

# Cibet Control Framework

## Reference Guide

version 2.0



# 1. Content

1.	Content .....	2
2.	Introduction .....	4
3.	Concepts of Control Theory .....	4
4.	Cibet Concepts .....	7
4.1	Control Events.....	7
4.2	Sensors .....	10
4.3	Actuators .....	11
4.4	Setpoints and Controller.....	12
4.5	Status of a business case or resource.....	13
5.	Installation .....	14
6.	Using Cibet.....	15
6.1	Scopes and Cibet Context .....	15
6.2	User and Tenant .....	18
6.3	EntityManager.....	21
6.4	Using JPA sensor in a JDBC application.....	23
7.	Integrating a sensor .....	24
7.1	EJB sensor.....	24
7.2	EJB-CLIENT sensor .....	25
7.3	JPA sensor.....	26
7.4	JPAQUERY sensor .....	27
7.5	ASPECT sensor.....	28
7.6	HTTP-FILTER sensor.....	31
7.7	HTTP-PROXY sensor.....	32
7.8	JDBC sensor.....	34
8.	Configuration .....	35
8.1	Definition of setpoints.....	35
8.2	Tenant Control .....	37
8.3	Event Control .....	39
8.4	Target Control .....	39
8.5	State Change Control .....	40
8.6	Method Control .....	41
8.7	Invoker Control .....	42
8.8	Condition Control .....	43
8.9	Implementing Custom Controls .....	45
8.10	Controls in the context of the sensor .....	47
8.11	Actuator configuration .....	48
8.12	Runtime Configuration changes .....	49
9.	Actuators .....	50
9.1	INFOLOG actuator .....	50
9.2	TRACKER actuator .....	51
9.3	FOUR_EYES actuator .....	51
9.4	SIX_EYES actuator .....	56
9.5	TWO_MAN_RULE actuator .....	57
9.6	PARALLEL_DC actuator.....	60
9.7	SCHEDULER actuator .....	63
9.8	ARCHIVE actuator .....	68
9.9	ENVERS actuator .....	70
9.10	SPRING_SECURITY actuator .....	71

9.11	SHIRO actuator .....	75
9.12	LOCKER actuator .....	78
9.13	LOADCONTROL actuator .....	79
9.14	Implementing Own Actuators .....	93
10.	Pre- and Post- Control Functionality .....	94
10.1	Post- Checking Control Results .....	94
10.2	Pre- Checking Control Results .....	95
10.3	Releasing and Rejecting Dual Control Events .....	96
10.4	Releasing and Rejecting Scheduled Business Events .....	99
10.5	Comparing objects.....	100
10.6	Checking Archive Integrity .....	101
10.7	Searching, Redo and Restore of archived Business Cases .....	101
10.8	Security.....	103
10.9	Assignment and Annotation of Dual Control Events.....	104
10.10	Notifications .....	105
10.11	Locking Business Cases .....	109
11.	Appendix .....	110
11.1	Schema Definition .....	110
11.2	Guideline for migration of serialized objects .....	110

## 2. Introduction

Trust is good, control is better. While it may not be a good idea to apply this proverb against your partner in love there are areas in life where you should. For example the modification of highly critical business data by an application user is subject to human failure. Typos, wrong customer number by mistake, lack of information, there are numerous possibilities why users unintentionally make mistakes. In such cases **control is better**. The Cibet framework helps to control the execution of important business processes. To control an application in the sense of Cibet is not meant on a security level but on the reliance and trustfulness level.

Cibet is not only the acronym for **control is better**, cibet (from the Arab word zabad) or civet is also a distinctive secretion from the anal glands of the civet cat. In the natural state this is a substance with a penetrating foul-smelling odor. In a highly diluted state however, it develops a pleasant, musk-like fragrant flavour which is used in perfume manufacturing.

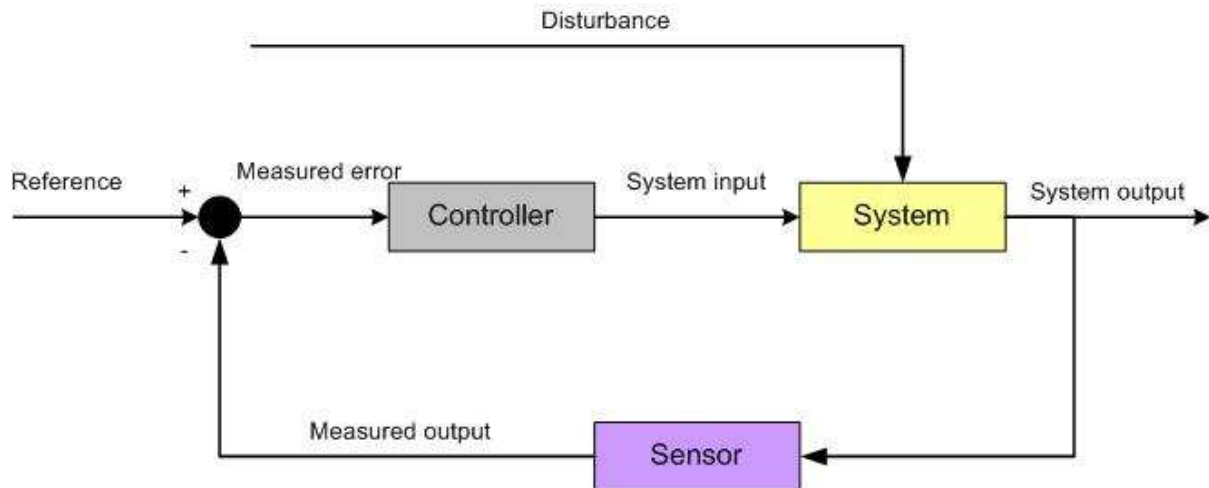
This describes exactly the position of control and the Cibet framework in an application. If you apply too much control it stinks: The application developer gets frustrated when he has to spend his time in applying control and security mechanisms instead of implementing business processes. The application user of such an application will have a feeling of "Big brother is watching you" and will get nerved and frustrated when he has to take care of complicated control mechanisms while executing business processes. Control in the right proportion and with an appropriate tool like Cibet however doesn't distract the developer from business process implementation and lets the application user feel safe, comfortable and well-guarded in his operations because he is sure that his actions can do no harm and his data could not get lost or compromised.

## 3. Concepts of Control Theory

Control theory is a discipline originated in mathematics and engineering science but was adopted also in psychology, social sciences and other research domains. It deals with influencing the behavior and operational conditions of dynamical systems. The fundamentals of control theory can also be applied to software architectures that have a need for controlling dynamical behavior. Control theory deals with topics that are also of importance in software development:

- **Stability** of a dynamical system is described by Lyapunov stability criteria (business rules must be obeyed and functionality must be granted)
- **Observability** is the possibility of observing the state of a system through output measurements
- **Controllability** is the possibility of forcing the system into a particular state by using an appropriate control signal
- **Robustness** of a control system is given if a controllers properties do not change much if applied to a system slightly different from the one used for its synthesis

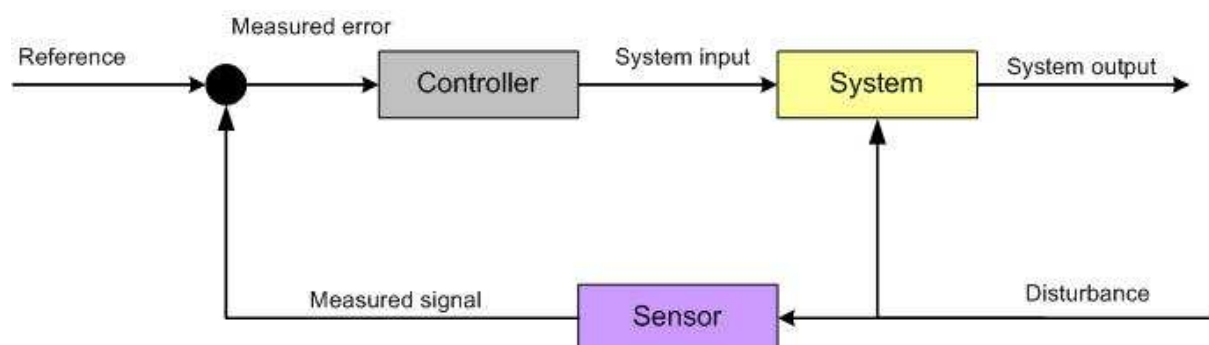
In control theory a lot of models have been developed but they all can be put down to two basic designs, the closed loop and the open loop controller. In the following figure a common closed loop controller design is shown:



**Figure 1: Closed loop control system**

An example for a closed loop control system is the heating system of a house for controlling temperature. A sensor monitors the output of the system which is in this case the air temperature. The controller compares the measured values with a reference or setpoint and calculates from the reference rules if the heater must be switched on or a hot water valve must be opened. The heater or hot water valve can be regarded as an actuator. In control theory an actuator is a mechanical device for controlling the system. It is operated by a source of energy, usually in the form of an electric current or hydraulic fluid pressure, and converts that into some kind of motion. The heater increases the temperature which is then an input variable of the system.

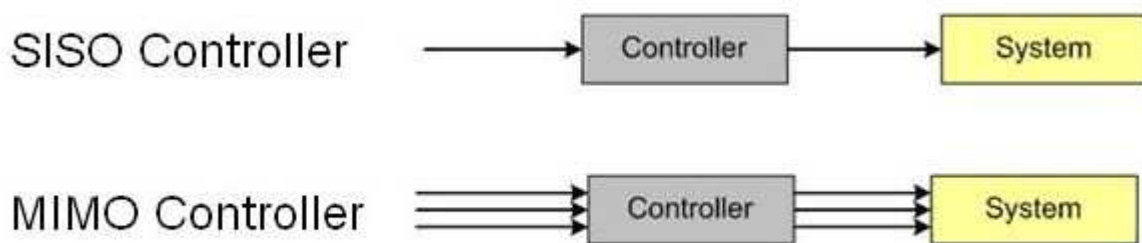
This is an example of a feedback controller: The variable measured by the sensor is the same as the controller is trying to control. The controlled variable is "fed back" into the controller. Feedback control usually results in intermediate periods where the controlled variable is not at the desired setpoint. This slowness of feedback can be minimized by using an appropriate open loop controller:



**Figure 2: Open loop control system**

In this design a disturbance is measured by the sensor before it has an effect on the system. If in the above example it is known that opening a window will decrease the air temperature a sensor could measure the opening of the window and switch on the heater before the air temperature actually has gone down. This is an example of a feed-forward controller: The controller acts before a disturbance affects the system. The difficulty with feed-forward controllers is that all possible disturbances and their influence on the system must be known, accounted for and observed by an appropriate sensor. If in the example the opening of the door is not observed the feed-forward controller will let the house cool down.

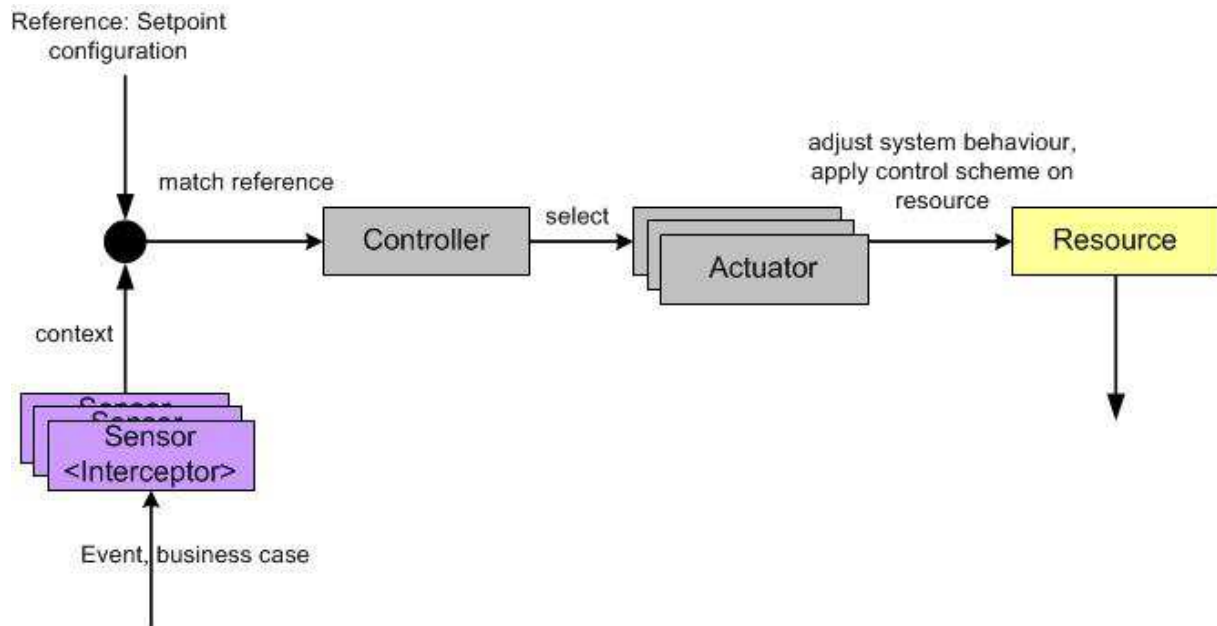
Another characteristic of controllers is how many variables of the system they control:



**Figure 3: SISO and MIMO Controller**

Single-Input-Single-Output controllers (SISO) measure one system variable like air temperature in the above example of the house heating system. Multi-Input-Multi-Output controllers control more than one system variables. In real control systems often a synthesis of open and closed loop, feedback and feed-forward, SISO and MIMO controllers are used in sequential, hierarchical, nested or networked combination.

The Cibet framework is a control system for controlling business processes on resources which can be described in the following figure:



**Figure 4: Cibet control system**

In fact it is an open loop controller. An event is observed by a sensor which is appropriate to the controlled resource. The controller checks this signal, compares it with the configuration and actual context and decides which actions to take and which actuator to apply. The actions of the actuators are then an input to the resource.

## 4. Cibet Concepts

### 4.1 Control Events

A control event is a context specific action that is performed on a Resource like a persistence entity or an object's method. This could be a persistence event on a domain object or a method call on an object. These actions are monitored in its specific context by Cibet. That means for example, that a database update can be done in a dual control context or a restoring context or in a simple update context. The following event types are the basic ones:

- **INSERT**

A newly created domain object which is not already in the database is persisted. This event corresponds to an SQL INSERT into the database. With JPA this is done with the `EntityManager.persist()` method.

- **UPDATE**

An already persistent object is updated in the database. This event corresponds to an SQL UPDATE in the database. With JPA this is done either implicitly when the object

is already in the persistent context and the transaction is committed or with the `EntityManager.merge()` method if the object is not in the persistent context.

- **DELETE**

A persistent object is removed from the database. This event corresponds to an SQL DELETE in the database. With JPA this is done with the `EntityManager.remove()` method.

- **SELECT**

A persistent entity is selected from the database. This event corresponds to an SQL SELECT statement. With JPA this is effectuated with the `EntityManager.find()` methods.

- **INVOKE**

This event is the execution of a service or business process. For example, the business process to update some entity in the database may also include sending an information message to an external system. You don't want the message to be sent if the update is set under dual control, but only when it is released. An INVOKE event could be a method call in a service class or an http request on a URL

- **RELEASE, RELEASE\_INSERT, RELEASE\_UPDATE, RELEASE\_DELETE, RELEASE\_SELECT, RELEASE\_INVOKE**

when a dual control mechanism is applied on a business case it is not executed instantaneously but is postponed. A second user must release or reject the business case. These events are the release actions on insert, update, delete, select or invoke business methods.

- **FIRST\_RELEASE , FIRST\_RELEASE\_INSERT, FIRST\_RELEASE\_UPDATE, FIRST\_RELEASE\_DELETE, FIRST\_RELEASE\_SELECT, FIRST\_RELEASE\_INVOKE**

when a six-eyes dual control mechanism is applied on a business case, a third person must release the business case. In this case the second user does not issue a release event but a first release event. These events are the first release actions on insert, update, delete, select or invoke business cases.

- **REJECT , REJECT\_INSERT, REJECT\_UPDATE, REJECT\_DELETE, REJECT\_SELECT, REJECT\_INVOKE**

when a dual control mechanism is applied on a business case it is not executed instantaneously but is postponed. A second user must release or reject the business case. These events are the reject actions on insert, update, delete, select or invoke business cases.

- **PASSBACK, PASSBACK\_INSERT, PASSBACK\_UPDATE, PASSBACK\_DELETE, PASSBACK\_SELECT, PASSBACK\_INVOKE**



when a dual control mechanism is applied on a business case it is not executed instantaneously but is postponed. A second user can pass back the business case to the user who initiated it. The business case remains in the responsibility of the initiating user.

- **SUBMIT, SUBMIT\_INSERT, SUBMIT\_UPDATE, SUBMIT\_DELETE, SUBMIT\_SELECT, SUBMIT\_INVOKE**

when a dual control mechanism is applied on a business case it is not executed instantaneously but is postponed. A second user can pass back the business case to the user who initiated it. The business case remains in the responsibility of the initiating user. He can modify the business case or make corrections and can submit it again to the dual control process.

- **REDO**

An archived service call invocation can be redone as all necessary metadata are stored with the archive. The redo event is the re-execution of a service call or the re-sending of an http request from an archive with exactly the same parameters.

- **RESTORE**

The state of an entity can be restored from an archived persistence business case. Restoring is either an update persistence action if the entity still lives in the database or an insert if the entity has been deleted.

These are the events that can be used for configuration. The events are organized in a hierarchical order. This means, if a configuration is defined for an event in the upper hierarchy, it applies also for all sub-events. The event hierarchy is shown in Figure 5.

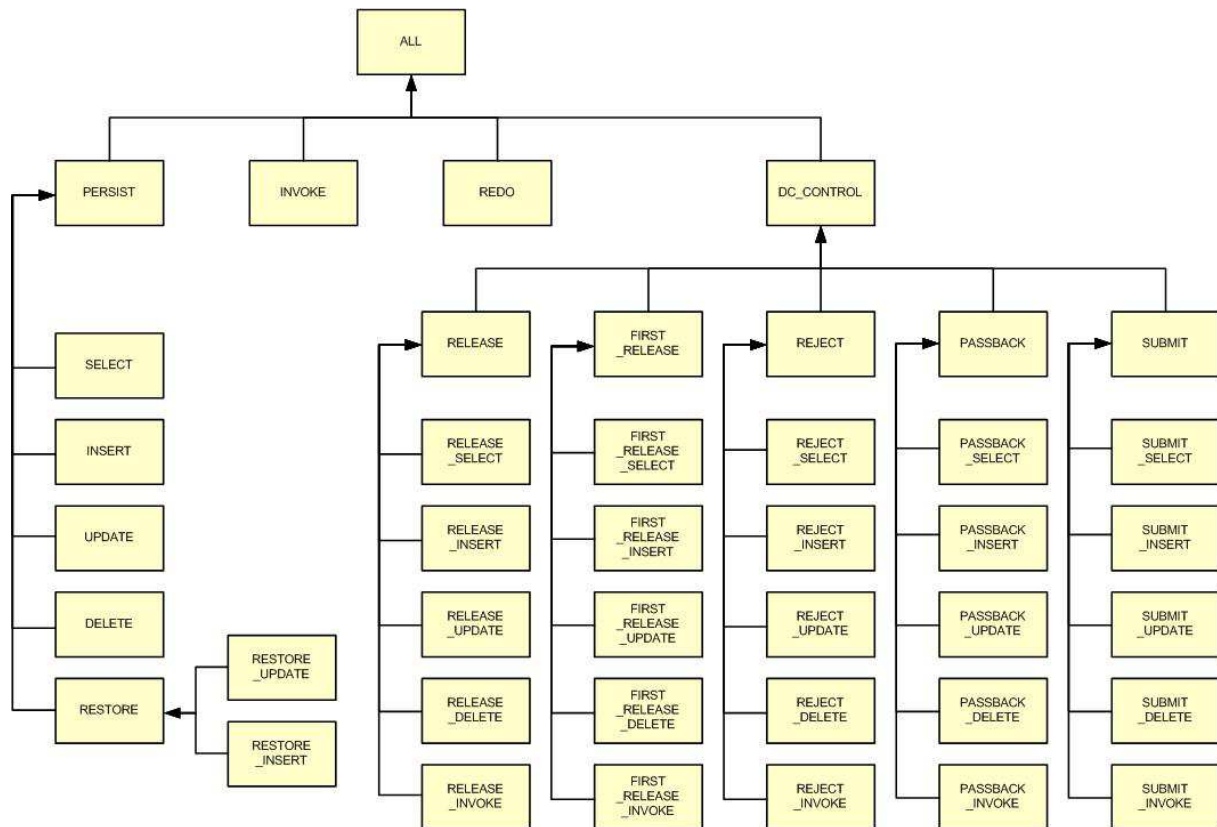


Figure 5: Hierarchy of control events

## 4.2 Sensors

The sensors are the part of the Cibet framework that must be integrated into an application in order to detect and control resources and business events and allow Cibet to evaluate setpoints and apply actuators. The following sensors are included in Cibet:

- **EJB-CLIENT, EJB**

These sensors detect control for EJB method invocations on the client and server side

- **JPA**

This sensor detects control for JPA database persistence actions like select, insert, update and delete of domain entities.

- **JPAQUERY**

With this sensor JPA queries can be controlled. Any type of queries can be controlled, let it be named queries, on-the-fly JPA queries or native queries. The control event is irrelevant as it is deduced from the query implicitly.

- **ASPECT**

This sensor detects control for invocations of public methods on any POJO. For dual control actuators and the REDO of a POJO method invocation the object on which to invoke the method must be recreated. Cibet supports the following object instantiation mechanisms:

- Default constructor
- Constructor with String parameter
- Static Singleton method without parameter
- Static Singleton method with String parameter
- Static method in a factory class
- Factory method in a factory class with default constructor
- Factory method in a singleton factory class
- Instantiation as Spring bean

- **HTTP-PROXY, HTTP-FILTER**

These sensors allow controlling of http requests on the client and server side

- **JDBC**

This sensor detects control for plain JDBC requests. Insert, update and delete in database tables can be controlled with the help of this sensor.

## **4.3 Actuators**

Actuators are one of the central concepts around which Cibet is built. An actuator implements the functionality that Cibet applies in order to control a resource. There exist a whole bunch of built-in actuators and it is also possible to create own actuators. Generally, the actuators provide cross-cutting functionality which can be applied on all resources but there exist also actuators which make sense only on special resources. For example, actuators which offer dual control or monitoring and archiving functionality can be applied on all resources while the ENVERS actuator which controls auditing of JPA entities with Hibernate Envers is applicable only with resources controlled by JPA sensor.

The built-in actuators are listed and described in chapter ‘Actuators’ and the instructions for creating custom actuators are given in chapter ‘Implementing Own Actuators’.

Actuators can be combined. For example in most cases it makes sense to combine a FOUR\_EYES or a SIX\_EYES dual control actuator with the ARCHIVE actuator in order to follow up the life cycle of objects. On the other hand it makes no sense to combine FOUR\_EYES and SIX\_EYES actuators. 4 + 6 is not 10 in this case!

## 4.4 Setpoints and Controller

Setpoints bring together the actuators with the control events and the control conditions. The Cibat controller decides by comparing the actual control event context with registered setpoints. If the control conditions of a setpoint match the context of the business case the actuators of this setpoint are executed. The actual control event context could comprise all data and metadata of the business case, for example:

- event type
- tenant
- logged in user
- affected domain object
- state of the object
- affected service call
- service call parameters
- service call context
- Cibat context properties
- time
- environment conditions
- URL
- http headers
- http querystring and form parameters

A setpoint defines which of these parameters should be evaluated with what parameters in order to decide if the actuators of this setpoint should be executed. Setpoints can be defined in `cibat-config.xml` configuration file or in program code with the Configuration API. Here is an example of xml configuration:

```
<setpoint id="Account Control 1">
  <controls>
    <event>UPDATE</event>
    <target>com.app.accounting.Account</target>
  </controls>
  <actuator name="FOUR_EYES" />
  <actuator name="ARCHIVE" />
</setpoint>
```

This setpoint defines that updating of Account objects shall be set under four eyes principle and the state changes be archived. A setpoint definition can be regarded as an IF THEN statement: IF the controls match the actual runtime context THEN the given actuators are applied.

The same setpoint as above could also be defined in program code:

```
Setpoint sp = new Setpoint("Account Control 1");
sp.setEvent(ControlEvent.UPDATE.name());
sp.setTarget(Account.class.getName());
sp.addActuator(cm.getActuator(FourEyesActuator.DEFAULTNAME));
sp.addActuator(cm.getActuator(ArchiveActuator.DEFAULTNAME));
Configuration.instance().registerSetpoint(sp);
```

The Cibet Controller executes evaluation of setpoints and makes decisions by applying instances of the Control interface. The Control instances are the counterpart of the setpoint definitions.

#### **4.5 Status of a business case or resource**

When a business case or an event on a resource is controlled by Cibet it will in an execution status which is determined by the applied setpoints and actuators. This status can be seen in the outcome of various actuators, for example in the stored Archive, a stored Controllable of a postponed business case in dual control or in the tracker output. The possible status values and transitions are shown in Figure 6. Not all possible transitions are shown for clarity, ERROR and DENIED are for example reachable from any other non-final status.

A business case starts always in status EXECUTING and ends in one of the final status EXECUTED, DENIED, REJECTED or ERROR.

- EXECUTING: the business case is currently executing and has not finished yet.
- EXECUTED: the business case has been executed
- POSTPONED: the business case has not been executed but postponed due to a dual control actuator. It must be released before it is executed.
- SCHEDULED: the business case is scheduled for later execution
- FIRST\_POSTPONED: the business case has not been executed but postponed due to a SIX\_EYES dual control actuator
- FIRST\_RELEASED: a SIX\_EYES controlled postponed business case has been released by a first user but is still postponed and awaiting release by a second user.
- REJECTED: a postponed business case has not been executed due to a user rejection
- PASSEDBACK: a postponed business case has not been executed but passed back to the originator of the business case
- DENIED: the business case has not been executed. User is not authorized.
- ERROR: the business case has not been executed cause of an error

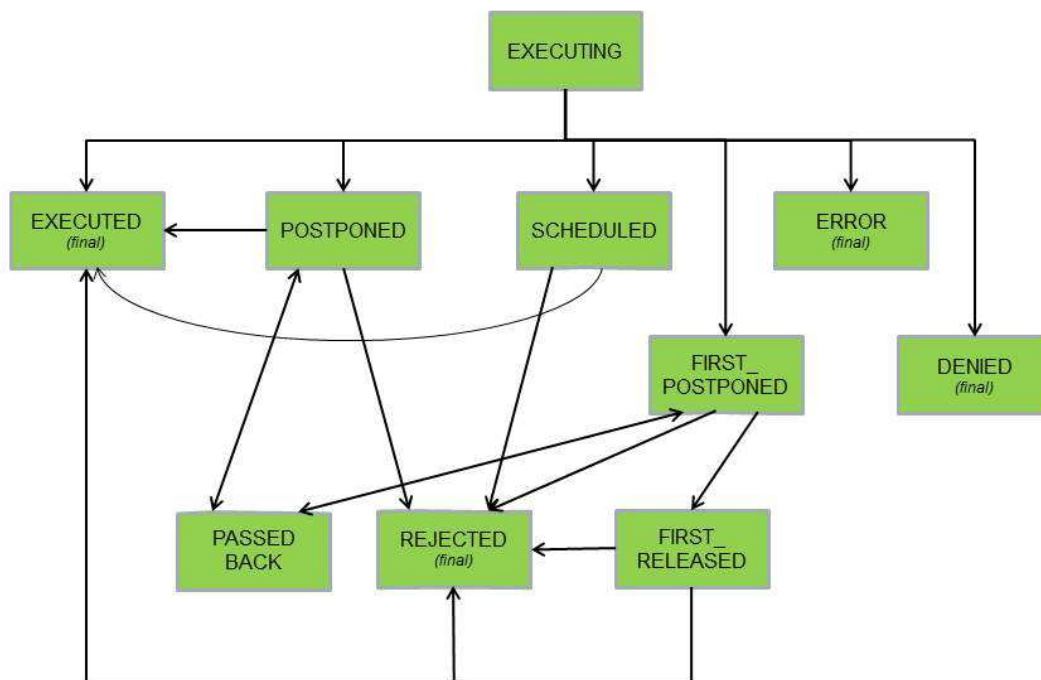


Figure 6: execution status of a controlled business case (simplified)

## 5. Installation

Cibet consists of several sub projects and it depends on the functionality which library must be included in your application. Cibet has a component- based architecture and it depends on which sensors and actuators you want to apply in your application.

All binaries can be downloaded from Maven Central repository at <http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22com.logitags%22%20AND%200a%3A%22cibet%22>. The easiest way to make use of Cibet is to add a Maven dependency into your pom.xml. For the basic functionality just add

```

<dependency>
  <groupId>com.logitags</groupId>
  <artifactId>cibet-core</artifactId>
  <version> the current version </version>
</dependency>

```

to the pom.xml of your project. The SQL scripts to set up the database tables are included in the cibet archive in folder sql.

The other Cibet libraries are cibet-envers, cibet-jpa, cibet-shiro and cibet-springsecurity. All these libraries have cibet-core as transitive dependency so that it is not necessary to add it if one of the other libraries is included. These four libraries extend and implement classes and interfaces of hibernate-envers, jpa, apache shiro and spring security and depend therefore on the versions of these libraries. Please see <http://www.logitags.com/cibet/download.html>.

The following chapters about sensors and actuators list which Cibet library must be included in which case.

## 6. Using Cibet

One of the greatest advantages of the Cibet framework is that it is non-intrusive. Existing applications can be enhanced by Cibet control functionality with no change of existing code. No change of domain objects, manager classes or database schemes is necessary. All control and actuator functionality can be added by configuration. All other Cibet code is optional and only necessary if the additional functionalities of Cibet should be applied.

### 6.1 Scopes and Cibet Context

Cibet retrieves and stores properties and data that are used internally in a context. Developers can use the Cibet context to make application specific data accessible to the Cibet framework. One example for a context property is the logged in user that must be made known to Cibet in order to apply controllers and actuators.

There exist three scopes of the Cibet context:

- Request scope: Data in request scope are only available within one http request. It can be accessed with `com.logitags.cibet.context.Context.requestScope()`
- Session scope: Data in session scope are available within the same http session. Cibet session scope is linked to the http session. It can be accessed with `com.logitags.cibet.context.Context.sessionScope()`
- Application scope: Data in application scope are available in all threads and sessions. It can be accessed with `com.logitags.cibet.context.Context.applicationScope()`

All three contexts have generic methods to set, get and remove any user-defined property. For scope-specific additional properties with explicit getter and setter methods see the API Javadoc of RequestScope, SessionScope and ApplicationScope.

The Cibet session and request scope contexts are thread-safe. Every logged in user has an own context. On each thread, the contexts must be initialized and at the end of a thread the contexts must be closed. This is important as memory leaks could occur if the contexts are not properly cleaned up.

The management of contexts is done by Cibet dependent on how a thread is started. Several explicit and implicit possibilities exist to start and close the contexts. It is not harmful to start the Context multiple times. If it is already started, starting it again is ignored.

## HTTP requests

In a web application each http request runs in a different thread. Therefore, the Cibet session context must be reset on each request from the http session. The linking of the Cibet session context to the http session is done by the `com.logitags.cibet.context.CibetContextFilter`. This http filter is responsible for setting the Cibet context on each request.

In order to make use of the `CibetContextFilter`, add the following to the `web.xml`:

```
<filter>
  <filter-name>cibetContextFilter</filter-name>
  <filter-class>
    com.logitags.cibet.context.CibetContextFilter
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>cibetContextFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Often it is not desirable to run all requests through `CibetContextFilter`. Static resources should be excluded for example. This is possible with the `excludes` parameter which takes a comma separated list of exclude regular expression patterns:

```
<filter>
  <filter-name>cibetContextFilter</filter-name>
  <filter-class>
    com.logitags.cibet.context.CibetContextFilter
  </filter-class>
  <init-param>
    <param-name>excludes</param-name>
    <param-value>.*<u>img</u>/.*, .*<u>css</u>/.*</param-value>
  </init-param>
</filter>
```

Please note that you don't have to configure `CibetContextFilter` if you use `HTTP-FILTER` sensor with the `CibetFilter` (see chapter `HTTP-FILTER` sensor). The `CibetFilter` inherits from `CibetContextFilter` and contains already its functionality.

## EJB invocations

When remote EJBs or Message-Driven Beans are invoked the above algorithm cannot be applied because there is no login and no HTTP session. In order to initialize and close the contexts in these cases the EJB/MDB must be decorated with an interceptor like this:

```
@Stateless
@Remote
@Interceptors(CibetContextInterceptor.class)
public class RemoteEJBImpl implements RemoteEJB {
```

The non-intrusive way to configure this interceptor is in the EJB descriptor XML file.



## ***POJO object invocations***

Non-EJB objects can be invoked for example as a message listener on a JMS queue or a batch process that is started by a timer. Objects of this kind cannot be decorated by `CibetContextInterceptor` but must be annotated by `@CibetContext`:

```
@CibetContext
public class SomeService {

    @CibetContext
    public void doSomeService() {
        ...
    }
}
```

The `@CibetContext` can be applied on class level or on method level. In the first case, a context is initialised and closed when any of the public methods of this class are invoked, in the second case only when the annotated method is invoked.

`@CibetContext` is based on aspectJ. Therefore the aspectj runtime library must be in the classpath. It can be included with the following Maven dependency:

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>???</version>
</dependency>
```

Cibet uses aspectj's runtime weaving of application classes. Therefore the runtime weaver must be set as a java agent when your application is started. This can be achieved by adding the following parameter to the java start command:

```
java -javaagent:<path_to>\aspectjweaver-???.jar
```

The non-intrusive way is to configure an aspect for `CibetContext` in aspectj's `aop.xml` file.

## ***Manual initialization***

The context can be started manually with

```
Context.start();
```

Don't forget to close the context with

```
Context.end();
```

### ***Implicit by sensors***

If the context has not been started by one of the explicit methods described before the context is initialized and closed when a sensor is executed.

### ***Implicit by Service classes***

If the context has not been started by one of the explicit methods described before the context is initialized and closed when one of the methods of ArchiveLoader, DcLoader, Locker, SchedulerLoader or other business logic is invoked. For this to work, the application must be started with the aspectweaver javaagent (see POJO object invocations)

### ***Implicit by the Scheduler Task***

If the context has not been started by one of the explicit methods described before the context is initialized and closed when a scheduler task is executed.

## ***6.2User and Tenant***

Some controller and actuators must know the user who executes the business process and optionally the tenant/client in a multi-client application. The user must be set into the session scope context to be accessible by Cibat. Normally, this can be done after the user has been authenticated during the login process. If the user is authenticated by Apache Shiro, Spring Security or JavaEE Security Cibat can detect the logged in user by itself and nothing has to be done.

If the user is authenticated by other means, the user can be set into the HTTP session:

```
session.setAttribute("CIBET_USER", user);
```

If no HTTP session is available the user can be set directly into the Cibat context. This is best done after login, when the user has been authenticated:

```
public void login(String user, String password) {  
    // authentication  
    ...  
    Context.sessionScope().setUser(user);  
}
```

Cibat detects the user by applying implementations of the AuthenticationProvider interface in a sequence. The first AuthenticationProvider that returns a user stops the sequence and subsequent AuthenticationProviders are no more queried.

It is possible to apply an own customized AuthenticationProvider implementation. A customized implementation must be registered either by code or by configuration. In code execute

```
Configuration.instance().registerAuthenticationProvider(  
    new MyCustomAuthProvider());
```

Alternatively add in cibet-config.xml:

```
<authenticationProvider>  
    <class>com.myapp.MyCustomAuthProvider</class>  
</authenticationProvider>
```

A custom AuthenticationProvider can be configured with properties:

```
<authenticationProvider>  
    <class>com.myapp.MyCustomAuthProvider</class>  
    <properties>  
        <myProp>any</myProp>  
    </properties>  
</authenticationProvider>
```

In this case the implementation must have a property myProp with getter and setter according to the Java beans convention.

In a multi-tenant system, the tenant to whom the user belongs must be set into the Cibet session context. The tenant can be set into the HTTP session:

```
session.setAttribute("CIBET_TENANT", user);
```

If no HTTP session is available the tenant can be set directly into the Cibet context:

```
Context.sessionContext().setTenant(tenant);
```

Alternatively, a customized implementation of the AuthenticationProvider interface like described above that implements the getTenant() method could be registered. It is also possible to inherit from one of the Cibet AuthenticationProviders and override the getTenant() method. Registration can be done by code or by configuration like demonstrated above.

## **HTTP requests**

When the CibetContextFilter is applied in a web application, Cibet detects the user by applying implementations of the AuthenticationProvider interface in the following sequence:

1. If a user is already in Context.sessionScope this one is taken
2. If Apache Shiro libraries are in the classpath, the authenticated user is taken from Shiro SecurityUtils if any
3. If Spring Security libraries are in the classpath, the authenticated user is taken from Spring Security SecurityContextHolder if any

4. If `HttpServletRequest.getUserPrincipal()` is not null, the user name is taken from that Principal
5. If in an EJB container the `SessionContext.getCallerPrincipal()` is not equal to 'guest' or 'ANONYMOUS', the user name is taken from that Principal
6. If the user is set in the HTTP session under key `CIBET_USER` this is taken
7. If the filter parameter `allowAnonymous` is set to true (default is false), the IP and port of the remote client will be set as user. The filter parameter can be set like this:

```
<filter>
  <filter-name>cibetContextFilter</filter-name>
  <filter-class>
    com.logitags.cibet.context.CibetContextFilter
  </filter-class>
  <init-param>
    <param-name>allowAnonymous</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
```

## ***EJB invocations***

The `CibetContextInterceptor` sets the user applying the following sequence:

1. The sequence described above for http requests is applied
2. If no user is found up to here and a property 'USER\_NAME' is found in the `invocationContext.contextData` map the value is set as user. EJB context data could be set when invoking remote EJBs for example with the JBoss EJB client API.
3. If the caller principal name in the EJB context is not anonymous or guest, this is taken as user
4. The user is set to ANONYMOUS

For the tenant the following algorithm is applied:

1. If a tenant is found in `Context.sessionScope` which is not the default tenant this one is taken
2. If a property 'TENANT\_NAME' is found in the `invocationContext.contextData` map the value is set as tenant. EJB context data could be set when invoking remote EJBs for example with the JBoss EJB client API.
3. If the tenant is set in the HTTP session under key `CIBET_TENANT` this is taken
4. The default tenant is used

## ***POJO object invocations***

The `@CibetContext` annotation has an optional property `allowAnonymous` which is defaulted to false.

```
@CibetContext(allowAnonymous=true)
public class SomeService {
```

In addition to the sequence described above for http requests, the user will be set to 'ANONYMOUS' if `allowAnonymous` is set to true.

## **6.3EntityManager**

Many Cibet actuators depend on a database (see description of actuators if a database is needed). Cibet can be used in applications that are based on JPA database access but also in applications based on plain JDBC and DataSources. Internally, Cibet uses JPA to persist and query entities.

An instance of the `EntityManager` interface serves as API for persistence operations. The persistence unit must be defined in the `persistence.xml` of the application and the `EntityManager` that Cibet should use must be provided to the Cibet request-scope context. This is done automatically when the context is started with `CibetContextFilter`, `CibetFilter`, `CibetContextInterceptor` or `@CibetContext`. All that has to be done is to start the context as described in chapter 6.1, define a persistence unit in the `persistence.xml` and make the persistence unit available in JNDI. The filter, interceptor or annotation care for finding and instantiating the correct `EntityManager` for Cibet. Also in JDBC applications you have to include the Java persistence API and a `persistence.xml` must be present. The configuration of the persistence unit depends on the environment:

- In a Java EE application, the transaction-type must be JTA and the unit name must be Cibet
- In a Java SE application the transaction-type must be `RESOURCE_LOCAL` and the unit name must be `CibetLocal`.
- In a JDBC application the `<provider>` must be set to `com.logitags.cibet.sensor.jdbc.bridge.JdbcBridgeProvider`. Transaction-type and unit name could be either JTA/Cibet or `RESOURCE_LOCAL/CibetLocal`

In a Java EE environment the `EntityManager` is injected into `CibetEEContext` EJB. In most cases Cibet will detect the JNDI name of this EJB itself, but in some environments it can't find out if it is not a standard name. If you observe an error in the log that Cibet failed to retrieve the JNDI name of `CibetEEContext` EJB when executing in an EJB container, you can set it as a parameter of the `CibetContextFilter`:

```
<filter>
  <filter-name>cibetContextFilter</filter-name>
  <filter-class>
```

```

        com.logitags.cibet.context.CibetContextFilter
    </filter-class>
    <init-param>
        <param-name>EJB_JNDI_NAME</param-name>
        <param-value>CibetEEContextEJBLocal</param-value>
    </init-param>
</filter>

```

The Cibet entities must be declared in the persistence unit with the <jar-file> tag. Add

```
<jar-file>cibet-<version>.jar</jar-file>
```

to the Cibet / CibetLocal persistence unit. Please note that according to Java EE specification jar files are specified relative to the directory or jar file that contains the root of the persistence unit. It can be quite tricky to find out what the root is, depending on your archive type (ear or war) and the persistence provider. Read my entry in java.net Glassfish forum (<http://www.java.net/forum/topic/glassfish/glassfish/whats-root-web-archive-jar-file-entry-persistence.xml>) and see Java EE specification for details and examples. You can also define the classes explicitly:

```

<class>com.logitags.cibet.actuator.archive.Archive</class>
<class>com.logitags.cibet.actuator.common.Controllable</class>
<class>com.logitags.cibet.core.EventResult</class>
<class>com.logitags.cibet.resource.ResourceParameter</class>
<class>com.logitags.cibet.sensor.ejb.EjbResource</class>
<class>com.logitags.cibet.sensor.http.HttpRequestResource</class>
<class>com.logitags.cibet.sensor.jdbc.driver.JdbcResource</class>
<class>com.logitags.cibet.sensor.jpa.JpaResource</class>
<class>com.logitags.cibet.sensor.jpa.JpaQueryResource</class>
<class>com.logitags.cibet.sensor.pojo.MethodResource</class>
<class>com.logitags.cibet.resource.Resource</class>

```

A JTA persistence unit must be made available in JNDI. This can be done by declaring it in a deployment descriptor. In web.xml add the following:

```

<persistence-unit-ref>
    <persistence-unit-ref-name>java:comp/env/Cibet</persistence-unit-ref-name>
    <persistence-unit-name>Cibet</persistence-unit-name>
</persistence-unit-ref>

```

Transaction management of Cibet / CibetLocal EntityManager is different in EE and SE applications:

- In an EE application, the Cibet EntityManager will join an active JTA transaction. It is therefore necessary, that the controlled business case is executed in a transacted environment.
- In an SE application Cibet will start and commit a transaction for the CibetLocal EntityManager. The CibetLocal EntityManager is independent from a transacted environment. If an application transaction is rolled back during execution of a business case, the CibetLocal transaction should also be rolled back with

```
httpSession.setAttribute("CIBET_ROLLBACKONLY", "true");
```

or

```
Context.requestScope().setRollbackOnly(true);
```

## 6.4 Using JPA sensor in a JDBC application

With `JdbcBridgeProvider` all sensors and actuators can be used the same way as with a JPA `EntityManager`. Even the JPA sensor can be used, though it makes not much sense to use `JdbcBridgeProvider` in a JPA application. If `JdbcBridgeProvider` should be used with the JPA sensor, some additional implementation effort has to be done.

If you want to use JPA sensor in a JDBC application (whatever sense that makes), you have to give Cibat some extra information. The `JdbcBridgeEntityManager` translates the JPA API into calls to the JDBC API. Cibat does not know however, how the entities of your application are persisted; it does not know primary keys, table and column names. Therefore this functionality must be implemented by you. Cibat provides the interface `com.logitags.cibat.sensor.jdbc.EntityDefinition` and an abstract implementation `com.logitags.cibat.sensor.jdbc.AbstractEntityDefinition`, from which you can inherit an own implementation per entity that you want to persist. The interface defines the following methods:

```
void persist(Connection jdbcConnection, Object obj) throws SQLException;
```

This method will be called when the `EntityManager.persist(Object)` is called. Implement this method with an INSERT SQL statement. An implementation of this method must not commit or close the connection.

```
<T> T merge(Connection jdbcConnection, T obj) throws SQLException;
```

This method will be called when the `EntityManager.merge(Object)` method is called. Implement this method with an UPDATE SQL statement and return the updated object. An implementation of this method must not commit or close the connection.

```
void remove(Connection jdbcConnection, Object obj) throws SQLException;
```

This method will be called when the `EntityManager.remove(Object)` method is called. Implement this method with a DELETE SQL statement. An implementation of this method must not commit or close the connection.

```
<T> T find(Connection jdbcConnection, Class<T> clazz, Object primaryKey);
```

This method will be called when the `EntityManager.find(Class, Object)` method is called. Implement this method with a SELECT SQL statement. An implementation of this method must not commit or close the connection.

```
List<Object> createFromResultSet(ResultSet rs) throws SQLException;
```

This method will be called when one of the `EntityManager.createQuery(String)`, `createNamedQuery(String)` or `createNativeQuery(String)` methods is called and a `getResultList()` or `getSingleResult()` is executed on the returned Query object. The `ResultSet`

contains the result of the query. From each record in the `ResultSet` an object must be created and returned in a list.

```
Map<String, String> getQueries();
```

returns a map of the names of all defined queries of the entity, mapped to the native SQL statement. The name could be

1. the name of a JPA named query
2. the native SQL query itself (if it is a native query)
3. a JPA query (if it is an on-the-fly query)

Example: Entity `Contract` is persisted in table `app_ctrct` and defines a JPA named query:

```
@NamedQuery(name = "SEL_CTRCT", query = "SELECT c FROM Contract c WHERE  
c.contractNumber = :number")
```

In order to use this query with `EntityManager.getNamedQuery()`, the `getQueries()` method in `ContractEntityDefinition` implementation of `EntityDefinition` must return a map which contains the native SQL statement for this named query:

```
String sql = "SELECT id, ctrctNumber, create_date, reference FROM app_ctrct  
WHERE ctrctNumber = ?";  
map.put("SEL_CTRCT", sql);
```

If the same query is used also as a native query with `EntityManager.getNativeQuery()`, the following entry must be added into the returned map:

```
map.put(sql, sql);
```

If the same query is used as a JPA query with the `EntityManager.getQuery()` method, add

```
map.put("SELECT c FROM Contract c WHERE c.contractNumber = :number", sql);
```

You have to implement one `EntityDefinition` for each entity that you want to persist with the `EntityManager` API. These implementations must be registered in `JdbcBridgeEntityManager`. You can do this with a static registering method in `JdbcBridgeEntityManager`:

```
JdbcBridgeEntityManager.registerEntityDefinition(  
    MyJdbcEntity.class,  
    new MyJdbcEntityDefinition());
```

## 7. Integrating a sensor

### 7.1 EJB sensor

This sensor controls the invocation of local EJB methods and remote EJBs on the server side. The resource is in this case the EJB class name together with the method name and the call parameters.



**Requirements:**

The javaee library must be in the classpath.

**Configuration:**

EJB service methods can be controlled by Cibat. With ARCHIVE actuator this means that an archive entry with class, method name and parameters is written. If the invoked method's return is not void the result value is stored with the archive entry. With dual control schemes this means that the method will not be invoked but suspended until a second user released the INVOKE action.

There is only one thing to do in order to let Cibat control EJB method invocations: Declare the CibatInterceptor interceptor in your EJB:

```
@Stateless
@Remote
@Interceptors({com.logitags.cibat.sensor.ejb.CibatInterceptor.class})
public class MyEJBImpl implements MyEJB {
    ...
}
```

## **7.2EJB-CLIENT sensor**

This sensor controls the invocation of remote EJBs on the client side. The resource is in this case the EJB class name together with the method name and the call parameters.

**Requirements:**

The javaee library must be in the classpath.

**Configuration:**

The EJB-CLIENT sensor behaves like the EJB sensor only that the invocation of the EJB is controlled on the client side before the RPC call is done. Remote EJB calls are controlled by applying the Cibat implementation of the InitialContextFactory. Usually, remote EJB calls are done by configuring the remote connection data in a map and calling InitialContext with this map. The sensor is applied like this:

```
props.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
          CibatRemoteContextFactory.class.getName());
props.put(CibatRemoteContext.NATIVE_INITIAL_CONTEXT_FACTORY,
          "org.jboss.naming.remote.client.InitialContextFactory");
props.put(javax.naming.Context.PROVIDER_URL, "remote://localhost:4447");
props.put("jboss.naming.client.ejb.context", true);
InitialContext ctx = new InitialContext(props);
```

The original InitialContextFactory must be declared in a property with key 'com.logitags.cibat.naming.factory.initial'. The key is defined as a constant in CibatRemoteContext.

If an EJB is looked up with this InitialContext and a method is called on this EJB it will be intercepted on the client side.

## 7.3 JPA sensor

This sensor controls insert, update and delete of JPA entities. The resource is in this case the entity.

### Requirements:

The JPA API library must be in the classpath. It is included in the javaee library. This actuator needs dependency cibet-jpa instead of cibet-core:

```
<dependency>
  <groupId>com.logitags</groupId>
  <artifactId>cibet-jpa</artifactId>
  <version>${version}</version>
</dependency>
```

### Configuration:

Use the Cibet persistence provider in the persistence unit that shall be controlled. The original provider of your persistence framework must be set in property `com.logitags.cibet.persistence.provider`:

```
<persistence-unit name="myAPP-PU" transaction-type="JTA">
  <provider>com.logitags.cibet.sensor.jpa.Provider</provider>
  ...
  <properties>
    <property name="com.logitags.cibet.persistence.provider"
      value="org.eclipse.persistence.jpa.PersistenceProvider"/>
  </properties>
</persistence-unit>
```

Only **explicit** persistence actions of `EntityManager` are controlled by Cibet, that is `persist()`, `merge()`, `remove()` and `find()`. **Implicit** persistence actions are not controlled. When for example the following method is executed in a transaction and the `Account` is updated in the database implicitly this is not controllable by Cibet

```
public void updateAccount(String accountNumber, int amount) {
    Account account = cibetEM.find(Account.class, accountNumber);
    account.setBalance(account.getBalance() + amount);
}
```

When using dual control, `ARCHIVE` or `SCHEDULER` actuators, the entity is serialized and stored in a Cibet table. When the entity has lazy loading properties and is not completely loaded, a `LazyInitialisationException` may be thrown. Performance issues must be taken into account if entities contain large collections. It is not a good idea to control entities with the

Cibet EntityManager that have many associations like for example an Enterprise entity with a collection of Employee. Per default, the actuators load the entity completely before storing it. The find() method of the Cibet EntityManager loads entities lazily per default. It can be forced to load eagerly by setting

```
<property name="com.logitags.cibet.persistence.loadEager" value="true"/>
```

in the persistence unit.

## 7.4 JPAQUERY sensor

This sensor controls named queries, on-the-fly queries and native queries. The resource is in this case the query name, the JPA query or the native SQL query together with any parameters.

### Examples:

Named query:

```
<setpoint id="t1">
  <controls>
    <event>INVOKE</event>
    <target>com.logitags.cibet.helper.TEntity.SEL_BY_OWNER</target>
  </controls>
  <actuator name="ARCHIVE" />
</setpoint>
```

JPA query:

```
<setpoint id="t2">
  <controls>
    <event>INVOKE</event>
    <target>DELETE FROM TEntity a WHERE a.owner = :owner</target>
  </controls>
  <actuator name="FOUR_EYES" />
</setpoint>
```

Native query:

```
<setpoint id="t3">
  <controls>
    <event>INVOKE</event>
    <target>
      select COUNTER from CIB_TESTENTITY where OWNER = ?1
    </target>
  </controls>
  <actuator name="ARCHIVE" />
</setpoint>
```

### Requirements:

The JPA API library must be in the classpath. It is included in the javaee library. This actuator needs dependency cibet-jpa instead of cibet-core:

```
<dependency>
  <groupId>com.logitags</groupId>
  <artifactId>cibet-jpa</artifactId>
  <version>${version}</version>
</dependency>
```

### Configuration:

Use the Cibet persistence provider in the persistence unit that shall be controlled. The original provider of your persistence framework must be set in property `com.logitags.cibet.persistence.provider`:

```
<persistence-unit name="myAPP-PU" transaction-type="JTA">
  <provider>com.logitags.cibet.sensor.jpa.Provider</provider>
  ...
  <properties>
    <property name="com.logitags.cibet.persistence.provider"
      value="org.eclipse.persistence.jpa.PersistenceProvider" />
  </properties>
</persistence-unit>
```

The `getSingleResult()` and `getResultList()` methods of the Cibet Query load entities lazily per default. It can be forced to load eagerly by setting

```
<property name="com.logitags.cibet.persistence.loadEager" value="true" />
```

in the persistence unit.

## 7.5 ASPECT sensor

This sensor controls the invocation of public methods of any object. The resource is in this case the object's class name together with the method name and the call parameters.

### Requirements:

The ASPECT sensor is based on aspectj and uses aspectj's runtime weaving of application classes. Therefore the runtime weaver must be set as a java agent when your application is started. This can be achieved by adding the following parameter to the java start command:

```
java -javaagent:<path_to>\aspectjweaver.jar
```

When dual control mechanisms or the REDO functionality are applied and the invoked class is a Spring bean you need the Spring libraries on the classpath. In your Spring config file add the following lines:

```
<context:component-scan base-package="com.logitags.cibet"/>
<aop:config proxy-target-class="true"/>
```

## Configuration:

Cibet supports two mechanisms to set an ASPECT sensor:

1. by annotation. This allows some optional configuration which may be necessary for the redo or release of the method invocation.
2. Non-intrusively by configuration of an AspectJ aspect

### 1. Setting the sensor by annotation

There is only one thing to do in order to let Cibet control method invocations by the ASPECT sensor: Declare an interceptor in your class with the `com.logitags.cibet.sensor.pojo.CibetIntercept` annotation:

```
@CibetIntercept
public class MyPojoClass {
    ...
}
```

The `@CibetIntercept` annotation can be set on class or on method level. If set on class level, all public methods of that class are under Cibet control; if set on method level only this method is under Cibet control.

When dual control actuators or the REDO functionality out of an archived method invocation are applied the instantiation mechanism of the controlled object must be declared. Cibet provides three factory classes:

```
@CibetIntercept(factoryClass = PojoFactory.class)
public class MyPojoClass {
    public MyPojoClass(){
    }
}
```

This is the default. The `PojoFactory` must be used if the `MyPojoClass` object is instantiated with the default constructor or as a Singleton. If the constructor or Singleton method takes a String parameter it can be also set in the Intercept annotation:

```
@CibetIntercept(factoryClass = PojoFactory.class, param="123")
public class MyPojoClass {
    public MyPojoClass(String aParameter){
    }
}
```

If the `MyPojoClass` object is instantiated with the help of a factory class, the `FactoryFactory` class must be used and the actual factory class must be set in the param value:

```
@CibetIntercept(factoryClass = FactoryFactory.class,
                 param="my.app.MyFactory")
public class MyPojoClass {
```

The FactoryFactory class will search in MyFactory for a method that will return an instance of the MyPojoClass class:

```
public class MyFactory {
    MyPojoClass createMyPojoClass(){
        ...
    }
}
```

You can also set the factory method explicitly:

```
@CibetIntercept(factoryClass = FactoryFactory.class,
                  param="my.app.MyFactory.createMyPojoClass()")
public class MyPojoClass {
```

If the object on which to invoke the method is a Spring bean the SpringBeanFactory must be set:

```
@CibetIntercept(factoryClass = SpringBeanFactory.class)
public class MyPojoClass {
```

If the object is declared with a bean id in the Spring config, this id can be set in the param value:

```
@CibetIntercept(factoryClass = SpringBeanFactory.class, param="myPojo")
public class MyPojoClass {
```

## 2. Setting the sensor with an aspect

Instead of setting the sensor in the code it can also be defined as a concrete aspect in `aop.xml` which must be in the META-INF directory in the classpath of the application. See the AspectJ documentation for detailed information about pointcuts and concrete aspects and how to define them in `aop.xml`. An example of an aspect configuration in `aop.xml` that controls invocations of the `getName()` method in class `SubArchiveController` is:

```
<aspectj>
  <aspects>
    <concrete-aspect name="com.xyz.tracing.AspectTest"
      extends="com.logitags.cibet.sensor.pojo.CustomAspect">
      <pointcut name="cibetIntercept"
        expression="target(com.helper.SubArchiveController) &&
          execution(public String getName())"/>
    </concrete-aspect>
  </aspects>
</aspectj>
```

With an annotation this aspect would look like this:

```
public class SubArchiveController {
    @CibetIntercept
    public String getName() {
```

```

    } ...
}

```

The rules for defining aspects in aop.xml for the ASPECT sensor are:

- The name of the aspect can be any name
- The aspect must extend `com.logitags.cibet.sensor.pojo.CustomAspect`
- The pointcut name must be `cibetIntercept`

It is not possible to configure the factory class like with the annotation. However, Cibet tries to find out the factory class by itself. If you want to have more control about the factory class the annotation variant must be used.

It is also possible to use a custom aspect to intercept EJB methods instead of using `com.logitags.cibet.sensor.ejb.CibetInterceptor`.

## 7.6 HTTP-FILTER sensor

This sensor controls http requests to an URL. The resource is in this case the URL together with header, attributes, querystring and form parameters.

### Requirements:

This sensor needs also the servlet API 2.5

### Configuration:

The HTTP-FILTER sensor is in fact a servlet filter. In order to install the filter add to your web.xml something like this:

```

<filter>
  <filter-name>cibetFilter</filter-name>
  <filter-class>com.logitags.cibet.sensor.http.CibetFilter
</filter-class>
</filter>

<filter-mapping>
  <filter-name>cibetFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

```

Often it is not desirable to run all requests through CibetFilter. Static resources should be excluded for example. This is possible with the `excludes` parameter which takes a comma separated list of exclude regular expression patterns (see also chapter 6.1).

Please note that you don't have to configure the `CibetContextFilter` if you use HTTP-FILTER sensor with the `CibetFilter` (see chapters `Scopes` and `Cibet Context` and ). `CibetFilter` inherits from `CibetContextFilter` and contains already its functionality.

## 7.7 HTTP-PROXY sensor

While HTTP-FILTER controls http requests on the server side, this sensor controls on the client side. It does this with the help of a proxy server. It is possible to start several proxy servers with different configuration. Each proxy server has a unique name.

### Requirements:

### Configuration:

The following properties can be configured per proxy (see also JavaDoc of class ProxyConfig):

Property	Data type	Default	Description
cibet.proxy.mode	ProxyMode	NO_PROXY	The mode in which the proxy is run: NO_PROXY: no proxy is started MITM: Proxy is started as Man-in-the-middle. SSL requests can be controlled but no upstream proxy can be addressed. This should be the default for control of https requests CHAINEDPROXY: Proxy is started in proxy mode. SSL requests are opaque. An upstream proxy can be configured. Only http requests can be controlled.
cibet.proxy.port	int	8078	Proxy port
cibet.proxy.chainedProxyHost	String		Optional host name of an upstream proxy. Ignored when MITM mode is set.
cibet.proxy.chainedProxyPort	int		Optional port of an upstream proxy. Ignored when MITM mode is set.
cibet.proxy.bufferSize	int	1024*1024	Chunked requests are aggregated in a buffer. Increase buffer size if the body content of the requests exceeds the default.
cibet.proxy.timeout	int	10000	the timeout for connecting to the upstream server on a new connection, in milliseconds
cibet.proxy.clientKeystore	String		optional name of a keystore in the classpath for SSL with client certificate authentication
cibet.proxy.clientKeystorePassword	String		optional password of a keystore in the classpath for SSL with client certificate authentication



cibet.proxy.excludes	Comma separated list of String		a list of regex patterns for URLs that shall be excluded from sensing
----------------------	--------------------------------	--	---

Multiple proxies with different properties can be started which must have a unique name. The proxy can be configured and started by one of four ways:

### 1. Filter parameters

The proxy can be configured in the web.xml for the CibetContextFilter or CibetFilter filter. The proxy name is appended to the configuration parameter. This will start a proxy with name MyProxy:

```
<filter>
  <filter-name>cibetContextFilter</filter-name>
  <filter-class>
    com.logitags.cibet.context.CibetContextFilter
  </filter-class>
  <init-param>
    <param-name>cibet.proxy.mode.MyProxy</param-name>
    <param-value>MITM</param-value>
  </init-param>
  <init-param>
    <param-name>cibet.proxy.port.MyProxy</param-name>
    <param-value>10113</param-value>
  </init-param>
  ...
</filter>
```

### 2. System properties

The proxy paramters can be set as system properties in code or when starting the application:

```
System.setProperty(Configuration.PROXY_CLIENT_KEYSTORE + ".MyProxy",
  "clientKeystore.jks");
System.setProperty(Configuration.PROXY_CLIENT_KEYSTOREPASSWORD +
  ".MyProxy", "test");
System.setProperty(Configuration.PROXY_PORT + ".MyProxy", "10113");
System.setProperty(Configuration.PROXY_MODE + ".MyProxy",
  ProxyMode.MITM.name());
Configuration.instance().initialise();
```

### 3. Manual

The proxy can be configured and started directly in code:

```
ProxyConfig config = new ProxyConfig();
Config.setName("MyProxy");
config.setMode(ProxyMode.CHAINEDPROXY);
config.setPort(10113);
```

```
config.setChainedProxyHost("192.168.55.1");
config.setChainedProxyPort(11113);
Configuration.instance().startProxy(config);
```

#### 4. MBean

The proxy can be started with a JMX console by calling the method

```
String startProxy(String name, String mode, int port, int bufferSize, int
    timeout, String chainedHost, int chainedPort, String clientKeystore,
    String keystorePassword, String excludes)
```

of MBean ConfigurationService. All parameters except name, mode and port are optional.

#### Usage

Start the application with system properties http.proxyHost and http.proxyPort set to the Cibet proxy or set the proxy according to your http client framework like:

```
HttpHost proxy = new HttpHost("localhost", 10113);
CloseableHttpClient cl = HttpClientBuilder.create().setProxy(proxy).build();
```

and send http requests as usual. Actuators are executed on the request according to the setpoint configurations.

Context propagation: As the proxy is running in an own thread the Cibet context of the calling thread must be propagated to the proxy thread. This is necessary if context information like user name, tenant, remarks, playing mode etc. should be available in the proxy thread. The Cibet context can be propagated by adding a header to the HTTP request like:

```
method.addHeader(Headers.CIBET_CONTEXT.name(), Context.encodeContext());
```

### 7.8JDBC sensor

This sensor controls inserts, updates and deletes in a database table. The resource is in this case the SQL query together with parameters.

#### Requirements:

A JDBC driver of your preferred database must be in the classpath. This actuator needs dependency cibet-jpa instead of cibet-core:

```
<dependency>
  <groupId>com.logitags</groupId>
  <artifactId>cibet-jpa</artifactId>
  <version>${version}</version>
</dependency>
```

## Configuration:

JDBC calls can be intercepted and controlled by Cibat. The sensor is implemented as a JDBC driver with class name `com.logitags.cibat.sensor.jdbc.driver.CibatDriver`. The connection URL has the following format:

```
jdbc:cibat:<native sql driver class>:<native URL>
```

where `<native sql driver class>` is the class name of the original JDBC driver and `<native URL>` is the original connection URL without leading `jdbc:`. For an Oracle database the URL would be for example like this:

```
jdbc:cibat:oracle.jdbc.OracleDriver:oracle:thin:@192.168.1.64:1521:xe
```

Not all JDBC queries can be controlled. Generally, only INSERT, UPDATE and DELETE statements are controlled. Batch updates and deletes are not controlled. Furthermore, only updates and deletes where the primary key column is in the WHERE clause can be controlled. Tables with multi-column primary keys are not supported.

RESTORE event is not supported for JDBC sensor.

## 8. Configuration

Configuration of Cibat consists of parameterizing actuators and defining setpoints. Optionally, providers for security, notification and authentication and custom controls can be defined.

This can be done either in a configuration file `cibat-config.xml` which is found in the classpath or in code. Cibat looks for all files named `cibat-config.xml` in the classpath and merges its configuration.

If for example a setpoint should be defined in code, do something like

```
Setpoint setpoint1 = new Setpoint("sp1");
...
Configuration.instance().registerSetpoint(setpoint1);
```

### 8.1 Definition of setpoints

The general structure of a setpoint in xml is:

```
<setpoint>
  <controls>
    ...
  </controls>
  <actuator />
```

```

    <actuator />
    ...
</setpoint>

```

The controls elements represent the IF part where the conditions are defined for which the following declared actuators should be applied.

## Unique identifier

Each setpoint has a unique identifier by which it is identified during runtime and in the log files.

XML:

```

<setpoint id="sp1">
    ...

```

Code:

```

Setpoint setpoint1 = new Setpoint("sp1");

```

## Setpoint inheritance

Setpoints can inherit from other setpoints. This allows a significant reduction of configuration code in complex control scenarios. Only the control definitions which is the IF condition are inherited. The actuators or the THEN part is not inherited.

XML:

```

<setpoint id="Basic">
    ...
</setpoint>

<setpoint id="sp1" extends="Basic">
    ...
</setpoint>

```

Code:

```

Setpoint basicSP = new Setpoint("Basic");
Setpoint setpoint1 = new Setpoint("sp1", basicSP);

```

If e.g. in setpoint Basic a target control element and some other controls are defined and in setpoint sp1 the same target should be controlled but with other conditions, the target element must not be repeated in sp1. If the parent setpoint contains a control that you don't want apply in a child setpoint just set an empty control in the child setpoint.

Please see the scenarios for controlling a domain object (<http://www.logitags.com/cibet/scenario-domain.html>) and controlling a service call (<http://www.logitags.com/cibet/scenario-usecase.html>) for complete examples for definition of setpoints.

## Input tolerance

The Cibet principle of ‘Keep it simple’ is applied in the Setpoint API. The methods for adding a control accept input in various styles. If you want for example set a method control for methods doThis() and makeThat() you can achieve this in the following ways:

```
sp.setMethod("doThis()", "makeThat()");
```

```
sp.setMethod("doThis(), makeThat()");
```

The result is the same. So if you are unsure about how something works in Cibet just try it out! If you don’t find it out by intuition, please tell us what we can approve to make it simpler.

Often a control definition accepts more than one value, like in the above case, where two methods are defined for the setpoint. In the xml configuration, the values are separated by comma or semicolon:

```
<target>
    SELECT t FROM TEntity t WHERE t.owner=:owner,
    SELECT t FROM TEntity t WHERE t.counter>=:counter
</target>
```

Sometimes the control values may contain themselves comma or semicolon. In this case, surround the value with “”:

```
<target>
    SELECT t FROM TEntity t WHERE t.owner=:owner,
    "SELECT t.id, t.counter FROM TEntity t WHERE t.counter>=:counter"
</target>
```

In the following all existing Control implementations and setpoint definitions are described. The Controller evaluates them in their declared order.

## 8.2 Tenant Control

Evaluates the tenant in the setpoint against the tenant which is set in the context. Evaluates also tenant hierarchies. In tenant hierarchies the tenant ids are separated by a |. If no setpoints for the actual tenant exist the Control checks if setpoints for one of the parents up to the default tenant exist. This is an optional control and can be omitted if the application is not tenant-specific. A setpoint can be defined for multiple tenants.

XML:

```
<!-- setpoint matches for the German client of IBank -->
<tenant>IBank|Germany</tenant>

<!-- setpoint matches for the German and French client of IBank -->
<tenant>IBank|Germany, IBank|France</tenant>
```

Code:

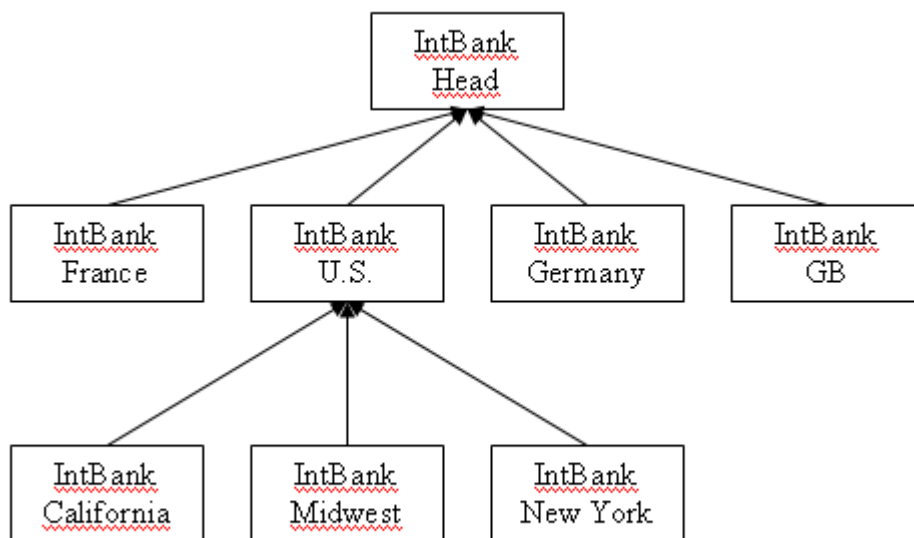
```
setpoint.setTenant("IBank|Germany");
```

```
setpoint.setTenant("IBank|Germany", "IBank|France");
```

Cibet supports multi-tenant systems. A multi-tenant application hosts several clients or tenants where each client has its own view on the system without interference between the clients. Each tenant can view only his own data and accesses the application without influence on the other tenants. Of course this could be achieved by deploying an own instance of the application and database independently for each tenant but often this means much higher effort for maintenance and support. Therefore it is sometimes easier, especially if there are many clients to host, to install them all on one instance of the application and to use one database for all. One possibility to keep data visibility restricted to the tenant is to assign to each domain object (and table record) the tenant to whom it belongs.

In Cibet multi-tenancy is achieved through introduction of a tenant attribute to all internal Cibet domain objects. Cibet supports also tenant hierarchies. What does that mean?

Let's say you run an application for a big international bank with name IntBank. The bank has national branches and in some countries subsidiaries. The bank's business rules allow headquarters to see data of all branches and subsidiaries. The branches may see only their national data including the data of the national subsidiaries. The subsidiaries at last can see only their local data. You may have the following tenant hierarchy for IntBank:



**Figure 7: Example tenant hierarchy**

The Cibet tenant attribute is a concatenation of the tenant ids separated by a |. In the example the tenant attributes may then be:

Head

Head|France, Head|US, Head|Germany, Head|GB

Head|US|California, Head|US|Midwest, Head|US|New\_York

If an application is not tenant-specific Cibet uses '\_\_DEFAULT' as default for the tenant attribute.

## 8.3 Event Control

Evaluates the event(s) in the setpoint against the current event. A setpoint can be defined for multiple events. It depends on the sensor and the controlled resource which control events can be applied (see chapter ‘Controls in the context of the sensor’).

XML:

```
<!-- matches for UPDATE, INSERT and DELETE persistence operations -->
<event>UPDATE, INSERT, DELETE</event>

<!-- matches all entity persistence operations -->
<event>PERSIST</event>
```

Code:

```
setpoint.setEvent(ControlEvent.UPDATE.name(), ControlEvent.INSERT.name(),
                  ControlEvent.DELETE.name());

setpoint.setEvent(ControlEvent.PERSIST.name());
```

## 8.4 Target Control

Evaluates the target for this setpoint. The target could be a class name of a domain object when its persistence should be controlled, the class name of a service class if a service call should be controlled or a URL if a http request should be controlled. In the context of JDBC sensor the target is the table name which is controlled. It depends on the sensor and the controlled resource what the target is (see chapter ‘Controls in the context of the sensor’).

The target element can contain more than one target name. The target element accepts patterns with \*. Here are some examples:

XML:

```
<!-- a concrete full qualified class name -->
<target>com.app.accounting.Account</target>

<!-- all classes of a package -->
<target>com.app.client.*, com.app.gui.*</target>

<!-- all classes of the service package that start with Txn -->
<target>com.app.service.Txn*</target>

<!-- a concrete URL -->
<target>http://www.logitags.de/cibet/documentation.html</target>

<!-- all resources in cibet/ -->
<target>http://www.logitags.de/cibet/*</target>

<!-- a JPA query -->
<target>SELECT a FROM TEntity a WHERE
      a.owner = ?1 AND a.xCaltimestamp = ?2
</target>
```

Code:

```
setpoint.setTarget("com.app.accounting.Account");
setpoint.setTarget("com.app.client.*", "com.app.gui.*");
setpoint.setTarget("com.app.service.Txn*");
setpoint.setTarget("http://www.logitags.de/cibet/documentation.html");
setpoint.setTarget("http://www.logitags.de/cibet/*");
setpoint.setTarget("SELECT a FROM TEntity a WHERE
    a.owner = ?1 AND a.xCaltimestamp = ?2");
```

## 8.5 State Change Control

Evaluates the state of object attributes or the state of columns of an updated table (JDBC). This evaluation is executed only for UPDATE control events, for other events this control is ignored. The control compares the actual state of the object under control against its persistent state. The stateChange element can contain more than one property name. The control has an attribute 'exclude', which defines how the comparison should be applied:

- exclude=false: This is the default. The setpoint matches if at least one of the given properties is modified.
- exclude=true: The setpoint matches if at least one property is modified that is not in the given list.

Attributes in associated objects of the controlled object can be declared with the point notation in stateChange elements. If for example an object has an attribute addr of type Address which in turn has a String attribute city the transitive property is written as addr.city

Example classes:

```
public class Customer {
    private String name;
    private boolean active;
    private Address addr;
    ...
}

public class Address {
    private String city;
    private String email;
    private Contact contact;
    ...
}

public class Contact {
    private String telephone;
    private String telephone2;
    private String telephone3;
    ...
}
```

XML:

```
<!-- exclude Customer state changes of active property and
    transitive property telephone3 in Contact object -->
<target>com.app.Customer</target>
```



```

<stateChange exclude="true">active, addr.contact.telephone3</stateChange>

<!-- include Address state changes of email property and all
      properties of associated Contact object -->
<target>com.app.Address</target>
<stateChange>email, contact</stateChange>

```

Code:

```

setpoint.setStateChange(true, "active", "addr.contact.telephone3");

setpoint.setStateChange("email", "contact");

```

StateChange control will not work if uncommitted modifications exist on the controlled target entity. Because the dirty check is done in a second transaction a LockAcquisitionException could occur.

## 8.6 Method Control

Evaluates the actual executed method against the method configured in the setpoint. The Method Control can be applied for Java method invocations as well as for http requests. In the latter case the http method is evaluated. The method element accepts patterns as method name. A setpoint can be defined for multiple methods. Here are some examples:

XML:

```

<!-- a concrete method without parameters -->
<method>doSomething()</method>

<!-- a concrete method with parameters -->
<method>doSomething(String, com.logitags.cibet.TEntity, int)</method>

<!-- all overloaded methods of the same name -->
<method>doSomething</method>

<!-- all methods starting with do or make -->
<method>do*, make*</method>

<!-- http POST method -->
<method>POST</method>

```

Code:

```

setpoint.setMethod("doSomething()");
setpoint.setMethod("doSomething(String, com.logitags.cibet.TEntity,
    int)");
setpoint.setMethod("doSomething");
setpoint.setMethod("do*", "make*");
setpoint.setMethod("POST");

```

## 8.7Invoker Control

For events detected by JPA, JDBC, EJB and ASPECT sensors this control evaluates the direct and indirect invoker methods in the current thread against the invokers configured in the setpoint. For INVOKE events from a HTTP-FILTER sensor, the invoker is looked up in the server chain from where the request originated. This is the remote host IP or DNS name. If the request has been passed through one or more proxy servers the list of invokers is constructed from the HTTP\_X\_FORWARDED\_FOR header if present. An invoker chain looks something like:

client, proxy1, proxy2, proxy3

The Invoker Control checks in this case if the request has been invoked or passed through one of the given IPs.

The control has an attribute 'exclude', which defines how the comparison should be applied:

- `exclude=false`: This is the default. The setpoint matches if the actual event has been invoked directly or indirectly by one of the given methods or an http request has passed through one of the given IPs.
- `exclude=true`: The setpoint matches if the actual event has not been invoked directly or indirectly by one of the given methods or a http request has not passed through one of the given IPs.

This control allows for example, that an actuator is executed for a business case if it is called from the GUI, but is not when called from a batch process.

The invoker element accepts patterns and multiple tokens. Here are some examples:

XML:

```
<!-- no control if invoked by any method of class BatchProcessor -->
<invoker exclude="true">com.app.batch.BatchProcessor</invoker>

<!-- control only if invoked by any method of any class in the
      com.root package or from any method of any class in the
      webservice package -->
<invoker>
  com.root.*,
  com.app.webservice.*
</invoker>

<!-- control only if invoked by receive method in class
      AccountService. The receive method can be overloaded,
      method signature is not evaluated -->
<invoker>com.app.web.AccountService.receive()</invoker>

<!-- control only if http request comes from an IP or
      has passed through a proxy with this IP -->
<invoker>192.168.1.45</invoker>

<!-- control only if http request comes from an IP range or
      has passed through a proxy within this range -->
<invoker>192.168.*</invoker>
```

Code:

```

setpoint.setInvoker(true, "com.app.batch.BatchProcessor");
setpoint.setInvoker("com.root.*", com.app.webservice.*);
setpoint.setInvoker("com.app.web.AccountService.receive()");
setpoint.setInvoker("192.168.1.45");
setpoint.setInvoker("192.168.*");

```

## 8.8 Condition Control

Evaluates JavaScript conditions. The condition element can contain any JavaScript statements that return a boolean value at the end. The setpoint matches if the condition evaluates to true. The condition could also be externalized into a file. This is more convenient if several conditions must be concatenated or you want to use functions. In order to read the condition from a file start the condition with 'file:' and give then the location of the file. The location can be set absolute or as a classpath entry.

The following variables can be used in a condition definition:

- \$REQUESTSCOPE: the context request scope
- \$SESSIONSCOPE: the context session scope
- \$APPLICATIONSCOPE: the context application scope
- \$EVENT: the actual control event

Additional variables are available depending on the resource that is being controlled.

EJB and POJO method invocations:

- \$TARGETOBJECT: the actual object which is invoked
- \$TARGET: the class name of the object which is invoked
- \$PARAM0, \$PARAM1 ... \$PARAMn: method parameter objects in declared order

JPA:

- \$TARGETOBJECT: the actual persistent object
- \$TARGET: the class name of the persistent object
- \$ + *simple class name*: the actual persistent object (same as TARGET)
- \$PRIMARYKEY: the unique id value of the persistent object

JPAQUERY:

- \$TARGET: named query, JPA query or native query

JDBC:

- \$TARGETOBJECT: the SQL statement
- \$TARGET: the table name in the SQL statement
- \$PRIMARYKEY: the primary key value in the WHERE clause of the SQL statement
- \$COLUMNS: the parameters of an SQL statement as a list of SqlParameter

HTTP request:

- \$TARGET: the requested URL
- \$HTTPATTRIBUTES: map of http attributes
- \$HTTPHEADERS: map of http headers

- \$HTTPPARAMETERS: map of http querystring or form parameters

XML:

```
<!-- This is a special setpoint for Werner -->
<condition>$SESSIONSCOPE.getUser() == 'Werner'</condition>

<!-- this setpoint matches only for Test environment (the property
      must have been set before) -->
<condition>
  $APPLICATIONSCOPE.getProperty('environment').equals('TEST')
</condition>

<!-- this setpoint matches if an object is in the Ciber context under key
      currentSession and the getCounter() method
      of this object returns 7 -->
<condition>
  $SESSIONSCOPE.getProperty("currentSession") != null
  &and;
  $SESSIONSCOPE.getProperty("currentSession").getCounter() == 7
</condition>

<!-- setpoint matches if the getCounter() method of the actual object
      under control of type AccountManager yields > 10 -->
<condition>$AccountManager.getCounter() > 10</condition>

<!-- setpoint matches if first parameter of the actual
      method under control has value 2 -->
<condition>$PARAM0 == 2</condition>

<!-- setpoint matches if second parameter of the actual method under
      control which is of type Date is in the past -->
<condition>
  importPackage(java.util);
  var DATE = new Date();
  println('todays date: ' + DATE);
  println('Date parameter: ' + $PARAM1);
  $PARAM1.before(DATE);
</condition>

<!-- setpoint matches if third parameter is null -->
<condition>$PARAM2 == null</condition>

<!-- setpoint matches if a String http attribute equals to Hase, a
      boolean http header value is true and an int http
      parameter is 67 -->
<condition>
  $HTTPATTRIBUTES.get('p1')=='Hase' &and;
  $HTTPHEADERS.get('head1') == true &and;
  $HTTPPARAMETERS.get('param1') == 67
</condition>

<!-- setpoint matches if the condition in file testscript2.js which is
      in the classpath is fulfilled -->
<condition>file:testscript2.js</condition>

<!-- setpoint matches if the condition in file testscript1.js which is
      located under the given URL is fulfilled -->
<condition>
```

```
file:C:\projects\cibet/src/test/resources/testscript1.js
</condition>
```

Code:

```
setpoint.setCondition("$SESSIONSCOPE.getUser() == 'Werner'");
setpoint.setCondition("$APPLICATIONSCOPE.getProperty('environment')
    .equals('INVOKE')");
setpoint.setCondition("$SESSIONSCOPE.getProperty(\"currentSession\")
    != null
    && $ SESSIONSCOPE.getProperty(\"currentSession\").getCounter() == 7");
setpoint.setCondition("$AccountManager.getCounter() > 10");
setpoint.setCondition("$PARAM0 == 2");
setpoint.setCondition("importPackage(java.util); var DATE = new Date();
    println('todays date: ' + DATE); println('Date parameter: '
    + $PARAM1); $PARAM1.before(DATE);");
setpoint.setCondition("$PARAM2 == null");
setpoint.setCondition("$HTTPATTRIBUTES.get('pl')==Hase' &&
    $HTTPHEADERS.get('head1') == true &&
    $HTTPPARAMETERS.get('param1') == 67");
setpoint.setCondition("file:testscript2.js");
setpoint.setCondition(
    "file:C:\projects\cibet/src/test/resources/testscript1.js");
```

## 8.9 Implementing Custom Controls

Own custom Control implementations can be registered in the configuration file or with the Configuration API. In cibet-config.xml add a control element and set the class tag. If the Control implementation uses own properties, they can be defined with property tags. The implementation must have a setter method with a name following the Java Beans convention and taking a String argument as parameter:

XML:

```
<control name="MYCONTROL">
    <class>com.company.MyControl</class>
    <properties>
        <myAttribute>someValue</myAttribute>
    </properties>
</control>
```

Code:

```
Configuration.instance().registerControl(new MyControl());
```

Control implementations must implement interface `com.logitags.cibet.control.Control`, provide a default constructor and must provide a unique name which is returned by method `getName()`. It is recommended to inherit from `AbstractControl` and overwrite the methods the custom control is interested in. This ensures that the custom control works also in case of future interface enhancements.

In a setpoint the custom control is applied with the `<customControl>` tag.

For example you want a business case be executed only when some system object is in a special state. A custom Control may check custom context parameters. Own context properties can be set into application-, session- or request scope which provide methods to set, get and remove a property.

Register a custom SystemStateControl and set the state into the context before executing the business case:

```
Configuration.instance().registerControl(new SystemStateControl());
...
Context.requestScope().setProperty(
    "SYSTEM_STATE", systemObject.getState());
```

Implement the evaluate(Object controlValue, EventMetadata metadata) method of SystemStateControl something like:

```
If (Context.requestScope().getProperty("SYSTEM_STATE") == OK) {
    return true;
} else {
    return false;
}
```

The method hasControlValue() should return always true in this case as the evaluation is not dependent on the value of the <customControl> tag. Apply the control in a setpoint like this:

```
<setpoint id="K1">
  <controls>
    <customControl name="MYSystemStateControl" />
  </controls>
  <actuator name="INFOLOG" />
</setpoint>
```

If the value of the <customControl> tag is a comma separated list of states for which the control should evaluate true SystemStateControl must be implemented like this:

- The resolve() method parses the comma separated list of states into a List object. The implementation of AbstractControl does right this.
- The parameter of the hasControlValue() method is the List object. It must return true if the list is not null and contains at least one element. The implementation of AbstractControl does right this.
- The first parameter of the evaluate() method is the List object. Implement it something like this:

```
if (((List)controlValue).contains(
    Context.requestScope().getProperty("SYSTEM_STATE"))) {
    return true;
} else {
    return false;
}
```

In the setpoint the control will then be applied like this:

```

<setpoint id="K1">
  <controls>
    <customControl name="MYSystemStateControl">OK, PENDING</customControl>
  </controls>
  <actuator name="INFOLOG"/>
</setpoint>

```

## 8.10 Controls in the context of the sensor

The meaning of the described controls can differ according to the applied sensor. The following table lists how controls are applied in the context of the different sensors and what values are allowed.

Control Sensor	Tenant	Event	Target	State Change	Method	Invoker	Condition
<b>EJB</b>	Tenant in request scope	INVOKE, RELEASE, REDO	name of EJB class whose method is controlled	Not applicable	EJB method	Direct or indirect class / method that invoked the controlled method	JavaScript with usage of specific properties \$... (method parameter names)
<b>EJB-CLIENT</b>			name of the class whose method is controlled		Pojo class method		
<b>ASPECT</b>			URL		http method	HTTP_X_FORWARDED_FOR header, remote host IP or DNS name	JavaScript with usage of Specific properties \$HTTP ATTRIBUTES, \$HTTP HEADERS, \$HTTP PARAMETERS
<b>HTTP-FILTER</b>							
<b>HTTP-PROXY</b>					Not applicable	Direct or indirect class / method that called the JPA / JDBC method	JavaScript
<b>JPAQUERY</b>			Query name, JPAQL statement, SQL statement				
<b>JPA</b>		INSERT, UPDATE, DELETE, SELECT, RELEASE, RESTORE	entity class name	entity attribute that is modified (only for UPDATE events)			JavaScript
<b>JDBC</b>		INSERT, UPDATE, DELETE, RELEASE	table name	column that is modified (only for UPDATE events)			JavaScript with usage of Specific properties \$COLUMNS, \$PRIMARYKEY

## 8.11 Actuator configuration

In the setpoints only the names of actuators are declared:

XML:

```
<actuator name="SPRING_SECURITY" />
```

Code:

```
Actuator act = Configuration.instance().getActuator(
    SpringSecurityActuator.DEFAULTNAME);
setpoint.addActuator(act);
```

Each built-in actuator has a default name.

Some actuators have properties which can be adjusted. This is done in the configuration file in actuator elements or in code by setting the properties directly. The `SPRING_SECURITY` actuator for example needs rules for authorization. If there is only one rule which should be applied in the whole application it can be defined as follows:

XML:

```
<cibet>

  <!-- set the preAuthorize property of the built-in default
        SpringSecurityActuator instance -->
  <actuator name="SPRING_SECURITY">
    <properties>
      <preAuthorize>hasRole('WALTER')</preAuthorize>
    </properties>
  </actuator>

  <setpoint>
    ...
</cibet>
```

Code:

```
SpringSecurityActuator act = (SpringSecurityActuator)
    Configuration.instance().getActuator(
        SpringSecurityActuator.DEFAULTNAME);
act.setPreAuthorize("hasRole('WALTER')");
```

If there are more rules to apply what normally is the case you have to instantiate another `SpringSecurityActuator`:

XML:

```
<actuator name="PermitAuthorizer">
  <class>
    com.logitags.cibet.actuator.springsecurity.SpringSecurityActuator
  </class>
  <properties>
    <permitAll></permitAll>
  </properties>
</actuator>
```

Code:



```
SpringSecurityActuator permitter = new
    SpringSecurityActuator("PermitAuthorizer");
permitter.setPermitAll(true);
Configuration.instance().registerActuator(permitter);
```

## **8.12 Runtime Configuration changes**

The configuration from file cibet-config.xml is read into memory when the application server is started. Afterwards the configuration in memory can be overwritten only by code. However, it is possible to force a rereading of the configuration file and to re-initialise the configuration at runtime with the help of a JMX MBean. At startup Cibet registers automatically a ConfigurationService MBean in the application servers MBean server. In order to make a reinitialisation start a JMX console like jconsole which is part of the standard Java SDK (Figure 8). You will find the ConfigurationService MBean which offers eight operations:

- initialize: reads the complete configuration file and re-initialises Cibet
- reinitActuators: re-initialises only the actuators from the configuration file
- reinitControls: re-initialises only the control definitions from the configuration file
- reinitSetpoints: re-initialises only the setpoints from the configuration file
- reinitAuthenticationProvider: re-initialises only the AuthenticationProvider from the configuration file
- reinitNotificationProvider: re-initialises only the NotificationProvider from the configuration file
- reinitSecurityProvider: re-initialises only the SecurityProvider from the configuration file
- start a proxy: starts a proxy for the HTTP-PROXY sensor with the given parameters

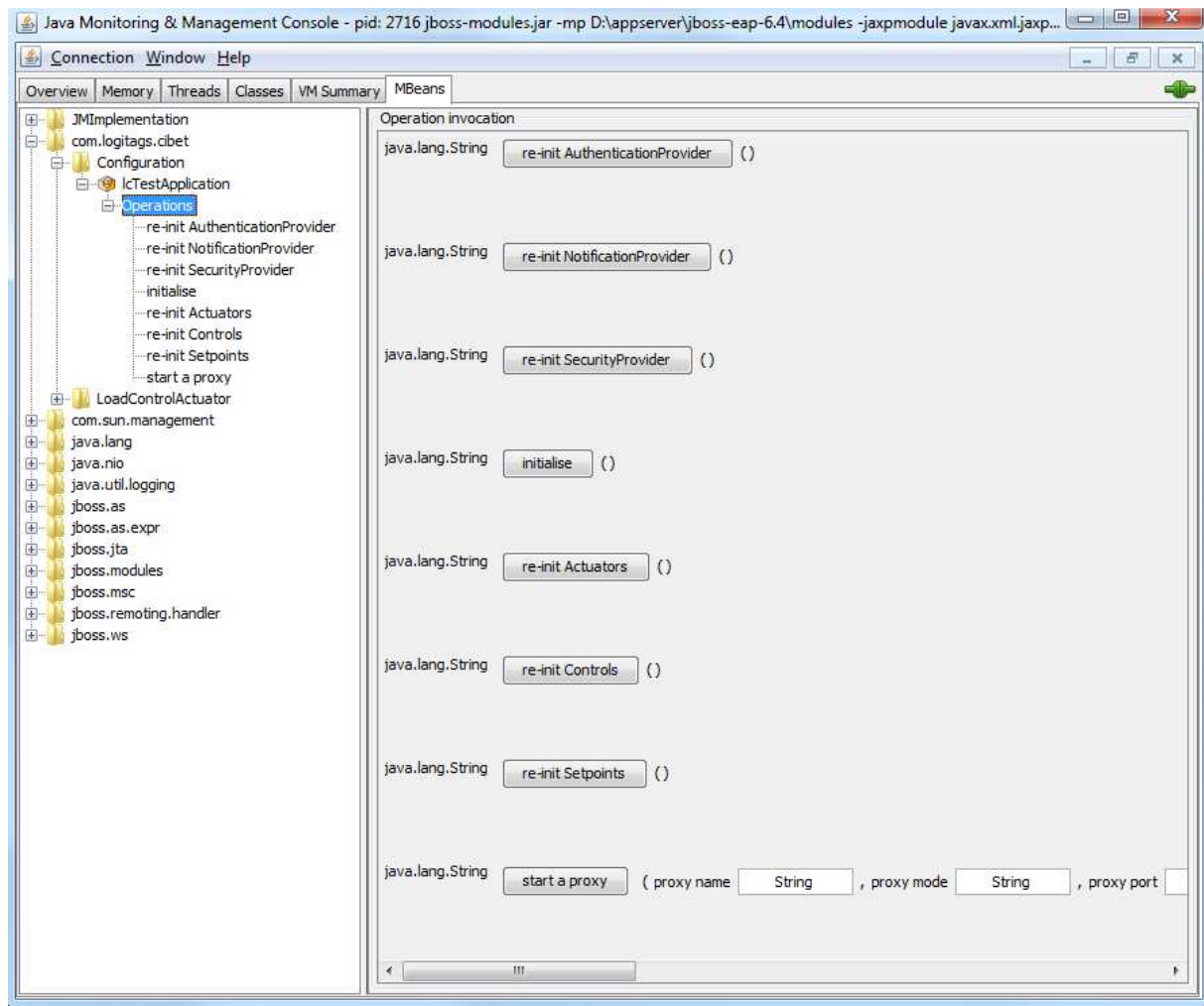


Figure 8: Cibet ConfigurationService deployed as MBean

## 9. Actuators

### 9.1 INFOLOG actuator

**Default name:** INFOLOG

**Class:** com.logitags.cibet.actuator.info.InfoLogActuator

#### Requirements:

None

#### Properties:

**Description:** Before and after the control event a message is logged in INFO level to the standard log channel. This actuator is for debugging and testing purpose. The logged data are not encrypted.

## 9.2 TRACKER actuator

**Default name:** TRACKER

**Class:** com.logitags.cibet.actuator.info.TrackerActuator

### Requirements:

This actuator needs a database. The database scheme can be created with the SQL scripts `<dbms>.sql` included in the release. There exist scripts for Derby, MySQL, PostgreSQL and Oracle database management systems.

### Properties:

**Description:** This actuator tracks what internally happens in Cibat. All controls executed by Cibat can be tracked and monitored as Event Results. The tracked data include applied sensor, setpoints and actuators, control result, user and so on. Nested control events are also tracked. When for example a method call is controlled by Cibat and within the execution of the method a nested persistence action is also controlled the second Event Result is a child of the first one.

This actuator writes all results of intercepted business cases into the database into table `cib_eventresult`. See chapter Post- Checking Control Results.

## 9.3 FOUR\_EYES actuator

**Default name:** FOUR\_EYES

**Class:** com.logitags.cibet.actuator.dc.FourEyesActuator

### Requirements:

This actuator needs a database. The database scheme can be created with the SQL scripts `<dbms>.sql` included in the release. There exist scripts for Derby, MySQL, PostgreSQL and Oracle database management systems.

### Properties:

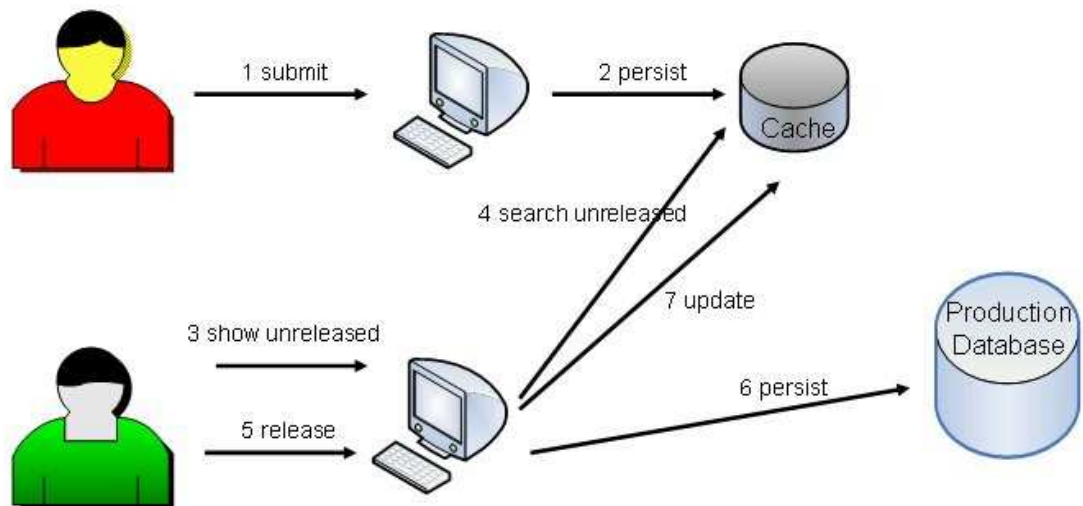
Property	Data type	Default	Description
throwPostponedException	boolean	false	throw an Exception when this actuator is applied
jndiName	String	null	jndi name of EJB under control
sendAssignNotification	boolean	true	Send a notification to assigned user
sendReleaseNotification	boolean	true	Send a release notification to the initiating user
sendRejectNotification	boolean	true	Send a reject notification to the initiating user
encrypt	boolean	false	If true sensible resource data

			are encrypted in the database.
storedProperties	Collection		list of JPA entity property names that will be stored as ResourceParameters with the Controllable. Only applicable for PERSIST events and JPA sensor
loadEager	boolean	true	If a JPA entity is controlled makes a complete eager load and detach of the entity. If set to false, performance may be increased but the entity may not be compared or released.

### Description:

With the FOUR\_EYES actuator a dual control mechanism is applied. If a user executes a persistence action or method invocation it will not be done directly in the productive environment. Instead, the action is stored and postponed. Only if a second user accepts and releases the action, it is performed in the productive system. If the second user rejects the action it will not be performed. Figure 9 shows the work flow of a four-eyes controlled process.

1. A user submits a business case which can be one of the before mentioned control events.
2. Cibat sensors detect that the business case is under dual control and postpones it. All relevant business case and context data are persisted in a cache.
3. A second user is responsible to check and release or reject postponed business cases
4. He searches for unreleased business cases and validates them
5. When the validation is positive he may release the business case, otherwise he rejects it and the following step 6 is not executed.
6. The business case is executed in the productive system, i.e. a persistence action is executed or a method is called on an object or a http request is sent.
7. The cache of postponed business cases is updated.



**Figure 9: Four-Eyes Principle**

Concerning the submitting and releasing users the following rules are applied:

- the two users must be different
- submitting user cannot release
- if submitting user rejects, the action will be discarded
- if releasing user release the action it will become productive
- if releasing user rejects the action it will be discarded

Let's make an example: User A changes the account number in an account object which is controlled by FOUR\_EYES actuator. The updated state of the object is stored in Cibat but the object itself is not updated. Later on user A remembers that not only the bank account but also the bank name must be changed. If he tries to update the account with the changed bank name he will receive a warning that an unreleased update exists already for this account. He has two choices now:

- He rejects the first update of the account number. While release must be done by a different user than the one who executed the controlled action, rejection can be done also by the same user. After rejection he changes account number and bank name and persists the update which will be postponed. Now user B accepts and releases the account change and the update is performed in the productive database.
- He waits until user B has accepted and released the change of the account number. After that user A can change the bank name and persist the update. Now user B accepts and releases the bank name change and the update is performed in the productive database.

When a dual control actuator is set in a setpoint the persistence action or service call will be suspended and postponed. If a business case is suspended can be checked by querying the EventResult object returned by the method

```
EventResult result = Context.requestScope().getExecutedEventResult();
```

The executionStatus property of EventResult will be POSTPONED, otherwise EXECUTED (see also chapter Post- Checking Control Results).

It is also possible to let Cibet throw an Exception when the business case is postponed. For this the throwPostponedException attribute must be set to true in the actuator, either in the Cibet config file or in code. The property can be set generally for the actuator or for a single setpoint (see chapter Actuator configuration).

XML:

```
<actuator name="FOUR_EYES">
  <properties>
    <throwPostponedException>true</throwPostponedException>
  </properties>
</actuator>
```

Code:

```
FourEyesActuator ac = (FourEyesActuator)
    Configuration.instance().getActuator("FOUR_EYES");
ac.setThrowPostponedException(true);
```

If a business case is set under dual control a PostponedException will then be thrown:

```
try {
    entityManager.insert(transaction);
} catch (PostponedException e) {
    ...
}
```

The PostponedException contains the Controllable object for the unapproved resource as a parameter.

For http requests which are under dual control the behavior is different. In http protocol it is not possible to transmit an exception. The request result is communicated by response codes. Therefore, the value of the throwPostponedException property has normally no effect in intercepted http requests. Instead, a response code 202 (ACCEPTED) is returned.

When a resource is in a postponed state, no other user can execute a business case controlled by Cibet. If for example updates and deleted of Contract entities are controlled by a dual control actuator in a setpoint and a user has done an update which has been postponed, another user will get an UnapprovedResourceException when he tries to delete the contract. The exception contains the Controllable object for the unapproved resource as a parameter. The same is valid for other resources: for a method invocation resource for example another user cannot invoke the same method with exactly the same parameters.

Release or Rejection of a FOUR\_EYES intercepted business cases are post control functionalities and are described in chapter Releasing and Rejecting Dual Control Events.

The jndiName property of the FOUR\_EYES actuator must not be set normally. When a dual control actuator is applied on an EJB method, the release method must look up the EJB from JNDI context. JNDI names are not standardized in Java EE specification < 3.1. When an EJB

service is invoked in a release or redo action Cibet tries to find out the JNDI name of the EJB applying various strategies. If the JNDI name cannot be figured out and an exception is thrown, it must be configured explicitly in property jndiName.

XML:

```
<actuator name="FOUR_EYES">
  <properties>
    <jndiName>
      com.logitags.cibet.javaee.CibetTestEJBImpl/remote
    </jndiName>
  </properties>
</actuator>
```

Code:

```
FourEyesActuator ac = (FourEyesActuator)
    Configuration.instance().getActuator("FOUR_EYES");
ac.setJndiName("com.logitags.cibet.javaee.CibetTestEJBImpl/remote");
```

The postponed business case will be stored in the database as a Controllable object. It is possible to encrypt sensible Controllable data. See chapter Security. It depends on the postponed resource which data are encrypted. The resource target, the result and the values of resource parameters will be encrypted.

When a JPA entity is controlled by FOUR\_EYES it is possible to store properties of this entity together with the Controllable object. It is then possible to create SQL queries for searching after these properties. When you have for example a Company entity under FOUR\_EYES control and you want to find all postponed business cases of Company with name 'Father & Son' or where the business type is Iron & Steel then you must store these properties with the Controllable. This can be done like this

XML:

```
<actuator name="FOUR_EYES">
  <properties>
    <storedProperties>companyName, businessType</storedProperties >
  </properties>
</actuator>
```

Code:

```
FourEyesActuator ac = (FourEyesActuator)
    Configuration.instance().getActuator("FOUR_EYES");
ac.getStoredProperties().add("companyName");
ac.getStoredProperties().add("businessType");
```

Another possibility is grouping Controllables to search for Controllable groups. By default, Controllables are grouped by a sensor-specific default. For the JPA sensor for example this is 'targetType'-'primaryKeyId'. The default group id can be overwritten by setting it in the request scope context like this:

```
Context.requestScope.setGroupId("myGroupId");
```

The DcLoader provides a method

```
List<Controllable> DcLoader.loadAllByGroupId(String groupId)
```

to search for archives by their group id. This possibility exists for all dual control and scheduler actuators.

## 9.4SIX\_EYES actuator

**Default name:** SIX\_EYES

**Class:** com.logitags.cibet.actuator.dc.SixEyesActuator

### Requirements:

This actuator needs a database. The database scheme can be created with the SQL scripts <dbms>.sql included in the release. There exist scripts for Derby, MySQL, PostgreSQL and Oracle database management systems.

### Properties:

Property	Data type	Default	Description
throwPostponedException	boolean	False	throw an Exception when this actuator is applied
jndiName	String	Null	jndi name of EJB under control
sendAssignNotification	boolean	true	Send a notification to assigned user
sendReleaseNotification	boolean	true	Send a release notification to the initiating user
sendRejectNotification	boolean	true	Send a reject notification to the initiating user
encrypt	boolean	false	If true sensible resource data are encrypted in the database.
storedProperties	Collection		list of JPA entity property names that will be stored as ResourceParameters with the Controllable. Only applicable for PERSIST events and JPA sensor
loadEager	boolean	true	If a JPA entity is controlled makes a complete eager load and detach of the entity. If set to false, performance may be increased but the entity may not be compared or released.

### Description:



The SIX\_EYES actuator applies an even higher dual control concept than the FOUR\_EYES actuator using the same mechanism. Additionally to FOUR\_EYES a third user must release the controlled action in order to get it productive. The applied rules are as follows:

- the three users must be different
- submitting user cannot release
- if submitting user rejects, the action will be discarded
- if first and second releasing user release the action it will become productive
- if first releasing user releases and second releasing user rejects the action will be discarded
- if first releasing user rejects the action will be discarded at once without any action of a second releasing user

SIX\_EYES actuator inherits from FOUR\_EYES actuator and has the same properties and behavior.

## **9.5TWO\_MAN\_RULE actuator**

**Default name:** TWO\_MAN\_RULE

**Class:** com.logitags.cibet.actuator.dc.TwoManRuleActuator

### **Requirements:**

This actuator needs a database. The database scheme can be created with the SQL scripts `<dbms>.sql` included in the release. There exist scripts for Derby, MySQL, PostgreSQL and Oracle database management systems.

### **Properties:**

Property	Data type	Default	Description
throwPostponedException	boolean	false	throw an Exception when this actuator is applied and no second user is authenticated
jndiName	String	null	jndi name of EJB under control
removeSecondUserAfter Release	boolean	false	The second authenticated user is removed from context after release if set to true
sendAssignNotification	boolean	true	Send a notification to assigned user
sendReleaseNotification	boolean	true	Send a release notification to the initiating user
sendRejectNotification	boolean	true	Send a reject notification to the initiating user
encrypt	boolean	false	If true sensible resource data are encrypted in the database.

storedProperties	Collection		list of JPA entity property names that will be stored as ResourceParameters with the Controllable. Only applicable for PERSIST events and JPA sensor
loadEager	boolean	true	If a JPA entity is controlled makes a complete eager load and detach of the entity. If set to false, performance may be increased but the entity may not be compared or released.

### Description:

This is another dual control actuator that imposes an additional requirement (see [http://en.wikipedia.org/wiki/Two-man\\_rule](http://en.wikipedia.org/wiki/Two-man_rule)). The controlled event executed by a submitting user must not only be released by a second user, but with this actuator the submitting and releasing user must be present at the same time. This means that the event must be authorized by two users simultaneously.

This control scheme can be observed for example at the cash desk of a supermarket when a customer wants to return a purchased product. While the cashier remains logged in into the terminal, a second employee has to authorize the cash return.

The following rules are applied:

- the submitting and releasing users must be different
- if at submitting time a second releasing user is authenticated the action will be executed at once.
- If at submitting time no second releasing user is authenticated the action will be postponed.
- Only the submitting user can request the release of a postponed action. If at that time a second releasing user is authenticated the action will be executed.
- Any user can reject a postponed action. The action will be discarded

TWO\_MAN\_RULE actuator inherits from FOUR\_EYES actuator and has the same properties and behavior with some exceptions:

As two users must be authenticated at the same time, the second authenticated user must be set into the session scope context with method `setSecondUser()`. If at the time of executing the controlled event a second user is set into the context, the event is released and executed directly. If no second user is set into the context the event is postponed and the presence of a second user is checked during release. The release of a TWO\_MAN\_RULE controlled event requires that the same user who initiated the event must be logged in and be set into session scope with `setUser()` method and a second user must be set into session scope with `setSecondUser()`.

With the property `removeSecondUserAfterRelease` it can be controlled whether the second user should be removed automatically from the context after release. Set this property to true to have a higher security. If several events should be released in one use case, set this property to false so that the second user must not authenticate himself after each release. After the last release the second user must be removed from context manually.

Authentication of the second user is out of scope of Cibat. It depends on the security and access system that is applied how this is achieved. Most security frameworks do not allow out of the box authentication of a second user. For the Spring Security and Apache Shiro frameworks Cibat provides functionality to log on and off a second user while the first user is still authenticated in the current session/thread:

```
// For Spring Security use Spring bean
com.logitags.cibat.actuator.springsecurity.SpringSecurityService from
cibet-springsecurity:
try {
    Authentication request = new UsernamePasswordAuthenticationToken(
        name, password);
    SpringSecurityService man =
        applicationContext.getBean(SpringSecurityService.class);
    man.logonSecondUser(request);
} catch (AuthenticationException e) {
    System.out.println("Authentication failed: " + e.getMessage());
}

// For Apache Shiro use com.logitags.cibat.actuator.shiro.ShiroService from
cibet-shiro project:
try {
    AuthenticationToken token = new UsernamePasswordToken(name, password);
    ShiroService.logonSecondUser(token);
} catch (AuthenticationException e) {
    System.out.println("Authentication failed: " + e.getMessage());
}
```

After the two-man rule use case has been executed the second user should be logged off. If the property `removeSecondUserAfterRelease` of `TwoManRuleActuator` has been set to true this is done automatically after release. If not it can be done with `SpringSecurityService` / `ShiroService` with method:

```
logoffSecondUser();
```

When instead of the logged in user the second user should be authorized with `SPRING_SECURITY` actuator the property `secondPrincipal` of `SPRING_SECURITY` actuator must be set to true. Likewise, if the second user should be authorized with `SHIRO` actuator the property `secondPrincipal` of `SHIRO` actuator must be set to true. In this case the authorization rules are not applied on the user who is logged into the current session but on the user who has logged in with the above method `logonSecondUser()`.

The `executionStatus` of the `EventResult` object retrieved by

```
EventResult result = Context.requestScope().getExecutedEventResult();
```

is always `POSTPONED`, regardless of whether a second user was present and the business case has been released directly or no second user was present and the business case has been postponed. That is because if such a business case is released directly it consists of two

control events, the originating one, e.g. UPDATE or INVOKE and the RELEASE event. The first one is POSTPONED while the second event will be EXECUTED. This is reflected by the parent – child relationship of EventResult (see also chapter Post- Checking Control Results):

```
// parent
result.getEvent() → INVOKE
result.getExecutionStatus() → POSTPONED

// child
result.getChildResults().get(0).getEvent() → RELEASE
result.getChildResults().get(0).getExecutionStatus() → EXECUTED
```

## 9.6 PARALLEL\_DC actuator

**Default name:** PARALLEL\_DC

**Class:** com.logitags.cibet.actuator.dc.ParallelDcActuator

### Requirements:

This actuator needs a database. The database scheme can be created with the SQL scripts <dbms>.sql included in the release. There exist scripts for Derby, MySQL, PostgreSQL and Oracle database management systems.

### Properties:

Property	Data type	Default	Description
throwPostponedException	boolean	False	throw an Exception when this actuator is applied
jndiName	String	Null	jndi name of EJB under control
sendAssignNotification	boolean	true	Send a notification to assigned user
sendReleaseNotification	boolean	true	Send a release notification to the initiating user
sendRejectNotification	boolean	true	Send a reject notification to the initiating user
executions	int	1	Minimum number of executions of the business case before it can be released
timelag	int	0	Time lag between subsequent executions in seconds
differentUsers	boolean	true	The users who execute the business case must be different
encrypt	boolean	false	If true sensible resource data are encrypted in the database.

storedProperties	Collection		This property is not used in this actuator because it is only applied for PERSIST events and JPA sensor
------------------	------------	--	---

### Description:

This actuator realizes a dual control scheme where a business case is executed in parallel more than once by one or more users, without having any impact on the system. Another releasing user compares the results together with the input parameters and decides which of the variants to reject and which to release.

An example of such a scenario is for example in an accountancy application when one employee physically counts all the valuables and arrives at a balance without knowledge of the balance expected by the paperwork balance. A second employee compiles the transaction information using supporting paperwork and tabulates the teller or store balance. A third employee then compares their findings. If a discrepancy exists, the two employees each verify the others work in an attempt to reconcile the difference. If the employees agree upon a balance and identify that a loss has occurred, they would then follow their appropriate procedures. If there is no difference the third employee releases the business case and the two employees sign the paperwork indicating the agreed upon balance.

This actuator allows defining if the users who execute the business case must be different, if there should be a time lag between the executions and how many executions must be done at minimum before the business case could be released. The releasing user must always be a different user.

PARALLEL\_DC actuator inherits from FOUR\_EYES actuator and has the same properties with some extensions.

This actuator requires that the controlled business case is structured into parts that don't have any impact on the system and parts that have an impact. The parts with no impact are executed always when a user executes the business case but the parts which have an impact are postponed and executed only when the business case is released. Naturally, this actuator cannot be applied on all resources. The above requirement could for example not be fulfilled by resources monitored by JPA sensor. The actuator makes on the other hand sense for controlling methods (ASPECT sensor), EJBs (EJB sensor) or servlets (HTTP-FILTER sensor).

The resource parts which have an impact on the system can be differentiated by querying the isPostponed flag in the Request context. Let's make an example: The following method is controlled by PARALLEL\_DC:

```
public CalculationResult calculateBalance(List<Valuable> vals,
                                         List<Transaction> trans) {

    // now do the calculations which have no impact
    CalculationResult result = doDetailedCalculations(vals, trans);
```

```

        // now this is the part which has an impact. It is executed
        // only when the business case has been released.
        if(!Context.requestScope().isPostponed()) {
            // store the results in the database
            entityManager.persist(result);
            // send the results to another application
            sendToBillingSystem(result);
        }

        return result;
    }
}

```

If a user calls now this method only the calculation is done and returned, but it is not stored nor send to the billing application. The business case is postponed and stored as Controllable object in the database. If another user calls the method, optionally with another set of parameters, the business case is again postponed and stored as Controllable object. The two Controllable objects belong to the same business case instance and are linked by the case ID. Controllables for the same business case instance are assigned the same case ID. The case ID can be retrieved from the EventResult object (see also chapter Post- Checking Control Results) like this:

```

CalculationResult result = calculator.calculateBalance(vals, trans);
EventResult er = Context.requestScope().getExecutedEventResult();
Assert(er.getExecutionStatus() == ExecutionStatus.POSTPONED);
String caseId = er.getCaseId();

```

If another user wants to execute the same business case and link it to the first one he has to set this case ID into the Request context before calling the method:

```

// ... another user
Context.requestScope().setCaseId(caseId);
CalculationResult result = calculator.calculateBalance(vals, trans);

```

If he doesn't set the case ID into the Request context, a new case ID is created and the postponed Controllable object opens a new instance of the calculateBalance() business case. The actuator parameter 'differentUsers' defines if executing the business case with the same case ID must be done by different users or if the same user can execute it more than once. The parameter 'timelag' defines if there must be a minimum time lag between subsequent executions.

The method can now be called by other users and when the same case ID is set into the Request context, always another linked Controllable object is created and stored in the database.

Eventually, someone has to decide which calculation is the good one and release the business case so that the result is stored in the database and the billing system is informed. The business case can only be released when it has been executed (and postponed) at least the minimum number defined by the actuator parameter 'executions'. The releasing user can retrieve all Controllable objects with the same case ID:

```

List<Controllable> list = DcLoader.loadByCaseId(caseId);
for (Controllable co : list) {
    // check and compare the CalculationResult objects
}

```

```

        CalculationResult res = (CalculationResult)
            co.getResource().getResultObject();
        // check and compare the method parameters
        List<Valuable> vals =
            co.getResource().getParameters().get(0).getValue();
        List<Transaction> trans =
            co.getResource().getParameters().get(1).getValue();
    }

    // now decide which one to release
    transaction().begin();(use any mechanism to begin and commit)
    CalculationResult finalResult = (CalculationResult)
        list.get(1).release(entityManager, comment);
    transaction().commit();

    // alternatively, a Controllable from the list can also be rejected:
    list.get(0).reject(entityManager, comment);

```

When one Controllable variant is released, all other Controllable instances with the same case ID are automatically rejected.

The procedure is a little different when PARALLEL\_DC is applied on a resource monitored by HTTP-FILTER sensor due to the fact that it is a remote call on a URL. In contrast to other dual control actuators, the http response code is not 202 (ACCEPTED) when the request is postponed but 200 (OK) and the response body can be retrieved. The EventResult can be retrieved with

```

String evReHeader = response.getFirstHeader(
    CibetFilter.EVENTRESULT_HEADER).getValue();
EventResult result = CibetUtil.decodeEventResult(evReHeader);

```

Setting the case ID into the client's request scope context will not have the desired effect because the server doesn't know something about it. Instead the case ID must be set into the http request as header with name CIBET\_CASEID. This could for example be done like this when using apache-httpClient:

```

HttpGet g = new HttpGet(url);
g.setHeader(HttpRequestInvoker.CASEID_HEADER, caseId);
HttpResponse response = client.execute(g);

```

## **9.7 SCHEDULER actuator**

**Default name:** SCHEDULER

**Class:** com.logitags.cibet.actuator.scheduler.SchedulerActuator

### **Requirements:**

This actuator needs a database. The database scheme can be created with the SQL scripts `<dbms>.sql` included in the release. There exist scripts for Derby, MySql, PostgreSQL and Oracle database management systems.

### **Properties:**

Property	Data type	Default	Description
throwPostponedException	boolean	false	throw an Exception when this actuator is applied and the business case is scheduled
throwScheduledException	boolean	false	throw an Exception when this actuator is applied and an event is already scheduled for the resource
jndiName	String	null	jndi name of EJB under control
encrypt	boolean	false	If true sensible resource data are encrypted in the database.
storedProperties	Collection		list of JPA entity property names that will be stored as ResourceParameters with the Controllable. Only applicable for PERSIST events and JPA sensor
autoRemove ScheduledDate	boolean	true	when true, the scheduled date is removed from the context after this actuator has been applied
timerStart	Date	2 am	First time at which the schedule task is to be executed. Default is the following 2 am. Format in cibet-config.xml is yyyy.MM.dd HH:mm:ss or '+s' to define a timestamp relative from current time in seconds
timerPeriod	long	86400 (one day)	time in seconds between successive task executions for executing scheduled business cases.
datasource	String		Optional JDBC DataSource if the scheduled event is on a JDBC resource
persistenceContextReference	String		Optional reference to a persistence context if the scheduled event is on a JPA resource in a JavaEE environment
persistenceUnit	String		Optional persistence unit name if the scheduled event is on a JPA resource in a JavaSE environment.
batchInterceptor	SchedulerTask Interceptor		Callback class invoked during batch execution of scheduled business cases
loadEager	boolean	true	If a JPA entity is scheduled makes a complete eager load and detach of the entity. If set to false, performance may be



			increased but the entity may not be compared or released.
--	--	--	---

### Description:

This actuator allows scheduling business cases in the future. An event on a resource controlled by SCHEDULER actuator is not executed instantly but postponed until the scheduled date is reached. A batch job executes then the business case at the scheduled date. The execution of the business case at the scheduled time is done as RELEASE event or one of its child events. The release of the business case can then be logged or archived by one of the archiving actuators by defining a setpoint for the release event.

The scheduled date must be set into the request context before the controlled business case is executed. There exist methods for setting an absolute date or a relative date:

```
# set the scheduled date to an absolute Date
httpSession.setAttribute("CIBET_SCHEDULEDDATE", scheduledDate);

or

Context.requestScope().setScheduledDate(scheduledDate);

# set the scheduled date to two days in the future counted from
actual date
httpSession.setAttribute("CIBET_SCHEDULEDDATE", "2");
httpSession.setAttribute("CIBET_SCHEDULEDFIELD",
Calendar.DAY_OF_MONTH);

or

Context.requestScope().setScheduledDate(Calendar.DAY_OF_MONTH, 2);
```

If the property `autoRemoveScheduledDate` is set to false the scheduled date is not removed from the context after the business case was executed. This may be useful if several business cases shall be scheduled in one request.

The property `throwScheduledException` allows checking if a scheduled business case exists already on a resource. If two updates on a resource are scheduled in parallel it should be checked if these updates do not interfere with each other. If `throwScheduledException` is true, a `ScheduledException` is thrown which allows making a decision if the scheduled event is reasonable or not:

```
try {
    entity = entityManager.merge(entity);
} catch (ScheduledException e) {
    // check what is already scheduled
    for (Controllable d : e.getScheduledControllables()) {
        print(d);
        // if the entity has a scheduled update, check the differences
        if(d.getControlEvent() == ControlEvent.UPDATE) {
            List<Difference> l = e.getDifferences(d);
            for (Difference dif : l) {
                print(dif);
            }
        }
    }
}
```

```

    }
  }
}

... decide what to do ...
}

```

When after inspection the second merge should be scheduled, CibeT must be told to ignore any `ScheduledException` for the next request. This can be done by setting a flag in the request context:

```

try {
    Context.requestScope().ignoreScheduledException(true);
    entity = entityManager.merge(entity);
} catch (ScheduledException e) {
    // will never be thrown
}

```

If the controlled resource is a JPA entity or JDBC query, the batch job needs an `EntityManager` or `DataSource` to execute the persistence event. As the job runs in an own thread it has no knowledge of any `EntityManagers` or connections in the CibeT context. The persistence resource is defined by one of the properties `datasource`, `persistenceContextReference` or `persistenceUnit`. Only one of these properties must be set. If the controlled resource is not a persistence resource but for example a method invocation, it is not necessary to set these properties.

#### **datasource:**

If the resource is a JDBC query controlled by JDBC sensor this property defines the JNDI name of a `DataSource` which will be used to execute the query. The `DataSource` must be defined in the `java:comp/env` context. In Tomcat this can be done for example like this:

```

<Resource name="jdbc/myAppDataSource" auth="Container"
    type="javax.sql.DataSource" maxActive="100"
    maxIdle="30" maxWait="10000" username="sepp"
    password="jdf3434SK"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://192.168.1.64:3306/test"
/>

```

#### **persistenceContextReference:**

If the application runs in a JavaEE application server and the scheduled resource is controlled by JPA or JPAQuery sensor, the `persistenceContextReference` property must be set. The persistence context reference makes an `EntityManager` available in JNDI. This can be done for example with the `persistence-context-ref` tag in the `web.xml` like this:

```

<persistence-context-ref>
  <persistence-context-ref-name>
    persistence/myAppPersistenceCtx
  </persistence-context-ref-name>
  <persistence-unit-name>myAppPU</persistence-unit-name>
</persistence-context-ref>

```

where `<persistence-context-ref-name>` is the JNDI name in the `java:comp/env` context, and `<persistence-unit-name>` is the persistence unit name in `persistence.xml`. This persistence context reference will be set like this in the actuator:

```
<actuator name="schedEE">
  <class>com.logitags.cibet.actuator.SchedulerActuator</class>
  <properties>
    <timerStart>2014.12.01 12:00:00</timerStart>
    <timerPeriod>40000</timerPeriod>
    <persistenceContextReference>
      java:comp/env/persistence/myAppPersistenceCtx
    </persistenceContextReference>
  </properties>
</actuator>
```

### **persistenceUnit:**

If the application runs in a non-EJB JavaSE environment and the scheduled resource is controlled by JPA or JPAQuery sensor, the `persistenceUnit` property must be set. This is simply the name of the persistence unit in `persistence.xml` that should be used for the persistence execution.

It can be checked if scheduled versions of a resource exist with the `SchedulerService` interface:

```
// find all scheduled Controllables for the tenant set in
// session scope. Returns all scheduled objects if no tenant is
// set in context.
List<Controllable> list = SchedulerLoader.findScheduled();
// returns all scheduled objects of the given target type
List<Controllable> list2 = SchedulerLoader.findScheduled(targetType);
// check if there exist scheduled events for a JPA entity. The method
// returns a map where the key is a scheduled Controllable object. If
// the event is other than UPDATE, the map value is null. For UPDATE
// events, the map value contains a list of the scheduled updates
Map<Controllable, List<Difference>> map =
    SchedulerLoader.findScheduledDifferences(entity);
```

The scheduled version may be compared with the current version to check the differences, see chapter 10.5. The scheduled business case could also be released or rejected by a user before the scheduled date is reached (see chapter 10.4).

It is possible to do additional work in the batch process just before and after the business case is executed when a callback class is defined for the `SchedulerActuator` with property `interceptorClass`. The callback class must implement `SchedulerTaskInterceptor` which defines a `beforeTask(Controllable)` and a `afterTask(Controllable)` method. If in the `beforeTask()` method a `RejectException` is thrown, the business case will not be executed and the `Controllable` will be set into status `REJECTED`.

```
<actuator name="SCHED1">
  <class>
    com.logitags.cibet.actuator.scheduler.SchedulerActuator
  </class>
  <properties>
    <timerStart>+ 6</timerStart>
```

```

        <throwPostponedException>true</throwPostponedException>
        <autoRemoveScheduledDate>true</autoRemoveScheduledDate>
        <interceptorClass>
            com.logitags.cibet.helper.SchedulerPersistIntercept
        </interceptorClass>
    </properties>
</actuator>

```

## 9.8 ARCHIVE actuator

**Default name:** ARCHIVE

**Class:** com.logitags.cibet.actuator.archive.ArchiveActuator

### Requirements:

This actuator needs a database. The database scheme can be created with the SQL scripts `<dbms>.sql` included in the release. There exist scripts for Derby, MySql, PostgreSQL and Oracle database management systems.

### Properties:

Property	Data type	Default	Description
integrityCheck	boolean	false	create message digest for each archive. This is a static attribute and is valid for all instances of this actuator
encrypt	boolean	false	If true sensible resource data are encrypted in the database.
jndiName	String	null	jndi name of EJB under control
storedProperties	Collection		list of JPA entity property names that will be stored as ResourceParameters with the Archive. Only applicable for PERSIST events and JPA sensor
loadEager	boolean	true	If a JPA entity is archived makes a complete eager load and detach of the entity. If set to false, performance may be increased but the archived entity may not be compared or restored.

### Description:

The controlled event is archived. In case of an updating persistence action on an entity the state of the entity is archived. For a new (insert) persistence action, the state of the newly created entity is archived. For a delete persistence action, the last state of the removed entity is archived. For a method invocation archive, the method parameters and, if applicable the constructor parameters of the object on which the method has been invoked are archived as well as the method result. For an http request, the URL with headers and query parameters are archived. Metadata of the business case like executing user and timestamp are stored with the archive.

The archived state can be restored or re-executed at any time. That means that from its archive, an entity could be reconstructed or if the archive is a method invocation archive, the method could be re-invoked with the same parameters. For an http request archive, the URL with headers and query parameters is archived and can be re-executed.

The archive entries in the database can be secured against manipulation to make them audit-proof and revision safe. It is not possible then to silently modify existing archives or to delete or add archive records without detection by Ciber control. Though it is not possible to prohibit entirely such manipulations except if an encrypted database is used all fraudulent actions can be detected. See chapter Checking Archive Integrity.

It is also possible to encrypt sensible archive data. See chapter Security. It depends on the archived resource which data are encrypted. The resource target, the result and the values of resource parameters will be encrypted.

The ArchiveService API provides methods to load archives for a certain domain class, primary key of domain object, method name or case id. All load methods are tenant-specific. Archives with the same case id belong to a common business case, e.g. a dual control INSERT and its release archives.

The compare methods work same as described in chapter Releasing and Rejecting Dual Control Events. Comparing archives makes only sense for state changing archives, not for service archives. Methods exist for comparing two archives, an archive with the actual domain object or two arbitrary objects.

When a JPA entity is archived it is possible to store properties of this entity together with the Archive object. It is then possible to create SQL queries for searching after these properties. When you have for example a Company entity and want to find all archived business cases of Company with name 'Father & Son' or where the business type is Iron & Steel then you must store these properties with the Archive. This can be done like this

XML:

```
<actuator name="ARCHIVE">
  <properties>
    <storedProperties>companyName, businessType</storedProperties >
  </properties>
</actuator>
```

Code:

```
ArchiveActuator ac = (ArchiveActuator)
    Configuration.instance().getActuator("ARCHIVE");
ac.getStoredProperties().add("companyName");
```

```
ac.getStoredProperties().add("businessType");
```

Another possibility is grouping archives to search for archive groups. By default, archives are grouped by a sensor-specific default. For the JPA sensor for example this is 'targetType'- 'primaryKeyId'. The default group id can be overwritten by setting it in the request scope context like this:

```
Context.requestScope.setGroupId("myGroupId");
```

The ArchiveLoader provides a method

```
List<Archive> ArchiveLoader.loadAllArchivesByGroupId(  
    String groupId)
```

to search for archives by their group id.

Configuring and checking archive integrity is post- control functionality and is described in chapter Checking Archive Integrity.

The main significance of the ARCHIVE actuator is that the archived business cases can be redone respective restored. This is described in chapter Searching, Redo and Restore of archived .

## **9.9ENVERS actuator**

**Default name:** ENVERS

**Class:** com.logitags.cibet.actuator.envers.EnversActuator

**Requirements:**

This actuator needs dependency cibet-envers instead of cibet-core:

```
<dependency>  
    <groupId>com.logitags</groupId>  
    <artifactId>cibet-envers</artifactId>  
    <version>${version}</version>  
</dependency>
```

The envers library must be in the classpath:

```
<dependency>  
    <groupId>org.hibernate</groupId>  
    <artifactId>hibernate-envers</artifactId>  
    <version>${version}</version>  
</dependency>
```

In persistence.xml set property hibernate.listeners.envers.autoRegister to false in the persistence unit that is controlled by Cibet JPA sensor:

```
<property name="hibernate.listeners.envers.autoRegister" value="false"/>
```

## Properties:

## Description:

The ENVERS actuator integrates Hibernate Envers (<https://docs.jboss.org/hibernate/core/4.2/devguide/en-US/html/ch15.html#d5e4338>). This Hibernate library allows auditing and historical versioning of an application's entity data. The add-on of the integration with Cibet is that conditional auditing as described in chapter 15.8 of the Envers documentation can be easily realized in the standard Cibet way by defining setpoints. As Envers audits only JPA entities that are stored with Hibernate persistence provider, the ENVERS actuator is only applicable in the following conditions:

- Hibernate persistence provider is defined as property 'com.logitags.cibet.persistence.provider' in the persistence unit
- The setpoint target is a JPA entity
- The setpoint event is one of the PERSISTENT events

Example: In order to audit Customers only when they are deleted in a batch process you may define a setpoint like this one:

```
<setpoint id="envers-2">
  <controls>
    <event>DELETE</event>
    <target>com.company.Customer</target>
    <invoker>com.company.services.BatchService.delete()</invoker>
  </controls>
  <actuator name="ENVERS" />
</setpoint>
```

## 9.10 SPRING\_SECURITY actuator

**Default name:** SPRING\_SECURITY

**Class:** com.logitags.cibet.actuator.springsecurity.SpringSecurityActuator

## Requirements:

This actuator needs dependency cibet-springsecurity instead of cibet-core:

```
<dependency>
  <groupId>com.logitags</groupId>
  <artifactId>cibet-springsecurity</artifactId>
  <version>${version}</version>
</dependency>
```

The Spring Security libraries and its dependent libraries must be on the classpath:

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-core</artifactId>
  <version>${version}</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>${version}</version>
</dependency>
```

In your Spring config file add the following line:

```
<context:component-scan base-package="com.logitags.cibet"/>
```

### Properties:

Property	Data type	Default	Description (see Spring Security documentation)
denyAll	boolean	false	deny all
permitAll	boolean	false	permit all
postAuthorize	String	null	define post authorize rule
postFilter	String	null	define post filter rule
preAuthorize	String	null	define a pre authorize rule
preFilter	String	null	define a pre filter rule
rolesAllowed	String	null	define allowed roles
secured	String	null	define allowed roles
throwDeniedException	boolean	false	throw an Exception when this actuator is applied and access is denied
urlAccess	String	null	allowed roles or expression for accessing an URL. Corresponds to the <intercept-url> element in the Spring configuration file
secondPrincipal	boolean	false	Authorization applied on second principal if true

### Description:

This actuator integrates Spring Security authorization <http://static.springsource.org/spring-security/site/index.html>. It allows defining access control rules for persistence actions, method invocations or http requests on URLs. Users and groups are managed by Spring Security as well as the evaluation of permissions and access rules. In a standalone Spring Security application the permissions are configured in the Spring config file or with annotations in the classes. With Cibet, permissions are defined in Cibet setpoints. The integration with Cibet allows a much more fine grained permission definition and a lot more possibilities. For example you can allow a dual control method invocation to one group of users and the release to another group. Or you may grant the update of a payment object to a user only when the



amount does not change. Please have a look here for more information about the integration of security frameworks like Spring Security into Ciber (<http://www.logitags.com/ciber/springsecurity.html>).

In order to use this actuator for method authorization, enable annotation-based security in your application with the `<global-method-security>` element in the Spring configuration. Read in Spring Security documentation (<http://static.springsource.org/spring-security/site/docs/3.1.x/reference/springsecurity.html>) about using the `<global-method-security>` element. With the `<global-method-security>` element access control can be defined by one of three kinds of annotations:

1. Spring Security native annotation `@Secured`
2. JSR250 annotations `@DenyAll`, `@PermitAll`, `@RolesAllowed`
3. Expression-based annotations `@PreAuthorize`, `@PreFilter`, `@PostAuthorize`, `@PostFilter`

In a Ciber controlled application these annotations can be used too but in order to make use of the enhanced control mechanisms of Ciber the access control configuration must be done in the ciber configuration file or in code. The configuration elements have the same names as the annotations and are parameterized in the same way as in Spring Security. Some examples:

XML:

```
<actuator name="AllPermitter">
  <class>
    com.logitags.ciber.actuator.springsecurity.SpringSecurityActuator
  </class>
  <properties>
    <permitAll/>
    <!--
    <preAuthorize>hasRole( 'WALTER')</preAuthorize>
    <secured>ROLE_BaseUsers</secured>
    <preAuthorize>
      hasPermission(#contact, 'admin') and hasRole(ROLE_ADMIN)
    </preAuthorize>
    <urlAccess>
      IS_AUTHENTICATED_ANONYMOUSLY
    </urlAccess>
    Throw Exception when permission denied:
    <throwDeniedException><throwDeniedException>
    This actuators permissions apply to the second logged in principal:
    <secondPrincipal />
    -->
  </properties>
</actuator>
```

Code:

```
SpringSecurityActuator act = new SpringSecurityActuator("AllPermitter");
act.setPermitAll(true);
// act.setPreAuthorize("hasRole( 'WALTER')");
// act.setSecured("ROLE_BaseUsers");
// act.setPreAuthorize("hasPermission(#contact, 'admin') and
//                      hasRole(ROLE_ADMIN)");
// act.setUrlAccess("IS_AUTHENTICATED_ANONYMOUSLY");
// act.setThrowDeniedException(true);
// act.setSecondPrincipal(true);
Configuration.instance().registerActuator(act);
```

Please note that the Ciber principle of being tolerant against user input is applied also here: The properties accept any kind of ' or " or even without any apostrophe and any empty spaces, Ciber corrects everything automatically.

The urlAccess property is for defining rules for accessing a URL. It corresponds to the

```
<intercept-url pattern="/**" access="ROLE_USER" />
```

elements in a Spring Security configuration file. In order to define URL access rules in the urlAccess property set in the Spring Security configuration file at least:

```
<sec:http />
```

If you want to use expressions in the urlAccess property like

```
<urlAccess>hasRole('ROLE_USER')</urlAccess>
```

set in the Spring Security configuration file

```
<sec:http use-expressions="true"/>
```

instead. Please refer to the Spring Security documentation (<http://static.springsource.org/spring-security/site/docs/3.1.x/reference/springsecurity.html>) for any details on how to configure access rules.

When the authorization should be applied on the second user in the release of a TWO\_MAN\_RULE controlled event the property secondPrincipal must be set to true. In this case the authorization rules are not applied on the Authorization object of the logged in user which is stored in SecurityContextHolder.context but on the Authorization object stored in Context.sessionScope().

When a Spring Security actuator is applied in a setpoint the persistence action or service call will be granted or denied. If a business case is denied can be checked by querying the EventResult object returned by the method

```
EventResult result = Context.requestScope().getExecutedEventResult();
```

The status property of EventResult will be DENIED, otherwise EXECUTED (see also chapter Post- Checking Control Results).

It is also possible to let Ciber throw an Exception when the business case is denied. For this the throwDeniedException attribute must be set to true in the actuator, either in the Ciber config file or in code (see also chapter Actuator configuration).

XML:

```
<actuator name="SPRING_SECURITY">
  <properties>
    <throwDeniedException>true</throwDeniedException>
```

```

    </properties>
</actuator>

```

Code:

```

SpringSecurityActuator ssa = (SpringSecurityActuator)
    Configuration.instance().getActuator("SPRING_SECURITY");
ssa.setThrowDeniedException(true);

```

If a business case is denied a `DeniedException` will then be thrown:

```

try {
    entityManager.insert(transaction);
} catch (DeniedException e) {
    ...
}

```

For http requests which are secured with `SpringSecurityActuator` the behavior is different. In http protocol it is not possible to transmit an exception. The request result is communicated by response codes. Therefore, the value of the `throwDeniedException` property has normally no effect in intercepted http requests. Instead, a response code 403 (FORBIDDEN) is returned if the access is denied.

## 9.11 SHIRO actuator

**Default name:** SHIRO

**Class:** `com.logitags.cibet.actuator.shiro.ShiroActuator`

**Requirements:**

This actuator needs dependency `cibet-shiro` instead of `cibet-core`:

```

<dependency>
    <groupId>com.logitags</groupId>
    <artifactId>cibet-shiro</artifactId>
    <version>${version}</version>
</dependency>

```

The Apache Shiro libraries and its dependent libraries must be on the classpath, at least the core library:

```

<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-core</artifactId>
    <version>${version}</version>
</dependency>

```

**Properties:**

Property	Data type	Default	Description (see Apache Shiro documentation)
hasAllRoles	String		authorized if the Subject is assigned all of the specified comma/semicolon separated roles
isPermittedAll	String		authorized if the Subject is permitted all of the specified semicolon separated permissions
requiresAuthentication	Boolean	false	requires the current Subject to have been authenticated during their current session
requiresGuest	Boolean	false	requires the current Subject to be a "guest", that is, they are not authenticated or remembered from a previous session
requiresUser	Boolean	false	requires the current Subject to be an application user, a Subject either known due to being authenticated during the current session or remembered from a previous session
throwDeniedException	boolean	false	throw an Exception when this actuator is applied and access is denied
secondPrincipal	boolean	false	Authorization applied on second Subject if true

### Description:

This actuator integrates Apache Shiro authorization (<http://shiro.apache.org/>). It allows defining access control rules for persistence actions, method invocations or http requests on URLs. Users and groups are managed by Shiro as well as the evaluation of permissions and access rules. In a standalone Shiro application the permissions are configured programmatically or with annotations in the classes. With Cibet, permissions are defined in Cibet setpoints. The integration with Cibet allows a much more fine grained and flexible permission definition and a lot more possibilities. For example you can allow a dual control method invocation to one group of users and the release to another group. Or you may grant the update of a payment object to a user only when the amount does not change. Please have a look here for more information about the integration of security frameworks like Shiro into Cibet (<http://www.logitags.com/cibet/springsecurity.html>).

Performing authorization in Shiro can be done in 3 ways: Programmatically, by annotations and by JSP/GSP tags.

In a Cibet controlled application there is a fourth possibility: In the Cibet configuration file. With Cibet integration a much more flexible distribution of permissions is possible. See in <http://www.logitags.com/cibet/springsecurity.html> for some examples what can be done with

Shiro - Cibat integration that cannot be done by standalone Shiro. In order to configure Shiro with Cibat an instance of ShiroActuator must be configured. The properties of ShiroActuator have the same names as the equivalent methods in Shiro's Subject class. Some examples:

XML:

```
<actuator name="AllPermitter">
  <class>
    com.logitags.cibat.actuator.shiro.ShiroActuator
  </class>
  <properties>
    <requiresGuest/>
    <!--
      Other permission possibilities:
      <hasAllRoles>SEC_ROLE; USER_ROLE; ADV_ROLE</hasAllRoles>
      <requiresUser>true</requiresUser>
      <isPermittedAll>
        lightsaber:* ;
        jaeger:schiess:*
      </isPermittedAll>
      Throw Exception when permission denied:
      <throwDeniedException><throwDeniedException>
      This actuators permissions apply to the second Subject:
      <secondPrincipal />
    -->
  </properties>
</actuator>
```

Code:

```
ShiroActuator act = new ShiroActuator("AllPermitter");
act.setRequiresGuest(true);
// act.setHasAllRoles("SEC_ROLE; USER_ROLE; ADV_ROLE");
// act.setRequiresUser(true);
// act.setIsPermittedAll("lightsaber:*; jaeger:schiess:*");
// act.setThrowDeniedException(true);
// act.setSecondPrincipal(true);
Configuration.instance().registerActuator(act);
```

When the authorization should be applied on the second user in the release of a TWO\_MAN\_RULE controlled event the property secondPrincipal must be set to true. In this case the authorization rules are not applied on the Subject object of the logged in user which is stored in the user session but on the Subject object stored in Cibat's session scope context accessible with getSecondPrincipal().

When a Shiro actuator is applied in a setpoint the persistence action or service call will be granted or denied. If a business case is denied can be checked by querying the EventResult object returned by the method

```
EventResult result = Context.requestScope().getExecutedEventResult();
```

The status property of EventResult will be DENIED, otherwise EXECUTED (see also chapter Post- Checking Control Results).

It is also possible to let Cibat throw an Exception when the business case is denied. For this the throwDeniedException attribute must be set to true in the actuator, either in the Cibat

config file or in code. The property can be set generally for the actuator or for a single setpoint (see chapter Actuator configuration)

XML:

```
<actuator name="SHIRO">
  <properties>
    <throwDeniedException>true</throwDeniedException>
  </properties>
</actuator>
```

Code:

```
ShiroActuator ssa = (ShiroActuator)
    Configuration.instance().getActuator("SHIRO");
ssa.setThrowDeniedException(true);
```

If a business case is denied a `DeniedException` will then be thrown:

```
try {
    entityManager.insert(transaction);
} catch (DeniedException e) {
    ...
}
```

For http requests which are secured with `ShiroActuator` the behavior is different. In http protocol it is not possible to transmit an exception. The request result is communicated by response codes. Therefore, the value of the `throwDeniedException` property has normally no effect in intercepted http requests. Instead, a response code 403 (FORBIDDEN) is returned if the access is denied.

## 9.12 LOCKER actuator

**Default name:** LOCKER

**Class:** com.logitags.cibet.actuator.lock.LockActuator

**Requirements:**

This actuator needs a database. The database scheme can be created with the SQL scripts `<dbms>.sql` included in the release. There exist scripts for Derby, MySQL, PostgreSQL and Oracle database management systems.

**Properties:**

Property	Data type	Default	Description
throwDeniedException	boolean	false	throw an Exception when this actuator is applied and access is denied

automaticUnlock	boolean	false	If true, the lock is set to unlocked after the first successful execution of the locked event
-----------------	---------	-------	---

### Description:

LOCKER actuator allows the temporary locking (reservation) of an event on a resource like persisting an entity, releasing of a dual control event, invocation of a method or requesting a URL. Only the user who sets the lock can execute that event until the lock is removed. This actuator is not a replacement for a sophisticated authorization framework like Spring Security or Apache Shiro. There is no concept of roles and groups. While an authorization framework defines static permissions which are applied to groups or roles, LOCKER actuator allows defining dynamic permissions applied to a user. LOCKER can be applied when a user wants to make a reservation for a resource so that no other user can do an action on it.

First requirement for the locking functionality is that the resource that should be locked is under Cibert control. This is done by a setpoint configuration, for example:

```
<setpoint id="id1">
  <controls>
    <event>UPDATE</event>
    <target>com.app.accounting.Account</target>
  </controls>
  <actuator name="LOCKER" />
</setpoint>
```

to set updates of Account entities under lock control. The LOCKER actuator checks now on each update of an Account object if it is locked by a user. When another user but the user who has set the lock tries to make an update of the account object, the action will be denied. If a business case is denied can be checked by querying the execute status in request scope context (see chapter Post- Checking Control Results) or by catching DeniedException if the throwDeniedException flag is set to true. Catching the DeniedException works the same as for SPRING\_SECURITY actuator and SHIRO actuator and is described there.

LOCKER actuator checks only if a lock on a business case exists, the locking and unlocking itself is pre- respective post control functionality and is described in chapter Locking Business Cases.

If the property automaticUnlock is set to true the lock will be removed immediately after the first successful execution of the locked event. This would be normally the case when the user who has set the lock or has made the reservation executes the business case.

## 9.13 LOADCONTROL actuator

**Default name:** LOADCONTROL

**Class:** com.logitags.cibet.actuator.loadcontrol.LoadControlActuator

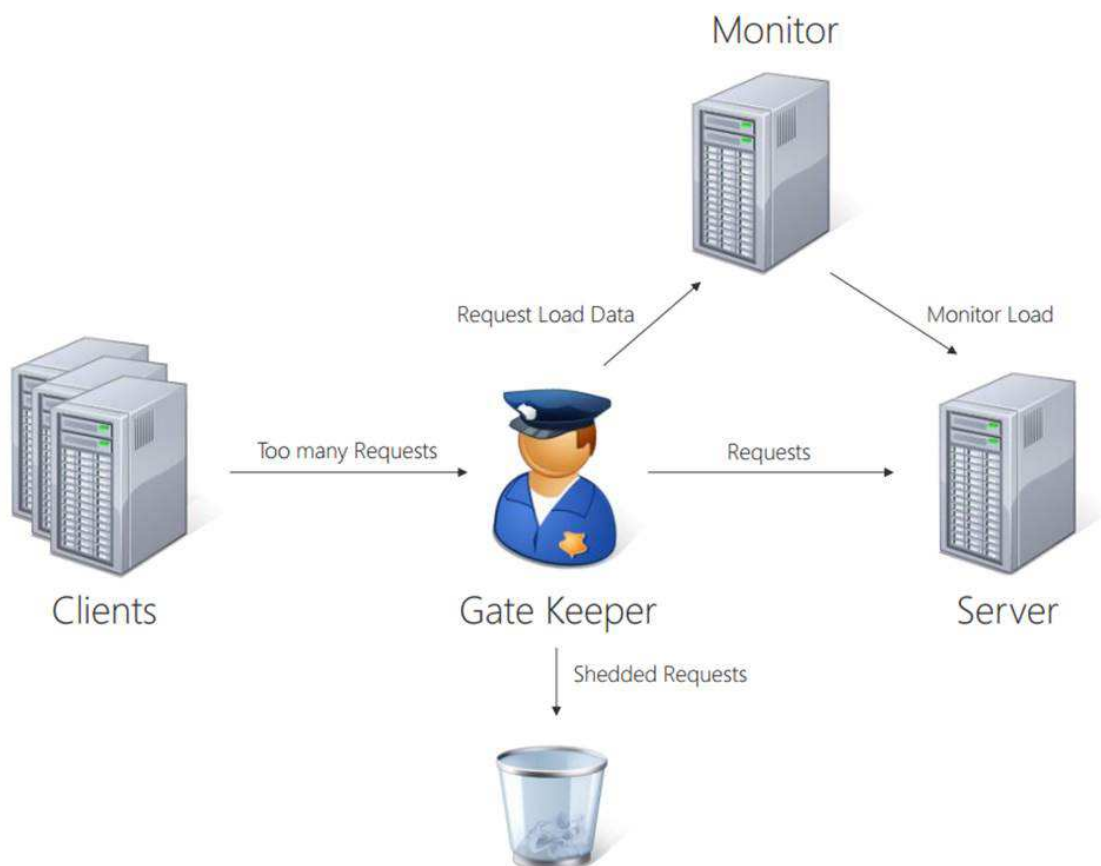
## Requirements:

## Properties:

Property	Data type	Default	Description
customMonitors	Monitor[]		List of custom Monitor implementations
loadControlCallback	LoadControl Callback		Implementation of LoadControl Callback which will be notified on load control events
+ monitor specific properties			See below

## Description:

The original intention of this actuator is to implement the shed load pattern which controls load by shedding requests when load is too high:



**Figure 10: Shed Load pattern**

Naturally, this actuator is best applied in server-side sensors like EJB, HTTP\_FILTER and ASPECT. Besides shedding requests the LOADCONTROL actuator provides three other modes for controlling load:



- Monitoring: in this mode the load is merely monitored without any impact on the system.
- Alarming: a callback class is notified when the load exceeds an alarm threshold.
- Valve Throttling: if the load exceeds a valve threshold, the request is held back and throttled until the load decreases below the threshold. If the load remains high, the request is eventually shed after a certain time.
- Shedding: if the load exceeds a shed threshold, the request is shed

If an implementation of interface

`com.logitags.cibet.actuator.loadcontrol.LoadControlCallback` is configured with the actuator it will be notified when requests are throttled or shed. This allows implementing custom logic to react on specific high load situations. A callback is configured in property `loadControlCallback` of `LoadControlActuator`.

The valve functionality is controlled by two parameters, `throttleMaxTime` and `throttleInterval`. The first parameter determines the maximum time in ms that a request is throttled before it is eventually shed when the load remains above the valve threshold. The second parameter is the interval in which `LoadControlActuator` checks if the load has decreased under the valve threshold. A value of 1000 ms and 200 ms for `throttleMaxTime` and `throttleInterval` means for example that during a period of 1 sec the load is determined at maximum 5 times. If the load is under the threshold within 1000 ms the request is passed through, otherwise it is shed.

Precondition to react on heavy load situations is a means to measure the load. Load on a system, a method or a service can be monitored in manifold ways. The `LOADCONTROL` actuator provides several alternatives which are realized as implementations of the `com.logitags.cibet.actuator.loadcontrol.Monitor` interface. It is also possible to create own implementations of this interface and configure them in the `customMonitors` property of the `LoadControlActuator`. Load can be checked either in code or with JMX. If for example a setpoint has been configured to control load of all methods in class `MyTestClass` that start with 'execute':

```
<setpoint id="SP2-javaMethod">
    <controls>
        <target>com.appl.MyTestClass</target>
        <method>execute*</method>
    </controls>
    <actuator name="LOADCONTROL"/>
</setpoint>
```

the average thread user time can be retrieved like this:

```
LoadControlActuator actuator = (LoadControlActuator)
    Configuration.instance().getActuator(LoadControlActuator.DEFAULTNAME);
int userTime = actuator.getThreadTimeMonitor()
    .getAverageThreadUserTime("SP2-javaMethod");
```

In a JMX console the values are displayed like this:

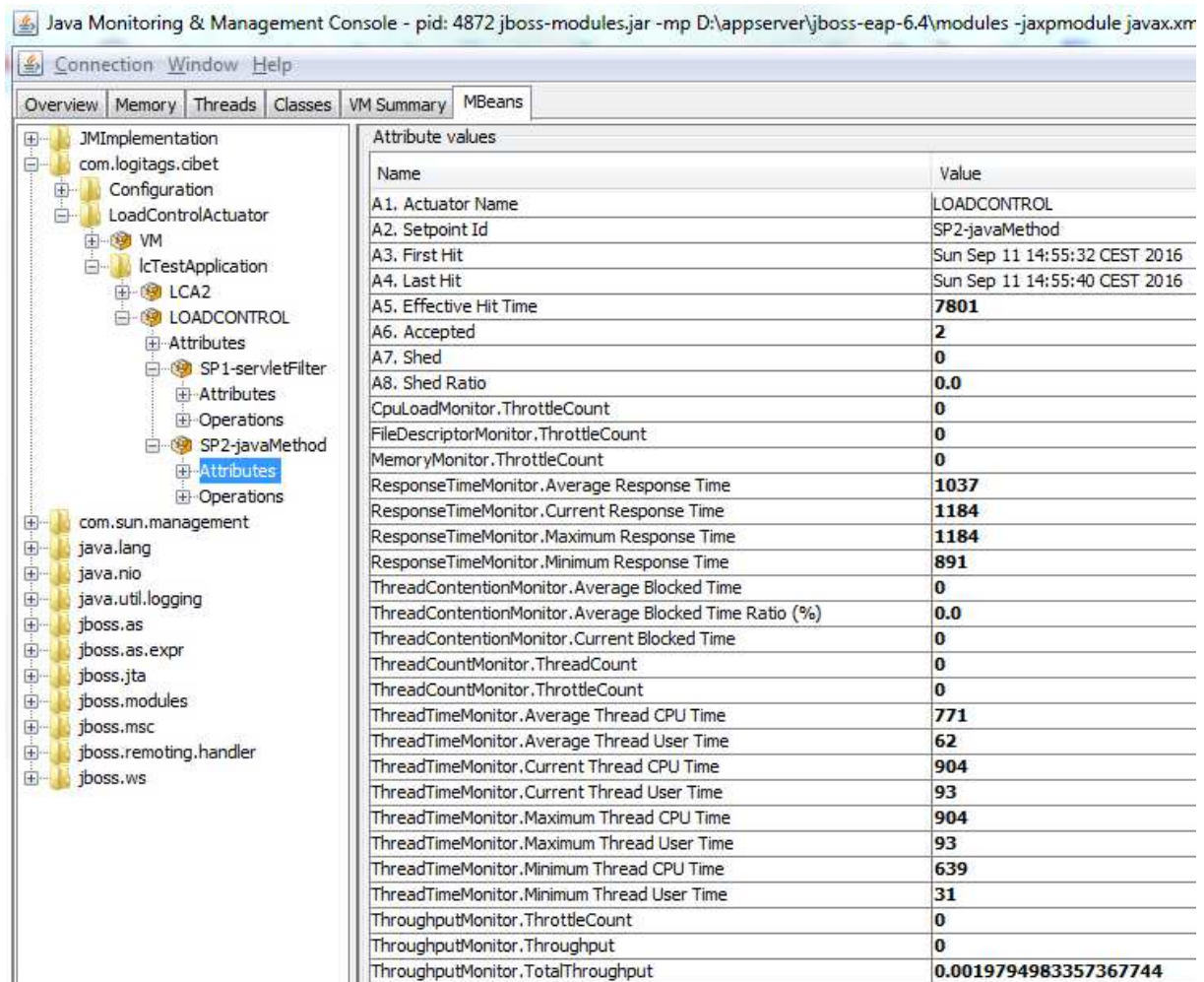


Figure 11: Monitored values in a JMX console

The JMX console displays the LoadControl monitored values and configuration parameters under the object name `com.logitags.cibet/LoadControlActuator`. The VM JMX bean shows monitored values of the virtual machine (Figure 13). Additionally, each application has its own set of JMX beans. In the example of Figure 11 there is one application monitored with name `lcTestApplication`. In this application two `LoadControlActuator`s with different configurations are deployed, `LCA2` and the default `LoadControlActuator`s with default name `LOADCONTROL`. The configuration parameters can be found in the `Attributes` node located directly under the `LOADCONTROL` node (see Figure 12). The default actuator is applied in two setpoints with names `SP1-servletFilter` and `SP2-javaMethod`. The corresponding `cibet-config.xml` may look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<cibet xmlns=http://www.logitags.com
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:schemaLocation="http://www.logitags.com
  http://www.logitags.com/cibet/cibet-config_1.3.xsd">

  <actuator name="LOADCONTROL">
    <properties>
      <threadCountMonitor.status>ON</threadCountMonitor.status>
      <threadCountMonitor.shedThreshold>15
        </threadCountMonitor.shedThreshold>
      <threadTimeMonitor.status>ON</threadTimeMonitor.status>
```

```

        <responseTimeMonitor.status>ON</responseTimeMonitor.status>
        <threadContentionMonitor.status>ON</threadContentionMonitor.status>
        <throughputMonitor.status>ON</throughputMonitor.status>
    </properties>
</actuator>

<actuator name="LCA2">
    <class>
        com.logitags.cibet.actuator.loadcontrol.LoadControlActuator
    </class>
    <properties>
        <threadCountMonitor.status>ON</threadCountMonitor.status>
        <threadTimeMonitor.status>ON</threadTimeMonitor.status>
        <responseTimeMonitor.status>ON</responseTimeMonitor.status>
        <threadContentionMonitor.status>ON</threadContentionMonitor.status>
    </properties>
</actuator>

<setpoint id="SP1-servletFilter">
    <controls>
        <target>http://localhost:8788/*</target>
    </controls>
    <actuator name="LOADCONTROL" />
</setpoint>

<setpoint id="SP2-javaMethod">
    <controls>
        <target>com.logitags.cibet.jmeter.MonitorTestClass</target>
        <method>cibet*</method>
    </controls>
    <actuator name="LOADCONTROL" />
</setpoint>

<setpoint id="SP3-ejb">
    <controls>
        <target>com.logitags.cibet.jmeter.MonitorEjb</target>
        <method>cibet*</method>
    </controls>
    <actuator name="LCA2" />
</setpoint>

<setpoint id="SP4-jpa">
    <controls>
        <!--target>SELECT a FROM JMEntity a*</target-->
        <target>com.logitags.cibet.jmeter.JMEntity.SEL</target>
    </controls>
    <actuator name="LCA2" />
</setpoint>
</cibet>

```

Each monitor has parameters to configure monitoring, alarm, valve and shed. The parameter 'status' can be OFF or ON. In status OFF monitoring is switched off and the monitor has no impact on the application. If in status ON monitoring is active and the parameters for alarm, valve and shed are taken into account. The status can also be NOT\_SUPPORTED. Some JAVA VMs and operating systems do not support all monitors. This is detected automatically. The configuration parameters can be set as for other actuators in code, in cibet-config.xml or in a JMX console.

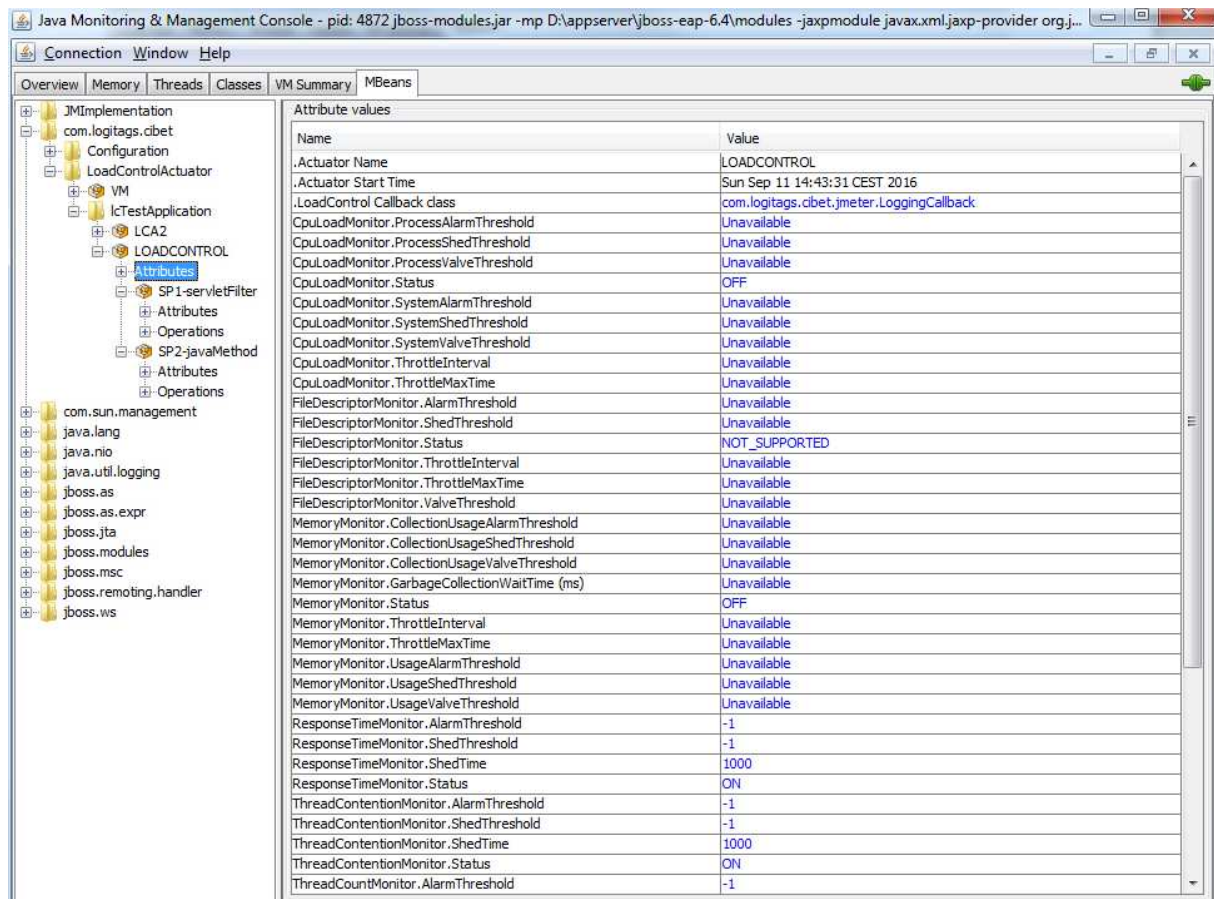


Figure 12: LoadControlActuator configuration parameters in a JMX console

The built-in monitors are the following:

## ThreadTime Monitor

monitors CPU time and user time of the current thread. The CPU time is the total time spent using a CPU for the thread's execution while the user time is the time spent running the threads's own code without the time spent running OS code on behalf of the thread (such as for I/O). The following monitored properties can be retrieved:

- Current thread CPU time: the current thread cpu time in ms that a thread has needed for execution
- Average thread CPU time: the average thread cpu time in ms that a thread has needed for execution
- Minimum thread CPU time: the minimum thread cpu time in ms that a thread has needed for execution
- Maximum thread CPU time: the maximum thread cpu time in ms that a thread has needed for execution
- Current thread user time: the current thread user time in ms that a thread has needed for execution
- Average thread user time: the average thread user time in ms that a thread has needed for execution
- Minimum thread user time: the minimum thread user time in ms that a thread has needed for execution

- Maximum thread user time: the maximum thread user time in ms that a thread has needed for execution

Thread time is not a measure for load but gives information about performance and specific execution times. Therefore the valve and shed modes are not implemented for the thread time monitor.

Property	Data type	Default	Description
threadTimeMonitor.status	enum		ON, OFF
threadTimeMonitor.alarm CpuTimeThreshold	numeric	-1	CPU time threshold for raising an alarm. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the average CPU time of this setpoint, otherwise it is taken as absolute in ms
threadTimeMonitor.alarm UserTimeThreshold	numeric	-1	User time threshold for raising an alarm. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the average user time of this setpoint, otherwise it is taken as absolute in ms

## ResponseTime Monitor

monitors the real-world elapsed time experienced by a user waiting for a task to complete. The following monitored properties can be retrieved:

- Current response time: the response time in ms of the current thread
- Average response time: the average response time in ms
- Minimum response time: the minimum response time in ms
- Maximum response time: the maximum response time in ms

Property	Data type	Default	Description
responseTimeMonitor. status	enum		ON, OFF
responseTimeMonitor. alarmThreshold	numeric	-1	Response time threshold for raising an alarm. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the average response time of this setpoint, otherwise it is taken as absolute in ms
responseTimeMonitor.	numeric	-1	Response time threshold for

shedThreshold			shedding. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the average response time of this setpoint, otherwise it is taken as absolute in ms
responseTimeMonitor. shedTime	numeric	1000	time span in ms in which requests are shed after the threshold is exceeded. If the shedThreshold is exceeded, following requests are shed for shedTime ms. If the value ends with % it is interpreted as percentage of the average response time of this setpoint, otherwise it is taken as absolute in ms

### Throughput Monitor

monitors the throughput in requests per time unit. The following monitored properties can be retrieved:

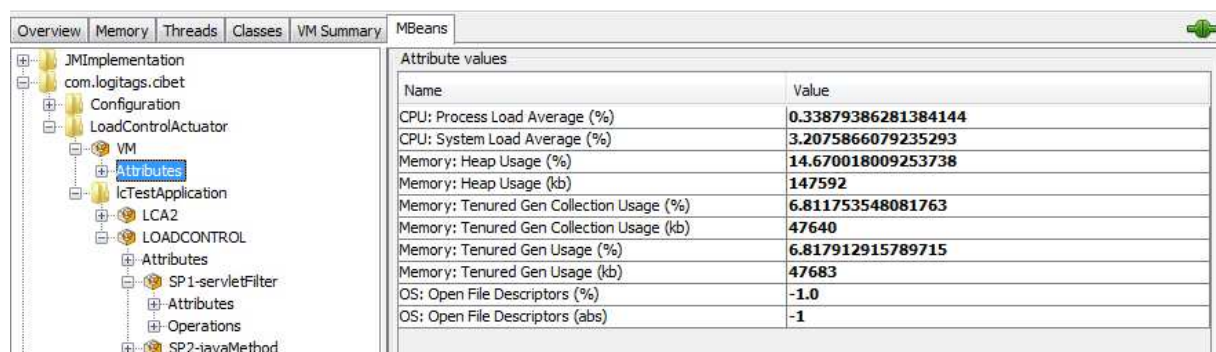
- Throughput per sliding window: the current throughput per time unit in a sliding window
- Total throughput per elapsed time: Throughput as accepted requests per elapsed time since application start in s

Property	Data type	Default	Description
throughputMonitor.status	enum		ON, OFF
throughputMonitor. alarmThreshold	numeric	-1	throughput threshold for raising an alarm. A value of -1 means no threshold. The threshold is given as number of requests in the given windowWidth period.
throughputMonitor. valveThreshold	numeric	-1	throughput threshold for throttling a request. A value of -1 means no threshold. The threshold is given as number of requests in the given windowWidth period.
throughputMonitor. shedThreshold	numeric	-1	throughput threshold for shedding a request. A value of -1 means no threshold. The threshold is given as number of requests in the given windowWidth period.
throughputMonitor.	numeric	1000	The width of the throughput

windowWidth			measuring window in ms. A value of 1000 means the throughput is measured in requests/1000 ms
throughputMonitor.throttleInterval	numeric	200	Interval in ms in which the load is checked when a request is throttled by a valve. A value of 200 means the load is checked every 200 ms beginning from the throttling. If the load decreases under the valveThreshold, the request is passed through.
throughputMonitor.throttleMaxTime	numeric	1000	Maximum time in ms that a request is throttled before it is eventually shed. A value of 1000 and a throttleInterval of 200 mean the load is checked 1000/200 times. If the load is then still above the valveThreshold the request is shed.

## Memory Monitor

monitors the memory usage of the system. Memory usage is not measured for a single thread or the controlled application but for the complete virtual machine. Therefore memory usage data are displayed in a VM JMX bean:



Name	Value
CPU: Process Load Average (%)	0.33879386281384144
CPU: System Load Average (%)	3.2075866079235293
Memory: Heap Usage (%)	14.670018009253738
Memory: Heap Usage (kb)	147592
Memory: Tenured Gen Collection Usage (%)	6.811753548081763
Memory: Tenured Gen Collection Usage (kb)	47640
Memory: Tenured Gen Usage (%)	6.817912915789715
Memory: Tenured Gen Usage (kb)	47683
OS: Open File Descriptors (%)	-1.0
OS: Open File Descriptors (abs)	-1

**Figure 13: Virtual machine monitored values in the VM JMX bean**

The following monitored properties can be retrieved:

- Heap memory usage (absolute and relative)
- Tenured generation usage (absolute and relative): memory usage of the tenured or old generation pool. This memory pool consists of long living objects. If an object survives some number of minor garbage collections it will be transferred to the tenured generation pool. Eventually the old generation needs to be collected by a major garbage collection.

- Tenured generation collection usage (absolute and relative): After the Java virtual machine has expended effort in reclaiming memory space by recycling unused objects at garbage collection time, some number of bytes that are garbage collected will still be in use which is called the collection usage.

Property	Data type	Default	Description
memoryMonitor.status	enum		ON, OFF
memoryMonitor.collectionUsageAlarmThreshold	numeric	-1	memory threshold on the collection usage for raising an alarm. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the maximum amount of memory that can be used for memory management, otherwise it is taken as absolute in bytes
memoryMonitor.collectionUsageShedThreshold	numeric	-1	memory threshold on the collection usage for shedding. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the maximum amount of memory that can be used for memory management, otherwise it is taken as absolute in bytes
memoryMonitor.collectionUsageValveThreshold	numeric	-1	memory threshold on the collection usage for throttling by a valve. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the maximum amount of memory that can be used for memory management, otherwise it is taken as absolute in bytes
memoryMonitor.usageAlarmThreshold	numeric	-1	memory usage threshold for raising an alarm. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the maximum amount of memory that can be used for memory management, otherwise it is taken as absolute in bytes
memoryMonitor.usageShedThreshold	numeric	-1	memory usage threshold for shedding. A value of -1 means no threshold. If the value ends



			with % it is interpreted as percentage of the maximum amount of memory that can be used for memory management, otherwise it is taken as absolute in bytes
memoryMonitor. usageValveThreshold	numeric	-1	memory usage threshold for throttling by a valve. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the maximum amount of memory that can be used for memory management, otherwise it is taken as absolute in bytes
memoryMonitor.garbage CollectionWaitTime	numeric	3000	Maximum time in ms after which a garbage collection is forced to execute if requests keep getting shed
memoryMonitor. throttleInterval	numeric	200	Interval in ms in which the load is checked when a request is throttled by a valve. A value of 200 means the load is checked every 200 ms beginning from the throttling. If the load decreases under the valveThreshold, the request is passed through.
memoryMonitor. throttleMaxTime	numeric	1000	Maximum time in ms that a request is throttled before it is eventually shed. A value of 1000 and a throttleInterval of 200 mean the load is checked 1000/200 times. If the load is then still above the valveThreshold the request is shed.

### Thread Count Monitor

monitors the total number of parallel threads that are executing the same observed http end point, Java method or service.

Property	Data type	Default	Description
threadCountMonitor. status	enum		ON, OFF
threadCountMonitor. alarmThreshold	numeric	-1	Thread count threshold for raising an alarm. A value of -1

			means no threshold.
threadCountMonitor. valveThreshold	numeric	-1	Thread count threshold for throttling a request. A value of -1 means no threshold.
threadCountMonitor. shedThreshold	numeric	-1	Thread count threshold for shedding a request. A value of -1 means no threshold.
threadCountMonitor. throttleInterval	numeric	200	Interval in ms in which the load is checked when a request is throttled by a valve. A value of 200 means the load is checked every 200 ms beginning from the throttling. If the load decreases under the valveThreshold, the request is passed through.
threadCountMonitor. throttleMaxTime	numeric	1000	Maximum time in ms that a request is throttled before it is eventually shed. A value of 1000 and a throttleInterval of 200 mean the load is checked 1000/200 times. If the load is then still above the valveThreshold the request is shed.

## CpuLoad Monitor

monitors the cpu load in percentage. CPU load is not measured for a single thread or the controlled application but for the complete virtual machine or operating system. Therefore load data are displayed in the VM JMX bean (see image above). The following monitored properties can be retrieved:

- System CPU load: the "recent cpu usage" for the whole system. A value of 100% means that all CPUs were actively running 100% of the time during the recent period being observed.
- Process CPU load: the "recent cpu usage" for the Java Virtual Machine process. A value of 100% means that all CPUs were actively running threads from the JVM 100% of the time during the recent period being observed. Threads from the JVM include the application threads as well as the JVM internal threads.

Property	Data type	Default	Description
cpuLoadMonitor. status	enum		ON, OFF
cpuLoadMonitor. processAlarmThreshold	numeric	-1	CPU process threshold for raising an alarm in percent. A value of -1 means no threshold.
cpuLoadMonitor.	numeric	-1	CPU process threshold for

processValveThreshold			throttling a request in percent. A value of -1 means no threshold.
cpuLoadMonitor. processShedThreshold	numeric	-1	CPU process threshold for shedding a request in percent. A value of -1 means no threshold.
cpuLoadMonitor. systemAlarmThreshold	numeric	-1	CPU system threshold for raising an alarm in percent. A value of -1 means no threshold.
cpuLoadMonitor. systemValveThreshold	numeric	-1	CPU system threshold for throttling a request in percent. A value of -1 means no threshold.
cpuLoadMonitor. systemShedThreshold	numeric	-1	CPU system threshold for shedding a request in percent. A value of -1 means no threshold.
cpuLoadMonitor. throttleInterval	numeric	200	Interval in ms in which the load is checked when a request is throttled by a valve. A value of 200 means the load is checked every 200 ms beginning from the throttling. If the load decreases under the valveThreshold, the request is passed through.
cpuLoadMonitor. throttleMaxTime	numeric	1000	Maximum time in ms that a request is throttled before it is eventually shed. A value of 1000 and a throttleInterval of 200 mean the load is checked 1000/200 times. If the load is then still above the valveThreshold the request is shed.

### Thread Contention Monitor

monitors the accumulated elapsed time that the controlled process has blocked for synchronization or waited for notification. A contention occurs when a thread is waiting for a resource that is not readily available; it slows the execution of the code and is therefore a measure for heavy load situations. The following monitored properties can be retrieved:

- Current thread blocked time: elapsed time that the controlled process, method or service was in the blocked state
- Average thread blocked time: average time that a thread was in the blocked state in the controlled process, method or service
- Blocked time/response time ratio: ratio of current blocked time to response time

<b>Property</b>	<b>Data type</b>	<b>Default</b>	<b>Description</b>
threadContentionMonitor.status	enum		ON, OFF
threadContentionMonitor.alarmThreshold	numeric	-1	Thread contention time threshold for raising an alarm. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the thread contention time, otherwise it is taken as absolute in ms
threadContentionMonitor.shedThreshold	numeric	-1	Thread contention time threshold for shedding. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the thread contention time, otherwise it is taken as absolute in ms
threadContentionMonitor.shedTime	numeric	1000	time span in ms in which requests are shed after the threshold is exceeded. If the shedThreshold is exceeded, following requests are shed for shedTime ms. If the value ends with % it is interpreted as percentage of the average thread contention time of this setpoint, otherwise it is taken as absolute in ms

## File Descriptor Monitor

monitors the number of open file descriptors. A file descriptor is an abstract indicator used to access a file or other input/output resource, such as a pipe or network socket. The number of open file descriptors can therefore be seen as a measure for load in terms of usage of i/o and system resources. Open file descriptors are displayed absolute or as percentage of the maximum file descriptors.

<b>Property</b>	<b>Data type</b>	<b>Default</b>	<b>Description</b>
fileDescriptorMonitor.status	enum		ON, OFF
fileDescriptorMonitor.alarmThreshold	numeric	-1	File descriptor threshold for raising an alarm. A value of -1 means no threshold. If the value ends with % it is interpreted as

			percentage of the maximum available file descriptors, otherwise it is taken as absolute value.
fileDescriptorMonitor.shedThreshold	numeric	-1	File descriptor threshold for shedding. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the maximum available file descriptors, otherwise it is taken as absolute value.
fileDescriptorMonitor.valveThreshold	numeric	-1	File descriptor threshold for throttling by a valve. A value of -1 means no threshold. If the value ends with % it is interpreted as percentage of the maximum available file descriptors, otherwise it is taken as absolute value.
fileDescriptorMonitor.throttleInterval	numeric	200	Interval in ms in which the load is checked when a request is throttled by a valve. A value of 200 means the load is checked every 200 ms beginning from the throttling. If the load decreases under the valveThreshold, the request is passed through.
fileDescriptorMonitor.throttleMaxTime	numeric	1000	Maximum time in ms that a request is throttled before it is eventually shed. A value of 1000 and a throttleInterval of 200 mean the load is checked 1000/200 times. If the load is then still above the valveThreshold the request is shed.

## 9.14 Implementing Own Actuators

Actuators are pluggable and it is possible to extend actuators or to define new ones with other business logic. Actuator logic is implemented in classes that implement interface `com.logitags.cibet.actuator.Actuator`. Actuator implementations can be declared in `cibet-config.xml` configuration file or dynamically registered with Configuration API.

In `cibet-config.xml` add an actuator element and set the class tag. If the actuator implementation uses own properties, they can be defined with property tags. The

implementation must have a setter method with a name following the Java Beans convention and taking a String argument as parameter:

XML:

```
<actuator name="Sub4Eyes">
  <class>com.company.MyActuator</class>
  <properties>
    <myAttribute>someValue</myAttribute>
  </properties>
</actuator>
```

Code:

```
Actuator act = new MyActuator();
act.setMyAttribute("someValue");
Configuration.instance().registerActuator(act);
```

Actuator implementations must have a default constructor and must provide a unique name which is returned by method `getName()`. This is the name that is set in the name attribute in `cibet-config.xml`. The other methods define logic which is executed before respective after a control event. It is recommended to inherit custom actuators from `AbstractActuator` and overwrite the methods the actuator is interested in. This ensures that the actuator works also in case of future interface enhancements.

## 10. Pre- and Post- Control Functionality

The following chapters describe Cibat functionality which enhance the basic control patterns described in the actuators chapter. Most of this functionality is optional but some is required in order to make the configured actuators useful.

### 10.1 Post- Checking Control Results

If a business case is controlled by a sensor, it can be summarized after execution of the business case which actuators have been applied and the actuator results. This can be done by the following method:

```
EventResult result = Context.requestScope().getExecutedEventResult();
```

This method returns an object of type `EventResult` or null if no sensor has been applied or the intercepted event is still executing and the final result could not yet be determined.

The `EventResult` object contains the following information:

- **sensor:** The sensor that intercepted the event
- **resource: information about the resource that is controlled, for example target class, method, URL, JPA query**
- **caseId:** the case id which is assigned to this business case
- **actuators:** comma separated list of the applied actuators
- **setpoints:** comma separated list of applied setpoint IDs

- **executionTime:** timestamp of interception
- **event:** the intercepted event
- **ExecutionStatus:** result of the event execution. One of EXECUTING, EXECUTED, POSTPONED, REJECTED or DENIED
- **childResults:** a list of child EventResult objects. If during execution of this event a second event is controlled by a setpoint, the result of the second event is added to this list. Empty list, if there is no child controlled event
- **parentResult:** if this event is executed during execution of another parent event controlled by a setpoint, the result of the parent event is stored in this property. Null, if there is no parent controlled event

The last two properties need maybe some more explanation. If for example a service EJB method is controlled and within the implementation of this method the JPA persistence of an entity is controlled, we have a parent – child relationship. If the Context.requestScope().getExecutedResult() method is called after execution of the EJB method it will return an EventResult object representing the execution of the EJB method containing as a child an EventResult object representing the execution of the JPA method. If the Context.requestScope().getExecutedResult() method is called within the EJB service method after execution of the JPA persistence the returned EventResult object represents the JPA persistence object which has no child element in this case but has a parent EventResult representing the EJB method which is in status EXECUTING.

With the HTTP-FILTER sensor, the EventResult object cannot be retrieved from the client Request scope context because the EventResult is produced on the server. It is however transmitted in an encoded format in the http response as header with name CIBET\_EVENTRESULT. It can be retrieved as follows:

```
String evReHeader = response.getFirstHeader(
    CibetFilter.EVENTRESULT_HEADER).getValue();
EventResult result = CibetUtil.decodeEventResult(evReHeader);
```

The EventResults can also be tracked in the database with the TRACKER actuator. If this actuator is applied on a business case, all EventResult objects including the child objects are stored into the database in table cib\_eventresult (see chapter TRACKER actuator).

## 10.2 Pre- Checking Control Results

Sometimes it is desirable to know the expected control results before the business case is actually executed. For example on a web page a button to trigger a business case may be disabled if the current user is not allowed to execute it. It is often a better style to not offer an action instead of telling the user afterwards that he isn't allowed to do it.

However, with Cibet checking the expected results is not simply a matter of looking into the configuration. As Cibet control is not static but highly dynamic the result depends not only on the configuration but may depend on various other context parameters. It is therefore necessary to simulate the business case in the current context in order to check the applied actuators and control results. Cibet provides this functionality with the Play mode. When the system is in Play mode, the business case is not executed, nor are actuators applied or other

impacts on the system are induced. It can be seen however which actuators will be applied and what will be the execution status of the business case.

A business case is executed in Play mode like this:

```
Context.requestScope().startPlay();

// now the business case is simulated:
TEntity entity = persistTEntity();

// and the Play mode is stopped:
EventResult er = Context.requestScope().stopPlay();
```

Stopping the Play mode returns an EventResult object that can be checked like described in previous chapter Post- Checking Control Results.

When using the HTTP-FILTER sensor, the procedure is a little different because it is a remote request which runs not in the same thread. When sending an HTTP request the Play mode is started by adding the header CIBET\_PLAYING\_MODE with a value 'true':

```
// add CIBET_PLAYING_MODE header
postMethod.addHeader(CibetFilter.PLAYINGMODE_HEADER, "true");

// send request
HttpResponse response = client.execute(postMethod);

// now the EventResult object can be retrieved like described in
// previous chapter Post- Checking Control Results
String evReHeader = response.getFirstHeader(
    CibetFilter.EVENTRESULT_HEADER).getValue();
EventResult result = CibetUtil.decodeEventResult(evReHeader);
```

It is not necessary to stop the Play mode after the http request. It is automatically stopped on the server side.

Please be aware that the Play mode has only the desired effect when the executed business case is controlled by a Cibet sensor. Methods and other actions which are not controlled by Cibet will be executed in Play mode!

### ***10.3 Releasing and Rejecting Dual Control Events***

If persistence or method invocation actions are suspended and postponed due to a dual control actuator a second user must check the actions and release or reject them. All suspended actions are represented by an instance of Controllable. Here is how unreleased actions can be found:

```
// find all unreleased Controllables for the tenant set in
// session scope. Returns all unreleased objects if no tenant is
// set in context.
List<Controllable> list = DcLoader.findUnreleased();
```



```
// or be more specific: This method finds all unreleased Controllables
// for the tenant set in session scope and entity BankAccount. Returns
// all unreleased objects of BankAccount if no tenant is set in context.
list = DcLoader.findUnreleased(BankAccount.class);
```

The load...() methods of DcLoader allow loading Controllables regardless of their states. If the controlled resource is a JPA entity and additional properties have been stored with the Controllable object (see property storedProperties of dual control actuators chapter 9.3) one can search after these properties:

```
List<Controllable> list = DcLoader.loadByProperties(
    Class<?> entityClass, Map<String, Object> properties);
```

where the properties map key is the name of the property and the map value is the search value. If for example an actuator and setpoint are configured like this:

```
<actuator name="my 4-Eyes for customer">
  <class>com.logitags.cibet.actuator.dc.FourEyesActuator</class>
  <properties>
    <storedProperties>customerName, country</storedProperties>
  </properties>
</actuator>

<setpoint id="spl">
  <controls>
    <target>com.comp.Customer</target>
    <event>UPDATE,DELETE,INSERT</event>
  </controls>
  <actuator name="my 4-Eyes for customer"/>
</setpoint>
```

to control Customer entities, you will find all stored Controllables with name Becker from Germany with

```
Map<String,Object> props = new HashMap<>();
props.put("customerName", "Becker");
props.put("country", "DE");
List<Controllable> list = DcLoader.loadByProperties(
    Customer.class, props);
```

Of cause you can define any other query on the cib\_controllable table.

Now you can display the data of the Controllables and the controlled Resource object in a GUI to the user who wants to check and release the data. GUI is out of scope of Cibet because every application will have its own technology and layout. Use the getter methods of Controllable to display the data, e.g.:

```
for (Controllable obj : list) {
  // the class name of the entity or the object on which to
  // invoke the method
  String affectedClassName = obj.getResource().getTargetType();

  // the unique ID of Controllable
  long uniqueCOId = obj.getControllableId();

  // the user who initiated the action
```

```

String user = obj.createUser();

// the action on the entity (invoke, insert, update ...)
String event = Obj.getControlEvent();

// get the method name if any
String methodName = (MethodResource)obj.getResource().getMethod();

// get the method or http parameters, http attributes and headers
List<ResourceParameter> parameters = obj.getResource().getParameters();

// get the unique ID of the entity
String id = ((JpaResource)obj.getResource()).getPrimaryKeyId();

// get the persisted object
Object entity = obj.getResource().getUnencodedTargetObject();
}

```

If the controlled event modifies the persistence state of an entity in the database the releasing user wants to compare the modified state with the productive unmodified state in order to see which attributes of the domain object have changed. The DcLoader interface provides functionality for comparing two objects of the same type:

```

// ... find an unreleased object with DcLoader.findUnreleased() method
Controllable obj;
// compare the state of the modified object with the actual state
List<Difference> list = DcLoader.differences(obj);

```

See chapter 10.5 for details.

After checking the data the user releases, rejects or passes back the event using the DcService interface. Releasing a business case means, he accepts the business case as it is and it will be executed. When he rejects the business case it will not be executed. When the releasing user does basically agree with the business case but wants to have some minor corrections he can pass the business case back to the user who initiated it. There is a functional difference between rejecting and passing back a business case:

When a business case on a resource is postponed due to a dual control actuator this resource is locked until it is either rejected or released. No other user has access to this resource. That means for example that a JPA entity resource cannot be updated by a third user when there exists a postponed business case on this entity. When a postponed method call business case exists, other users cannot invoke the method with the exact parameters.

When a business case is rejected this lock is removed and other users have access to the resource. When the business case is passed back, the lock remains and only the initiating user can access the resource.

```

// release the event represented by the Controllable.
// The user may give a remark which will be stored.
EntityManager em;
// transaction.begin(); (use any mechanism to begin and commit)
try {
    Object result = controllable.release(em, remark);
} catch (ResourceApplyException e) {
    // if the release fails
    ...
}

```

```

// transaction.commit();

// reject the event represented by the Controllable.
// The user may give a remark which will be stored.
EntityManager em;
// transaction.begin(); (use any mechanism to begin and commit)
controllable.reject(em, remark);
// transaction.commit();

// pass back the event represented by the Controllable.
// The user may give a remark which will be stored.
EntityManager em;
// transaction.begin(); (use any mechanism to begin and commit)
controllable.passBack(em, remark);
// transaction.commit();

```

The result of the release method is either the object on which a persistence event has been performed, the result of the method invocation or null. The release, reject and passBack methods must be executed within a transaction, either bean- or container- managed. If the method invocation throws an exception during release, or the database persistence action throws an exception the transaction is rolled back.

When the releasing user has passed back the business case, the initiating user has the possibility to reject it himself or to make corrections and submit it again to be released by a second user. Submitting is done with the submit method:

```

// submit the business case represented by the Controllable.
// The user may give a remark which will be stored.
EntityManager em;
// transaction.begin(); (use any mechanism to begin and commit)
controllable.submit(em, remark);
// transaction.commit();

```

## **10.4 Releasing and Rejecting Scheduled Business Events**

Scheduled business cases can be released or rejected by a user before the scheduled date is reached. This is done very similar to releasing and rejecting dual control business cases (see last chapter) with the SchedulerService API. All scheduled business cases are represented by an instance of Controllable. Here is how unreleased actions can be found:

```

// find all scheduled Controllables for the tenant set in
// session scope. Returns all scheduled objects if no tenant is
// set in context.
List<Controllable> list = SchedulerLoader.findScheduled();
// returns all scheduled objects of the given target type
List<Controllable> list2 = SchedulerLoader.findScheduled(targetType);
// compare the state of a scheduled JPA entity with the actual state.
// The method returns a list of differences
List<Difference> list = SchedulerLoader.differences(controllable);

```

Now the scheduled business case can be released or rejected:

```
// release the event represented by the Controllable.
// The user may give a remark which will be stored.
EntityManager em;
// transaction.begin(); (use any mechanism to begin and commit)
try {
    Object result = controllable.release(em, remark);
} catch (ResourceApplyException e) {
    // if the release fails
    ...
}
// transaction.commit();

// reject the event represented by the Controllable.
// The user may give a remark which will be stored.
EntityManager em;
// transaction.begin(); (use any mechanism to begin and commit)
controllable.reject(em, remark);
// transaction.commit();
```

The result of the release method is either the object on which a persistence event has been performed, the result of the method invocation or null. The release and reject methods must be executed within a transaction, either bean- or container- managed. If the method invocation throws an exception during release, or the database persistence action throws an exception the transaction is rolled back.

## 10.5 Comparing objects

Often it is interesting to know which properties have changed in two versions of the same object. This could be when an updated entity under dual control is released comparing the release version with the current version, comparing a scheduled update of an entity with the current version or comparing an archived version with the current or another archived version.

The general interface for comparing objects is in class CibatUtil:

```
List<Difference> list = CibatUtil.compare(Object newO, Object oldO);

List<Difference> list = CibatUtil.compare(Resource newR, Resource oldR);
```

The two compared objects must be of the same type. The second method is a convenient method and is the same as

```
List<Difference> list =
    CibatUtil.compare(newR.getObject(), oldR.getObject());
```

All compare() methods use library java-object-diff for determining the differences. Static fields and @Transient and @Version annotated fields or methods are skipped. Fields of super classes and transitive fields from dependent objects are considered. Fields of type Map or Collection are compared member by member disregarding the sequence. The result of the

comparison is a list of Difference objects each of which represents a modified property. The Difference class has methods to get the type of modification, the old and new values and the name of the modified property in different representations, see JavaDoc.

## 10.6 Checking Archive Integrity

Per default the archive entries are not secured against fraudulent manipulation with checksums. Integrity check functionality for the ARCHIVE actuator can be switched on in ciber-config.xml or in code. All modifications of existing archives will then be detected.

XML:

```
<actuator name="ARCHIVE">
  <properties>
    <integrityCheck>true</integrityCheck>
  </properties>
</actuator>
```

Code:

```
ArchiveActuator act = (ArchiveActuator)
    Configuration.instance().getActuator(ArchiveActuator.DEFAULTNAME);
act.setIntegrityCheck(true);
```

Cibet uses an implementation of the com.logitags.cibet.security.SecurityProvider to generate checksums. See chapter ‘Security’ for more information about the security provider.

When the archive entries are secured integrity should be controlled regularly. In a productive system it is good practice to check integrity on a snapshot of the database in a consistent state. Check the archive integrity with ArchiveLoader API:

```
List<Archive> checkList = archiveLoader.checkIntegrity();
```

The returned list contains all Archive objects where the checksum is not correct.

## 10.7 Searching, Redo and Restore of archived Business Cases

The load...() methods of ArchiveLoader allow loading Archives of the current tenant or of all tenants. If the controlled resource is a JPA entity and additional properties have been stored with the Archive object (see property storedProperties of Archive actuator chapter 9.8) one can search after these properties:

```
List<Archive> list = ArchiveLoader.loadArchivesByProperties(
    Class<?> entityClass, Map<String, Object> properties);
```

where the properties map key is the name of the property and the map value is the search value. If for example an ARCHIVE actuator and setpoint are configured like this:

```
<actuator name="myArchiveControl">
  <class>com.logitags.cibet.actuator.archive.ArchiveActuator</class>
```

```

    <properties>
      <storedProperties>company, state</storedProperties>
    </properties>
  </actuator>

  <setpoint id="sp3">
    <controls>
      <target>com.comp.Contract</target>
      <event>UPDATE,DELETE,INSERT</event>
    </controls>
    <actuator name="myArchiveControl"/>
  </setpoint>

```

to control Contracts, you will find all stored Contract archives of company 'Fiji Sales' in state 'open' with

```

Map<String,Object> props = new HashMap<>();
props.put("company", "Fiji Sales");
props.put("state", "open");
List<Archive> list = ArchiveLoader.loadArchivesByProperties(
    Contract.class, props);

```

Archive entries created by the ARCHIVE actuator store the state of a method invocation, a http request or a domain object. Methods and http requests can be invoked a second time from the archive with the exact same parameters. This can be done with the redo method of the Archive class. A notice can be added as an explanation:

```

// get an Archive that represents a method invocation resource, e.g. with
// ArchiveLoader.loadArchivesByMethodName("AccountManager", "transfer");

Object result = archive.redo("transferred again as bonus");

```

With the redo method the disclosure of sensible data to the executing user can be prevented as for example account and payment data in the above example. Controls and actuators are applied also to repeated method invocations when a setpoint for the REDO event exists.

For redoing http requests there is a limitation: If a POST or PUT http request contains a body, the body is archived only when the request is postponed by a dual control actuator at the same time. This is because the body of an http request is streamed in and therefore once read it is consumed. It is therefore not possible to read the body in the HTTP-FILTER sensor and read it again in the receiving servlet.

If an archive contains the state of a deleted or modified object which has been created by a SELECT, DELETE or UPDATE event this state can be restored with the restore method of the Archive class:

```

// get an Archive that represents a persistence resource, e.g. with
// ArchiveLoader.loadArchivesByCaseId("...");

EntityManager em;
// transaction.begin();(use any mechanism to begin and commit a
transaction)
Object restoredObject = archive.restore(em,
    "restored because user error");
// transaction.commit();

```

Configured controls and actuators are applied to the restore event if a setpoint for the RESTORE event exists.

## 10.8 Security

Integrity check functionality and encryption of sensible data is ensured by an implementation of interface `com.logitags.cibet.security.SecurityProvider`. The default implementation is `com.logitags.cibet.security.DefaultSecurityProvider` which stores secrets in an internal map. The map key is an identifier for the secret. The actual secret can be changed at any time. The map contains the current secret and previous secrets identified by their secret keys.

`DefaultSecurityProvider` can be fed with secrets and their secret keys by configuration in `cibet-config.xml` or by code:

XML:

```
<securityProvider>
  <class>com.logitags.cibet.security.DefaultSecurityProvider</class>
  <properties>
    <secrets mapKey="key2" current="true">2366Au37nBB.0ya?</secrets>
    <secrets mapKey="key1" current="false">1234567</secrets>
  </properties>
</securityProvider>
```

Code:

```
DefaultSecurityProvider sec = (DefaultSecurityProvider)
    Configuration.instance().getSecurityProvider();
sec.getSecrets().put("key2", "2366Au37nBB.0ya?");
sec.getSecrets().put("key1", "1234567");
sec.setCurrentSecretKey("key2");
```

This configuration means that current encryption and checksums will be created with the secret stored under key 'key2'. In the database old records may exist which were encrypted or have a checksum created with the secret stored under 'key1'.

Managing of passwords, keys or certificates is out of scope of Cibet. In any case, the secrets should be kept in a safe place like an encrypted configuration file or a hardware security box. It is also possible to create an own implementation of the `SecurityProvider` interface. Custom implementations must implement interface `com.logitags.cibet.security.SecurityProvider`, provide a default constructor and must define properties according to the Java Beans convention. Registration of a security provider can be done by code or by configuration. In code execute:

```
Configuration.instance().registerSecurityProvider(
    new MySecurityProvider());
```

Alternatively add in `cibet-config.xml`:

```
<securityProvider>
  <class>com.my.security.MyHsmSecurityProvider</class>
</securityProvider>
```

The properties can be set like this in ciber-config.xml:

```
<securityProvider>
  <class>com.my.security.MyHsmSecurityProvider</class>
  <properties>
    <hsmHost>com.my.secserver</hsmHost>
    <hsmPort>9000</hsmPort>
  </properties>
</securityProvider>
```

## 10.9 Assignment and Annotation of Dual Control Events

The release of business cases controlled by a dual control actuator can be directly assigned to a specific user. If a postponed business case is assigned, only the assigned user can execute the release. Rejection of business cases cannot be assigned.

Assignment will be automatically done if an approval user is set in the session scope with method

```
Context.sessionScope().setApprovalUser(userName);
```

During execution of the dual control actuators, the session property approvalUser is evaluated and if it is not null, the postponed business case is assigned to that user. Don't forget to set the approval user null after execution of the business case if you don't want to assign subsequent other business cases to the same user.

In the SIX\_EYES actuator there are two releases, both of which can be assigned. If during the initiation of the business case an approval user is set in session context, the first release will be assigned to that user. If during the first release an approval user is set in session context, the final release will be assigned to that user.

The initiating user can add a remark to the postponed business case to inform the releasing user about the context of the event. When in the request context a remark is found it will be automatically added to the metadata of the postponed business case. A remark can be added with method

```
httpSession.setAttribute("CIBET_REMARK", remark);
or
Context.requestScope().setRemark(remark);
```

A remark can also be added to the release and reject of a business case. In these cases, the remark is set in the release/reject methods of Controllable and overwrites a remark set in the request scope context:

```
Object controllable.release(EntityManager entityManager,
                           String remark)

void controllable.reject(EntityManager entityManager, String remark)
```



## **10.10 Notifications**

Cibet can send notifications of control events. Notifications may be sent for the following events:

- **FIRST\_ASSIGNED:** A business case is controlled by a SIX\_EYES process. The first releasing user has been assigned by the initiating user. He will be notified of the postponed business case.
- **FIRST\_RELEASED:** The first user has released a business case controlled by a SIX\_EYES process. The initiating user will receive a notification.
- **ASSIGNED:** A FOUR\_EYES controlled business case has been assigned to a user for release or a SIX\_EYES controlled business case has been assigned to a user for the second final release. He will be notified of the postponed business case.
- **RELEASED:** A FOUR\_EYES controlled business case has been released or a SIX\_EYES controlled business case has been finally released. The initiating user receives a notification
- **REJECTED:** a dual controlled business case (FOUR\_EYES or SIX\_EYES) has been rejected. The initiating user receives a notification

Sending of notifications requires three configurations:

- Registration of the notification provider
- Setting of the recipient address
- Activation of notifications in the actuators

### **Registration of the Notification Provider**

The notification provider implements the protocol and technique how notifications are sent. Cibet provides two implementations:

`com.logitags.cibet.notification.HttpNotificationProvider`

sends notifications as a http POST requests. The request body contains all metadata and target data of the business case under dual control.

`com.logitags.cibet.notification.EmailNotificationProvider`

sends notifications as an email. The subject and email text can be customized. This provider has five properties:

- `smtpHost`: SMTP server IP

- smtpPort: SMTP server port
- smtpUser: optional user name if the SMTP server requires authentication
- smtpPassword: optional password if the SMTP server requires authentication
- from: standard email address of the sender

It is possible to register own implementations of a Notification Provider, for example for SMS notifications. Custom implementations must implement interface `com.logitags.cibet.notification.NotificationProvider`, provide a default constructor and must define properties according to the Java Beans convention.

Registration of a notification provider can be done by code or by configuration. In code execute:

```
Configuration.instance().registerNotificationProvider(
    new HttpNotificationProvider());
```

Alternatively add in `cibet-config.xml`:

```
<notificationProvider>
  <class>
    com.logitags.cibet.notification.HttpNotificationProvider
  </class>
</notificationProvider>
```

The properties can be set like this in `cibet-config.xml`:

```
<notificationProvider>
  <class>
    com.logitags.cibet.notification.EmailNotificationProvider
  </class>
  <properties>
    <smtpHost>192.168.48.10</smtpHost>
    <smtpPort>25</smtpPort>
    <from>cibetNotifier@company.com</from>
  </properties>
</notificationProvider>
```

## Setting of the recipient address

The nature of the recipient address depends on the notification provider that has been registered. An `EmailNotificationProvider` requires an email address while an `HttpNotificationProvider` requires a URL.

The recipient address is set into the session scope context. The session scope provides two methods:

```
Context.sessionScope().setUserAddress(String)
```

This is the address of the logged in user, who initiates a dual controlled business case. The address can be set after login when the user has been authenticated and his properties are known. The other method

```
Context.sessionScope().setApprovalAddress(String)
```

sets the address of the user who has been assigned for first or final release. The approval address must be set before the business case is executed or before a SIX\_EYES controlled business case is released by the first user to notify the next user for the final release. Normally it makes sense to not only notify a user but also to assign the business case to him (see chapter Assignment and Annotation of Dual Control Events):

```
Context.sessionScope().setApprovalUser(String)
```

## Activation of notifications in the actuators

The last point to enable notifications is to activate it in the actuators. The Dual Control actuators FOR\_EYES, SIX\_EYES, PARALLEL\_DC and TWO\_MAN\_RULE have the flags `sendAssignNotification`, `sendReleaseNotification` and `sendRejectNotification`. If set to true, an address is found in session scope and a `NotificationProvider` has been registered, the respective notification is sent. Per default these properties are true.

This three-step configuration allows a fine-grained tuning for what business cases and events notifications should be sent. It is for example possible to instantiate dual control actuators which send notifications, others that do not send any and apply them in different setpoints.

## Customization of email notification templates

The default templates for email notifications are packed within the `cibet.jar` archive. If you want to customize or translate the email texts, create own templates and put them in the classpath of your application. The templates must have the following names:

<NotificationType>-emailsubject.vm for the email subject

<NotificationType>-emailbody.vm for the email body text

with <NotificationType> one of FIRST\_ASSIGNED, FIRST\_RELEASED, ASSIGNED, RELEASED and REJECTED according to the five possible notifications.

The templates are Velocity templates (<http://velocity.apache.org/>). Cibat puts all properties of class `Controllable` into the Velocity context and are accessible by the same names as in the classes. Additionally, properties of the `Resource` class are set into Velocity context dependent on the resource type.

EJB / POJO method invocation:

- target: class name of the object on which the method is invoked

- method: method name which is invoked
- resultObject: the return value of the method invocation, if any

JPA persistence:

- target: class name of the persisted object
- targetObject: the persisted object
- primaryKeyId: unique id of the persisted object

JPAQUERY queries:

- target: named query, JPA query or native SQL query

JDBC SQL statement:

- target: table name in the SQL statement
- targetObject: the SQL statement
- primaryKeyId: primary key id value in the SQL statement

HTTP servlet request:

- target: requested URL
- method: HTTP method of the request

For example the default template for the ASSIGNED event starts with

Hello \$approvalUser,

A business case under dual control has been assigned to you for final approval.  
You may release  
or reject the case. Please visit the dialogue for releasing/rejecting.

The dual controlled business case is registered under id: \$controllableId (case  
id: \$caseId)

```
control event:          $controlEvent

controlled target:      $target
#if( $method.length() > 0 )
($method)

#end
```

Please see the Velocity documentation for details on how to create these templates.

## 10.11 Locking Business Cases

The LOCKER actuator checks if a lock on a business case exists, the locking itself must be done before.

If a user wants to set a lock on a business case, he uses one of the lock methods of the Locker API. If a user wants e.g. set a lock on updates of a special account object he does something like:

```
Account account = getAccount();
Locker.lock(account, ControlEvent.UPDATE, "locked because ...");
```

There exist also methods to set a lock on all instances of a domain object:

```
Locker.lock(Account.class, ControlEvent.UPDATE, null);
```

or on a method of a class:

```
Locker.lock(AccountManager.class, "createAccount", "INVOKE", "account  
creation reserved for me only!");
```

or on a URL:

```
Locker.lock("http://www.mycompany.com/createAccount", "RELEASE", null);
```

If a lock exists already on the resource, an `AlreadyLockedException` will be thrown.

A lock can be unlocked, that means the lock is kept in the database for history reasons but the status is set to unlocked. Unlock of a lock exist in three different flavors:

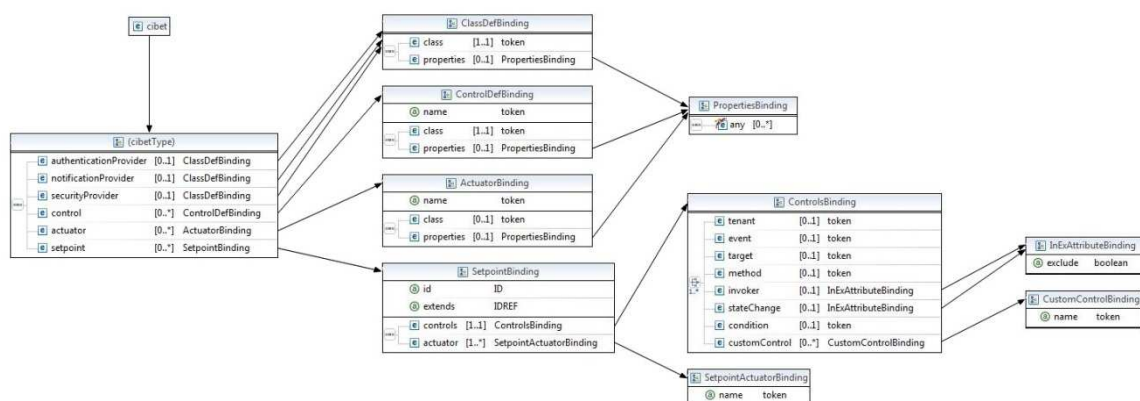
- method `unlockStrict()` in Locker API: Only the user who has locked the resource can execute this method.
- method `unlock()` in Locker API: Any user can execute this method
- `automaticUnlock`: If property `automaticUnlock` of `LockActuator` is set to true, the lock status is set to unlocked when the user who has set the lock executes the locked action for the first time. This method of unlocking is very convenient as it is done automatically. It is especially useful for deletes and dual control releases, because after executing the delete or release the lock is useless.  
Example: User John locks deletion of a Customer. No user except John can delete this customer now. If John actually deletes the customer the lock is automatically removed.

Hint: Often it is not a good idea to set a lock on a RELEASE event. If a lock on a RELEASE exists and a user rejects the dual control action the RELEASE lock would remain active though this is often not what is wanted. Though Cibet tries to reduce the developer's code from such processing functionality this kind of situations cannot be decided automatically. The easiest way to solve this issue is to lock `DC_CONTROL` instead of RELEASE event. In this case the lock will be removed whether the action is rejected or released.

## 11. Appendix

### 11.1 Schema Definition

XML Schema Definition cibet-config\_1.3.xsd: [http://www.logitags.com/cibet/cibet-config\\_1.3.xsd](http://www.logitags.com/cibet/cibet-config_1.3.xsd)



### 11.2 Guideline for migration of serialized objects

Cibet stores the state of objects in the database using Java serialization. When the classes of these objects are modified during development special care must be taken in order to allow a smooth deserialization of objects stored with an old class version. Depending on the controlled resource this may affect columns targetobject and result in table CIB\_RESOURCE and column encodedvalue in table CIB\_RESOURCEPARAMETER.

In most cases Java serialization is transparent to the developer and objects serialized with an old class version are deserialized to a new class version without problems when a compatible modification has been applied to the class. About compatible and incompatible class modifications read

<http://docs.oracle.com/javase/8/docs/platform/serialization/spec/version.html>.

The following considerations should be taken into account when a class has been modified and serialized objects of the old class version must be read from the above mentioned tables:

- Objects that shall be serialized by Cibet must implement java.io.Serializable and must have a developer defined serialVersionUID. When the class is modified the serialVersionUID must not be changed.

- When a property has been added to the class the value of new property is set to the default value
- When a property has been changed from static to non-static or transient to non-transient the value of the non-static/ non-transient property is set to the default
- When a property has been removed from the class, this property and its value is silently discarded
- When a property has been changed from non-static to static or non-transient to transient, the value of the static/transient property is set to the default
- When the class hierarchy has changed, for example when the super class has changed, the properties of the old super class are lost and the properties of the new super class are set to the default values
- When the data type of a property has been changed, an `java.io.InvalidClassException` is thrown

If the class has changed in an incompatible manner and the loss of data is not acceptable or an `InvalidClassException` is thrown, the serialized objects of the old class version must be migrated to the new class version. This can be achieved by using two classloaders and mapping old and new properties manually like in the following example where the attribute amount has changed incompatibly from type `Double` to type `String`:

```
// read serialized object from database
bytes[] oldBytes = ...;
URLClassLoader oldLoader = URLClassLoader.newInstance(new URL[] { new URL(
    "file:/D:/projects/test/oldVersion.jar") });
Object oldObject = CibetUtil.decode(oldLoader, oldBytes);

URLClassLoader newLoader = URLClassLoader.newInstance(new URL[] { new URL(
    "file:/D:/projects/test/newVersion.jar") });
Class<?> newClass = newLoader.loadClass("com.projects.test.SomeClass");
Object newObject = newClass.newInstance();

Method getter = oldObject.getClass().getMethod("getAmount");
Double dblAmount = (Double) getter.invoke(oldObject);
String strAmount = String.valueOf(dblAmount);
Method setter = newClass.getMethod("setAmount", String.class);
setter.invoke(newObject, strAmount);

// repeat for all other properties
...

byte[] newBytes = CibetUtil.encode(newObject);
// store serialized object in database
```