Socket has been closed it will still appear in the list of statistics.

Operations in the SocketStatistics M(X)Bean

Apart from simple monitoring, there are also two operations available in the MBean.

- **exportStatisticsCSV** exports all gathered statistics of open and closed sockets to a .csv file. As argument, a full path to the csv is required (e.g. `C:\outputdir\stats.csv`).
- **deleteStatistics** deletes all previously gathered statistics.

Usage Example(s)

On Tomcat with Method2

This is a simple, straight forward example, how to monitor all connections on a tomcat server using `SocketStatistics`.

To enable `SocketStatistics` for a tomcat server (on windows) search for the catalina.bat file inside the tomcat\bin directory. In this file, replace the line

```
set JAVA_OPTS=%JAVA_OPTS -
Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager [...]
```

with this line

```
set JAVA_OPTS=%JAVA_OPTS -Xbootclasspath/p:[PATH TO THE SocketStatistics
JAR] -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
[...]
```

-- JonasHauenstein - 23 Sep 2009

The Hibernate Offliner

Introduction

Purpose

We have a database that we access via hibernate in a client/server environment. We would like to be able to offline a subset of the data on the client and work with it offline, then resynchronize with the server.

The offliner does this by using a second database local to the client (and known as the local database). This means at any time the user can work with a database (daos, queries etc.). In addition, he can perform offliner operations.

Site map/ structure of documentation

- HibernateOffliner The offliner main page.
    - OfflinerSpec Offliner specifications.
        - OfflinerKeyVersion Key and version handling.
        - OfflinerStrategies Chunking strategies design.
        - OfflinerTests List of all offliner tests and what they test.
    - OfflinerImpl Offliner implementation details.
        - OfflinerOffline The offline operation.
        - OfflinerSync The synchronization operation.
        - OfflinerMappingEntries Mapping entries implementation and meaning.
        - OfflinerKeyManagement Implementation of key management.
    - OfflinerGraphWalker The graph walker package of the offliner.
    - OfflinerAspects The aspects package of the offliner. (Renamed to objectwrapper.)

        - OfflinerAspectsList List of currently available aspects and implementations.
    - OfflinerVersions Version history, differences and future plans.
    - OfflinerLimitationsAndPerformance
- Powerpoint presentation under this link.

Offliner Terminology

Remote

Refers to the remote database which holds the master copy of the data. We can either work with it directly or offline some data, use it then resynchronize. Where appropriate, the full term remote database is used.

Local

Refers to the client or the client's local copy of some data.

Database

Unqualified, this is the remote database. "Local database" is used where necessary to mean the client's local database seen as a database.

Offline (noun)

Refers the the client's local copy of some data.

offline (verb)

To copy data from the database into the local database.

(re)synchronize

To copy data from the local database back to the database, managing versioning and optimistic locking conflicts in the process.

Usage

The offliner implements DaoRegistry. Applications must get all their daos from the offliner and not cache them in between calls that change the offliner's state. At any time, the offliner returns a dao for the currently active data source (remote database or local one). The offliner can be set to the states online and offline to switch between the two.

The offliner supports the two operations offline and synchronize. Offline copies a set of objects from the database into the local database, either the objects themselves can be passed to the method or a hibernate query in which case the offliner executes it on the database and puts the results in the local database.

Synchronize takes no parameters, copies all changed data to synchronize the local database with the database and returns the conflicts (objects changed both in the local database and the database, which it won't overwrite) encountered.

The best way to use the offliner is

1. Offline some data.
2. Go offline.
3. Use and edit the local data.
4. Synchronize.
5. Repeat the last two steps as often as you like in any order.

Offliner demo

The demo is under `internal/sandbox/beanbrowser-offliner-demo` (read the readme file there). FYI: The demo under `internal/applications/demos/imputations-offliner` has some issues.

Offliner setup

Both the client and the server part of the offliner use the object wrapper package. This is documented separately; it is enough to copy the provided configurations for it.

On the server, run an OffliningServer which needs a DaoRegistry for the database and an object wrapper implementation. On the client, run the OfflinerClientImpl and link it to a OfflinerInfo object which you have filled with the necessary data:

- The client-side object wrapper implementation.
- The client-side DaoRegistry for the local database (the local database).
- A client-side DaoRegistry for the server dao remoting proxies.
- The domain classes you wish to use with the offliner and a chunking strategy for each.

- The offlining server (remoting proxy).
- An offlining state table (just use an instance of the default implementation).

Setup in detail

Common beans (client and server)

```
   <!-- Common wrappers. -->

   <bean id="objectWrapper"
class="ch.elca.el4j.util.objectwrapper.ObjectWrapper">
      <property name="wrappables">
         <map>
            <entry
key="ch.elca.el4j.util.objectwrapper.interfaces.KeyedVersioned" value-
ref="keyedVersioned" />
            <entry key="ch.elca.el4j.util.objectwrapper.interfaces.Linked"
value-ref="linked" />
            <entry
key="ch.elca.el4j.services.persistence.hibernate.offlining.objectwrapper.Ma
pped" value-ref="mapped" />
            <entry
key="ch.elca.el4j.services.persistence.hibernate.offlining.objectwrapper.Ty
ped" value-ref="typed" />
            <entry
key="ch.elca.el4j.services.persistence.hibernate.offlining.objectwrapper.Un
iqueKeyed" value-ref="uniqueKeyed" />
            <entry
key="ch.elca.el4j.services.persistence.hibernate.offlining.objectwrapper.Of
fliningStateWrappable" value-ref="offliningState" />
         </map>
      </property>
   </bean>

   <bean id="keyedVersioned"
class="ch.elca.el4j.util.objectwrapper.impl.KeyedVersionedHibernateImpl">
```

```
      <constructor-arg ref="sessionFactory" />
   </bean>

   <bean id="linked"
class="ch.elca.el4j.util.objectwrapper.impl.LinkedHibernateImpl">
      <constructor-arg ref="sessionFactory" />
   </bean>

   <!-- Mapped is separate for local and remote. -->

   <bean id="typed"
class="ch.elca.el4j.services.persistence.hibernate.offlining.objectwrapper.
impl.TypedImpl" />

   <bean id="uniqueKeyed"
class="ch.elca.el4j.services.persistence.hibernate.offlining.objectwrapper.
impl.UniqueKeyedImpl" />

   <bean id="offliningState"
class="ch.elca.el4j.services.persistence.hibernate.offlining.objectwrapper.
impl.OffliningStateTableImpl">
      <constructor-arg ref="stateTable" />
   </bean>

   <!-- The state table. Although this is in common, there are two
different
   tables on the client and the server. -->

   <bean id="stateTable"
class="ch.elca.el4j.services.persistence.hibernate.offlining.util.Offlining
StateTable" />
```

Server-side beans

```
   <!-- It is assumed the database is set up as usual in EL4J. -->
   <!-- The mapping table server implementation (in-memory). -->
   <bean id="mappingTable"
class="ch.elca.el4j.services.persistence.hibernate.offlining.util.ServerMap
pingTable" />

   <!-- The mapped server implementation. -->
   <bean id="mapped"
class="ch.elca.el4j.services.persistence.hibernate.offlining.objectwrapper.
impl.MemoryMappedImpl">
      <constructor-arg ref="mappingTable" />
   </bean>

   <!-- The offlining server. -->
   <bean id="offliningServer"
class="ch.elca.el4j.services.persistence.hibernate.offlining.impl.Offlining
ServerImpl">
      <constructor-arg ref="daoRegistry" />
      <constructor-arg ref="mappingTable" />
      <constructor-arg ref="objectWrapper" />
      <constructor-arg ref="stateTable" />
   </bean>
```

Client-side beans

```
   <!-- It is assumed the client-side database is set up correctly and the
server daos are exported as proxies to the client. -->

   <!-- The dao registry. We must restrict this not to inlcude server dao
proxies. One possibility is to name client-side daos *Dao as usual and
server proxies *Rao. -->
   <bean id="daoRegistry"

class="ch.elca.el4j.services.persistence.generic.dao.impl.DefaultDaoRegistr
y">
      <property name="namePattern" value="*Dao" />
   </bean>
   <!-- A second DaoRegistry that collects the server-side dao proxies. -->
   <!-- A "RAO" is a "remote access object", to keep them distinct from the
local "DAO" ones. -->
   <bean id="daoRegistryRemote"

class="ch.elca.el4j.services.persistence.generic.dao.impl.DefaultDaoRegistr
y">
      <property name="namePattern" value="*Rao" />
   </bean>

   <!-- The client-side mapping table dao. Note that this is not
autocollected by default as it is in a different package from the domain
classes. This is ok because only the offliner uses it (in the mapping table
implementation) and needs an explicit reference to it anyway. -->
   <bean id="mapDao"
class="ch.elca.el4j.services.persistence.hibernate.offlining.impl.MappingTa
bleDao" />

   <!-- The offliner properties dao. -->
   <bean id="propertyDao"
class="ch.elca.el4j.services.persistence.hibernate.offlining.util.PropertyD
ao" />

   <!-- The client-side mapping table implementation that uses the mapping
dao. -->
   <bean id="mapped"
class="ch.elca.el4j.services.persistence.hibernate.offlining.objectwrapper.
impl.DatabaseMappedImpl">
      <constructor-arg ref="mapDao" />
   </bean>

   <!-- Replace this with a proxy for the server-side offlining server. -->
   <!-- <bean id="server"
class="ch.elca.el4j.services.persistence.hibernate.offlining.test.RmiOfflin
ingServerSimulator" scope="singleton"/> -->
```

```
   <!-- The offliner info object. All client-side configuration is done
here. -->
   <bean id="info"
class="ch.elca.el4j.services.persistence.hibernate.offlining.impl.OfflinerI
nfo">

      <!-- The object wrapper. -->
      <property name="wrapper" ref="objectWrapper"/>

      <!-- The client-side dao registry. -->
      <property name="clientDaoRegistry" value="daoRegistry" />

      <!-- The dao registry for the server proxies. -->
      <property name="serverDaoRegistry" value="daoRegistryRemote" />

      <!-- The server. -->
      <property name="server" ref="server" />

      <!-- The state table. (We should really instantiate this internally,
      but then we can't use spring for the wrapper package either.) -->
      <property name="stateTable" ref="stateTable" />

      <!-- Create one singleton instance of each strategy you want to use.
      <bean id="allStrategy"
class="ch.elca.el4j.services.persistence.hibernate.offlining.chunk.AllStrat
egyImpl" scope="singleton" />
      -->

      <!-- The classes map. This must be a linked hash map to preserve
order of the keys.
      It has two purposes: One, as a sorted list of all classes to iterate
over when
      synchronizing. Two, to provide a chunking strategy for each class. --
>
      <property name="classes">
```

```
         <util:map map-class="java.util.LinkedHashMap">
            <!-- Enter domain classes and strategies here. -->

            <!--
            <entry key="test.testclasses.Person" ref="allStrategy"/>
            -->
         </util:map>
      </property>
   </bean>

   <!-- The actual offliner. Note that we use the spring implementation as
it is required to ensure
   the context is ready before we access any DAOs. If you create the
offliner in java, use the
   OfflinerClientImpl directly. -->
   <bean id="offliner"
class="ch.elca.el4j.services.persistence.hibernate.offlining.impl.OfflinerS
pringImpl">
      <constructor-arg ref="info" />
   </bean>
```

Setting up the database

The database set-up is the user's responsibility. The remote database will usually be given, the local database must be identical to hibernate but use keys from a disjoint set. For example, if you have a domain object SimplePerson with the following table definition in the remote database:

```
CREATE TABLE SIMPLEPERSON (
    ID BIGINT NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY
      (START WITH 1, INCREMENT BY 1),
    VERSION BIGINT NOT NULL,
    NAME VARCHAR(40) NOT NULL,
    EMAIL VARCHAR(40) NOT NULL
);
```

You could do the same locally except that you replace the key generator with (START WITH -1, INCREMENT BY -1). The Typed implementation must be compatible with your strategy, this (positive/negative) is the default strategy although it does not work under Oracle.

You must implement Typed yourself if you want to change this scheme, here is the default:

```
   /** {@inheritDoc} */
   public KeyType getType() {
      Serializable key = m_wrapper.wrap(KeyedVersioned.class, m_target)
         .getKey();
      if (key instanceof Long) {
         Long keyAsLong = (Long) key;
         return (keyAsLong.equals(0L) ? KeyType.NULL
            : (keyAsLong > 0L ? KeyType.REMOTE : KeyType.LOCAL));
      } else if (key instanceof Integer) {
         Integer keyAsInt = (Integer) key;
         return (keyAsInt.equals(0) ? KeyType.NULL
            : (keyAsInt > 0 ? KeyType.REMOTE : KeyType.LOCAL));
      } else {
         throw new IllegalArgumentException("Key not of type Long."
            + " The default implementation requires this.");
      }
   }

   /** {@inheritDoc} */
   public void nullKey() {
      try {
         m_wrapper.wrap(KeyedVersioned.class, m_target).setKey(0L);
      } catch (ClassCastException ex) {
         throw new IllegalStateException("The default implementation
requires "
            + "keys of type long.");
      }
   }
```

Finally, you must set up the mapping and property tables in the local database. The create script is in the offliner's test resources, the relevant part is

```
CREATE TABLE KEYMAP (
   ID INT NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY
      (START WITH 1, INCREMENT BY 1),
   LOCALBASEVERSION VARCHAR(128) NOT NULL,
   REMOTEBASEVERSION VARCHAR(128) NOT NULL,
   DELETEVERSION BIGINT NOT NULL,
   SYNCVERSION INT,
    LOCALKEY VARCHAR(128) NOT NULL,
    REMOTEKEY VARCHAR(128) NOT NULL
);

CREATE TABLE OFFLINERPROPERTIES (
   ID INT NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY
      (START WITH 1, INCREMENT BY 1),
   PROPNAME VARCHAR(30) NOT NULL UNIQUE,
    PROPVALUE VARCHAR(40)
);
```

Offliner Specifications

Functionality

An offliner implementation adds offlining functionality to a database accessed via DAOs.

To use offlining, the client must get his DAOs from the offliner by using it as a DaoRegistry. These DAOs must not be cached by the client between state changes in the offliner (which the client initiates).

An offliner has two states: online and offline. In online mode, it forwards all calls to DaoRegistry.getFor(Class) to the database's DaoRegistry. In other words, it performs as if you were working directly on the database with no offliner present. In offline mode, the offliner returns DAOs for a local database. At all times, the offliner's DaoRegistry implementation is guaranteed to return a valid DAO for the active database (unless no such DAO exists).

The offliner offers two operations: offline and synchronize.

After performing a set of offline operations, the client may go offline and work with the local database. He can perform multiple offline or synchronize operations in sequence.

In addition, the offliner allows the client to clear the local database completely.

Conflicts

A conflict can occur on offlining or synchronizing the local with the remote database. It represents an object that caused an exception while trying to save it.

Exceptions during `synchronize` that occur in a `saveOrUpdate` or `delete` operation are caught by the offliner and wrapped in conflict objects. These contain, at least, the original exception, the phase during which the exception occurred and the object that caused the exception if present (the only time it is not present is if we are trying to delete an object, already deleted on the client, on the server). One special case is dependent conflicts: If an object has one or more children that caused a conflict, the parent object is automatically marked as conflicted and it is not even attempted to save to the database.

`synchronize()` returns a `Conflict[]` that is of length 0 if the operation was successful, otherwise it contains all conflicts that occurred. For each conflicted object, exactly one conflict must be returned. If an object is both locally and dependently conflicted, the dependent conflict takes precedence as no attempt can be made to save a dependently conflicted object.

Deleting

There are two kinds of delete imaginable on the local database:

The first is removing an object to prevent it from being resynchronized, to cancel local changes, or simply because we do not need it in the local database anymore. This kind of delete functions as if the object had never been offlined and is accessible over `Offliner.evict()`.

The second is requesting an object be deleted from the server, in which case the offliner has to mark it as deleted and process the deletion on synchronization. This is currently implemented by wrapping all local DAOs and firing `Offliner.markforDeletion()` in the delete methods.

Offliner Key and Version handling

Hibernate assumes it alone is responsible for the primary keys and versions of persistent objects and the user will never do something unusual like change them himself. Unfortunately, when dealing with two databases, we have to do exactly this and cheat, hack and trick our way around hibernate's assumptions.

Hibernate's assumptions

- An object with a key of 0 (*For numeric keys. Possibly `null` for string keys, not tested.*) is new. It will recieve a key on saving into the database. The version field is saved as it is, no version conflicts can occur for new objects.
- An object with a non-zero key is assumed to be a saved instance. Hibernate looks for the database entry with this key and
    - If none exists, assumes the object was deleted in the meantime. Optimistic locking conflict.
    - If it exists, compares the versions:
        - If they are equal, the object is saved and the version incremented.
        - If the versions differ, a versioning (optimistic locking) conflict is thrown.

Getting past these assumptions

The basic principle of "cheating" hibernate is *whenever an object is saved to a database, it must carry the exact key and version it last had when it came from that database*.

The second principle is whenever an object is newly saved to the local database, its key must be nulled first.

From this, it is clear that we somewhere have to store metadata for each object holding at least the "other" key and version, so we can swap them with the object's current ones when we move it from one database to the next. This metadata for an object is called its ***mapping entry***.

The mapping entry

In fact we hold all data in the mapping entry - that is, local and remote keys and versions. This allows us to tell easily when an object has been changed: For X being one of "local" or "remote", if an object has an X key then it has been changed since the last offline/sync (and thus needs some kind of update in the next one) if and only if its X version differs from its X "base" version in the mapping entry.

The current state of an object can be found by looking at its key: If is from the REMOTE subset of all possible keys (as defined in the TypedAspect implementation) it is from the database, otherwise (LOCAL) from the local database (NULL keys are an exception in any case, the offliner should never see any when moving objects between databases).

The mapping entry stores UniqueKey instances instead of the actual keys. As unique keys and actual keys can be converted back and forth, this is just an indirection and not a problem.

Key and Version Modification

The operations try to copy objects from one database to the other. They fail if changed data would be overwritten. Force operations are meant for conflict resolution and overwrite their target even if it has been changed.

Note that a force(obj) will only force obj but not its children. Therefore, forcing objects which are dependent-conflicted will not work (and leave the mapping entry unchanged). The object that caused the conflict first must be determined from the conflict type and forced.

Adding an object to the local database that is new there

1. The object's remote key and version are saved to the mapping entry to allow us to copy it back.

2. The object's key is nulled and the object saved to the local database.

3. The object's local key and version are saved to the mapping entry.

Updating an object in the local database with a new server version

We assume the server version is greater than the remote base one. If they are equal, we can skip the save; if it is smaller something is wrong with the server. We are only allowed to overwrite a local object of local base version (i.e. not a changed local object) except in a force operation.

1. The remote version is copied into the mapping entry.

2. The object's key and version are set to the current local one (queried from the local database) to allow the overwrite.

3. The object's local version is copied to the metadata from the saved instance. (Unless the last step failed).

Synchronizing a locally changed object with the server (Phase 1)

This procedure is done on the server.

1. The object's local version is saved temporarily.

2. The object's key and version are set to the remote ones.

3. The update is performed.

If the update suceeds,

1. The object's remote version is saved in the mapping entry.

2. The object's local version is set to the one saved in step one. This makes the object "unchanged".

If the update fails, the mapping entry versions are left as they are so the object does not appear unchanged. The version is left one the remote one in case the user needs to investigate it from the Conflict object.

Synchronizing a new object in the local database with the server (Phase 1)

1. The object's local version is saved temporarily.
2. The key is nulled.
3. The object is saved to the database. (If it fails, we abort here.)
4. The local key is saved from the temporary to the mapping entry.
5. The remote key and version are saved to the mapping entry.

Synchronizing a remotely changed object (Phase 3)

The object is processed as in "updating an object in the local database with a new server version". In fact, phase 3 just fires an offline operation on all changed objects.

Forcing an overwrite in the local database

1. The object's remote version in the metadata is set to its current remote version.
2. The key and version are set from the current local one to force the update.
3. The object is saved to the local database.
4. The local version from the saved object is saved to the mapping entry.

Forcing an overwrite on the server

1. The object's local version is saved to the mapping entry.
2. The remote instance is loaded from the database and the object's key and version set from it.
3. The object is saved to the database.
4. The object's remote version is saved into the mapping entry.

Offlining/Batching strategies

Purpose

Synchronizing a set of data requires each element to be copied at least once. The strategies described here aim to achieve that a) each element is copied only once and b) objects can be copied in chunks, that is not all objects need to be copied at once. A trivial implementation of chunking risks increasing the copy effort for `n` objects from `O(n)` to `O(n^2)`. THe reason for this is that whenever we copy an object, all objects it has references to are copied as well.

Strategies

A class is "out-independent" if no two instances of this class depend on each other. (For instance, several Imputations may depend on the same ImputationNumber, but no two Imputations depend on each other in the sense that following any chain of pointers from one leads you to the other. So Imputation is an out-independent class.)

For such classes we can batch transfers by collecting any number of instances in an Object[] and transferring the whole array at once. This can only transfer an object several times within a batch if two instances hold a pointer to the same logical element of another class, but they are different instances i.e. a and b of class C1 both have pointers to x of class C2 and a.c2.equals(b.c2) `= true but a.c2` b.c2. This can be fixed by an identity fixing scheme.

If a and b are not in the same batch, the client will either have to resend x when b is transferred or have an option to transfer b without its dependency x and relink them on the server.

We can avoid ever sending an object twice if each object graph is sent in one batch. This leads to the following problems: if a --> x and b --> x (--> = has pointer to), we need to somehow notice that a and b belong to the same graph. The only way is to iterate over all objects in the local database and give each a "graph id". For n objects this takes up to 3n database read/writes (first pass: read objects and save ids, keeping id links in memory for speed, total 2n operations; second pass: readjust ids that point to the same graph, time 1n). If we have n elements with no links between them at all, these are 3n unnecessary operations on the database.

This strategy also allows us to send any number of disjoint object graphs in one batch.

Call a class "independent" if no two instances can ever have links that meet. (This covers all classes with no outgoing links at all or with only 1:* multiplicity outgoing links. Imputation is not independent because two Imputations can use the same ImputationNumber.) For such classes we do not need any object graph processing and can batch them in any way we like. Independent is a strict subset of out-independent.

In the case that all instances of a class form a connected graph (quite possible in real-world applications), if we transfer objects (trasferring an object takes all its dependencies with it) we either need to send the whole set of instances in one batch or resend some instances where we split into batches. If we could somehow transfer each object individually, breaking the links before sending and relinking on the server, we can batch however we want. (If we serialize locally, send the result as binary data in whatever chunk size we wish, and deserialize on the server again we can control the chunk size but still need to send the whole graph at once.)

Here is one way to achieve this: Assume each object can a) be saved as and reconstructed from a mapping of bean properties and b) has a unique id. If a has a link to b, instead of transferring b with a we look up b's id and only transfer the id along with a. (Which is pretty much what SQL does: hold a foreign key instead of the object itself.) We still cannot use a on the server without b, but if we transfer b first we can send a later (in another batch) and do not need to retransfer b.

Together with graph component searching, this allows us to batch pretty much anything in any way we want, subject only to the condition that when we want to save an object to the server database, all its dependencies must be there too. (If save does not cascade, we might get away with only the direct dependencies being present. Transitive dependencies just need to be ignored by hibernate.)

Consider a Person class with a "Person parent" and "Set children". In the database, only the parent pointer is saved (as foreign key) and children is marked as transient. If we serialize, we risk the children set being serialized along with any person we send to the server. (We could mark it as transient, but then when we deserialize we have to somehow restore the links). Even if the connected component of n Persons is small enough that this does not matter normally, a bad offlining strategy might send the whole graph n times.

Any serialization must somehow restore the links, unless all we want to do is save the person to the database in which case we can set children to null. (Getting the person back is harder, but we can save him to the local database, drop the Person object and reload from the local database and let hibernate do that work for us. However, if a child is missing we get "garbage in = garbage out".)

Again if we can transfer ids instead of object references, we send each object once only, but we do need to track which objects we have already sent so we never try and reconstruct a partially sent graph.

Offliner test cases

Running tests

The tests have to run for two databases (db2, oracle) with different implementation details and use two separate schemes (local, remote) in either case. This pushes the capabilities of maven and the standard plugins (database, env) to their limits and uses several "hacks" to go beyond those limits. Despite this, the tests should run automatically when building el4j under either database at the time of writing.

Strategy-independent tests

testKeyStrategy

Check that LOCAL and REMOTE key ranges are correct in both databases.

testOffline

Check offline operation works.
Check dependent objects are offlined too.

testOfflineIdentity

Check several offline operations in sequence do not produce clones in the local database.

testOfflineServerNew

Check offline corrrectly updates new objects from the server.

testOfflineFailOnNew

Check the local database fails if the local databased instance was updated.

testForceRemote

Create a conflict then resolve it by forcing the remote version.

Also check a force on a dependent conflict fails.

testForceLocal

Create a conflict then resolve it by forcing the local version.

Also check a force on a dependent conflict fails.

testDeleteResolution

Provoke a deletion conflict and ensure the resolutuion strategy works.

Strategy dependent tests

All these tests are rerun with all chunking strategies. Each strategy must derive a subclass.

testOfflineAndCommit

Read test data and save it back again.

This is simple offline-commit operation without modifying anything in the local database.

It must not produce any problems.

testModification

Modify object contents and references in the local database.

Ensure they are committed correctly.

testCreateInOffline

Create new objects in the local database.

They only acquire a server key on synchronization.

Ensure they are written back correctly.

testDelete

Delete an object in the local database causing it to be marked for deletion in the metadata and deleted on sync. Ensure this happens.

testDeleteConflict

Delete an object in the local database that has a non-offlined object pointing to it on the server.

Ensure the conflict is reported correctly.

testDeleteOrder

Ensure deletes are performed in the correct order on the server (important when there are references between the objects to delete).

Ensure deletes that already fail in the local database do not contribute to the order.

testWrongDeleteOrder

Force a conflict by messing with the delete order.

Most of this method is like testDeleteOrder, except for a modification to break the delete order.

testConcurrentFailure

Modify data on the server which is also in the local database, then try and recommit.

Check the conflict and its dependent conflicts are reported correctly.

testDependentConflict

Check that in a graph of objects where some are conflicted, those that are not conflicted are updated correctly.

testServerVersion

Ensure that a server version > 0 of data does not change offliner semantics.

testEvict

Test the offliner's evict function that removes an object from the local database without causing a delete on the server.

testMultipleSync

Check multiple synchronizations work as expected.

Cases:

- Change in local database.
- Change on server.

Offliner Implementation

The mapping table

Hibernate allows us to work with POJOs and let it handle all the key generation and management issues. For access to a single database, this makes it easier, for copying between databases, it is unhelpful. In particular, hibernate will not allow us to insert a row with a given primary key into a table unless that key already exists there.

To work around this, we store a mapping table which allows us, for each class, to look up the local and database key. For example, if we have a Bean with key 100 in the database and want to offline it, we first set the key to zero then store the bean locally. This makes hibernate generate a new key, say -42. We store the pair (100, -42) in the mapping table. On resynchronizing, we look up the local key and set the bean's key back to the one it had in the database.

The bare minimum requirements for keys are that

1. Keys are serializable and have proper equals/hashcode implementations (this is a hibernate requirement).
2. Local and remote keys are distinguishable. No key can appear in both the local and the remote database. This is required on resynchronisation to preseve object identity.

The current implementation is that all keys must be of type Long (that is capital-L to get Serializable) and the database must use positive, the local database negative keys. It would be nice to do this more generically but there are pitfalls with java generics if we do not have the precise type available at compile-time (which is unrealistic).

The offliner accesses keys generically through the aspects package. Extensions to key type or distinguishing scheme can be made by swapping out the aspects implementations.

Versioning

Because an object may be locally saved several times in the offlining process, we store the remote version while offlining and restore it on synchronizing. Further, we manually update the local versions to eliminate conflicts: If, during the offlining process, an object is offlined three times

- The first time, it will offline with version 0 (or whatever the server version was).
- The second time, if it is the same object reference we detect it by the fact it now has a local key. (The same then happens on pass three.) If it is a different reference, it will be inserted with version 0 causing the local version to rise to 1.

- The third time, if we did nothing, we would be committing an object of version 0 when the local database version is 1. This is a conflict.

To prevent this, the local save process is

1. Read the current local version.
2. Set the version on the object to commit to the current local version. This effectively disables versioning locally.
3. Save the object.

Object graph traversal

Where objects reference other objects, several issues arise:

- Hibernate's lazy fetching strategies mean we cannot access referenced objects once the session that got the root object is closed, unless they were already fetched by some other operation.
- When we save an object into the local database, its references become foreign keys. These must also be adapted to contain the local database's foreign keys for the objects in question.

While offlining or synchronizing an object - these two are known as operations - the process is as follows:

1. (For all objects)
    1. Iterate over all associated objects.
    2. Recursively perform the operation on the associated objects. This sets the primary keys of the associates correctly, which means hibernate will generate foreign keys correctly.
2. Perform the operation on the object in question.

This guarantees that whenever an object is saved, all its properties are set correctly. It is not, however, particularly efficient: saving n objects can generate O(n^2) save statements.

Synchronization and conflicts

While synchronizing, exceptions from the server are wrapped in Conflict objects and returned at the end of the operation. Mapping entries are marked as PROCESSED after the first attempt to save them so no object is synchronized more than once. (This works because synchronize() is called once for the entire contents of the local database. This technique cannot be used for offline(...) as it allows for multiple independent offline operations.) If a conflict is encountered synchronizing a child object, the parent is automatically marked as conflicted (dependent conflict).

Algorithm choice

The current synchronisation algorithm guarantees that all changed objects are synchronized by iterating over all objects and cascading on each object's graph. This has one efficiency drawback : If the objects have cascade&eq;SAVE set, hibernate will cascade too potentially resulting in O(n^2) database commits for n objects. This can be avoided if hibernate notices itself when it is trying to recommit an object it has just saved already - n^2 operations im memory are certainly cheaper than on the database.

The offliner design was based on the abstraction that only remoting dao-proxies are available server-side. If we could have a separate offlining process running on the server and use this instead, we could handle the server's session management ourselves and ensure that objects are neither sent multiple times from client to server during one synchronisation, nor - if we can do the whole synchronisation in one server session - written multiple times to the database (hibernate's session cache would then prevent this).

Another algorithm we discussed involved having the offliner store the list of operations on the local database (updates etc.) and then send this list to the server which executes it again during synchronisation. As I did not implement this, I cannot say for certain whether it would work - I see no reason why it should not in theory. However, instead of sending the end result of the user's edits in the local database, it would mean sending the whole history. For example, changing an object 10 times would mean sending it not once but 10 times to the server and saving it 10 times to the database there. (In the case of a single object, this could of course be optimized trivially - it is multiple modifications of object assocations I am worried about).

The concept of an object's identity plays a central role here. With one database, a primary key is a simple choice of identifier. With two, we need to manage two primary keys (local and remote) for each object and further take into account newly created objects that have no key yet. Transferring a list of object operations like "A.setParent(B)" to the server would mean matching the client-side and server-side identities of A and B up. I also noticed that a "unique object identifier" of type (Class X PK) is a central building block of object identity management.

Further, if a client makes 10 changes to an object locally then synchronizes, it seems logical to me that the server's version of this object rises by 1. Currently, this is the case. If we replay the edit history, it would rise by 10.

Offlining is an example of where an abstraction (hibernate, ORM) that makes one thing easier (persisting objects in a single database) creates a whole host of new problems when something that the abstraction was not designed for (offlining) is attempted. For example, the whole association management becomes more or less straight-forward when done in plain SQL. However, we want a solution that does not depend on low-level details of our database (and thus is open to incompatibilities between different database types, for instance) and so switching back to plain SQL is not an option.

What would be possible - and interesting - is a new kind of database-independent "object-oriented query language" which can be used for transmitting object changes to the server. I envisage something like "UPDATE Person:1000 parent=Person:2000" to mean "On the instance of Person with key 1000, call setParent using the Person with key 2000 as argument". This would mean no actual domain objects ever have to get sent to the server. (Person:1000 is again an "unique object identifier" as stated above.)

Offlining

This page describes the offliner's offline operation.

Offline copies a selection of objects to the local database. They must be database instances in the sense that

1. They are instances of entity classes for which a table exists in the local database.

2. Their class is declared in the offliner's list of classes to manage (when resynchronizing, the offliner iterates over this list).
3. They have an id set that makes them uniquely identifiable and recognizable as coming from the database. In particular, this id must not be null/zero. (To save new instances of entity classes into the local database, get a dao from the local database and use its save method instead.)
4. All FK associations of an object must be resolved at the time it is passed to offline. (We cannot save objects with and unresolved lazy FK references in their object graph to a database.)

If the objects passed to offline do not exist yet in the local database they are created. If they exist under the same version as the new one to offline, the objects are ignored. If the version in the local database is older than on the server, the object is updated as long as it has not been modified in the local database.

One consequence of this is that if you check out an already offlined object from the server, modify some data then try to offline it, it will not be offlined again because the version is unchanged. This would lose your changes. The correct procedure is first to save it back to the server, then reload it (with the new version) and then offline it. Alternatively, load the local databased version and modify that then save it back to the local database with a regular save. This procedure is required to ensure the object can be recommitted to the server afterwards. As a general rule, anything passed to offline should come directly from the server's database.

Any object added to the local database is on `synchronize` returned (updated) to the server with the same id/PK and version. Any other changes made to it are kept.

You can offline the same object many times in a row. The offliner ensures there will not be an exception here. *Internally, the offliner may offline something several times in a row during processing of an object graph.* The version in the local database should not rise in this case.

Offline only runs on the object graph rooted at the object(s) you pass it. It is primarily (when called by the user) meant to add new objects to the local database that are not there yet. To update all objects already in the local database, use synchronize. *Internally, synchronize calls offline on the objects changed on the server.*

If the local database is non-empty, you should synchronize before offlining any new objects. Offline can only fail if an object has been changed both in the local database and the database. Objects unchanged on the server since the last offline are ignored.

Offline does not provide conflict management - if you synchronize before offlining and the synchronize succeeds, there will not be any offlining conflicts.

There is one other rare case that causes an offline exception: Trying to update an object of an older version that you offlined it as earlier. This is a bug in the application calling the offliner, as database versions do not decrease over time.

Synchronization

Synchronize ensures that, unless there is a conflict,

- All changes made in the local database are written to the server.
- All offlined objects that have in the meantime been updated on the server are updated in the local database too.

If there is a conflict,

- All objects that have been updated both in the local database and on the server cause a Conflict.

During the whole of a synchronize operation, neither local nor database data should be externally modified.

Each object - identified uniquely by the pair (class, PK) - is processed exactly once by the synchronization process. (However, if an object's save results in a cascade, these extra saves are not under the control of the synchronization process.)

Phases

Synchronization runs in three phases. If a conflict occurs in a phase, subsequent phases are skipped. This allows us to assume previous phases were successful and depend on them in later phases.

The phases are

1. Synchronize changed data in the local database.
2. Synchronize deleted data in the local database.
3. Synchronize unchanged data in the local database.

Changed data

All objects that have been changed in the local database are passed to the server once for synchronization. They can either be successful or fail. This step runs as in offliner 1.1 except that the metadata may be differently implemented and organized. Further the metadata is updated by the server as, unlike 1.1, it is not deleted but written back after the synchronization.

From the database's point of view, it must act as if the objects had been loaded by the user, modified and saved back directly. In particular, an object's id and version must be the same when synchronize saves it back than when the user offlined it.

Deleted data

This is the simplest to process. All deleted objects are deleted on the server in the order they were deleted on the client using the metadata.

Unchanged data

The issue here is that it may have been changed on the server thus producing stale data on the client. This phase only runs when the previous two completed successfully so we can never get a client-side conflict in this phase.

All unchanged entries are queried by version from the server which checks if they have been updated and sends them for updating if they have.

Conflict resolution

There are several resolutions for confclicts. The former concern objects updated both on the client and the server since the last sync, the latter deletion conflicts.

For an object updated both on the client and the server, we recommend showing the user the two versions and deciding which one he wants to keep. Then, calling either forceLocal or forceRemote forces the given version to overwrite the other ignoreing versions. This requires some trickery to break hibernate's optimistic locking and must only be done during a period of time when you have an exclusive lock on the database. Note that force only forces the object passed as parameter. If it failed to sync because something it depends on failed, you must sort out the source of the conflict first before dealing with dependent conflicts.

Deletion fails in the following scenario: On the server you have objects A and B where a depends on B. You offline only B and delete it in the local database. This is ok because there is no A there. On sync, the offliner tries to delete B in the database and gets a constraint violation from A. You have two options: Delete A manually on the server and resync, or declare the deletion of B void. After you have deleted all you want to delete on the server, eraseDeletes declares all pending deletions void.

Mapping entries

Each domain object has offliner metadata associated with it. This is represented by the MappingEntry class and can be accessed as if it were a bean property of the domain class via the aspects package.

Offliner metadata is persistent in the local database. However, where an object's metadata would only contain default values or values derivable from the object itself it is allowed not to store any metadata and return a new instance of MappingEntry in the getter.

Mapping entries contain the following properties:

- Local and remote unique keys that identify the object.
- The base version under which the object was last offlined.
- A delete version. This is 0 unless the object is pending deletion in which case it indicates the order in which the objects must be deleted.

The local database status was a property of hte mapping entry in 1.1. It is now recompupted when needed based on the object an the mapping entry. Because the mapping entry for RMI efficiency reasons cannot contain a link to its object, offline status is a separate aspect. It takes the following values:

- NEW if an object has been created in the local database and never synchronized.
- OFFLINED if the object is offlined and has not been changed in the local database.
- CHANGED if the object is offlined and has been changed in the local database.
- DELETED if the object has been deleted from the local database but is pending deletion on the server.

The status can be queried by comparing the version and keys of the object with those of the mapping entry. In addition, during synchronization operations the following states can occur:

- PROCESSED if the entry has been processed successfully in the current sync operation.
- CONFLICTED if the entry has caused a conflict.

These must be set back to other values (PROCESSED becomes OFFLINED, CONFLICTED CHANGED or DELETED) when the sync operation ends even if it ends in failure.

Key Management and Requirements

This page contains the requirements and implementation of the offliner's key management.

Requirements

1. The offliner must be able to save a generic key in a field of the mapping table.
2. Keys must form three disjoint nonempty sets NULL, LOCAL and REMOTE.
3. New instances of objects must have a key from NULL.

4. Objects with NULL keys must be insertable into any database and get a generated non-NULL key, namely from REMOTE in the remote database and LOCAL in the local one. No key of these sets may ever be generated twice (except if the database is reset) even between different runs of the application. *Notes: Clearing the local database after a synchronize* ***is*** *a database reset in the local database. While NULL can and usually will contain one element, LOCAL and REMOTE must contain at least as many elements as there will ever be objects.*

5. equals() on domain objects fulfills the hibernate contract.

6. The offliner must be able to reset keys to a NULL value. equals() must then treat the object as new.

The hibernate contract for equals() is

1. For any object a, a.equals(null) is false. *This is part of the java contract.*

2. For any object a, a.equals(a) is true. In fact a `= b implies a.equals(b) =` true. *This is part of the java contract.*

3. For any object a that has not yet been saved to the database and got a key, a.equals(b) nust be false except if a == b.

4. For any objects a, b (of the same class) that both have been saved and thus have keys, a.equals(b) if and only if the keys are equal.

Default implementation (for derby/db2)

1. The key field is of type VARCHAR, keys of String, Integer and Long can be saved and read via an extra class. They are prefixed with S, I or L to distinguish.

2. Keys must be of type Long. 0L is NULL, >0 is REMOTE and <0 is LOCAL. *Negative keys are not possible in Oracle!*

3. New objects get their field of type long automatically initialized to 0L by java.

4. The NULL convention matches that of hibernate. The remote database uses a sequence (start with 1, increment by 1) and the local database starts with and increments by -1 respectively.

5. equals() in the example base class tests for key == 0L and acts acordingly.

6. setKey(0L) is a reset.

Test implementation for oracle

Here we use key ranges of 0-2000000000 for the REMOTE and 2000000000-4000000000 for the LOCAL database. The boundaries are not included. This class is currently available in the tests package only.

Custom implementations

Of the default implementation,

1. New types can be added by extending GenericSerializableUtil.

2. For a new type added in 1. or a String or Integer type, you need to provide a TypedAspect implementation that distinguishes key types.

3. The default value new instances of your domain objects get must be a NULL value.

4. It is your own responsibility to make your conventions hibernate- and database-compatible.

5. This is your own responsability.

6. Your TypedAspect must provide a nullKey implementation that resets a key to a NULL value.

The Graph Walker

The object graph walker algorithm traverses an object graph starting from a root object and following all *links*. The precise meaning of link is defined externally by the Linked implementation passed in the aspects object to the graph walker. Object graphs are directed; no other assumptions are made (loops, multi-edges and cycles are all allowed).

The graph walker requires two parameters: An ObjectWrapper that provides Linked and UniqueKeyed and a NodeVisitor that it can run on each node.

The following node states are used: All nodes that have not yet been seen are NEW. When a node is first discovered, it becomes PENDING. It is then passed to preVisit which can override its state; if an override happens it is treated as if it had been that state before (i.e. if it is overridden as PROCESSED, no further recursion will be done). Each node can be reached as NEW at most once (exactly once unless overridden).

A PENDING node becomes PROCESSED once all of its children have been successfully processed. Immediately before becoming PROCESSED, a node is passed to `visit`. PROCESSED nodes are not touched again.

A node can also become ERROR, indicating a problem. All nodes with links to ERROR nodes are guaranteed to end up as ERROR as well. Nodes become ERROR in three ways:

1. Visit throws a NodeException.
2. A linked/child node is ERROR.
3. preVisit overrides the node as ERROR.

ERROR nodes are not touched again by the graph walker.

The node state determination is shared between the walker and the visitor. The walker resets its state memory every time it is run on a new root; any stored state information between runs must be handled by the visitor. This is why the override mechanism exists. Only nodes that are NEW from the walker's point of view are passed to preVisit and are guaranteed to lose NEW status afterwards (PENDING if the visitor does nothing, ERROR or PROCESSED if it overrides).

The justification for this is that it is best suited to the purposes of the offliner. A more general graph walker would manage state information itself, allowing it to correctly handle a node linking to another seen in a previous run itself.

The contract between the walker and its visitor is as follows:

1. All nodes that are reachable from the current root will be passed to the visitor's `preVisit` exactly once, namely at the time they are first discovered.

The visitor can return NEW to indicate normal processing or override with ERROR or PROCESSED.

2. Nodes that the visitor does not override are passed exactly once to the visitor again after all their children have been processed; they are passed to

   1. `visit` if all children are PROCESSED.

   2. `markError` if any child is ERROR.

The walker's algorithm on a node is:

1. If the node is not NEW, ignore it.

2. If the node is NEW, pass it to preVisit. If it overrides, mark it with the new state and ignore it.

3. If preVisit keeps it NEW, set it to PENDING and recurse on all children.

4. If any children are ERROR, mark it as ERROR and call markError on the visitor.

5. If all children are PROCESSED, call visit. If it succeeds, make the node PROCESSED, otherwise ERROR.

Node identity is determined by UniqueKey objects. Two nodes are equal if and only if their unique keys are equal. If a different instance im memory is seen that is equal to a previously seen one, it is treated as the same node for state purposes (*states are in fact stored as a map of UniqueKey to NodeState*).

The Object Wrapper

The ObjectWrapper is now part of module-hibernate. It was developed for the offliner and formerly known as the "aspects package".

Motivation

The offliner works with the concept of a "generic domain object". These can be of any domain class and are known to have some properties like "primary key" or "version". In addition, they all have "properties" imposed by the offliner like a mapping entry (mapping entries and domain objects are in a 1:1 relationship) or a unique key.

This package came into being because the code for dealing with this became messy and introduced unwanted dependencies. For instance, the object graph walker needs information about an object's identity (i.e. its unique key). This can be got from hibernate. Earlier, this meant the object graph walker had a dependency on hibernate's session factory which is not good design.

The wrapper package separates the different interfaces (Keyed etc.) that the walker or offliner use from the implementation. In fact the mapping table has two different implementations on the client and on the server.

Other Approaches

This was written at the time of development when it was unclear which solution would be best. The current implementation using the wrapper package is the one that was chosen.

Having the graph walker load keys from the hibernate session factory is an unwanted and unnecesary dependency.

Requiring domain objects to implement the necessary interfaces (Keyed) themselves i.e. in an AbstractDomainObject class would be an elegant object-oriented solution. But it is implausible for two reasons:

1. We want to be able to plug the offliner into an existing application. The application should have the offliner as a dependency and not the other way round. Therefore we cannot mandate a base class for domain objects nor access such a class in both the application and the offliner.
2. Wrappers like UniqueKeyed or Mapped are offliner-specific, so it makes no sense to require application classes to implement them. Also it prevents us from choosing different implementations (on the client and server).

Having one utility class to deal with all wrappers might look like this:

```
public class DomainObjectUtils {
   public static Serializable getKey(Object object);
   public static void setKey(Object object, Serializable key);
   public static long getVersion(Object object);
   public static void setVersion(Object object, long version);
    // ...
}
```

You might be able to do this in C, in java this kind of code smells. It is one of the motivating examples in Object-Oriented Programming 101 that this kind of code can be done better.

Having one big wrapper class instead of several small ones. This may well be the final form of this package, but I find several small (fit on one page) classes with a clear purpose easier to use and maintain than one big class for five purposes. Also, if one of the apsects needs to be varied from client to server (mapping table) whereas the others stay the same, there is no way around using several classes.

This does not preclude merging some wrappers, for instance KeyedVersioned, UniqueKeyed and Typed could all become something like DomainObject - this would lose us the option of casting to Keyed objects for which we do not want to do offlining and so never need unique key or type information but admittedly such objects are never used by the offliner at the moment. All offliner-related wrappers could be united in one OfflineableWrapper too. I still think it is cleaner to separate unrelated wrappers but it won't harm anyone to merge them.

This decision proved valuable when the tests were adapted to run on oracle too. Only one wrapper had to be re-implemented.

Using Wrappers

We have a domain object obj given and want its key. What we would really like to do is `"key = obj.getKey();"`. Wrappers allow us to do this with only one extra line of code:

```
Keyed objKeyed = wrapper.wrap(Keyed.class, obj);
key = objKeyed.getKey();
```

objKeyed can be kept around for further key-related operations. A setKey() on it sets the key in the actual object obj, to which it holds a reference.

The main class of the wrappers package is called ObjectWrapper. To set it up, create a prototype of each wrapper implementaion (the reasoning is explained below), create a wrapper object and link them up:

```
ObjectWrapper wrapper = new ObjectWrapper();
UniqueKeyed uk = new UniqueKeyedImpl();
wrapper.add(UniqueKeyed.class, uk);
```

Or using spring:

```
<bean id="wrapper" class="ch.elca.el4j.util.objectwrapper.ObjectWrapper">
   <property name="wrappables">
      <map>
         <entry
key="ch.elca.el4j.services.persistence.hibernate.offlining.objectwrapper.Un
iqueKeyed"
            value-ref="uniqueKeyed" />
      </map>
   </property>
</bean>

<bean id="uniqueKeyed"
class="ch.elca.el4j.services.persistence.hibernate.offlining.objectwrapper.
impl.UniqueKeyedImpl" />
```

To use a wrapper on an object, call `wrap(wrappable, object)`. If the object already implements this interface, it is returned unchanged. If not, the wrappable is looked up in the wrapper object's map of registered apsects. If it is not found, an ObjectWrapperRTException is thrown. If it is found, the wrapper is set up for this object as described below. This too can throw an ObjectWrapperRTException if it is impossible to handle the wrapper for this object (like Keyed on something that is not a domain object).

To check if a wrapper is registered, `boolean wrappablePresent()` exists. The version taking an object also returns true if the object already implements the wrappable interface. None of these methods can guarantee that wrapper creation will not fail due to an exception from the wrappable implementation, though.

The ObjectWrapper package implementation

The wrappable interfaces all extend the `Wrappable` interface.

Implementations are named by adding an optional identifier (when several implementations exist) and then "Impl" to the name or the wrappable. For example, Typed becomes TypedImpl and KeyedVersioned has KeyedVersionedHibernateImpl and KeyedVersionedReflectionImpl.

ObjectWrapperRTException is thrown by any object wrapper package class when an illegal operation is attempted. This inlcudes

- Trying to use a wrapper for which there is no implementation (at creation time)
- Trying to use a wrapper on an object that it can not be used on (it is best to complain at creation time, but it is feasible that in some situations the problem can only be detected later when a wrapper method is actually called).

InternalObjectWrapperRTException is thrown if a "this can never happen" line is reached in ObjectWrapper or an implementation and indicates a serious bug in the object wrapper package.

The base class of all implementations is AbstractWrapper. This is required for now. Because interfaces or base classes cannot mandate constructors and I do not want to make this requirement implicit (via reflection), I use the prototype design pattern. Prototypes of the implementations are added to the ObjectWrapper class, when one is required it is cloned (Cloneable is declared in AbstractWrappable and all implementations must allow cloning) and these protetced properties are set: m_target is the object we are creating a wrapper for, m_wrapper is the object wrapper itself. (This allows one wrappable to depend on another.) Next, create() is called. This is the effective "constructor" and must prepare the wrapper or throw ObjectWrapperRTException if there is a problem like an object that cannot be used with this wrapper. If create() returns the wrapped object is returned to the user.

If a wrapper requires external dependencies like hibernate's session factory, they can be passed in the true constructor (for the prototype). This is independent of the ObjectWrapper class which only sees the finished prototype. The prototype pattern thus allows instantiation of the wrapper class itself to be decoupled from instantiation for a specific object. One nice side-effect of this pattern is that all required classes can easily be declared as spring beans. As the cloning is done by the ObjectWrapper class, no spring prototype qualifier is required.

Writing your own wrappers

Suppose you want to create a ValidatableWrapper that performs validation on objects.

1. Create an interface Validatable extends Wrappable.
2. Create an implementation ValidatableImpl extends AbstractWrapper implements Validatable.
3. In the constructor of your implementation, load any dependencies (to the validator, perhaps) that your wrapper will need. Remember that this is a prototype.
4. In create(), you can check if an object actually is validatable. When create() is called, the wrapper object creating the implementation (a clone of your prototype) will have m_wrapper set to itself and m_target to the object this clone should refer to. Throw an ObjectWrapperRTException if it is not a valid object. If you need to "import" other wrappers, check they are present with m_wrapper.wrappablePresent(wrappableClass) and throw an excpetion if not.
5. Implement the wrapper methods (validate, for example). They can use m_target and m_wrapper as they please. They can also throw ObjectWrapperRTException if necessary.

In your application,

1. Instantiate the implemantation once (prototype) with its dependencies.

2. Register it with your wrapper object.

Naming

I originally called this package aspects because that was what I was designing it as. However, aspects can be quite complicated both as a concept and to implement.

The package has since been renamed to "object wrapper".

There are no dynamic proxies, bytecode modification, cglib or anything else involved in this package. The original objects can be used further after they have been wrapped.

Version History and Future Plans for Cacher/Offliner

Before "0"

Before the name Cacher was used, the project started off as UniversalData / DataExpressionBrowser. It displayed simple tables of data. Its advantage was that it worked with the abstraction of a DataExpression which was very generic. This approach was found to be insufficient when dealing with database objects that carry associations between each other.

The next installation was a generic DatabaseBrowser. It allowed database entities to be accessed in a strongly typed way and references to be manipulated. It still exists (so does the DE-browser) and could be very useful in other scenarios than the one the cacher was made for. One of its modules, namely the DetailWidgets for displaying and editing typed data (and for editing database instances by displaying a list of DetailWidgets for all the object's properties) provided some of the ideas for the current version of a table data widget in the EL4J framework.

Caching was a submodule in each of these projects. The difference to the cacher ("0" upwards) is that the cacher is designed to be a separate pluggable module independent of the client/user interface/gui/business logic. However, an imputations demo was made that shows the cacher in operation and that is based on, though much less generic than, the database browser. This demo itself is split into a generic GUI/database part and a specific imputations part.

Cacher "0"

This first version of the cacher provided an independent module for all caching-related tasks and a first version of graph walking with a separated walker/visitor implementation and algorithm. It defined the cacher interface with synchronize and cache operations.

Cacher 0.1

This was completed on the 31st of July 2008 and features, compared to "0"

- An aspects package. This separates aspects the cacher or graph walker used like "keyed" or "link" (i.e. from a parent to a child object) from implementations (hibernate, reflection etc.).
- A new graph walker and node visitor package. The walker was completely rewritten to take advantage of the aspects package and use only generic access to its graph objects and to allow for some optimizations. The node visitor can override objects' states in one graph if it (or the cacher) has the necessary information from a previously handled graph reducing unnecessary database or remoting calls.
- Split between client and server. A caching server now runs on the server to allow chunking operations.
- Chunking. A strategy can be given for each class that allows chunks of objects to be sent to the server rather than a new remoting call for each object.
- Recovery from confclicts. All conflicted objects are returned with their key and version set correctly for the remote database allowing the user to attempt a manual recommit if he can deal with the conflict's source.
- More tests and example for the new graph walker.

Cacher 0.5

Cacher 0.5 allows multiple edit cache/edit server/sync operations to run in any order. To allow this, it saves the base version of an object both in the database and cache while it operates on it. One nice side-effect is that versions can be restored whenever an object moves from one database to the other and some potential unwanted overwrites are prevented during recaching.

Conflict resolution strategies are provided that allow you to choose the version of a conflicted object you want to use.

Instead of saving it in the mapping entry, the local database state is now recomputed when needed. This is necessary for multiple sync operations. Also, the last sync version of an object it saved in the mapping entry. This allows objects unchanged since the last sync to be skipped in the next one.

Offliner 0.6

- The aspects package has been renamed object wrapper and incoroporated into module-hibernate. The cahcer only provides extra wrappables and implementations.
- The generic to/from string handling is implemented cleanly and extensibly using strategy objects.
- The bug in the property dao is fixed; cacher properties (last commit) are written to database.
- Integer and long are both recognized by the default type distinguisher.
- Cacher renamed to offliner.

Offliner 1.0-SNAPSHOT

The offliner has since been integrated into EL4J internal as module-offliner; the first version number there was 1.0-SNAPSHOT.

Offliner 1.0.1-SNAPSHOT

A few minor changes: Since the EL4J base domain class was adapted to be offlining-capable, the offliner can use it directly. The strategy pattern for synchronization strategies was made cleaner and more extensible with a new interface; the chunk size for out-independent classes is now configurable via parameter. Offliner tests are finally automated and run through in a normal build thanks to a new test starter/stopper package. More detailed information is provided with conflict objects.

Project: Change tracking.

Change tracking can be used independently of the cacher to save unnecessary database/remoting operations when a set of data is checked out to memory, edited in place and recommitted. (This is caching too but the lack of a local database means that the whole key/version/cache management is unnecessary.)

Change tracking could be used to implement a wholly new kind of cacher which saves changes instead of objects and replays them to the server. Some solution to the problem of inefficiency incurred by having to send a complete version hsitory instead of just the latest changes will need to be found first.

Project: Identity fixing.

There is already an identity fixer but it does not work in the cacher scenario where object identities change. Also, it is not very efficient recursing on absolutely all fields like the static singletons of the map class internals. A new object identity fixer based on the generic concept of unique keys could be made as a graph walker node visitor.

-- DavidBernhard - 14 Jan 2009

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.

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

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.

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.

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.)

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.

Handling exceptions

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.

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.

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.

Related useful concepts and hints

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.)

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.

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/Assert.html][Assert] class. Please refer also the the AssertionUsageGuidelines for more details about assertion usage.

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.

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.

Exceptions and transactions

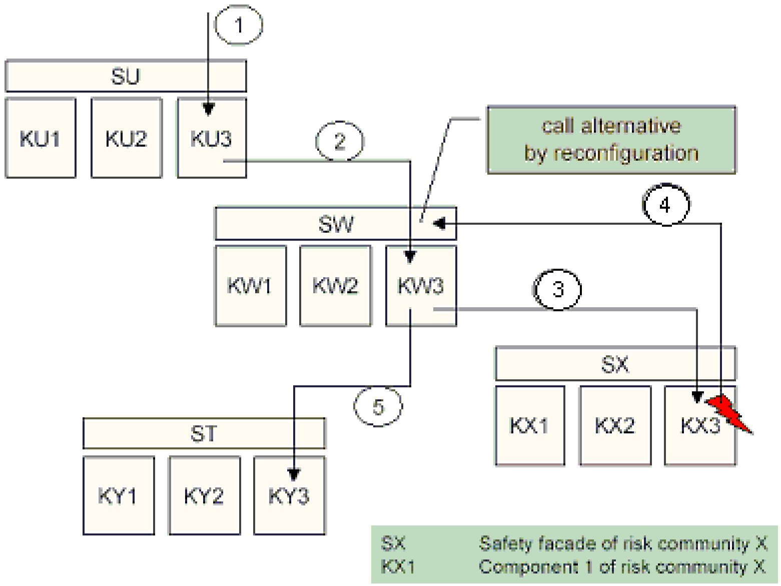
Transactions should often be rolled back after an exception occurs. Please refer to ModuleTransactionAttributes for a description on how to do this automatically.

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.

Emergency Handling and Safety Facades (2)


1
SU
KU1
KU2
KU3
2
call alternative by reconfiguration
4
SW
KW1
KW2
KW3
3
SX
KX1
KX2
KX3
5
ST
KY1
KY2
KY3
SX Safety facade of risk community X
KX1 Component 1 of risk community X

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

Correlation ID

A correlation id is a String value (usually a UUID) that is used to correlate (= link together) exceptions and log-entries that concern the same HTTP request or processing. The correlation ID is generated at the entry of a HTTP request and stored in a ThreadLocal? variable. When an exception is printed out or a log is written out, this correlation id is added to the message. On the level of log4j this is done via a special PatternLayout? string that includes `%X{correlationId}` (see below for a full example).

Running example: The `CorrelationIdManager` interface and an example implementation `CorrelationIdManagerSlf4jImpl` are contained in `module-core`. They are used by a `CorrelationIdFilter` in the internal `jsf-demo` which assures the creation of a fresh correlation id for each request sent to the server. There is also a `CorrelationIdImplicitContextPasser` in `module-core`, which is used in the sample configuration `scenarios/remoting/common/protocols-config.xml` of the `swing-demo-common` project. The ImplicitContextPasser allows to trace a request over process-borders.

```
2010-11-19 09:07:51,731 ERROR [org.jboss.seam.exception.Exceptions]
[[230516046]] handled and logged exception
javax.servlet.ServletException: #{entityManager.saveOrUpdate(employee)}:
org.springframework.dao.DataIntegrityViolationException: not-null property
```

The correlation id `230516046` is then also displayed on a error page in JSF.

Internally we use the MDC mechanism of slf4j and log4j configuration like the following:

```
 <appender name="business-file-appender"
class="org.apache.log4j.DailyRollingFileAppender">
        <param name="File" value="logs/business.log" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d{ISO8601} %-5p [%t]
[%c:%L]: %X{correlationId} %m%n" />
        </layout>
    </appender>
```

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.

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
- The trouble with checked exceptions, discussion with Bruce Eckel and Anders Hejlsberg: do we need to reconsider our choices? TBD http://www.artima.com/intv/handcuffs.html

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.

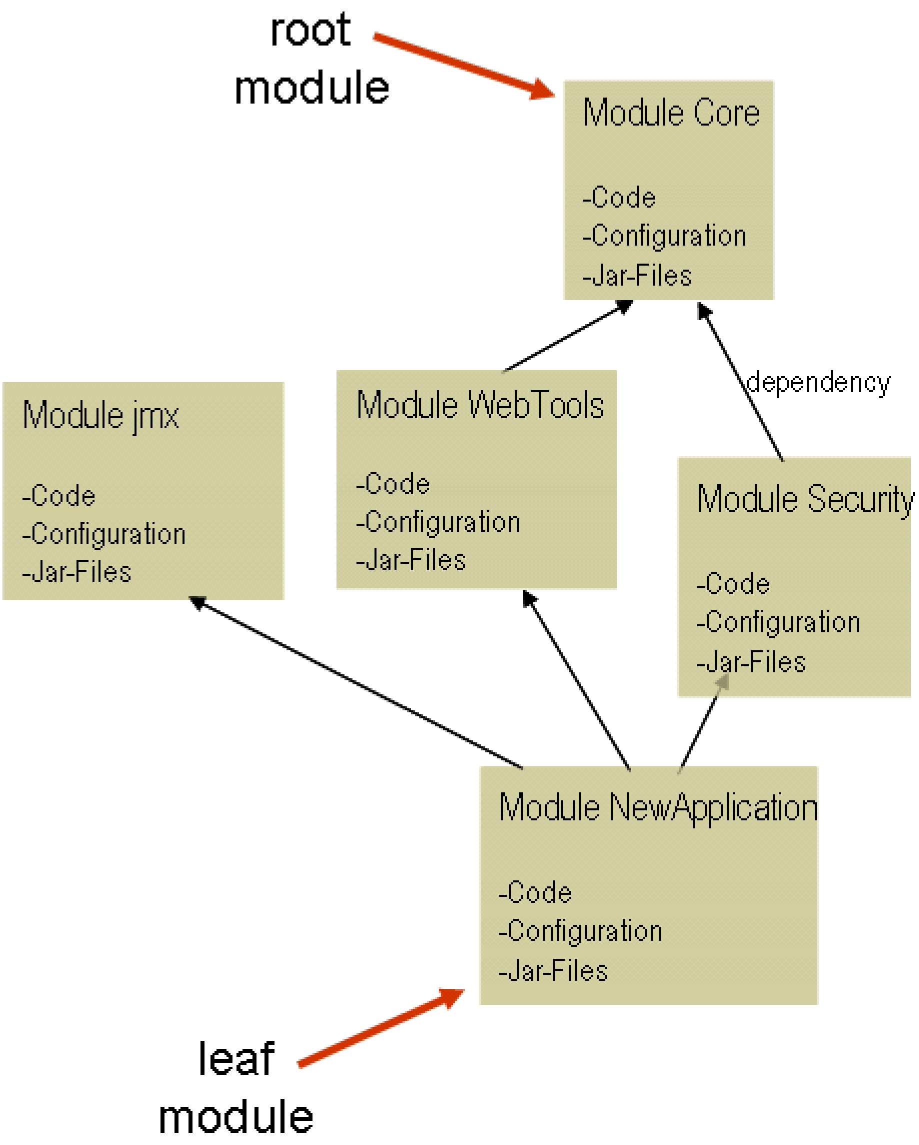
Database plugin

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.

Overview

This plugin applies all SQL scripts of your project and all projects it depends on. The scripts are applied in *correct* order (root module -> leaf module for creation, leaf module -> root module for destruction).

Here is a sample module set up that helps to illustrate a typical usage:



Create SQL scripts of the modules would be execute in the order module core, jmx, webTools, security, newApplication (the order of the modules jmx, webTools, security may be interchanged, as they are on the same level). SQL scripts inside a module get sorted alphabetically (since EL4J version 1.6.1). Destroy scripts are executed in the inverse order.

The idea that each project should be able to define its own SQL scripts to init and clear the database is part of the module abstraction of EL4J.

The goals of this plugin can be either run via the console or during automated builds (during tests). During most goal, the plugin executes sql scripts whose names have particular prefixes: the create `goal` executes files that start with `create`, the `update` goal executes files that have an `update` prefix.

Goals

Goal start

This goal can run in two modes:

1. Start the configured database server (the `db.name` maven property) through the console command `mvn db:start`. It only launches Derby (when other databases 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 if 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.

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.

Goal update

This goal works like the `create` goal, but executes statements from sql files starting with `update`.

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.

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.

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 when you actually want to get exceptions.

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 ends.

Goal block

Convenience goal to block until one hits Ctrl-C. Is unconditional (blocks also with a db.wait flag set to false). This goal is typically useful on the command line.

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>
        <goals>
```

```
          <goal>prepare</goal>
        </goals>
        <phase>pre-integration-test</phase>
      </execution>
    </executions>
  </plugin>
```

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

Goal destroy

The `destroy` goal launches the sql files with a destroy prefix.

Goal run

The `run` goal launches the sql files starting with a specified prefix. Example: `mvn db:run -DfilePrefix="archive"` executes all sql files starting with "archive".

There are two arguments:

- `filePrefix`: the file prefix
- `reverse`: should sql scripts be executed in reverse order. `false`: bottom up (in dependency tree), like `create` goal; `true`: top down, like `drop` goal

Two examples:

- `mvn db:run -DfilePrefix="create" -Dreverse="false"` is equal to `mvn db:create`
- `mvn db:run -DfilePrefix="drop" -Dreverse="true"` is equal to `mvn db:drop`

Properties

- `db.wait`: Specifies whether 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;":retriev
eMessagesFromServerOnGetMessage=true;
dataSource.username=refdb_user
dataSource.password=**********
```

If you don't provide a value for this parameter, a `.properties` file is selected automatically. If there is a file `env-placeholder.properties` (see module-env) in the root of the classpath and it contains the needed database properties, they will be taken. Otherwise the following naming pattern will be applied: `artifactId-override-{db.name}.properties` in directory `scenarios/db/raw/`, where *artifactId* is replaced by current ArtifactId. The base directory may be changed using the 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 dependency tree are checked (from the leaf module towards the root module), until a `.properties` file is found.

- `db.connectionPropertiesSourceTemplate`: Template for file names for `.properties` files used to read the db connection settings. 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, so it should "just work")). Sample content:

```
dataSource.driverClassName=org.apache.derby.jdbc.ClientDriver
dataSource.validationQuery=VALUES CURRENT TIMESTAMP
```

- `db.sqlSourceDir`: SQL source directories, i.e. the 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`: Separator for string lists (default value: ",").
- `delimiter`: Separator for sql statements. (default value: ";").

Example usage

Console

`cd newApplication; mvn db:prepare db:block`

Launch and re-init db (of current project), block until Ctrl-C:

`mvn db:silentDrop db:create`

The same without db launch

`mvn db:start`

Start the Derby Network Server in blocking mode.

Please refer also to the MavenCheatSheet for more examples.

Integrating plugin into build lifecycle

Add the following snippets to the section in the part of the `pom.xml` file of your project (this is already done for you in the application templates):

- Add the db initialization in the `pre-integration-test` phase. Actually it is shorter to just use the `prepare` goal (see the example under the `prepare` goal documentation above):

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

- In the `post-integration-test` phase we just launch the `cleanUp` goal (refer to `cleanUp` goal documentation).

Remarks:

- 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 (see the `separator` property).
- All the resources are taken from the classpath, which includes the project as well as all dependencies.
- `prepare` and `cleanUp` were called `prepareDB` and `cleanUpDB` before.
- 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.

Dependencies to external jars

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

References

- Plugin documentation: http://el4j.sourceforge.net/plugins/maven-database-plugin/plugin-info.html
- Another SQL plugin http://mojo.codehaus.org/sql-maven-plugin/

DepGraph plugin

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.

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.

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.

The depgraph-plugin has been revised. It now shows exactly the same order of the dependencies as they are added to the classpath by maven, e.g. if two dependencies point to the same artifact but with different version, the newer one wins (is put to the classpath). If the versions are the same, the one that is nearer to the root artifact wins. If you want detect duplicated artifacts, set the `depgraph.drawOmitted` property to true. This draws omitted artifacts in dotted boxes.

Goal depgraph

Creates the graph to the local project.

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.
- `depgraph.drawScope`: (default=true) Defines whether the edges should be labeled with dependency-scope (from the perspective of the root artifact).
- `depgraph.drawOmitted`: (default=false) Defines whether omitted artifacts should be drawn in dotted boxes.

Goal fullgraph

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

Properties

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

Links

- Graphviz Homepage

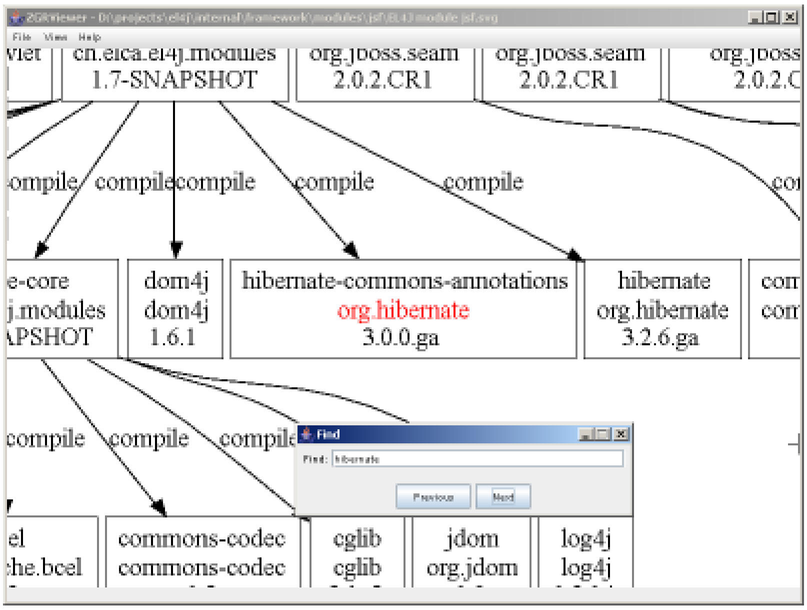
- The Java tutorials: Regex
- Introduction to the Dependency Mechanism

Advanced usage

As soon as the graph gets bigger it gets more and more difficult to find artifacts. For this use case the following can help:

```
mvn depgraph:fullgraph -Ddepgraph.ext=svg
```

The graph can be viewed and searched using http://zvtm.sourceforge.net/zgrviewer.html. Download and unzip it and put the installation path in `run.bat` otherwise it will not work properly. Open the generated svg file with `File` -> `Open` -> `Open SVG generated by GraphViz...` and use `Ctrl-F` to search for a string.



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

Examples

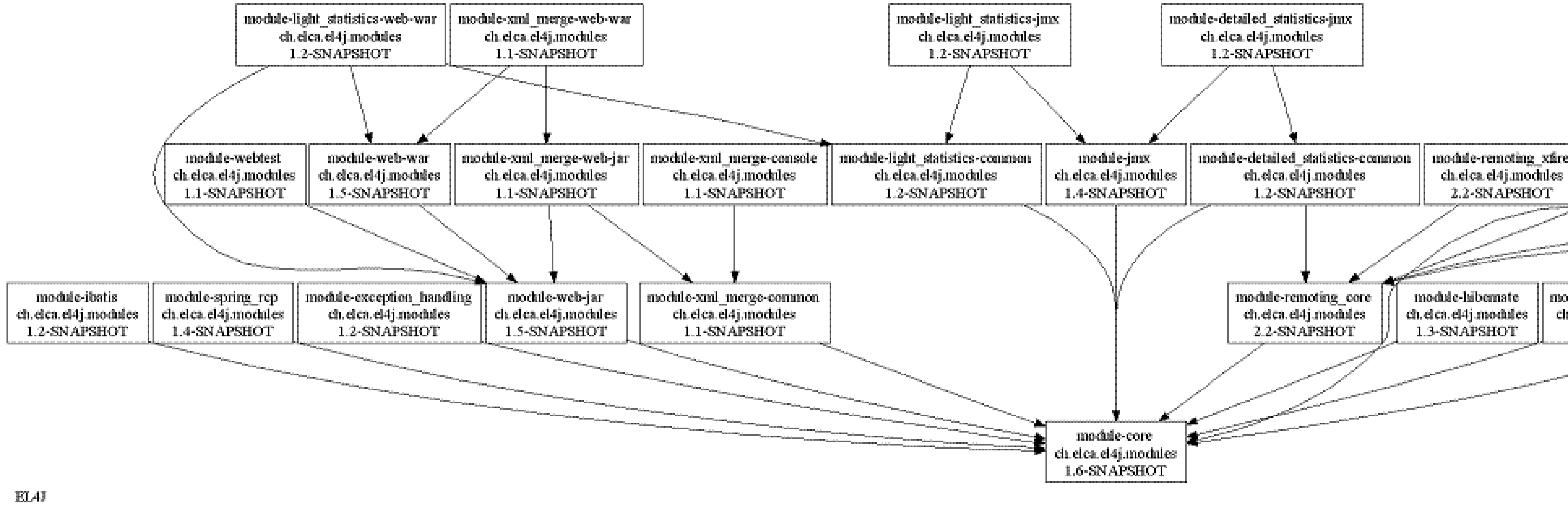
Command line

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

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

Sample Output graphs

Small output graph


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-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_xfire
ch.elca.el4j.modules
2.2-SNAPSHOT
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-remoting_core
ch.elca.el4j.modules
2.2-SNAPSHOT
module-hibernate
ch.elca.el4j.modules
1.3-SNAPSHOT
module-core
ch.elca.el4j.modules
1.6-SNAPSHOT
EL4J

All modules, different view

Medium graph

All modules, test modules and samples of EL4J:

Big graph

Please refer also to the MavenCheatSheet for more examples

-- ClaudeHumard - 14 Jan 2008

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.

Goals

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.

Properties

- `-Dversion.listall={true|false}` List all artifacts, including those that are up-to-date (*optional, defaults to* `false`)

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.

Properties

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

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.

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`)

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

Example Output

`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
```

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
```

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
```

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).

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 ?

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).

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?

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.

MavenRec plugin

Introduction

The MavenRecPlugin is used to execute one or more maven commands (e.g. `clean compile`) for a specific project and recursively for all local dependencies of this project. A dependency is local when it can be found in the same directory structure as the current project.

This is particularly useful if you work on the EL4J framework and you have checked out the whole "external" directory containing a lot of submodules. Now, if you make changes in 2 dependent modules (e.g. A depending on B), you need to rebuild the module the other depends on first (i.e. B) before you compile the second module (i.e. A). Normally, you would do this by changing to the respective directories and execute a `mvn clean compile` in each of them. Another (even worse) alternative would be to execute `mvn clean compile` in the external-folder and getting the whole EL4J framework compiled (including even unchanged modules), which is a slow task.

With the MavenRecPlugin, you just type the maven command(s) in the folder of the current project replacing `mvn` by `mvnrec`. The MavenRecPlugin (1) finds the root folder of the current project folder and from there, it (2) scans all sub-directories in the directory-tree which contain a `pom.xml` (i.e. "src" or "target" folders are ignored). The MavenRecPlugin then (3) executes the maven command(s) only for those projects which are a dependency of the current project.

Requirements

To ease the use of MavenRecPlugin, there are two scripts in the maven/bin folder: `mvnrec` for linux/cygwin and `mvnrec.bat` for the windows shell.

Usage

Just type `mvnrec` (cygwin/linux) or `mvnrec.bat` (windows-console) to get help on the usage.

The syntax is:

```
mvnrec [OPTIONS] MAVEN_COMMAND [MAVEN_COMMAND]...
```

Example:

```
mvnrec -b -ff clean install
```

In the windows shell, you have to type `mvnrec.bat` instead of `mvnrec`. The rest of the syntax and all options are exactly the same.

Options

The following options are available:

- `-b` force scanning of folders and creation of bootstrap-file (see next section)
- `-ff` fail-fast: Stop at first failure
- `-fae` fail-at-end: Only fail the build afterwards; continue execution in all non-impacted projects (default)
- `-v` verbose: produce mvnrec debug output

Bootstraping

As described above, in the first phase the MavenRecPlugin searches the folder of the root-project of the current project (e.g. the `external` folder in the el4j-framework). There, it looks out for a `mvn_rec_bootstrap.xml` file located in the target folder. If this file exists, MavenRecPlugin skips the directory scanning phase and uses the information stored in this file to locate the dependencies. Otherwise, MavenRecPlugin scans the directories as described above and stores the collected information about the visited projects (e.g. groupId, artifactId, pom-location etc.) in the mentioned `mvn_rec_bootstrap.xml` file. As the project dependencies don't change very often, this mechanism helps to reduce the execution time of MavenRecPlugin as the directory scanning phase may take while. On the other hand, if there are changes in the directory structure or if you add or remove a dependency in your project, you have to enforce MavenRecPlugin to rescan the directories and to overwrite the old `mvn_rec_bootstrap.xml`. This can be done by adding the `-b` parameter to the `mvnrec` command, e.g. `mvnrec -b clean compile`.

Example

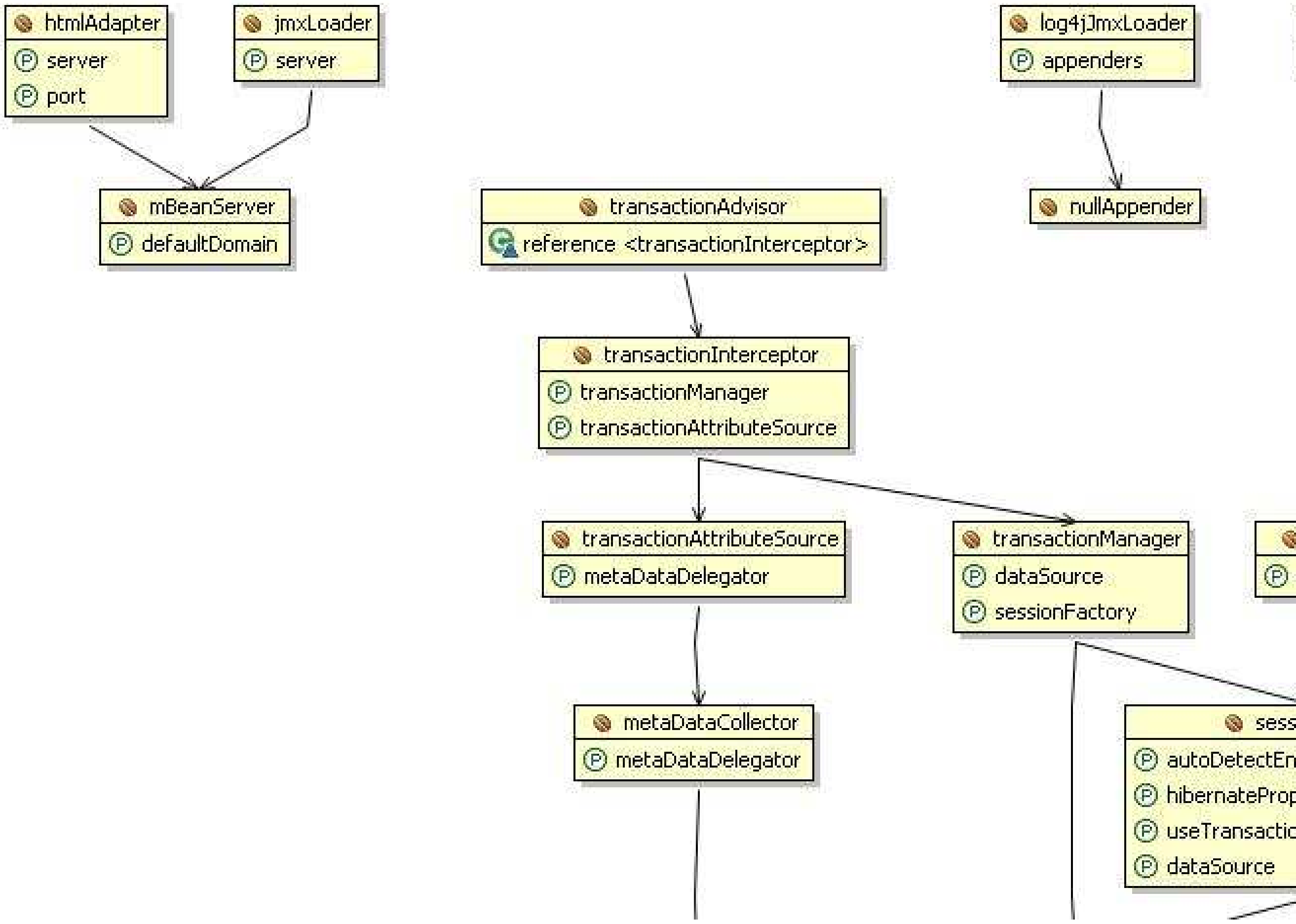
Let's assume you have the following directory and project structure:

- modules/
    - pom.xml
    - core/
        - pom.xml
    - hibernate/
        - pom.xml
    - remoting_core/
        - pom.xml
    - remoting_jaxws/
        - pom.xml

If you are currently working on the module "Remoting Jaxws" and made also some changes in the module "Core" it requires. Now, you can change to the directory `/modules/remoting_jaxws` and type in there `mvnrec clean install`. MavenRecPlugin scans the directories, finds all other modules and executes the command only in the dependent projects "Module Core" and "Module Remoting Core" before it is executed in module "Remoting Jaxws". The project "Hibernate" is ignored because it's not a dependency of module "Remoting Jaxws". MavenRecPlugin considers only projects with the packaging-types `jar` and `war`.

Spring IDE and maven-spring-ide-plugin

Spring IDE is an eclipse plugin that allows you to view graphs of your spring beans and their dependencies.



Important Note: This plugin needs at least version 2.2.1 of the SpringIDE? to work correctly.

Viewing a set of beans files in Spring IDE (general method)

- Right-click your project and select "Spring/Add spring project nature". Open the project properties and select Spring/Beans Support.
- On the config files tab, you can import a set of xml files containing beans. Import all your project's bean files here. On the config sets tab, make a new config set and add all the files you imported to this config set.
- Select Window/Show View/Other, then Spring/Spring Explorer. In the spring explorer, you see all your beans files and your config set.
- To view the beans of one file, right-click it and choose open graph. To view all beans in one graph, right-click the config set you created above and open graph there.
- You can right-click an element in the graph to open either the bean definition or the corresponding .java file.

What the maven beans plugin does:

- The plugin searches your source files or web.xml file for inclusive and exclusive bean locations as you would present them to a ModuleApplicationContext.
- Based on the found locations it creates the .springBeans file needed by SpringIDE
- It forces Spring Nature for the project in eclipse (by modification of the .project file)

What you get

- After execution the project has an entry in the Spring Explorer containing all defined beans.

Usage

For normal apps

To use it, add the following to your pom.xml:

```
<plugin>
  <groupId>ch.elca.el4j.maven.plugins</groupId>
  <artifactId>maven-spring-ide-plugin</artifactId>
   <executions>
    <execution>
         <phase>verify</phase>
            <goals>
               <goal>spring-ide</goal>
            </goals>
       </execution>
    </executions>
 </plugin>
```

If you don't specify a sourceFile, the plugin will go through all java files in the source directory and search for one that contains the following comment:

```
// $$ BEANS INCLUDE $$
```

In the source file, you must use the following schema:

```
// $$ BEANS INCLUDE $$

String[] included = {
  "classpath*:mandatory/*.xml",
  "classpath:demo/demo*.xml"
};
// $$ BEANS EXCLUDE $$

String[] excluded = {
  "classpath*:exclude-*.xml"
};
```

For web apps:

To use it, add the following to your pom.xml:

```
<plugin>
  <groupId>ch.elca.el4j.maven.plugins</groupId>
  <artifactId>maven-spring-ide-plugin</artifactId>
   <executions>
    <execution>
         <phase>verify</phase>
            <goals>
               <goal>spring-ide</goal>
            </goals>
       </execution>
    </executions>
 </plugin>
```

The plugin will automatically take the web.xml file in the WEB-INF directory. Add folling comment right before defining the module application context in web.xml

```
<!-- $$ BEANS INCLUDE $$ -->
```

Example:

```
<param-value>
<!-- $$ BEANS INCLUDE $$ -->
    classpath*:mandatory/*.xml,
    classpath*:mandatory/refdb/*.xml,
    classpath*:scenarios/db/raw/*.xml
</param-value>
```

Important notes

- The $$ comments must appear in the correct order and be typed exactly as shown.
- Any " " - quoted strings in between are assumed to be bean file entries (for .java files).

- Only one string at most per line.
- The name of the variables is not important, excluded is optional. The included/excluded `String[]` arrays are just the format that a ModuleApplicationContext can use.
- The included lines end as soon as a `}` (java) or a closing tag (xml) is read.
- If exclusive lines are present, they are preceded with `// $$ BEANS EXCLUDE $$`, the rest is the same for them.
- Commented lines (using // in front or /*...*/ around one or multiple lines) are not read

Cobertura runtime plugin

Introduction

Cobertura (http://cobertura.sourceforge.net/) is used to create code coverage reports. The normal Cobertura Maven plugin is used for encapsulated tests, so just unit test are used to check the code coverage. The idea of this plugin is to build and deploy the application nearly as normal with Maven and let Cobertura measure the coverage while working on the application.

What does the plugin do?

The application instruments the class files and collects the java source files. Instrumenting means modifying the ".class" files plus creating a special file (cobertura data file) where all lines of the (source) code are summarized. This is used as base for the report. Cobertura needs to know where its cobertura data file is stored. The solution is to share the configuration of Maven with the application, so we have to put the following properties-file cobertura.properties in the "env" directory of a module of the application (see http://el4j.sourceforge.net/plugins/maven-env-support-plugin/index.html).

While the application is running, there is a JMX server running too. Just connect to it you can generate a report via the MBean "coberturaRuntimeController". You can also pause the coverage-reporting.

Configuration properties, pom properties

- `cobertura-runtime.dataDirectory`
    - The directory where all needed files will be created/copied. Default: `${el4j.project.home}/cobertura-runtime` (property `el4j.project.home` is set in your `settings.xml`)
- `cobertura-runtime.dataFilename`
    - The name of the Cobertura data file. The file is placed in the data directory. Default: `el4j-cobertura.ser`
- `cobertura-runtime.sourceCollectorDirectoryName`
    - The directory name of the directory where the source code of instrumented classes will be copied. The directory is placed in the data directory. Default: `java-sources`
- `cobertura-runtime.keepReports`
    - Flag to mark if a newly generated report should be placed in a new directory (name with timestamp) or always the same directory should be used. Default: `false`
- `cobertura-runtime.jmxRmiRegistryPort`
    - The port where the RMI registry used for the JMX service should be installed. Default: `8199`
- `cobertura-runtime.jmxServiceHost`
    - The host name where the application, actually the JMX service is running. Default: `localhost`

All these properties and more are in `cobertura.properties`, which means that the values are fixed after building the application. But you can still change these values via setting the properties as system properties with the same property name. Take attention: Some values are repeated in other values (see "cobertura-runtime" properties on the bottom of EL4J's ROOT pom)!

How to use

Be sure that you have Maven of EL4J >=1.6 installed!

- Place the cobertura properties file in one of your modules
    - Copy cobertura.properties to directory "src/main/env" i.e. of your service module, or where you already have env properties files.
- Go to your project's home directory and build your application by activating profile "cobertura-runtime".
    - `cd YOUR_PROJECT_HOME`
    - `mvn -P+cobertura-runtime clean install`
      or `mvn -DskipTests=true -P-integrationTests,+cobertura-runtime clean install` to skip tests.
        - This instruments your code and collects the source files.
        - Via plugin configuration parameters "fileListIncludes" and "fileListExcludes" you can drive the list of files being instrumented.
        - Via plugin configuration parameters "sourceFileListIncludes" and "sourceFileListExcludes" you can drive the list of source files being collected.
        - Via plugin configuration parameter "includeTestFiles" you can include test files too.
        - Via plugin configuration parameter "ignoredMethods" you can ignore methods.
        - Here a configuration example (BTW these are the default values, except "ignoredMethods", it is by default empty):

```
<build>
    <pluginManagement>
        <plugins>
            <plugin>
```

```
                <groupId>ch.elca.el4j.maven.plugins</groupId>
                <artifactId>maven-cobertura-runtime-plugin</artifactId>
                <configuration>
                    <fileListIncludes>**\/*.class</fileListIncludes>
                    <fileListExcludes>**\/*Test.class</fileListExcludes>

<sourceFileListIncludes>**\/*.java</sourceFileListIncludes>

<sourceFileListExcludes>**\/*Test.java</sourceFileListExcludes>

<ignoredMethods>ch.elca.el4j.apps.jsf.security.Authenticator#login*,ch.elca
.el4j*#create</ignoredMethods>
                    <includeTestFiles>false</includeTestFiles>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
```

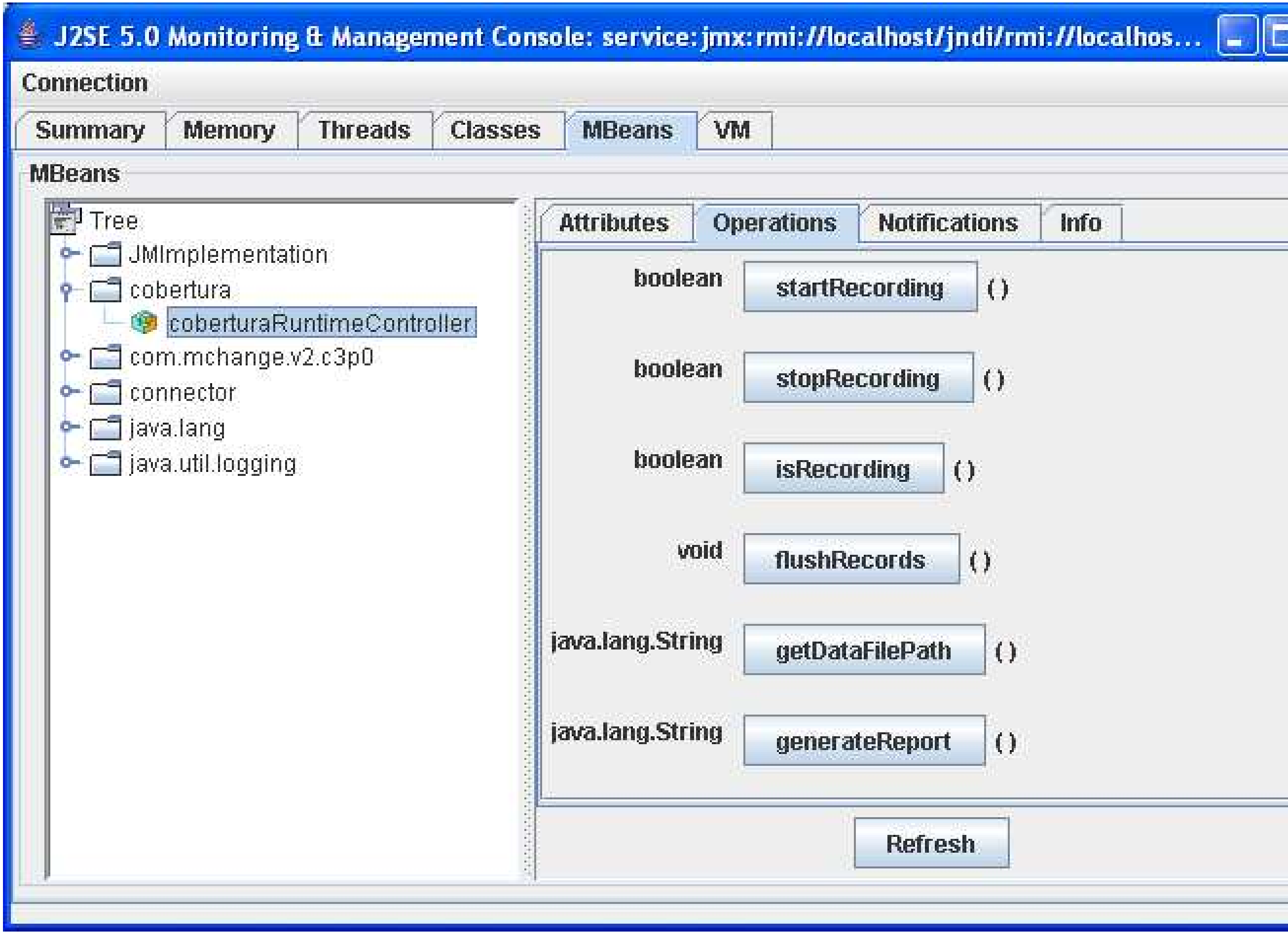
- **You can set all these properties also as Maven properties.** Here the example with the values above:

```
<properties>
    <cobertura-runtime.fileListIncludes>**\/*.class</cobertura-
runtime.fileListIncludes>
    <cobertura-runtime.fileListExcludes>**\/*Test.class</cobertura-
runtime.fileListExcludes>
    <cobertura-runtime.sourceFileListIncludes>**\/*.java</cobertura-
runtime.sourceFileListIncludes>
    <cobertura-runtime.sourceFileListExcludes>**\/*Test.java</cobertura-
runtime.sourceFileListExcludes>
    <cobertura-
runtime.ignoredMethods>ch.elca.el4j.apps.jsf.security.Authenticator#login*,
ch.elca.el4j*#create</cobertura-runtime.ignoredMethods>
    <cobertura-runtime.includeTestFiles>false</cobertura-
runtime.includeTestFiles>
```

```
</properties>
```

- Start your application
  - If you start your app via maven ("Simple" Java, Jetty, Cargo (Tomcat)) do not forget to always add the profile "cobertura-runtime" like below:
    - `mvn -P+cobertura-runtime exec:java`
    - `mvn -P+cobertura-runtime jetty:run`
    - `mvn -P+cobertura-runtime cargo:start`
    - Else you will get a `ClassNotFoundException` i.e. for `net.sourceforge.cobertura.coveragedata.HasBeenInstrumented`
- By having config location `classpath*:mandatory/*.xml` as usual in the `ModuleAppliactionContext`, the JMX service is now running beside the application by default at url `service:jmx:rmi://localhost/jndi/rmi://localhost:8199/coberturaRuntimeConnector`. Open the JConsole via the "jconsole" goal:
  - Change to the directory where you launched your application.
  - `mvn -P+cobertura-runtime cobertura-runtime:jconsole`

- Cobertura runtime controller @ JConsole:



  - **flushRecords**: Writes the cobertura-coverage-data to the configured file.
  - **getDataFilePath**: Returns the path to the cobertura-coverage-data file.
  - **generateReport**: Executes a flush and generates the cobertura coverage report. The returned path is the directory where the report has been written to.
  - **isRecording**: Returns `true` (default case) if cobertura is currently collecting data.
  - **stopRecording**: Stops cobertura coverage recording.
  - **startRecording**: Starts cobertura coverage recording again.

- If the application has been stopped you can still create the coverage report with `mvn -P+cobertura-runtime cobertura-runtime:report`
- To clean all coverage data execute `mvn -P+cobertura-runtime cobertura-runtime:clean`
  Now you have to start the instrumentation and source file collection of you files again.
- To permanently activate profile `cobertura-runtime` add it in your `settings.xml`:

```
<activeProfiles>
    <!-- DO NOT DELETE THE FOLLOWING LINE! -->
    <activeProfile>el4j.general</activeProfile>

    <!-- Web container -->
    <!--activeProfile>tomcat6x</activeProfile-->
    <!--activeProfile>weblogic10x</activeProfile-->

    <!-- Database -->
    <!--activeProfile>db2</activeProfile-->
    <!--activeProfile>oracle</activeProfile-->

    <!-- Cobertura-runtime -->
    <activeProfile>cobertura-runtime</activeProfile>
</activeProfiles>
```

Links

- Cobertura Runtime integration presentation

Acknowledgments

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

- Tobias Ammon
- Christoph Bäni
- David Bernhard
- Frank Bitzer
- Raphael Boog
- Reynald Borer
- Simon Börlin
- Laurent Bovet
- Andi Bur
- Alexander Deiss
- Do Phuong Hoang (PHD)
- Reto Fankhauser
- Christian Gasser
- Adrian Häfeli
- Jacques-Olivier Haenni
- Jonas Hauenstein
- Dominique Hügli
- Claude Humard
- Philippe Jacot
- Florian Keusch
- Vincent Larchet

- Marc Lehmann
- Yves Martin
- Alex Mathey
- Martin Meier
- Adrian Moos
- Philipp Oesch
- Philipp H. Oser
- Markus Pahs
- Andreas Pfenninger
- Stefan Pleisch
- Jean-François Poilpret
- Pham Quoc Ky (QKP)
- Fabian Reichlin
- Andreas Rüedlinger
- Nicola Schiper
- Marc Schmid
- Christoph Schwitter
- David Stefan
- Florian Süss
- Daniel Thomas
- Michael Vorburger
- Rachid Waraich
- Sandra Weber

- Stefan Wismer
- Martin Zeltner
- Dominik Zindel
- Martin Zingg

Thank you!

References

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

Record of changes

Filename	Version	Date	Description / Author
ReferenceDocumentation	1.7	15.12.09	Initial Version of Documentation for EL4J 1.7

References

Abbreviations