CleverBus

ESB made by developers for developers

Have you already noticed that **most of integration projects consists of 80% from routine and only 20% are really new topics**? Do you really want to spend so much effort on things which have already been solved before ? We don't.

In CleverBus world most of the operational infrastructure is ready out of the box. Also typical coding scenarios can be generated and only adjusted if you need to. And when you code someting, there is a big chance that somebody already did the same before. We offer you to profit from the wide community, not to code, just install. Also repeating business scenarios can be downloaded from the catalogue and just plugged in into your project.

Use code builders, components from the catalogue, design patterns. And if you reach the white area on the map, our community helps you to get through.

And if you really face to something special and new, **CleverBus does not limit you in any direction**. There are no dark corners in your project. Who has ever had to deal with black box, sure understands.

With CleverBus you have the whole source code in your hands, to the gunnels, and the community around, with many people ready to help.

Advantages

- Simple problems should be solved simply.
- Opensource ESB with community development.
- No limits for distribution.
- Real life needs turned into framework.
- Focus on maximum reliability during operation.
- Simplicity and high speed of the development.
- Maximum reuse of the code.
- 100% control over the code.
- Active architectural governance.

We believe CleverBus is useful and effective integration framework, see WHY?

Project artefacts

Issues:https://github.com/integram/cleverbus/issues

GitHub:https://github.com/integram/cleverbus

Current stable version: 1.1 (JavaDoc)

Current development version: 2.2.0-SNAPSHOT

Build server: https://travis-ci.org/integram/cleverbus

Forum:https://groups.google.com/d/forum/cleverbus

Main content

User guide

CleverBus – a modern information superhighway

The CleverBus integration platform offers a modern way of connecting IT systems intelligibly, with reasonable costs, and of reacting flexibly to all changing business requirements. CleverBus operates as a go-between for the exchange of information between IT systems, allowing them to communicate between each other by a system of messages and requests. It does this quickly, reliably, and in real time.

CleverBus significantly increases the accessibility of the entire environment and allows certain systems to be shut down temporarily for maintenance purposes without any direct impact on the customer. It simplifies servicing activities thanks to an integrated perspective on the customer profile regardless of which system or technology is administering its data, including external sources (e.g. ARES, SOLUS and other public registers). CleverBus also reduces the burden on individual applications, since it obtains each piece of information only once and then automatically distributes it to all necessary recipients regardless of their number. Operations are not burdened by the long response-time of individual systems, and everything is quickly available and up-to-date.

Among other integration platforms CleverBus occupies the position of a light, open product that can be quickly implemented, easily modified, and yet thanks to its embrace of open source technology involves minimum acquisition costs. During its development great emphasis is laid on high performance, security, reliability and problem-free operations.

Types of integration processes

The CleverBus lightweight integration platform is an ESB (Enterprise Service Bus) product. It involves minimal expenses while allowing for the implementation of all regular types of integration processes.

CleverBus supports synchronous and asynchronous methods of communication, distributes information using both the point-to-point and publish-subscribe methods, checks whether the information is up to date, guarantees the delivery of messages and their correct sequence.

It contains resources for ensuring data persistence, data update checks, postponed delivery, automatic correction and recovery from errors, as well as a wide range of resources for administrators enabling them to learn of errors quickly and remedy them.

It makes life easier for developers thanks to its extensive support of automatic tests and use of only regularly available technology such as Spring Framework, etc.

Drivers and adapters

It uses Apache Camel to process messages, which offers all the advantages associated with the large community of developers linked with this product, and the possibility of using the large set of drivers and extension components that exist for Apache Camel. This makes it possible to connect quickly and uniformly to basically any of the technologies used at present and restrict the negative impacts of such differences on the entire environment.

Not only Software

Successful integration is not only about technology but a suitably designed architecture. This is why CleverBus is not simply software but contains instructions on how to design processes so that they are simple, effective and problem-free.

CleverBus consists of:

- The core ESB
- Business components
- Documentation
- Suggested models

CleverBus features

- Open and certified technology.
- Generally used standards (XML, XSLT, WSDL, etc.), no proprietary formats.
- Zero licence costs from third parties.
- A wide range of drivers, adapters and other components.
- High performance and easy scalability.
- Integrated security mechanisms.
- Resources for ease of management and operations.
- Elimination of the technological differences of individual systems.
- Clear and intelligible architecture.
- Clear implications during changes.

Why CleverBus?

CleverBus is Apache Camel on steroids. Why not get more?

- CleverBus is based on matured, well-proofed and very popular integration engine **Apache Camel**
- CleverBus extends basic Camel functionality to **be more productive and effective** in common integration implementation
- CleverBus is **proven and stable** solution used in production environments
- CleverBus is **open-source with quality code,** covered by many unit tests
- CleverBus uses database as queue for asynchronous messages processing. Database is well-known technology for most people to administrate or use it. Nevertheless, we want to offer alternative, we want to support JMS as queue.
- CleverBus is **technologically neutral** to operation system, application server and database.
- CleverBus together with application server like Apache Tomcat and database like PostgreSQL represent **light-weight ESB solution** (server and database can be different)
- CleverBus uses another well-proven open-source libraries such as Spring framework, Spring security, Spring Web Services or HttpClient
- CleverBus can **use same tools which can be used by Apache Camel**, for example Red Hat® JBoss® Fuse or Hawtio
- CleverBus has own **web administration console**

We have many years of experience with integration projects, we know common problems, we know how to solve it effectively. CleverBus is established on this know-how.

Main CleverBus extensions

CleverBus extends Apache Camel in many ways, look at main points:

- **asynchronous message processing model**
  - **parent-child concept** that allows to divide main message into more child messages and process them separately
  - **obsolete messages** checking when messages impact same data
  - **funnel** component is for filtering concurrent messages at specific integration point. This filtering ensures that only one message at one moment will be processed, even in guaranteed order (optional choice).
  - **guaranteed message processing order**
  - algorithm is **configurable**
  - synchronous response that input request is saved in queue and **asynchronous**

    confirmation with processing result
  - **monitoring** of processing in Admin GUI and via JMX
- **throttling** - functionality that checks count of input requests to integration platform and if this count exceeds defined limit then new requests are restricted
- extended error handling with many new Camel events
- **tracking external systems communication** - storing requests and responses
- **web administration console**
  - searching in asynchronous messages
  - message details with requests/responses overview
  - searching in logs
  - manual cancel of next message processing
  - restart failed messages
  - error codes catalogue
  - exposed WSDLs overview
  - endpoints overview
- **extended logging** allows to group logs together of one request/message or process
- **direct call console** allows to send custom requests to external system
- **stopping mode** is useful function for correctly CleverBus shutdown
- **extensions** allow to encapsulate new CleverBus functionality
- **support for cluster** (in progress at this moment)

Competitors

When to use/not to use Apache Camel.

Spring Integration

http://projects.spring.io/spring-integration

Talend ESB

Product page: http://www.talend.com/products/esb

ESB specification: http://www.talend.com/products/specifications-esb

Basic technologies: Apache Camel, Apache Karaf (OSGi)

Same base ground - Apache Camel (http://www.talend.com/resource/apache-camel.html)

Camel is a mediation engine while Mule is a light-weight integration platform. See http://stackoverflow.com/questions/3792519/apache-camel-and-other-esb-products

Apache Camel and Talend ESB: monitoring and administration

WSO2 ESB

Product page: http://wso2.com/products/enterprise-service-bus/

Basic technologies: Apache Synapse and ODE (http://stackoverflow.com/questions/15361380/wso2-or-fusesource)

Getting started

What constitutes CleverBus

Following diagram descibes main technologies you came across when you start to develop with CleverBus. See Architecture section for more information.

What a developer needs to know

Before you start playing with CleverBus you should have knowledge of the following frameworks and tools:

- **Java JDK 1.6** (http://docs.oracle.com/javase/6/docs/)
- **JAXB** (http://docs.oracle.com/javase/6/docs/technotes/guides/xml/jaxb/index.html, https://jaxb.java.net/guide/)

    - basic concept
    - XmlRootElement peculiarities
- **JUnit** (http://junit.org/)
- **Spring Framework** (http://projects.spring.io/spring-framework/)
    - basic concept, XML configuration
    - autowiring
    - first-contract Spring-WS (http://docs.spring.io/spring-ws/site/reference/html/tutorial.html)
- **Apache Maven** (http://maven.apache.org/)
    - basic concept
    - profiles (http://maven.apache.org/pom.html#Profiles)
- **Apache Camel** (http://camel.apache.org/)
    - Basic concept
    - JavaDSL style (http://camel.apache.org/java-dsl.html)
    - Methods mostly used in CleverBus routes
        - from, to, routeId, id, log
        - unmarshal, marshal, jaxb, convertBodyTo, transform, validate, split, filter, method
        - simple, mvel, constant
        - loop, choice, when, otherwise, end, endChoice
        - doTry, doCatch, onException, continued, handled, throwException
        - process, pipeline, routingSlip
        - multicast, stopOnException
        - setProperty, setHeader, removeHeader, setBody, header, append
        - bean, beanRef
        - exchange: getIn, getOut
- **SoapUI** (http://www.soapui.org/)

What a developer needs to install

Necessary

- **Java JDK 6** (http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase6-419409.html)
- **Apache Maven3** (http://maven.apache.org/download.cgi)
- **JAVA IDE**
    - Eclipse IDE for Java EE Developers (https://www.eclipse.org/downloads)
    - Spring Tool Suite (http://spring.io/tools/sts)
    - IntelliJ IDEA (http://www.jetbrains.com/idea/download/)
- *all other needed libraries are fetched by Maven*

Optional

- **Apache Tomcat** (http://tomcat.apache.org/) â€“ in fact, during developing you can run your routes just in JUnit test. However for integration test you will need deploy application with routes on Tomcat.

- **PostgreSQL** (http://www.postgresql.org/download/) â€“ for JUnit tests run, H2 DB is used by default; for Tomcat deployment, PostreSQL db datasource is used
- **SoapUI** (http://www.soapui.org/)

How to

This section contains several "How to" tutorials for better understanding how to make specific tasks.

- How to build CleverBus
- How to start new project
- How to write routes
- How to write wiki

How to build CleverBus?

Building CleverBus is very easy:

1. download sources from GitHub or clone Git repository
2. call *mvn package*
3. deploy *cleverbus.war* (module *web-admin*) to application server and start it

Must be called *mvn package* and not *mvn compile* because there are dependencies in resources between Maven modules which use *maven-dependency-plugin* with *unpack-dependencies* goal.

You can continue with starting new integration project or with writing new integration routes.

How to start new project?

If you want to start to implement new integration project (implement new routes) then you have the following options:

1) implement new routes in separate project and use it with CleverBus

This approach supposes to use CleverBus admin as is (without any additional changes) with adding dependencies to external projects/libraries with route implementations. This approach is very similar how works CleverBus extensions, specifically look at How to implement new extensions.

There is one big difference between extensions and adding Project routes to CleverBus - extensions are initialized in own Spring child context but when you add external project routes then you use same Spring context like other routes implemented directly in CleverBus.

This approach has several steps:

- create separated project for route implementations with dependency to CleverBus API (*core-api* module)
- add dependency to project's artefact (JAR file is expected)
- adjust Spring configuration to initialize Project routes (all in *web-admin module*)
  - add Spring XML configuration file (*/src/main/webapp/WEB-INF/spring-ws-servlet.xml*)
  - add property files (*/src/main/webapp/WEB-INF/spring-ws-servlet.xml*)
  - extend Spring auto-scaning

You can use Dynamic extension loading functionality even if you don't implement extensions, you can use extension/external library concept for initialization.

Use *extensions.cfg* (or *extensions0.cfg*) and add parameter for external library.

This approach has advantage that it's not necessary to make changes in CleverBus at all (only set extension properties)

2) create new project from CleverBus web-admin

CleverBus *web-admin* module contains admin GUI and all necessary configurations for runnable and deployable CleverBus web application.

Create new (Maven) project with the following module structure:

- project routes implementation (analogy to *examples* module, see Maven and Spring)
- web admin with dependency to previous module with routes. This module is analogy to *web-admin* module in CleverBus.

The most simple approach is to copy *web-admin* module from CleverBus and adjusts it for new project. There are the following steps to do:

- change Maven dependencies (*pom.xml*)
- change package names for auto-scanning of route implementations (*/src/main/webapp/WEB-INF/spring-ws-servlet.xml*)
- if needed change security settings (*/src/main/webapp/WEB-INF/rootSecurity.xml*)

If you want to customize web admin (add new content or change look-at-feel or whatever else) then there are two ways how to do it:

- use or copy/duplicate default Admin GUI from CleverBus to your project and make changes. This approach has one big disadvantage that it will be tricky to upgrade Admin GUI to newer versions.
- second possibility is to leave Admin GUI as is (without changes) and create new web content

in separated web context or web application. In other words decouple Admin GUI content from your web content.

If you use different database to default H2 DB then you must initialize database structure with SQL scripts for specific target databases, e.g. */db/db_schema_postgreSql.sql* (*core* module) for PostgreSQL.

3) use Maven archetype

Use Maven archetypes for creating new project based on CleverBus. Maven archetype is implemented in *web-admin-archetype*.

Build customized project from archetype *cleverbus-web-admin-archetype* it **is possible by command**:

```
mvn archetype:generate
  -DarchetypeCatalog=(local|remote)
  -DarchetypeGroupId=org.cleverbus
  -DarchetypeArtifactId=cleverbus-web-admin-archetype
  -DarchetypeVersion=<version>
  -DcleverBusVersion=<version>
  -DcleverBusServerName=ESB
  -DcleverBusServerDescription="Enterprise Service Bus"
```

Note: one of the best practices is that the name of package ends on ".admin", because archetype contains only admin console, all other is as dependency.

How to write routes?

Description

This page contains list of steps and tips for implementation of new routes.

Look at How to start new project? for starting new project ...

Look at Development tips and Best practices.

Adhere to conventions how to write the code.

Create new routes

- class name should ends with suffix *Route*
- adhere to the rules of naming conventions for route IDs (use *getInRouteId()*, *getOutRouteId()* or *getRouteId()* functions) and URIs
- use parent class *org.cleverbus.api.route.AbstractBasicRoute*
- use *@CamelConfiguration* and define unique Spring bean name for this route - this annotation allows initialization by Spring auto-scanning funcionality. Don't forget to check auto-scanning configuration.
- define operation name

Unit tests

Each route implementation must have corresponding unit tests, at least one successful scenario.

- create unit test that extends *AbstractTest* or *AbstractDbTest *(both classes are from test *module*) if database support is needed
- you can use Spring test profiles - see *com.cleverlance.cleverbus.modules.TestProfiles*
- use *@ActiveRoutes* annotation that defines which routes will be activated for specific unit test

You can use *RouteBeanNameGenerator* for automatic bean names generation.

New synchronous route implementation

Synchronous route is route where source system waits for response. This type of requests are not stored in CleverBus evidence, there are mentions in log files only. Possible error is immediately propagated to source system.

Steps for implementation are identical to those mentioned in previous chapter.

Every new synchronous route must contains ***traceHeader*** element that contains ***traceIdentifier*** element with tracing information of incoming message.

SOAP example:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:com="http://cleverbss.cleverlance.com/ws/Common-v1"
xmlns:hel="http://cleverbus.cleverlance.com/ws/HelloService-v1">
  <soapenv:Header>
     <com:traceHeader>
        <com:traceIdentifier>
           <com:applicationID>APPL002</com:applicationID>
           <com:timestamp>2015-05-21T08:54:58.147+02:00</com:timestamp>
           <com:correlationID>${=java.util.UUID.randomUUID()}</com:correlationID>
           <!--Optional:-->
           <com:processID>process001</com:processID>
        </com:traceIdentifier>
     </com:traceHeader>
  </soapenv:Header>
  <soapenv:Body>
     <hel:syncHelloRequest>
        <hel:name>CleverBus team</hel:name>
     </hel:syncHelloRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

If traceHeader is not part of SOAP message, default traceHeader will be generated or route must implements TraceHeaderProvider interface for getting traceHeader.

Example of TraceHeaderProvider implementation: ``` java CamelConfiguration(value = SyncOutageRoute.ROUTE_BEAN) public class SyncOutageRoute extends AbstractBasicRoute implements TraceHeaderProvider {

private static final String ROUTE_ID_SYNC_OUTAGE = getRouteId(ServiceEnum.OUTAGE, OPERATION_NAME); ...

public String getRouteId() { return ROUTE_ID_SYNC_OUTAGE; }

public TraceHeader getTreaceHeader(Exchange exchange) { TraceIdentifier traceIdentifier = new TraceIdentifier(); traceIdentifier.setTimestamp(DateTime.now()); traceIdentifier.setCorrelationID(UUID.randomUUID().toString()); traceIdentifier.setApplicationID("SAP");

```
  TraceHeader result = new TraceHeader();
 result.setTraceIdentifier(traceIdentifier);
 return result;
```
}}```

New asynchronous route implementation

How to implement asynchronous routes is described in another page.

Route implementation tips

Checklist of features which can be used during route implementation:

- throttling - route for processing asynchronnous input requests contains throttling by default. You should add it to all synchronnous routes
- external call - each call to external system should go through "external call" funcionality
- route authorization is made by Camel policy, see Camel Security
- Guaranteed message processing order
- checking of obsolete messages via object ID
- look at components for use - msg-funnel, asynch-child, ...
- define senders *CloseableHttpComponentsMessageSender* for calling external system via Spring Web Service (SOAP messages). Add configuration into */META-INF/sp_ws_wsdl.xml*

Use SoupUI integration test for calling real web service to verify that everything works as expected. Unit test is good to have but it doesn't catch all possible problems.

Example implementations

Example routes are in *examples* module. There are two basic examples - synchronous and asynchronous *hello world* service.

WSDL is available on */ws/hello.wsdl* and endpoint on the URL */ws/hello/v1*

WSDL and XSD implementation

- WSDL and XSD is defined in Contract-First design approach - firstly define web service interfaces, secondly implement them in specific language.
- there are two files (in package *resources*/*org.cleverbus.modules.in.hello.ws.v1_0*): WSDL definition (e.g. *hello-v1.0.wsdl*) and XSD with requests/responses definition (*helloOperations-v1.0.xsd*). XSD contains request definitions, imports *commonTypes-v1.0.xsd* with common types

- publish WSDL/XSD files - add configuration to */META-INF/sp_ws_wsdl.xml.sws:static-wsdl* is used for publishing WSDL, XSD files have to be published separately. See Spring Web Services reference manual where you find more information because Spring WS is underlying library for web service communication.
- register XSD schemas for validation incoming/outgoing messages in configuration of "*validatingInterceptor*" (class *HeaderAndPayloadValidatingInterceptor*)
  - *traceHeader* is mandatory for asynchronous requests only - set synchronnous requests in *ignoreRequests* property to ignore this validation
- configure conversion to Java classes - use *jaxws-maven-plugin* Maven plugin for conversion from WSDL or *jaxb2-maven-plugin* for conversion from XSD

Synchronous Hello service

Synchronous implementation is in class *org.cleverbus.modules.in.hello.SyncHelloRoute*.

If comes web service request with element name *syncHelloRequest* and namespace *http://cleverbus.org/ws/HelloService-v1*) then this route is "activated" and request starts processing. Response is immediately send back to the source system.

Asynchronous Hello service

Asynchronous implementation is in class *org.cleverbus.modules.in.hello.AsyncHelloRoute*.

If comes web service request with element name *asyncHelloRequest* and namespace *http://cleverbus.org/ws/HelloService-v1*) then this route is "activated" and request starts processing. Firstly synchronous part is proceeded - input request is validated and saved into queue (database) and synchronous response is immediately send back to the source system with information that CleverBus accepted input request and ensures request processing. When request processing finishes then asynchronous confirmation with processing result is send to source system.

Unit tests implementation

There are corresponding unit tests to each route - *SyncHelloRouteTest* resp.*AsyncHelloRouteTest*.

How to write wiki?

Introduction

As long as we need to support GFM syntax which uses GiHub for our CE edition we have moved from Confluence to GFM. So that we have one source of documentation for both CE and EE edition. Versioning is done basically by git. For each released version of cleverbus we have also up-to-date documention in the same repo.

- Wiki pages are stored in GIT repo in doc/wiki folder
- Wiki pages are writen using [GitHub Flavored Markdown][A] syntax
- To modify/add content just use .md sources in this repo

Tools for editing

- Text editor of your choice eg. Notepad++
  - no HTML preview
- In favorite development IDE (HTML preview)
  - InteliJ MultiMarkdown plugin
  - NetBeans Flow Markdown plugin
- Specialized markdown editor eg. MarkdownPad
- Markdow table online generator/editor
  - with CSV file import

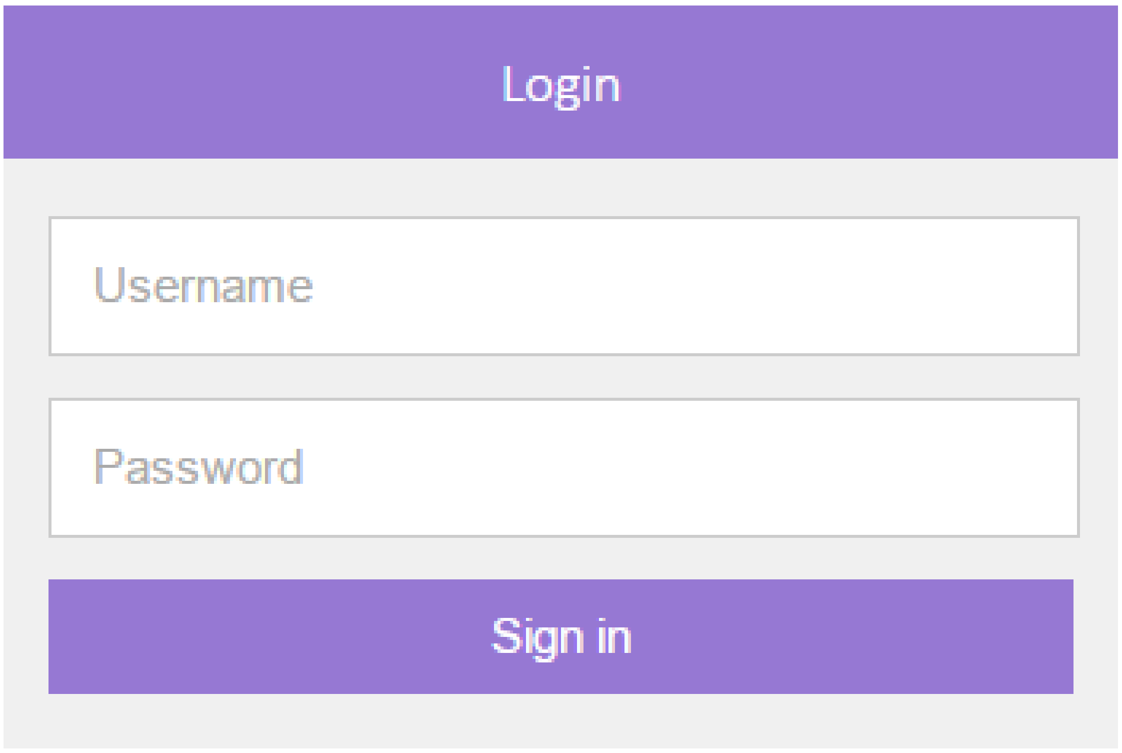
Rules for documentation writing

- Write new documentation in English only.
- There is one version of documentation for all CleverBus versions. If some information is valid for or from specific version then highlight it. Example:
  From version 0.2
- Use Markdown code macro for code snippets.
- If you write Java classes, configuration parameters or use other technical names then use italic format style.

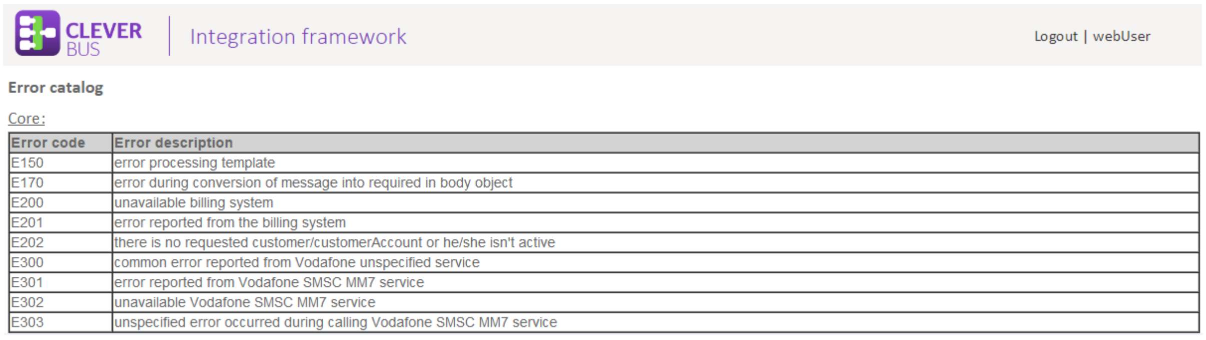
Admin GUI

Admin GUI is lightweight graphic user interface to support general and frequent scenarios for administration of CleverBus solutions such as observation and search information about processes in case of problems.

The default page without credentials allows you to view the current version:



After logging in the user sees the signpost and appropriate actions, which may be carried out on the basis of the authorization.



Error catalog

Core:

Error code	Error description
E150	error processing template
E170	error during conversion of message into required in body object
E200	unavailable billing system
E201	error reported from the billing system
E202	there is no requested customer/customerAccount or he/she isn't active
E300	common error reported from Vodafone unspecified service
E301	error reported from Vodafone SMSC MM7 service
E302	unavailable Vodafone SMSC MM7 service
E303	unspecified error occurred during calling Vodafone SMSC MM7 service

Endpoints overview

`Since version 0.4`

Link:*/esb/web/admin/endpoints*

Overview of inbound camel URI endpoints

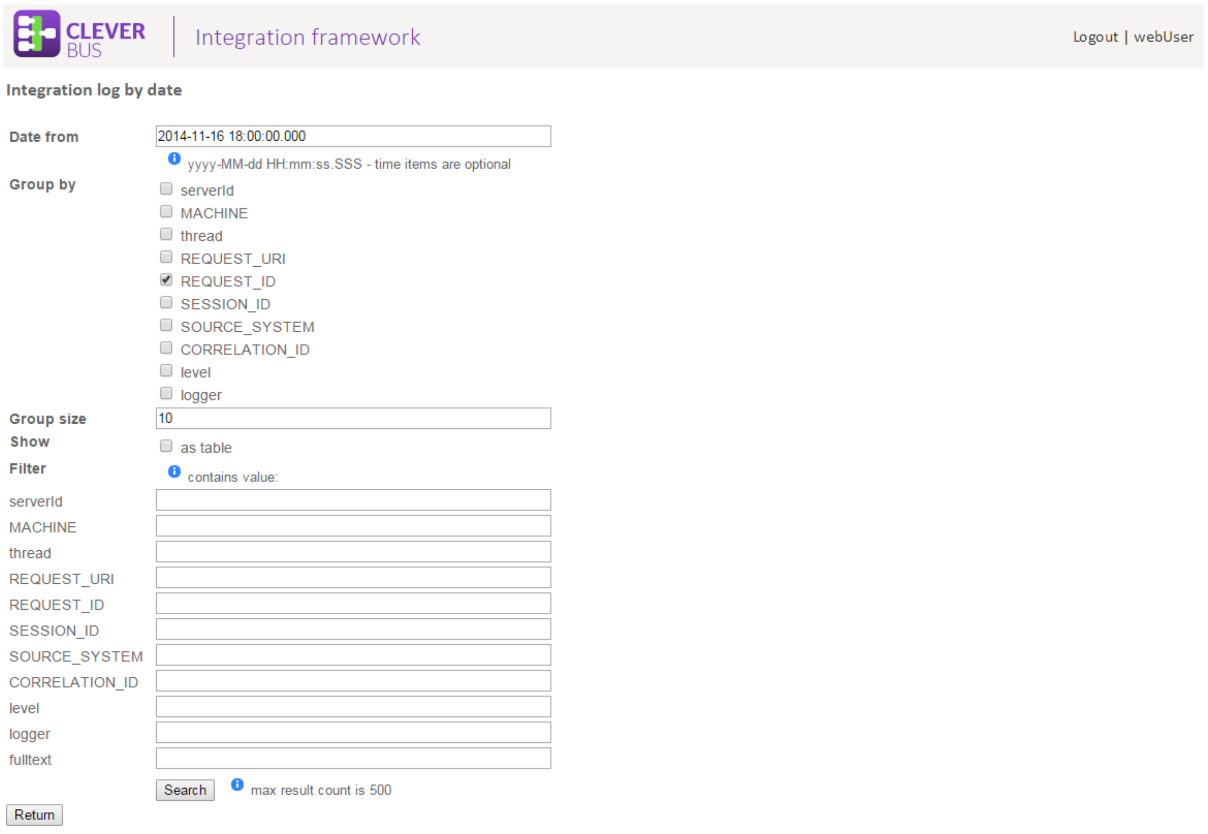
It is necessary to properly configure configuration parameter *endpoints.includePattern* which defines pattern for the URIs which are traced. See more information in chapter configuration.

Exposed WSDLs overview

`Since version 0.4`

Link:*/esb/web/admin/wsdl*

Overview of exposed WSDL contracts.


Logout | webUser

Integration log by date

Date from	2014-11-16 18:00:00.000
	yyyy-MM-dd HH:mm:ss.SSS - time items are optional
Group by	☐ serverId ☐ MACHINE ☐ thread ☐ REQUEST_URI ☑ REQUEST_ID ☐ SESSION_ID ☐ SOURCE_SYSTEM ☐ CORRELATION_ID ☐ level ☐ logger
Group size	10
Show	☐ as table
Filter	contains value:
serverId	
MACHINE	
thread	
REQUEST_URI	
REQUEST_ID	
SESSION_ID	
SOURCE_SYSTEM	
CORRELATION_ID	
level	
logger	
fulltext	

max result count is 500

The following funcion is available since version 0.4

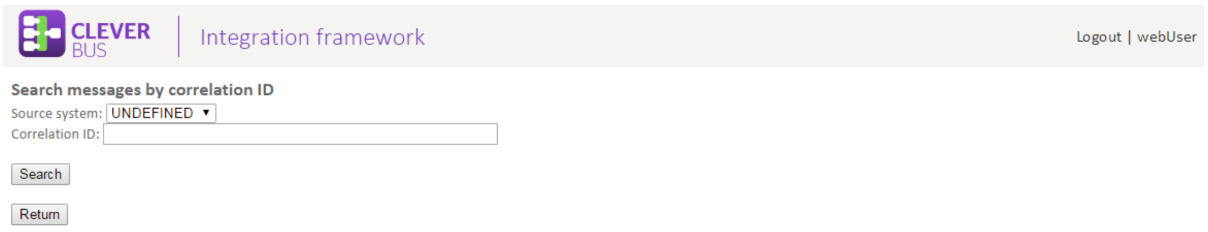
Important information such as requests and responses for CleverBus platform or requests and responses sent into external systems or errors are highlighted. If a message is asynchronous, each record in log contains also identification of external system (e.g. CRM) and correlation ID (sceptra tenens) such as stated in picture below. These highlighted parts are concurrently clickable links by which can go to detail of message. Next highlighted parts are for example request and response by which you can show modal window with request and response that are reformatted (if this is SOAP message).

Finding monitored requests to external systems

`Since version 0.4`

Link: Searching by parameters in all monitored (defined by configuration) requests and responses sent into external systems (*/esb/web/admin/reqresp/search*).

According defined filter where parameters *Date from* and *Date to* are required, can search records of monitored requests and responses, which were sent into external systems. Each records contains also state and potential error if is occurred. If the request was sent within asynchronous process the record also contains direct link to detail of asynchronous message. Integration platform but for performance reasons returns only up to 50 entries which are suitable by defined filter.



Search messages by correlation ID

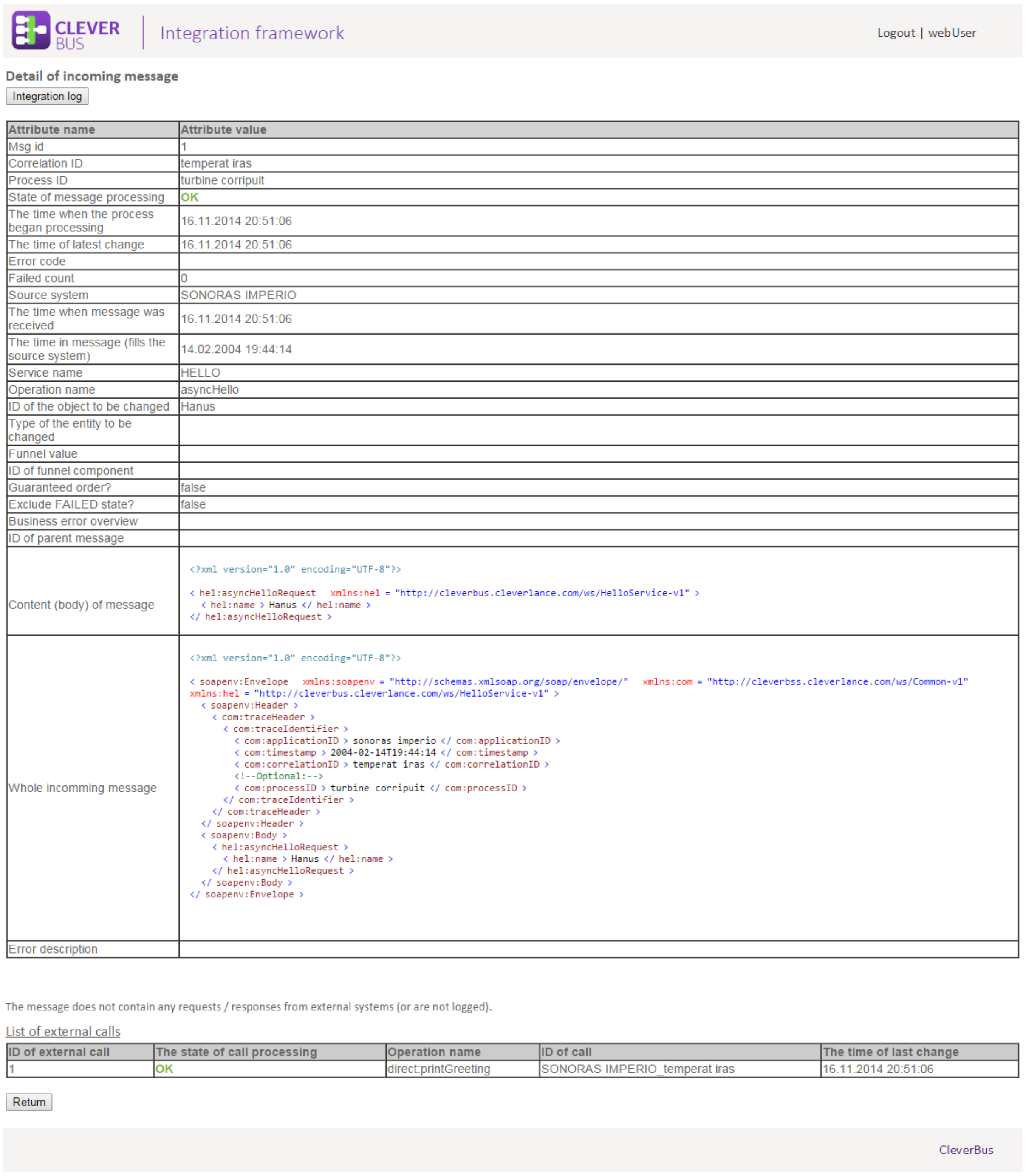
Source system: UNDEFINED

Correlation ID:

Finding asynchronous message which contain given string

Link: Searching asynchronous message which contain given string in body of message (*/esb/web/admin/messages/messagesByContent*)

Displaying all **asynchronous** messages in CleverBus integration framework which contain in inbound request body given string.



Detail of incoming message

Attribute name	Attribute value
Msg id	1
Correlation ID	temperat iras
Process ID	turbine corripuit
State of message processing	OK
The time when the process began processing	16.11.2014 20:51:06
The time of latest change	16.11.2014 20:51:06
Error code	
Failed count	0
Source system	SONORAS IMPERIO
The time when message was received	16.11.2014 20:51:06
The time in message (fills the source system)	14.02.2004 19:44:14
Service name	HELLO
Operation name	asyncHello
ID of the object to be changed	Hanus
Type of the entity to be changed	
Funnel value	
ID of funnel component	
Guaranteed order?	false
Exclude FAILED state?	false
Business error overview	
ID of parent message	

Content (body) of message

```
<?xml version="1.0" encoding="UTF-8"?>

< hel:asyncHelloRequest   xmlns:hel = "http://cleverbus.cleverlance.com/ws/HelloService-v1" >
  < hel:name > Hanus </ hel:name >
</ hel:asyncHelloRequest >
```

Whole incomming message

```
<?xml version="1.0" encoding="UTF-8"?>

< soapenv:Envelope   xmlns:soapenv = "http://schemas.xmlsoap.org/soap/envelope/"   xmlns:com = "http://cleverbss.cleverlance.com/ws/Common-v1"
xmlns:hel = "http://cleverbus.cleverlance.com/ws/HelloService-v1" >
  < soapenv:Header >
    < com:traceHeader >
      < com:traceIdentifier >
        < com:applicationID > sonoras imperio </ com:applicationID >
        < com:timestamp > 2004-02-14T19:44:14 </ com:timestamp >
        < com:correlationID > temperat iras </ com:correlationID >
        <!--Optional:-->
        < com:processID > turbine corripuit </ com:processID >
      </ com:traceIdentifier >
    </ com:traceHeader >
  </ soapenv:Header >
  < soapenv:Body >
    < hel:asyncHelloRequest >
      < hel:name > Hanus </ hel:name >
    </ hel:asyncHelloRequest >
  </ soapenv:Body >
</ soapenv:Envelope >
```

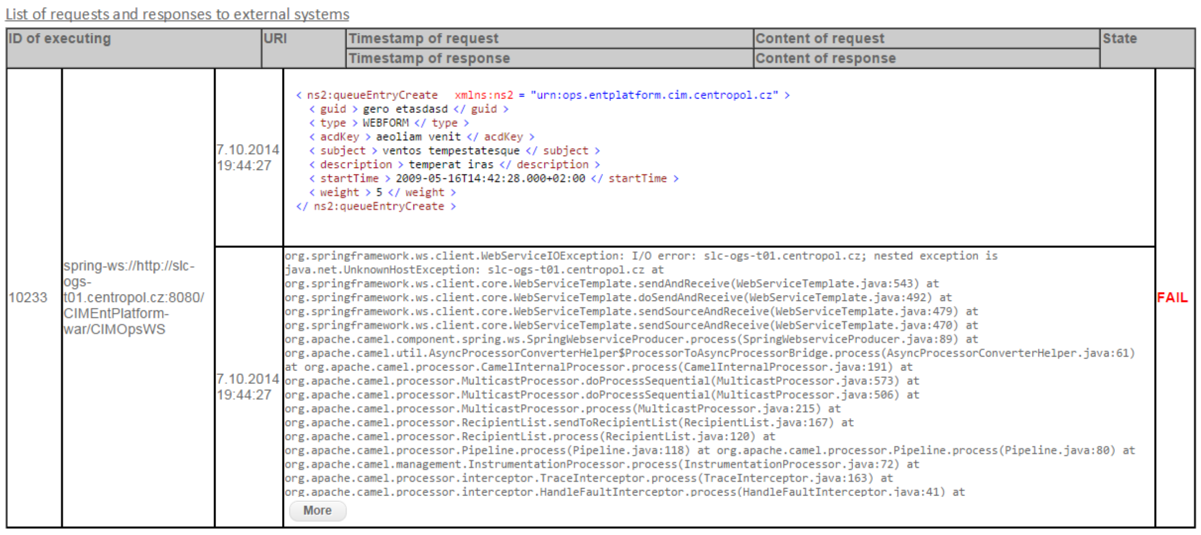
Attribute name	Attribute value
Error description	

The message does not contain any requests / responses from external systems (or are not logged).

List of external calls

ID of external call	The state of call processing	Operation name	ID of call	The time of last change
1	OK	direct:printGreeting	SONORAS IMPERIO_temperat iras	16.11.2014 20:51:06

Detail of message contains all necessary information which are stored by CleverBus integration platform and also contains the list of external calls and their statuses which identifies concrete place in process with potential error. If the message is not properly processed the state is FAILED with error code and description. You can show detail information about error by clickable error code.



List of requests and responses to external systems

ID of executing	URI	Timestamp of request / Timestamp of response	Content of request / Content of response	State

10233	spring-ws://http://slc-ogs-t01.centropol.cz:8080/CIMEntPlatform-war/CIMOpsWS	7.10.2014 19:44:27	`< ns2:queueEntryCreate xmlns:ns2 = "urn:ops.entplatform.cim.centropol.cz" >`
			`< guid > gero etasdasd </ guid >`
			`< type > WEBFORM </ type >`
			`< acdKey > aeoliam venit </ acdKey >`
			`< subject > ventos tempestatesque </ subject >`
			`< description > temperat iras </ description >`
			`< startTime > 2009-05-16T14:42:28.000+02:00 </ startTime >`
			`< weight > 5 </ weight >`
			`</ ns2:queueEntryCreate >`
				FAIL

		7.10.2014 19:44:27	`org.springframework.ws.client.WebServiceIOException: I/O error: slc-ogs-t01.centropol.cz; nested exception is`
			`java.net.UnknownHostException: slc-ogs-t01.centropol.cz at`
			`org.springframework.ws.client.core.WebServiceTemplate.sendAndReceive(WebServiceTemplate.java:543) at`
			`org.springframework.ws.client.core.WebServiceTemplate.doSendAndReceive(WebServiceTemplate.java:492) at`
			`org.springframework.ws.client.core.WebServiceTemplate.sendSourceAndReceive(WebServiceTemplate.java:479) at`
			`org.springframework.ws.client.core.WebServiceTemplate.sendSourceAndReceive(WebServiceTemplate.java:470) at`
			`org.apache.camel.component.spring.ws.SpringWebserviceProducer.process(SpringWebserviceProducer.java:89) at`
			`org.apache.camel.util.AsyncProcessorConverterHelper$ProcessorToAsyncProcessorBridge.process(AsyncProcessorConverterHelper.java:61)`
			`at org.apache.camel.processor.CamelInternalProcessor.process(CamelInternalProcessor.java:191) at`
			`org.apache.camel.processor.MulticastProcessor.doProcessSequential(MulticastProcessor.java:573) at`
			`org.apache.camel.processor.MulticastProcessor.doProcessSequential(MulticastProcessor.java:506) at`
			`org.apache.camel.processor.MulticastProcessor.process(MulticastProcessor.java:215) at`
			`org.apache.camel.processor.RecipientList.sendToRecipientList(RecipientList.java:167) at`
			`org.apache.camel.processor.RecipientList.process(RecipientList.java:120) at`
			`org.apache.camel.processor.Pipeline.process(Pipeline.java:118) at org.apache.camel.processor.Pipeline.process(Pipeline.java:80) at`
			`org.apache.camel.management.InstrumentationProcessor.process(InstrumentationProcessor.java:72) at`
			`org.apache.camel.processor.interceptor.TraceInterceptor.process(TraceInterceptor.java:163) at`
			`org.apache.camel.processor.interceptor.HandleFaultInterceptor.process(HandleFaultInterceptor.java:41) at`
			More	

Finding detail of message

Link: Searching the message by correlation ID (*/esb/web/messages/searchForm*)

Each asynchronous message is strictly identified by external system that sent this message and correlation ID. Correlation ID is unique identifier for each external system. With these two parameters can search detail of message. If message exists CleverBus shows detail of it with all information which are joined with this message - inbound request that will be processed, state of processing and list of potential errors and so on. If detail of message does not contain required information you can browse integration log by "Integration log" button (above the detail of message). This button shows in single page all grouped logged records for this message (log viewer).

Some records are highlighted (only informative):

- red parts are errors (exceptions)
- orange parts are executing of calls (request, response) into external systems

Notice: records displayed in this single page are searched only by correlation ID. Some records but don't contain correlation ID and thanks to it these records will not be displayed in this page.

Restart FAILED messages

Link: Admin - Message operation (*/esb/web/admin/messageOperations*)

Often happens that asynchronous messages fails (is in FAILED state) for error reason or general for unavailable external system. In this cases it is possible restart the message. In during processing of asynchronous message there are "through points" (external calls) defined by developers. It is our customized implementation of the idempotent component and it is used to filter out duplicate messages. This uses an expression to calculate a unique message ID string (as subpart of process) for a given message exchange; this ID can then be looked up in the idempotent repository (external_call table) to see if it has been before; if it has the message is consumed; if its not then the message is processed and the ID is added to the repository. Thanks to that each process can be divided into independent parts which have own status (OK, FAILED). During next iteration the message can be processed from the through point in which message was not processed correctly (with error). You can choose whether the message will be processed from scratch or from the point where message processing failed.

Cancel next processing of message - CANCEL

Link: Admin - Message operation (*/esb/web/admin/messageOperations*)

Administrator can manually cancel next iteration of processing the message. After that the message will be in final state *CANCEL.*

Steps to solve messages with errors

Recommendation steps:

1. search detail of message and check status, error and reason description
2. if information is insufficient can search detail integration log by button
3. each message contains message (body) of request that could be useful for reproducing
4. the largest percentage of errors is due to communication with external system - wrong format of data, unexpected value, error in external system and so on - for this reason it is necessary to search request into external system sent by integration platform

Direct call to external system (only for WS communication)

Sometimes it is necessary to call external system directly and send into it specific request, e.g. repair data during standard processing.

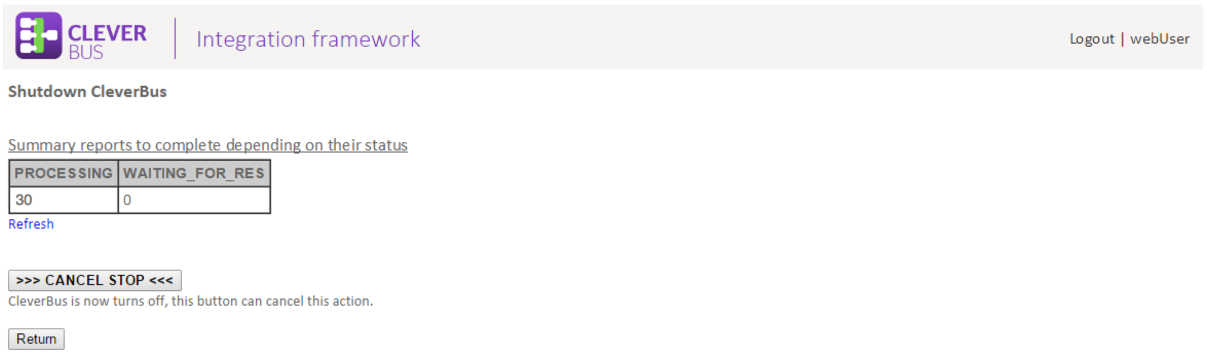
This function is mainly for production environment, because there are limited options to call external system and often it can be only through an integration platform for security reasons call.

Shutdown CleverBus

`Since version 0.4`

Stopping CleverBus is useful function for correctly shutdown when integration platform starts to reject new asynchronous messages. Current processing messages are processed (message in status *PROCESSING* and *WAITING_FOR_RES*)

This function correctly stop CleverBus for example for upgrade reason. If CleverBus is configured as clustered environment we can upgrade each node separately without stop whole solution (under certain conditions).



Shutdown CleverBus

Summary reports to complete depending on their status

PROCESSING	WAITING_FOR_RES
30	0

CleverBus is now turns off, this button can cancel this action.

Request/response tracking

Since version 0.4

Description

Request/response tracking functionality allows to save internal communication between routes or communication with external systems into database.

Functionality is disabled by default - see *requestSaving.enable* parameter to enable it. There is another parameter *requestSaving.endpointFilter* that defines pattern for filtering endpoints URI which requests/response should be saved. See configuration for more details.

Look at data model for more details about *request* and *response* tables.

Implementation and limitations

Implementation is in *sc-core* module, package *org.cleverbus.core.reqres.*

Default implementation uses Camel events that has one possible disadvantage - it's necessary to join request and response together (= two Camel events) and if exchange is changed from sending request until response receive (e.g. using *wireTap*) then it's not possible to join it. But this limitation is mainly for internal communication, there is no problem with saving request/response to/from external system.

Requests/responses are saved into database, *RequestResponseService* defines contract. *RequestResponseServiceDefaultImpl* is default implementation that saves them directly to DB in synchronous manner.

CleverBus: Alerts

Since version 0.4

Alerts REST API since version 2.2.1

Description

Alerts define metrics for watching database data and if any metric exceeds its limit then alert is activated and further operation can be executed. Alert metrics are also exposed via REST API, allowing the integration of CleverBus alerts with 3rd party monitoring tools. An integration with Zabbix monitoring tool is described here: Zabbix integration

Metrics are configured with the SQL query that retrieves the count of selected items and with the optional action threshold.

Examples of alerts (with follow-up operations):

- when count of failed messages for last 10 minutes exceeds 5 then send email to administrators
- when count of messages which wait for response from external system for more then 5 minutes exceeds 10 then send email to administrators

Alerts checking is scheduled operation that is determined by alerts.repeatTime configuration parameter - checking is executed every 5 minutes by default.

Alerts configuration

There are the following configuration possibilites:

- property files (default option)
- JMX

Configuration using property files

There the following property files (with increasing priority):

- *alertsCore.cfg* (in sc-core module)
- *alerts.cfg* (in sc-web-admin module)
- *alerts0.cfg* (in specific project module)

The configuration in the higher-priority files overrides the previous definitions.

Property file configuration format and example:

```
###############################################################################
#  Alerts core configuration file.
#
#   There the following property names:
#   - alerts.N.id: unique alert identification (if not defined then order number (=N) is used
instead)
#   - alerts.N.limit: limit that must be exceeded to activate alert
#   - alerts.N.sql: SQL query that returns count of items for comparison with limit value
#   - [alerts.N.enabled]: if specified alert is enabled or disabled; enabled is by default
#   - [alerts.N.mail.subject]: notification (email, sms) subject; can be used Java Formatter
placeholders (%s = alert ID)
#   - [alerts.N.mail.body]: notification (email, sms) body; can be used Java Formatter
placeholders (%d = actual count, %d = limit)
#   - [alerts.N.notificationType]: type of notification:
#                                  NONE - there will be no notification when the threshold is
reached
#                                  EMAIL - notification will be always send by email
#                                  EMAIL_EXCEEDED_LIMIT - notification will be send only when
the threshold is crossed (either up or down) (default value)
#
#
###############################################################################

# checks if there is any waiting message that exceeds time limit for timeout
alerts.900.id=WAITING_MSG_ALERT
alerts.900.limit=0
alerts.900.sql=SELECT COUNT(*) FROM message WHERE state = 'WAITING_FOR_RES' AND
last_update_timestamp < (current_timestamp - interval '3600 seconds')
alerts.900.notificationType=EMAIL
```

Alerts from core module start from 900, project-specific alerts start from 0. This order number is only for better description, doesn't have influence to behaviour.

SQL queries must be defined for specific database that is use.

Configuration using JMX

JMX configuration allows change alert limits and enable/disable selected alerts.

Spring configuration

The alert system is initialized in the following sequence:

- initialization of alerts from property files
- initialization of service for executing SQL queries and checking limits
- initialization of default listener that sends email to administrators for each alert that will be

activated

[*sp_camel_services.xml*]:

```
<bean id="alertsConfiguration" class="org.cleverbus.core.alerts.AlertsPropertiesConfiguration">
<constructor-arg ref="confProperties"/>
</bean>
<bean id="alertsCheckingService" class="org.cleverbus.core.alerts.AlertsCheckingServiceDbImpl"/>
<bean class="org.cleverbus.core.alerts.EmailAlertListenerSupport"/>
```

spring-ws-servlet.xml - adding property files:

```
<bean id="confProperties"
class="org.springframework.beans.factory.config.PropertiesFactoryBean">
<property name="ignoreResourceNotFound" value="true"/>
<property name="locations">
           <list>
               <value>classpath:applicationCore.cfg</value>
               ...
               <value>classpath:alertsCore.cfg</value>
               <value>classpath:alerts.cfg</value>
               <value>classpath:alerts0.cfg</value>
   </list>
</property>
</bean>
```

Reaction to alerts

There are listeners *org.cleverbus.spi.alerts.AlertListener* which are called when specified alert is activated.

There is default *org.cleverbus.core.alerts.EmailAlertListenerSupport* implementation that sends email notifications to admin emails.

If you want to implement more actions and you can implement *AlertListener *or extends default implementation *EmailAlertListenerSupport.*

REST API

Alerts definition and values are by default exposed via REST API.

Definition of all configured alerts in JSON format is available at the following URL:

```
https://<cleverbus server>/esb/rpc/v1/alert/getAll
```

The computed value of the specific alert item can be retrieved from this URL:

```
https://<cleverbus server>/esb/rpc/v1/alert/alertCalculation/<alert ID>
```

CleverBus Alerts: Zabbix Integration

Description

Cleverbus' Alerts functionality can be easily integrated with 3rd party monitoring tools via REST API. As an example, this article describes the integration with opensource monitoring and alerting tool Zabbix

The integration with Zabbix can be realized using the "User parameters" functionality, available from the Zabbix 2.0+ (https://www.zabbix.com/documentation/2.0/manual/config/items/userparameters):

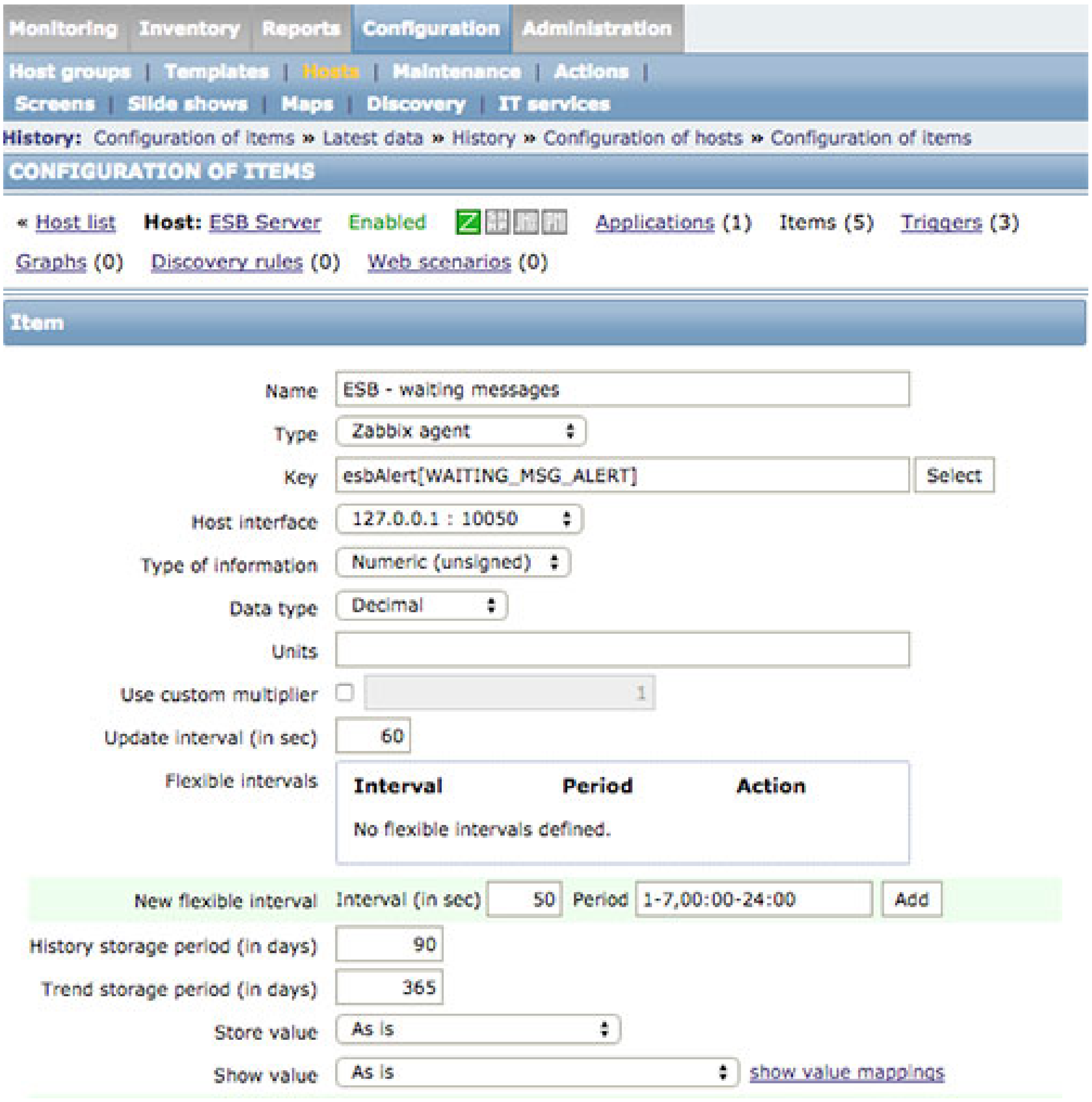
- Prepare the script that will process the ESB alert data (returned in JSON format) and output the value that will be tracked by Zabbix. The following getEsbAlert.py Python script can be used for this (do not forget to set <esb server>, <api user> and <api password> parameters):

```
#!/usr/bin/python
import urllib2, json, sys

baseUrl = "https://<esb server>/esb/rpc/v1/alertCalculation/"
username = "<api user>"
password = "<api password>"

alert_key = sys.argv[1]
url = (baseUrl + "{}").format(alert_key)

password_mgr = urllib2.HTTPPasswordMgrWithDefaultRealm()
password_mgr.add_password(None, url, username, password)
handler = urllib2.HTTPBasicAuthHandler(password_mgr)
opener = urllib2.build_opener(handler)
response = opener.open(url)

data = json.load(response)
print data["count"]
```

- Add an user parameter definition to the Zabbix Agent configuration:

```
UserParameter=esbAlert[*],/usr/local/bin/getEsbAlert.py $1
```

- Create a new Zabbix item with key esbAlert[<alert_id>]:


Monitoring
Inventory
Reports
Configuration
Administration
Host groups | Templates | Hosts | Maintenance | Actions |
Screens | Slide shows | Maps | Discovery | IT services

History: Configuration of items » Latest data » History » Configuration of hosts » Configuration of items

CONFIGURATION OF ITEMS

« Host list **Host:** ESB Server Enabled Applications (1) Items (5) Triggers (3) Graphs (0) Discovery rules (0) Web scenarios (0)

Item

Name	ESB - waiting messages
Type	Zabbix agent
Key	esbAlert[WAITING_MSG_ALERT] Select
Host interface	127.0.0.1 : 10050
Type of information	Numeric (unsigned)
Data type	Decimal
Units	
Use custom multiplier	☐ 1
Update interval (in sec)	60
Flexible intervals	**Interval** **Period** **Action** No flexible intervals defined.
New flexible interval	Interval (in sec) 50 Period 1-7,00:00-24:00 Add
History storage period (in days)	90
Trend storage period (in days)	365
Store value	As is
Show value	As is show value mappings

- The resulting graph of values can be viewed at Monitoring > Latest data > Graph

- Basic e-mail alerts can be configured directly within Cleverbus (see Alerts, more complex scenarios can be configured using Zabbix Trigger functionality)

CleverBus: CleverBus Management API

- externalsystem
- plannedoutages

CleverBus Management API: externalsystem

Operation: findAll (GET)

This method returns all external systems defined in the class implementing ExternalSystemProvider interface.

URI: /v1/externalsystem/findAll/ (GET)

Input params

N/A

Output params

Collection of system names

```
[
{"name":"ESB"},
{"name":"CRM"},
{"name":"BILLING"}
]
```

CleverBus Management API: plannedoutages

Retrieve an outage

Retrieves the speficic outage, identified by an unique id

URI: /v1/plannedoutagesystem/{id}
Method: GET

Input

{id}: (URL) numerical ID of the outage instance.

Output

Outage object:

(long) id: numerical ID of the outage instance
(string) externalSystemCode: code of the system being affected by the outage (dateTime) startOutageTimestamp: start of the outage (dateTime) endOutageTimestamp: end of the outage (string) description: Outage description

Example

```
{
"id":2,
"externalSystemCode":"CRM",
"startOutageTimestamp":"2015-06-29T23:05:00.000Z",
"endOutageTimestamp":"2015-06-30T19:45:00.000Z",
"description": "HW upgrade"
}
```

Delete an outage

Deletes the outage identified by an unique id

URI: /v1/plannedoutagesystem/{id}
Method: DELETE

Input

{id}: (URL) numerical ID of the outage instance.

Output

N/A

Update an outage

Updates the parameters of a specific outage.

URI: /v1/plannedoutagesystem/
Method: PUT

Input

Outage object - see above for details

Output

N/A

Create an outage

Creates a new outage and returns its outage ID.

URI: /v1/plannedoutagesystem/
Method: POST

Input

Outage object - see above for details

Output

numerical ID of the new outage instance.

Find all outages

Retrieve the list of all defined outages.

URI: /v1/plannedoutagesystem/findAll
Method: GET

Input

N/A

Output

Array of outage objects. See above for the description of outage object's fields.

```
[
{
"id":2,
"externalSystemCode":"CRM",
"startOutageTimestamp":"2015-06-29T23:05:00.000Z",
"endOutageTimestamp":"2015-06-30T19:45:00.000Z",
"description": "HW upgrade"
},
{
"id":3,
"externalSystemCode":"BILLING",
"startOutageTimestamp":"2015-06-29T23:05:00.000Z",
"endOutageTimestamp":"2015-06-30T19:45:00.000Z",
"description": "HW upgrade"
}
]
```

Find conflicting outage

For a given outage it verifies whether it collides with some already defined outage for the same system. Used eg. in GUI as a validator during the definition of a new outage.

URI: /v1/plannedoutagesystem/hasSystemOutage
Method: POST

Input

Outage object - see above for details

Output

TRUE: The

Error handling

Basic error handling concept is well-documented in Apache Camel.

Exceptions hierarchy

All CleverBus exceptions are in package *org.cleverbus.api.exception:*

- *IntegrationException*: parent Exception for all CleverBus exceptions
    - *LockFailureException*: unsuccessful getting lock for the record from DB (most often it means that another process was faster and acquired lock before)
    - *MultipleDataFoundException*: one record was expected but more records were found
    - *NoDataFoundException*: at least one records was expected but no record was found
    - *ThrottlingExceededException*: this exception is thrown when throttling limits were exceeded
    - *StoppingException*: when CleverBus is in stopping mode then this exception will be thrown when new requests arrive
    - *ValidationIntegrationException*: input data are not valid
        - *IllegalDataException*: wrong data

How it works?

Basic algorithm of error handling is implemented in parent class of routes - *org.cleverbus.api.route.AbstractBasicRoute*.

Is message asynchronous?

If **yes** then next processing steps are determined according to exception/error type:

- there is **error that can't be resolved/repaired during next tries** (e.g. wrong input data, wrong data in database etc.). Message state is changed to *FAILED* state because next processing doesn't have sense. Message processing is redirected to URI: *AsynchConstants.URI_ERROR_FATAL*
- there is **temporary error/exception where is chance to resolve/repair it in next tries** (e.g. external system is unavailable, error because of concurrent message procesing etc.). Message state is changed to *PARTLY_FAILED* state. Message processing is redirected to URI: *AsynchConstants.URI_ERROR_HANDLING* - message will wait in this state for specified interval and then starts processing again.

Example of error handling in communication via Web Services with external system:

asynchronous error handling

```
private static final Class[] FATAL_EXCEPTIONS = new Class[] {AlreadyExistsException.class,
   ValidityMismatchException.class, ChargingKeyNotFoundException.class,
NonExistingProductOfferingException.class, IllegalOperationException.class};
private static final Class[] NEXT_HANDLING_EXCEPTIONS = new Class[]
{CustomerNotFoundException.class,
   CustomerAccountNotFoundException.class};



.doTry()
   .to(getBillingUri())
   // explicitly converts to UTF-8
   .convertBodyTo(String.class, "UTF-8")
   // XML -> specific payload implementation of child for error check
   .unmarshal(getUnmarshalDataFormat())
.doCatch(WebServiceIOException.class)
   .setProperty(ExceptionTranslator.EXCEPTION_ERROR_CODE, constant(ErrorEnum.E600))
   .to(AsynchConstants.URI_ERROR_HANDLING)
   // we handle all exceptions in the same way there are two big catches here
.doCatch(NEXT_HANDLING_EXCEPTIONS)
   .setProperty(ExceptionTranslator.EXCEPTION_ERROR_CODE, constant(ErrorEnum.E602))
   .to(AsynchConstants.URI_ERROR_HANDLING)
.doCatch(FATAL_EXCEPTIONS)
   .setProperty(ExceptionTranslator.EXCEPTION_ERROR_CODE, constant(ErrorEnum.E601))
   .to(AsynchConstants.URI_ERROR_FATAL)
.end();
```

If **not** (=message is synchronous) then error handling is redirected to URL: *AsynchConstants.URI_EX_TRANSLATION* and then to *ExceptionTranslator.class*. Exception is propagated back to source system.

synchronous error handling

```
.onException(WebServiceIOException.class)
   .handled(true)
   .setProperty(ExceptionTranslator.EXCEPTION_ERROR_CODE, constant(ErrorEnum.E600))
   .process(ExceptionTranslator.getInstance())
   .end()
.onException(FATAL_EXCEPTIONS)
   .handled(true)
   .setProperty(ExceptionTranslator.EXCEPTION_ERROR_CODE, constant(ErrorEnum.E601))
   .process(ExceptionTranslator.getInstance())
   .end()
.onException(NEXT_HANDLING_EXCEPTIONS)
   .handled(true)
   .setProperty(ExceptionTranslator.EXCEPTION_ERROR_CODE, constant(ErrorEnum.E602))
   .process(ExceptionTranslator.getInstance())
   .end()

.to(getSapUri())
// explicitly converts to UTF-8
.convertBodyTo(String.class, "UTF-8")
// XML -> specific payload implementation of child for error check
.unmarshal(getUnmarshalDataFormat())
```

Custom error processing

Error handling concept in described in Apache Camel where you can find more details.

In common there are two types of error handling - *routes specific* and *global*, there are two ways how to catch exceptions - with *Exception Clause* or with *DoTry Clause*.

Mapping exceptions in WSDL

WSDL contract allows to define exceptions (aka faults) directly in *operation* definition.

```
 <wsdl:operation name="createSubscriber">
           <wsdl:input message="tns:createSubscriber" name="createSubscriber">
           </wsdl:input>
           <wsdl:output message="tns:createSubscriberResponse" name="createSubscriberResponse">
           </wsdl:output>
           <wsdl:fault message="tns:CustomerAccountNotFoundException"
name="CustomerAccountNotFoundException">
           </wsdl:fault>
           <wsdl:fault message="tns:CustomerNotFoundException" name="CustomerNotFoundException">
```

```
            </wsdl:fault>
            <wsdl:fault message="tns:AlreadyExistsException" name="AlreadyExistsException">
            </wsdl:fault>
        </wsdl:operation>
```

Example of fault response with exception detail:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
        <soap:Body>
            <soap:Fault>
                <faultcode>soap:Server</faultcode>
                <faultstring>Bussiness violation</faultstring>
                <detail>
                    <ns2:ValidityMismatch xmlns:ns2="http://ws.lbss.com/">
                        <message>
                       Product offering with externalNo:-1 is not valid on date:Sat May 25
00:00:00 CEST 2013
                        </message>
                    </ns2:ValidityMismatch>
                </detail>
            </soap:Fault>
        </soap:Body>
</soap:Envelope>
```

There is helper parent class *AbstractSoapExceptionFilter* for mapping SOAP fault exceptions (*SoapFaultClientException*) to internal Java exceptions from WSDL contract.

Example of *CrmSoapExceptionFilter*:

```
.onException(WebServiceIOException.class)
   .handled(true)
   .setProperty(AsynchConstants.EXCEPTION_ERROR_CODE, constant(ErrorEnum.E403))
   .process(ExceptionTranslator.getInstance())
   .end()
.onException(SoapFaultClientException.class)
   .handled(true)
   .process(new CrmSoapExceptionFilter(false))
   .end()
```

Configuration

Asynchronous algorithm configuration

CleverBus contains two configuration files:

- ***applicationCore.cfg*** - file placed in *core* module, contains basic configuration parameters
- ***application.cfg*** - file placed in *web-admin* module, contains web and project specific parameters

Common recommendations for configuration:

- set email addresses of administrators (*admin.mail*) because CleverBus sends notifications to these addresses, for example when
  - message processing failed
  - if message waits for response from external system for more then defined interval
- set username and password to access (web) services - parameters *ws.user* a *ws.password* in *application.cfg*
- path to log folder (*log.folder.path* in *application.cfg*)

applicationCore.cfg

Parameter	**Default value**	**Description**
asynch.countPartlyFailsBeforeFailed	3	Count of unsuccessful tries of message processing before message will be marked as completely *FAILED* Time interval between tries is defined by *asynch.partlyFailedInterval*.
asynch.repairRepeatTime	300	Max. interval in seconds how long can be message being processed, in other words how long can be in *PROCESSING* state. If message is still in processing then repair process (*org.cleverbus.core.common.asynch.repair.RepairProcessingMsgRoute*) will be started and the message state will be changed to *PARTLY_FAILED*. This parameter is not only for messages themselves but also for external calls and confirmations.
asynch.concurrentConsumers	30	Input asynchronous messages are waiting for processing in the priority queue. This parameter determines how many concurrent consumers (=threads) can take message from the queue and start processing. In other words this parameter detemines how many concurrent messages can be processed. For more informacetion see http://camel.apache.org/seda.html, parameter *concurrentConsumers*

asynch.partlyFailedRepeatTime	60	How often to run process (*org.cleverbus.core.common.asynch.queue.PartlyFailedMessagesPoolRoute*) for pooling *PARTLY_FAILED* messages (in seconds). This parameter is relevant to *asynch.partlyFailedInterval*.
asynch.partlyFailedInterval	60	Interval (in seconds) between two tries of *PARTLY_FAILED* messages. When this interval expires then can be message be processed again. This parameter is relevant to *asynch.partlyFailedRepeatTime*.
asynch.confirmation.failedLimit	3	Maximum count of confirmation fails when will finish further processing of confirmation, confirmation fails.
asynch.partlyFailedRepeatTime	60	How often to run process (*org.cleverbus.core.common.asynch.confirm.ConfirmationsPoolRoute*) for pooling failed confirmations (in seconds) This parameter is relevant to *asynch.confirmation.interval* .
asynch.confirmation.interval	60	Interval (in seconds) between two tries of failed confirmations. This parameter is relevant to *asynch.confirmation.repeatTime* .
asynch.waitForResponse.timeout	3600	**Parameter was removed in version 0.4 because new similar functionality was added - Alerts.**
asynch.externalCall.skipUriPattern		Regular expression that defines URIs which will be ignored by extcall component. Useful when you want to skip communication with an external system.
asynch.postponedInterval	5	Interval (in seconds) after that can be postponed message processed again.
asynch.postponedIntervalWhenFailed	300	Interval (in seconds) after that postponed messages will fail. See Guaranteed message processing order functionality for more details. Since version 0.4

Miscellaneous configuration

Parameter	Default value	Description
mail.admin		Administrator email(s) If more emails, then separated them with semicolon if empty then email won't be sent.
mail.from	CleverBus integration platform	Email address FROM for sending emails
mail.smtp.server	localhost	SMTP server for sending emails
dir.temp		Directory for storing temporary files, related to DefaultFileRepository
dir.fileRepository		File repository directory where files will be stored, related to DefaultFileRepository

contextCall.localhostUri	http://localhost:8080	URI of this localhost application, including port number. Related to external call in Admin GUI
disable.throttling	false	True for disabling throttling at all. See throttling component.
endpoints.includePattern	^(spring-ws|servlet).*$	Pattern for filtering endpoints URI - only whose URIs will match specified pattern will be returned, related to endpoints overview .
requestSaving.enable	false	True for enabling saving requests/responses for filtered endpoints URI.
requestSaving.endpointFilter	^(spring-ws|servlet).*$	Pattern for filtering endpoints URI which requests/response should be saved.
alerts.repeatTime	300	How often to run checking of alerts (in seconds) This parameter is enabled from 0.4 version.

application.cfg

Parameter	**Default value**	**Description**
db.driver	org.h2.Driver	Driver class name
db.url	jdbc:h2:mem:cleverBusDB	Database URL
db.username	sa	Database username
db.password		Database password
ws.user	wsUser	Username for accessing web services (Spring security configuration in *rootSecurity.xml*).
ws.password	wsPassword	Password for accessing web services (Spring security configuration in *rootSecurity.xml*).
web.user	webUser	Username for accessing web admin (Spring security configuration in *rootSecurity.xml*).

web.password	webPassword	Password for accessing web admin (Spring security configuration in *rootSecurity.xml*).
monitoring.user	monUser	Username for accessing JavaMelody tool (Spring security configuration in *rootSecurity.xml*).
monitoring.password	monPassword	Password for accessing JavaMelody tool (Spring security configuration in *rootSecurity.xml*).
log.folder.path	${log.folder}, value is from Maven profile	Path to folder with application logs, used in logs searching in admin GUI .
log.file.pattern	`(^.*\\.log$\|^.*\\.log\\.2\\d{3}-(0[1-9]\|1[0-9])-[0\|1\|2\|3]\\d?_\\d*\\.gz$)`	Defines format of log file names (include filter) which will be taking into logs searching in admin GUI. Since version 0.4

Configuration hiearchy

There is the following configuration files hierarchy that determines processing order:

```
applicationCore.cfg -> application.cfg -> application0.cfg
```

In other words parameters defined in last file (*application0.cfg*) have higher priority then those defined in *applicationCore.cfg* (where there are default values).

Throttling

Throttling functionality is implemented by throttling component where you can find configuration description.

Alerts

Alerts have own configuration files.

Restrict Spring bean inicialization

Since version 0.2

If there are many Spring beans with route definitions then startup time for loading ESB application can be quite long. This functionality enables to **include** only those Spring beans which should be initialized or **exclude** those Spring beans which we don't want to initialize.

It can be handy during development because it's possible via system properties to restrict set of Spring beans (=Camel routes) which will be initialized.

There are two handy classes for these purposes: *org.cleverbus.common.spring.SystemIncludeRegexPatternTypeFilter* and *org.cleverbus.common.spring.SystemExcludeRegexPatternTypeFilter*

Both type filters are handled by system or environment property. *springIncludePattern* for *SystemIncludeRegexPatternTypeFilter* class, *springExcludePattern* for *SystemExcludeRegexPatternTypeFilter* class. System property has higher priority.

Example of system property definition for exclude filter (excludes all Spring beans under *org.cleverbusu.modules* package and sub-packages):

```
-DspringExcludePattern=org\.cleverbus\.modules\..*
```

Example of Spring configuration with custom type filters:

```
<context:component-scan base-package="org.cleverbus.core, org.cleverbus.admin.routes,
org.cleverbus.modules" use-default-filters="false">
<context:include-filter type="custom"
expression="org.cleverbus.common.spring.SystemIncludeRegexPatternTypeFilter"/>
<context:exclude-filter type="custom"
expression="org.cleverbus.common.spring.SystemExcludeRegexPatternTypeFilter"/>
</context:component-scan>
```

Include filters are applied after exclude filters.

Configuration checking

Since version 0.4

Configuration checker (*org.cleverbus.core.conf.ConfigurationChecker*) enables to check selected configuration parameters during application start:

- *directCall.localhostUri* - checks URI availability (must be explicitly enabled)
- *endpoints.includePattern, requestSaving.endpointFilter* - checks regular expressions

You can implement your own checking functionality by *org.cleverbus.core.conf.ConfCheck* (since version 1.1).

Architecture

Main third party components

CleverBus extends Apache Camel and therefore base architecture came up from Apache Camel.

- Spring Framework
- Spring Security solves security issues
- Spring Web Services is main component for web services communication
- Hibernate for persistence implementation
- PostgreSQL database
- Apache Tomcat application server

CleverBus is technologically neutral to operation system, application server and database. PostgreSQL or Tomcat is one possible choice only.

Asynchronous messages

Receive asynchronous message

Receive asynchronous messages and storing them to DB:

- all essential things are solved in *AsynchInMessageRoute* (trace header recognition, create message, persist to DB, exception handling)
- use *AsynchRouteBuilder *to implement inbound asynchronous message

AsynchRouteBuilder is since 0.2 version

```
 /**
 * Route for asynchronous <strong>asyncHello</strong> input operation.
 * <p/>
 * Prerequisite: none
 * <p/>
 * Output: {@link AsyncHelloResponse}
 */
private void createRouteForAsyncHelloRouteIn() throws FailedToCreateRouteException {
    Namespaces ns = new Namespaces("h", SyncHelloRoute.HELLO_SERVICE_NS);

    // note: mandatory parameters are set already in XSD, this validation is extra
    XPathValidator validator1 = new XPathValidator("/h:asyncHelloRequest", ns, "h:name1");

    // note: mandatory parameters are set already in XSD, this validation is extra
    XPathValidator validator2 = new XPathValidator("/h:asyncHelloRequest", ns, "h:name2");

    // note: only shows using but without any influence in this case
    Expression nameExpr = xpath("/h:asyncHelloRequest/h:name").namespaces(ns).stringResult();

    AsynchRouteBuilder.newInstance(ServiceEnum.HELLO, OPERATION_NAME,
            getInWsUri(new QName(SyncHelloRoute.HELLO_SERVICE_NS, "asyncHelloRequest")),
            new AsynchResponseProcessor() {
                @Override
                protected Object setCallbackResponse(CallbackResponse callbackResponse) {
                    AsyncHelloResponse res = new AsyncHelloResponse();
                    res.setConfirmAsyncHello(callbackResponse);
                    return res;
```

```
                        }
                    }, jaxb(AsyncHelloResponse.class))
                    .withValidator(validator1, validator2)
                    .withObjectIdExpr(nameExpr)
                    .build(this);
    }
```

AsynchRouteBuilder creates uniform design of asynchronous routes (processes) with the following processing steps (low-level approach):

- settings the name of source service and operation. This information (based name conventions) are necessary to resolving of start endpoint to process asynchronous part.
- **execution of general validation (check required values)**. If a validation error occurs then CleverBus will throw *ValidationIntegrationException* or *org.apache.camel.ValidationException* (from Apache Camel).
- redirection to "*to(AsynchConstants.URI_ASYNCH_IN_MSG)*"
- using *AsynchResponseProcessor *- checking whether created message was successfully persisted or some error did not occur, after that correct response will be produced.

Hereby accepting an asynchronous message ends, is persisted in internal database for further processing and CleverBus sent confirmation to external system that message was adopted.

Asynchronous message processing

- at first incoming asynchronous message is persisted and almost immediately is stored into queue for further processing. New inbound messages are not queried from database and therefore are not locked for the specific node of cluster. CleverBus pulls only messages in *PARTLY_FAILED* and *POSTPONED* state - messages which failed or were postponed.
- messages are stored in SEDA queue. To version 0.4 included there is classic FIFO queue where messages were processed in the order in which they were inserted into the queue. Since version 0.4 this behavior is implemented by priority queue where new messages are processed earlier than postponed messages or in next attempt of processing.
- message is dynamically routed into start endpoint. Expected URI of this endpoint has to match with this format:
  `"direct:SERVICE_OPERATION_OUTROUTE"`
  where *SERVICE* is value from enum interface implementation *ServiceEnum*, *OPERATION* is name of operation and *OUTROUTE* is const AbstractBasicRoute#OUT_ROUTE_SUFFIX. Service and operation names are values configured via *AsynchRouteBuilder*, see also *AsynchConstants.SERVICE_HEADER* and *AsynchConstants.OPERATION_HEADER*
- main algorithm for processing of asynchronous messages you can find in class *AsynchMessageRoute*

Implementation of asynchronous route

Implementation of route for asynchronous message is the same as synchronous message but there are some differences which must be followed:

- header (Camel header) *AsynchConstants.MSG_HEADER* contains entity of *Message*, body contains storable (serialized, marshalled) payload of message (e.g. for communication via SOAP it will be XML of request) of original message
- **for each external system call (for example only one system with multiple calls) it must check duplicate calls**. One asynchronous message (=one correlationID) can be processed repeatedly - for example: to create subscriber in MVNOsolution the Billing and MNO systems have to be called. If the Billing system was called successfully but MNO failed the message is persisted with PARTLY_FAILED status and in future will be reprocessed (redelivered). When this message will be reprocessed, successfully calls have to be skipped.
    - **important**: asynchronous messages are processed repeatedly (number of attempts and how often are configurable values)
- if processing of message is simulated - fake processing ("blank" = unsuccessfully processing is not error > failed count must not be incremented) you have to setup properly header *AsynchConstants.NO_EFFECT_PROCESS_HEADER* as *true* value

Exceptions

- if in the frame of process an external system is called so this call can be unsuccessfully - either some expected error is thrown (= exception declared in operation) or another exception is thrown as error during processing message. The type of error is "business" exception (e.g. value of invoice is negatively and it does not make sense or we want to create customer in external system which already exists), second type is internal error in external system, where it is necessary to try to call external system for some time. But if this error is expected "business" exception so process should remember these exceptions for confirmation scenario to system that invokes this process.
    - **if some expected exception occurs so has to be catched and stored into property** with name that will end on** ***AsynchConstants.BUSINESS_ERROR_PROP_SUFFIX* (there may be more exceptions, and each of them will be saved as a separate value in property under name with appropriate suffix)
- **during processing of asynchronous message it is possible to call more external systems. The order of calling not has to be randomly therefore if any calling of external system fails, a error is thrown and process is stopped.** After some time (configurable) CleverBus will pick the message with *MessagePollExecutor *from DB and the message is processed again.
- **validate exceptions (=*org.apache.camel.ValidationException*) are resolved as**

business exceptions, so if this error occurs processing of message ends with status *FAILED*

- each call which defines route, is extended from *AbstractBasicRoute* where is basic mechanism of error handling implemented, see Error handling
- exceptions can be catched and processed immediately in the route or can be propagated to superior route, i.e. route whence the route was called - it is solved by *errorHandler(noErrorHandler())*. More information you can find in Apache Camel documentation - Error handling in Camel

Transferring state among attempts

- sometimes is necessary to transfer stateful information of message among individual attempts, for example information, which IDs of customer collection were processed
- for these purposes there is property *AsynchConstants.CUSTOM_DATA_PROP*, which can contain random string data (e.g. map of string and so on). The value of this property is at the end of message processing included in message entity: *Message#getCustomData*

Check obsolete messages in the queue

- control of obsolete messages is solved by extcall component
- the messages that failed the first time to process and amends existing data, it is necessary to check whether the data to be processed again. You can imagine that new message for same operation and same entity is received but was processed.
  ```
  MSG1 setCustomer(externalCustomerId=5) OK
  MSG2 setCustomer(externalCustomerId=5) PARTLY_FAILED
  MSG3 setCustomer(externalCustomerId=5) OK
  ```
  Message *MSG2* must not be further processed because there exists the message *MSG3*, which is newer and changes the same entity with same "object ID".
- **to make it possible to centrally check it so each entity has a specific unique identifier which we call "object ID". This identifier is mandatory and has to be set during processing inbound message via *AsynchConstants.OBJECT_ID_HEADER header.***
  ```
  .setHeader(AsynchConstants.OBJECT_ID_HEADER,
  ns.xpath("/cus:setCustomerRequest/cus:customer/cus1:externalCustomerID", String.class))
  ```
- for accurate entity identification CleverBus uses (by default) the name of operation and object ID, but sometimes it is not sufficient because for example change of customer is contained in several operations. To change this behaviour is necessary to use header value *AsynchConstants.ENTITY_TYPE_HEADER*, which is then used instead of the name of the operation.
  - Example: we have two operations *setCustomerExt* and *createCustomerExtAll*, which change customer. In this scenario the name of operation is not sufficient and therefore we use ENTITY_TYPE_HEADER.

- if CleverBus evaluates that message is obsolete then will have new status *SKIPPED* and hereafter already will not be processed
- this is not necessary to check for messages where new objects are created

Processing of message by splitter pattern to child (partial) messages

- if a message is too complex to process as altogether it is recommended (appropriate solution) to split into small child messages (partial messages). Main (parent) message will be successfully processed when all her child messages will be also successfully processed. Conversely, if any partial message will in *FAILED* status, then the main message will in *FAILED* status.
- **to split message into partial messages use *MessageSplitter***, where is necessary to implement *getChildMessages* method*:*

```
/**
* Gets child messages for next processing.
* Order of child messages in the list determines order of first synchronous processing.
*
* @param parentMsg the parent message
* @param body the exchange body
* @return list of child messages
*/
protected abstract List<ChildMessage> getChildMessages(Message parentMsg, Object body);
```

Implementation of *MessageSplitter* must be as Spring bean to resolve next dependencies:

- child messages are at first attempt processed synchronously in respectively order. When any processing of child message fails then the order during next processing is not guaranteed. **Therefore for this reason it is necessary to write implementation of child messages completely independent of the order.**

Confirmation the result of processing asynchronous messages

when asynchronous message is processed (is in final status) then CleverBus can transmit information about result of processing - OK, FAILED or CANCEL final status. main interface is *ConfirmationCallback*, which has now two implementations: - *DefaultConfirmationCallback* - default behaviour (rather suitable for tests), which only logs information about result - *DelegateConfirmationCallback* - based upon source system it chooses properly implementation of *ExternalSystemConfirmation* interface*, which *as callback calls external system to confirm result.

Design of this functionality is so flexible because:

- not every system wants be informed about result of message processing
- every system can have specific requirements to confirmation (confirmation via web service,

db call and so on)

- CleverBus provides own defined WSDL asynchConfirmation-v1.0.wsdl with XSD asynchConfirmationOperations-v1.0.xsd to auto confirmation solution

Asynchronous message request/response

Message request header

All header content is under ***traceHeader*** element that contains ***traceIdentifier*** element with tracing information of incoming message.

TraceIndentifier parameters

Parameter	Type	Description	Mandatory
applicationID	string	Identification of source application.	Yes
timestamp	datetime	Timestamp of sent request.	Yes
correlationID	string	Used to track calls from the service consumer to the API. This parameter is free text, unique value (e.g. UUID), generated by the caller. It is the responsibility of the application to generate a unique value and pass it during each call to the API. Unique identifier (at least for specific application) serves for pairing asynchronous request/response messages.	Yes
processID	string	Used to track calls connected to one process (e.g. one process is "creating new customer" and it means calling creating customer, then activating him etc.) This parameter is free text, unique value (e.g. UUID), generated by the caller. It is the responsibility of the application to generate a unique value and pass it to all calls of one process.	No

SOAP example:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:com="http://cleverbss.cleverlance.com/ws/Common-v1"
xmlns:hel="http://cleverbus.cleverlance.com/ws/HelloService-v1">
  <soapenv:Header>
     <com:traceHeader>
        <com:traceIdentifier>
           <com:applicationID>APPL001</com:applicationID>
           <com:timestamp>2013-05-21T10:33:58.147+02:00</com:timestamp>
           <com:correlationID>${=java.util.UUID.randomUUID()}</com:correlationID>
           <!--Optional:-->
           <com:processID>process001</com:processID>
        </com:traceIdentifier>
```

```
      </com:traceHeader>
   </soapenv:Header>
   <soapenv:Body>
      <hel:asyncHelloRequest>
         <hel:name>CleverBus team</hel:name>
      </hel:asyncHelloRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Validation of trace identifier from trace header

Since version 0.4

It's good practice to use validation functionality of allowed values in applicationID. This value is used for example in GUI Admin, auto confirmation delegation etc. For that we only have to register custom implementation of *ExternalSystemIdentifierValidator* interface. We can have an unlimited number of these implementations.

```
@Service
public class CenExternalSystemIdentifierValidator implements TraceIdentifierValidator {
   @Override
   public boolean isValid(TraceIdentifier traceId) {
       return EnumUtils.isValidEnum(ExternalSystemEnum.class, traceId.getApplicationID());
   }
}
```

SOAP example of failed response for sync operation:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
     <SOAP-ENV:Fault>
        <faultcode>SOAP-ENV:Server</faultcode>
        <faultstring xml:lang="en">
               E120: the trace identifier does not contain allowed values
(ValidationIntegrationException: the trace identifier
               'applicationID=ERP,timestamp=2013-09-
27T10:23:34.698+02:00,correlationID=da793349-b486-489a-9180-200789b7007f,processID=process123'
is not allowed)
        </faultstring>
        <detail>
           <errorCode xmlns="http://cleverbus.cleverlance.com">E120</errorCode>
        </detail>
     </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP example of failed response for async. operation:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
     <asyncHelloResponse xmlns="http://cleverbus.cleverlance.com/ws/HelloService-v1"
xmlns:ns2="http://cleverbss.cleverlance.com/ws/Common-v1">
        <confirmAsyncHello>
           <ns2:status>FAIL</ns2:status>
           <ns2:additionalInfo>
               E120: the trace identifier does not contain allowed values
(ValidationIntegrationException: the trace identifier
               'applicationID=ERP,timestamp=2013-09-
27T10:23:34.698+02:00,correlationID=da793349-b486-489a-9180-200789b7007f,processID=process123'
is not allowed)
           </ns2:additionalInfo>
        </confirmAsyncHello>
     </asyncHelloResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Message response

There is synchronnous response that indicates if input message was correctly saved into message queue for next processing.

SOAP example of successful response:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
     <asyncHelloResponse xmlns="http://cleverbus.cleverlance.com/ws/HelloService-v1"
xmlns:ns2="http://cleverbss.cleverlance.com/ws/Common-v1">
        <confirmAsyncHello>
           <ns2:status>OK</ns2:status>
        </confirmAsyncHello>
     </asyncHelloResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP example of failed response:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
     <asyncHelloResponse xmlns="http://cleverbus.cleverlance.com/ws/HelloService-v1"
xmlns:ns2="http://cleverbss.cleverlance.com/ws/Common-v1">
        <confirmAsyncHello>
           <ns2:status>FAIL</ns2:status>
           <ns2:additionalInfo>E106: error during saving asynchronous message into storage
(RuntimeException: some error)</ns2:additionalInfo>
        </confirmAsyncHello>
     </asyncHelloResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

It's good practice to use specific response for one request and not one common response for all requests: asyncHelloResponse -> asyncHelloRequest

If error occurred during incoming message processing (*ThrottlingExceededException* and *StoppingException* at this moment) then fault response is generated (identical for synchronnous message). Fault response contains error description and error code for clear indentification.

SOAP example with fault:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
     <SOAP-ENV:Fault>
        <faultcode>SOAP-ENV:Server</faultcode>
        <faultstring xml:lang="en">E117: Access is denied - there is no required authorization
role (AccessDeniedException: Access is denied)</faultstring>
        <detail>
           <errorCode xmlns="http://cleverbus.cleverlance.com">E117</errorCode>
        </detail>
     </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Guaranteed message processing order

Since version 0.4

Description

CleverBus allows to garant processing order of messages. This functionality garants that incoming messages with the same *funnel_value* will start processing in order by *msgTimestamp* (timestamp from source system) and next message won't start before previous message isn't finished.

Guaranteed message processing order takes all the following states into consideration: *PROCESSING*, *WAITING*, *WAITING_FOR_RES*, *PARTLY_FAILED*, FAILED and *POSTPONED*. *FAILED* state is used by default but can be excluded.

It can happen that first message failed and then all next messages will wait to start processing (message state will change to *POSTPONED* again and again).There is new configuration) parameter *asynch.postponedIntervalWhenFailed* that determines interval (in seconds) after that postponed messages will fail.

Comparison to msg-funnel

Msg-funnel component filters messages in one specific place of the processing but this gauranteed processing order functionality is for wholes routes at the beginning.

There are the following types of message filtering:

- **classic funnel** (msg-funnel component) - filters processing messages (states *PROCESSING* , *WAITING*, *WAITING_FOR_RES*) by *funnel_value* in specific place of the route, most often before communication with external system to ensure that only one specific value at one time is send to it. For example telco provider can process one request of one specific subscriber (msisdn) at one moment only and we need to filter out requests to this system if there are more requests for one subscriber (msisdn).
- **classic funnel with guaranteed order** (msg-funnel component) - component has optional parameter *guaranteedOrder* to turn on guaranteed order - messages are filtered out by *funnel_value* (it's same as classic funnel)* *and also *msgTimestamp* of the message is taken into consideration. Messages must be processed at specific funnel in order by *msgTimestamp*. *PARTLY_FAILED*, *FAILED* and *POSTPONED* states are used together with processing states
- **guaranteed message processing order** - this functionality that is about whole routes

How to use it?

Use *org.cleverbus.api.asynch.AsynchRouteBuilder* with methods *withGuaranteedOrder()* or *withGuaranteedOrderWithoutFailed()* (see Asynchronous messages for more details) or sets *AsynchConstants.GUARANTEED_ORDER_HEADER* or *AsynchConstants.EXCLUDE_FAILED_HEADER* to *true* when you create route definition manually.

```
private void createRouteForAsyncHelloRouteIn() throws FailedToCreateRouteException {
   Expression funnelValueExpr =
xpath("/h:asyncHelloRequest/h:name").namespaces(ns).stringResult();
   AsynchRouteBuilder.newInstance(ServiceEnum.HELLO, OPERATION_NAME,
           getInWsUri(new QName(SyncHelloRoute.HELLO_SERVICE_NS, "asyncHelloRequest")),
           new AsynchResponseProcessor() {
               @Override
               protected Object setCallbackResponse(CallbackResponse callbackResponse) {
                   AsyncHelloResponse res = new AsyncHelloResponse();
                   res.setConfirmAsyncHello(callbackResponse);
                   return res;
               }
           }, jaxb(AsyncHelloResponse.class))
           .withFunnelValue(funnelValueExpr)
           .withGuaranteedOrder()
           .build(this);
}
```

Maven and Spring

Maven modules

```
<groupId>org.cleverbus</groupId>
```

```
<artifactId>cleverbus-integration</artifactId>
```

CleverBus framework consists of the following Maven modules:

- **common**: module contains useful functions for other modules
- **core-api**: user API for writing new routes with CleverBus
- **core-spi**: interface for internal use in CleverBus components
- **core**: basic implementation module
- **test**: module with basic configuration and parent classes for tests
- **components**: CleverBus components
- **examples**: examples how to use CleverBus
- **web-admin**: admin GUI web application
- **web-admin-archetype**: Maven archetype for easy creation of new CleverBus project

```
         <dependency>
            <groupId>org.cleverbus</groupId>
            <artifactId>cleverbus-common</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>org.cleverbus</groupId>
            <artifactId>cleverbus-core-api</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>org.cleverbus</groupId>
            <artifactId>cleverbus-core-spi</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>org.cleverbus</groupId>
            <artifactId>cleverbus-core</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>org.cleverbus</groupId>
            <artifactId>cleverbus-components</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>org.cleverbus</groupId>
            <artifactId>cleverbus-test</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>org.cleverbus</groupId>
            <artifactId>cleverbus-examples</artifactId>
            <version>${project.version}</version>
        </dependency>
```

There were the following names before 1.1 version:

- groupId: *com.cleverlance.cleverbus*
- artifactIds start with *sc-* prefix

CleverBus depends on several Camel components and if you want to use another Camel component with same Camel's version then you can do it:

1. import dependencies from CleverBus

   ```
   <dependency>
   <groupId>org.cleverbus</groupId>
   <artifactId>cleverbus-integration</artifactId>
   <version>${cleverBus-version}</version>
   <scope>import</scope>
   <type>pom</type>
   </dependency>
   ```
2. use specific component which you want

   ```
   <dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-crypto</artifactId>
   </dependency>
   ```

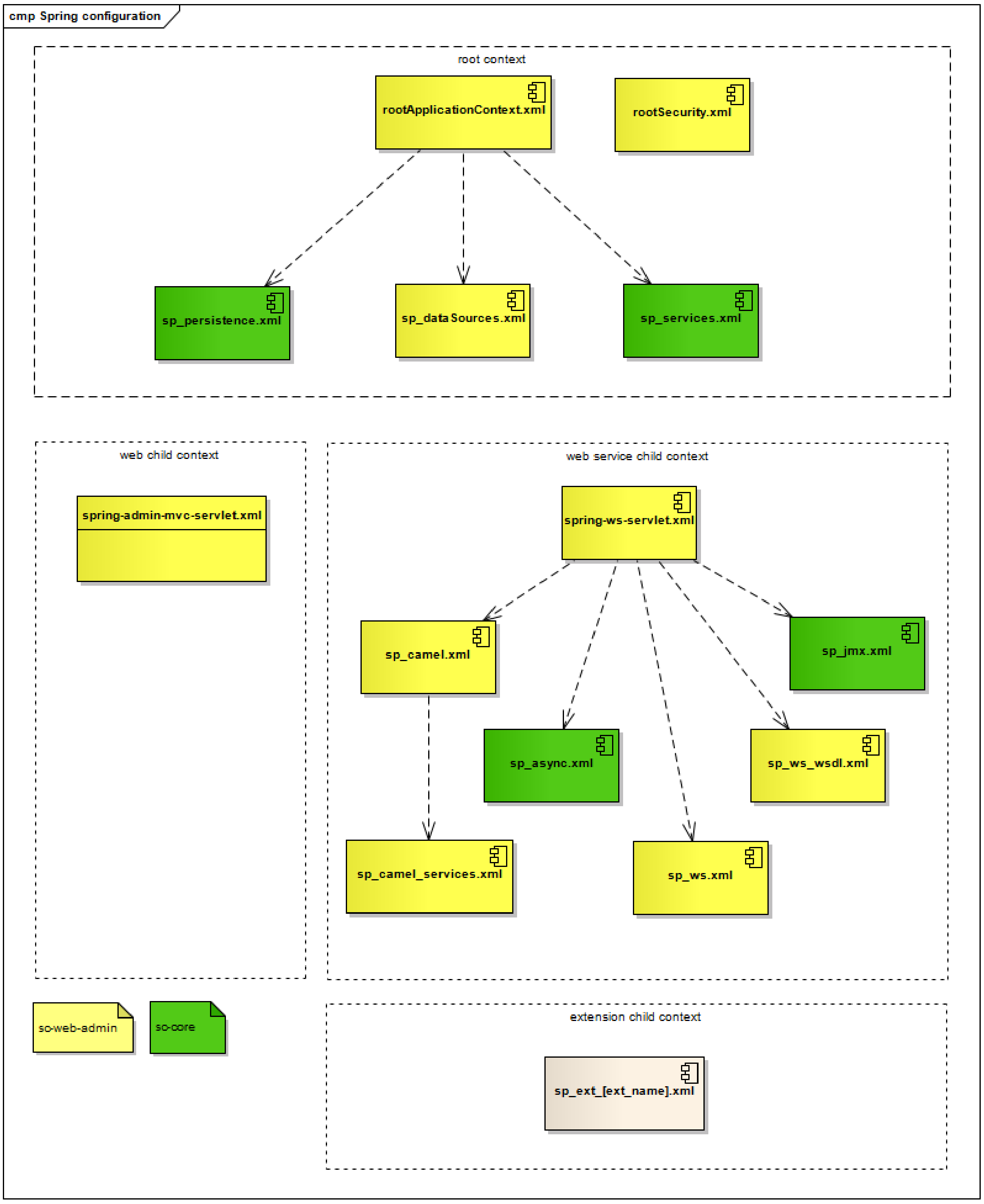
   Since version 1.1

Spring contexts

CleverBus uses and is configured by Spring framework.

There are the following **Spring contexts hierarchy** from the web application (=admin GUI) point of view:

- **root application context**; Apache Camel, database and security (Spring security) is initialized in this context
- **spring-ws**: Spring Web Services context
- **spring-admin-mvc**: Spring Web MVC context



Spring profiles

CleverBus framework uses **Spring profiles to simplify configuration for different target environments:**

- **dev (default)**: profile for development environment; this profile has disabled several functions which are enabled in production, e.g. scheduled tasks for asynchronous messages
- **prod**: profile for testing and production environment

Next Spring profiles are used for switching between databases (more in configuration file *META-*

INF/sp_dataSources.xml)

- **h2 (default):** database H2 in embedded in-memory mode, more information in chapter about H2 database
- **postgreSql**: profile for PostgreSQL database

Defaults Spring profiles are set in* web.xml*, it's possible to override them by system parameter *spring.profiles.active*:

```
  <context-param>
     <param-name>spring.profiles.default</param-name>
     <param-value>dev,h2</param-value>
 </context-param>
```

Spring configuration for tests

Maven profiles

There are also **Maven profiles** which correspond to Spring profiles. Maven profiles solve dependency to third-party libraries for specific target environment and settings of folders, logs etc. See *pom.xml* in module *sc-web-admin.*

- **esb.dev**
- **esb.prod**
- **esb.psSqlb -** profile for PostgreSQL database
- **esb.psSql.prod -** profile for PostgreSQL database but PostgreSQL libraries are *"provided"*

CleverBus compilation with Maven for local use (=dev) with PostgreSQL database:

```
mvn clean && mvn -DskipTests -Pesb.psSql package
```

Application server starts with the following system parameter *spring.profiles.active*:

```
postgreSql,dev
```

Data model

Table message

Table for storing asynchronous messages:

Attribute	Type	NULL	Description	Value description
msg_id	number	No	Unique identifier, primary key.	Automatically generated.
msg_timestamp	datetime	No	Message timestamp, generated by source system. This timestamp determines processing order of messages.	Source system, part of *TraceHeader* *TraceHeaderProcessor.TRACE_HEADER*
receive_timestamp	datetime	No	Timestamp when message arrives to integration platform, generated by CleverBus.	Automatically generated when input request arrives.
service_name	string	No	Service name, e.g. customer Implements *org.cleverbus.core.entity.ServiceExtEnum* interface	Source system *AsynchConstants.SERVICE_HEADER*
operation_name	string	No	Operation name, e.g. *createCustomer*	Source system *AsynchConstants.OPERATION_HEADER*
object_id	string	Yes	Object ID that is impacted by processing of this message. For example when this message updates customer object then it can be customer ID. This values serves for searching messages which impact identical object. When there are two messages in processing queue, both want to change identical object, then first message will be skipped and second one will be processed.	*AsynchConstants.OBJECT_ID_HEADER*
entity_type	string (enum)	Yes	Entity type (implements *org.cleverbus.api.entity.EntityTypeExtEnum* interface). In general it's enough to detect identical changed data by objectId and operation name but there are few different operations which can change the same data (e.g. setCustomer, setCustomerExt). If defined then it will be used for "obsolete operation call" detection instead of operation name.	*AsynchConstants.ENTITY_TYPE_HEADER*
correlation_id	string	No	ID generated by source system that is important for pairing request and asynchronous response in source system. Combination correlation_id and source_system is unique.	Source system, part of *TraceHeader* *TraceHeaderProcessor.TRACE_HEADER*
process_id	string	Yes	Process identifier serves for pairing messages of same process.	Source system, part of *TraceHeader* *TraceHeaderProcessor.TRACE_HEADER*

payload	string (XML)	No	Message payload	
envelope	string (XML)	No	The whole input request, e.g. SOAP envelope including headers and body	
source_system	string (enum)	No	Source system (implements *org.cleverbus.core.entity.ExternalSystemExtEnum* interface)	Source system, part of *TraceHeader* (applicationID) *TraceHeaderProcessor.TRACE_HEADER*
state	string (enum)	No	Message state; enum value: ● *NEW*: new saved message ● *PROCESSING*: message is being processed ● *OK*: successfully processed message ● *PARTLY_FAILED*: last processing ended with error, there will be next try ● *FAILED*: finally failed message, no next processing ● *WAITING*: parent message that waits for child messages ● *WAITING_FOR_RES*: message that waits for confirmation/response from external system ● *CANCEL*: message was canceled by external system or by administrator. This state isn't set by this application. ● *POSTPONED*: message is postponed because there was another message that was processed at the same time with same funnel values. See msg_states.png See overview of operations which change message state .	
failed_count	number	No	Count of unsuccessful tries. Default value is 0.	
failed_error_code	string	Yes	Error code when message processing failed (implements *org.cleverbus.core.common.exceptions.ErrorExtEnum* interface)	
failed_desc	string	Yes	Error description when message processing failed.	
last_process_timestamp	datetime	Yes	Timestamp when this message was last updated.	
custom_data	string	Yes	Field for saving custom data, suitable for sharing information between more tries of processing.	
parent_msg_id	number	Yes	Reference to parent message (parent-child binding).	Useful for parent-child binding, *ChildMessage#createMessage*

parent_binding_type	string (*SOFT*)	Yes	Binding type defines how tightly child message does influence parent message: ● *HARD*: result of child message influences result of parent message (for example when child message ends in FAILED state then parent message will in *FAILED* state too) ● *SOFT*: result of child message has no effect to parent message Useful for parent-child binding only, see asynch-child component.Since version 0.4	Useful for parent-child binding, *ChildMessage#createMessage*
funnel_value	string	Yes	Funnel value is for finding out if two or more concurrent messages impact identical target object. Funnel value can be anything, e.g. customer ID, subscriber mobile number, custom string etc. This value is for msg-funnel component use. What is difference between *object_id* and *funnel_value* ? *object_id* represents real object identifier and serves for analysis if any message should be skipped because message data are obsolete. On the other hand funnel value represents virtual value (it can be anything) and is used to ensure that there will be only one message with funnel value at the moment being processed in specific route point.	*AsynchConstants.FUNNEL_VALUE_HEADER*
funnel_component_id	string	Yes	Funnel identifier that specifies unique point during message processing. Messages are checked and filtered out to this point. This value is for msg-funnel component use. Since version 0.4	Useful for (CleverBus-components/msg-funnel)[msg-funnel] component.
guaranteed_order	boolean (*false*)	No	When this flag is true then mesage is processed in guaranteed order.Since version 0.4	*AsynchConstants.GUARANTEED_ORDER_HEADER*
exclude_failed_state	boolean (*false*)	No	Messages in *FAILED* state are taking into consideration for guaranteed order by default. If *FAILED* state should be excluded then this flag will be true. Value of this attribute has sense only when guaranteed_order is true. Since version 0.4	*AsynchConstants.EXCLUDE_FAILED_HEADER*

Table external_call

Table stores calls to external systems when extcall component is used.

Attribute	**Type**	**NULL**	**Description**
call_id	number	No	Unique identifier, primary key.

msg_id	number	No	Reference to asynchronous message.
operation_name	string	No	Operation name.
state	string (enum)	No	External call state: ● *PROCESSING*: external call is just processing ● *OK*: external call is successfully done ● *FAILED*: external call failed, try it next time ● *FAILED_END*: all confirmation tries failed, no further processing
entity_id	string	No	Entity identifier
msg_timestamp	datetime	No	Message timestamp (from source system).
creation_timestamp	datetime	No	Timestamp when this record was created.
last_update_timestamp	datetime	No	Timestamp when this record was lasttime updated.
failed_count	number	No	Count of unsuccessful external calls.

Tables request and response

From version 0.4

These tables stores requests/responses in communication with external systems.

Attribute	**Type**	**NULL**	**Description**
req_id	number	No	Unique request identifier, primary key.
msg_id	number	Yes	Binding to asynchronous message (NULL for synchronous messages).
res_join_id	string	Yes	Identifier for pairing/joining request and response together. ● It can be *Message* or correlation ID or exchange ID or some ID that is unique with *uri*. ● This attribute helps to associate response to the right request.
uri	string	No	Endpoint/target URI
req_envelope	text	No	The request content
req_timestamp	datetime	No	Timestamp when request was send to target URI.

Unique key: *uri* and *res_join_id*

Attribute	**Type**	**NULL**	**Default**	**Description**
res_id	number	No		Unique response identifier, primary key.
req_id	number	Yes		Binding to corresponding request to this response.
res_envelope	text	Yes		The response itself. If not specified then failed reason must be filled.
failed_reason	string	Yes		Reason (SOAP fault, exception, stackTrace, ...) why communication failed.
res_timestamp	datetime	No		Timestamp when response/failed reason was received back.
failed	boolean	No	false	Has been communication failed? If yes then failed reason must be filled.

msg_id	number	Yes		Binding to asynchronous message (NULL for synchronous messages).

Backups

With regards to stability of the database schema, database performance and the whole CleverBus solution is recommended to perform regular archiving of records with regards to the utilization of individual tables and the number of records to be found in the tables. As part of the solution is a distributed database procedure *archive_records* that has one input parameter. This parameter indicates the number of months, which is derived from the old record, which has to include archiving (default value is 2 months). The script can be expanded by additional tables to be archived. The actual procedure now includes database archiving tables: *message*, *external_call*, *request* and *response*. These records are stored in tables *archive_message*, *archive_external_call* , *archive_request* and *archive_response*.

Operations which change message state

The following table presents operations or actions which change message states (see Data Model with state workflow diagram):

Source state	**Target state**	**Operation/action**
NEW	PROCESSING	New request/message starts processing (*org.cleverbus.core.common.asynch.AsynchInMessageRoute*)
PROCESSING	POSTPONED	Message is being processed but there is conflict with another message with same " funnel " value.
PROCESSING	PARTLY_FAILED	Processing failed but there are next tries to finish it. ● error occured during message processing (*org.cleverbus.core.common.asynch.AsynchMessageRoute*) ● time processing exceeds limit for processing (*org.cleverbus.core.common.asynch.repair.RepairProcessingMsgRoute*)
PROCESSING	WAITING_FOR_RES	Message is being processed and waits for response from external system.
PROCESSING	WAITING	Message is being processed and waits for response from external system (valid for parent message only).
PROCESSING	OK	Message is successfully processed.
PROCESSING	FAILED	Processing of the message failed - there is no next try for processing.
POSTPONED	PROCESSING	Previous processing was postponed and started next try. *org.cleverbus.core.common.asynch.queue.PartlyFailedMessagesPoolRoute*
POSTPONED	CANCEL	Admin canceled further processing in Admin GUI
POSTPONED	FAILED	Message has been waiting for starting processing more then interval defined by *asynch.postponedIntervalWhenFailed. org.cleverbus.core.common.asynch.queue.MessagePollExecutor*
PARTLY_FAILED	PROCESSING	Previous processing was postponed and started next try. *org.cleverbus.core.common.asynch.queue.PartlyFailedMessagesPoolRoute*
PARTLY_FAILED	CANCEL	Admin canceled further processing in Admin GUI
WAITING_FOR_RES	PROCESSING	Message got response from external system and continues in processing.
WAITING_FOR_RES	CANCEL	Admin canceled further processing in Admin GUI
WAITING	FAILED	Parent message was waiting for processing of child message but at least one child message failed.
WAITING	OK	All child messages of parent message finished successfully.

FAILED	PROCESSING	Restart failed message from admin GUI .

CleverBus components

Description

Implementations of camel components are in *components* Maven module.

Components overview

Component	Description
asynch-child	This component creates new asynchronous message.
extcall	Component for wrapping external calls with checks for duplicate and outdated calls.
msg-funnel	Component for filtering concurrent asynch. messages which influence identical object.
throttling	Component for [throttling](http://en.wikipedia.org/wiki/Throttling_process_(computing)) functionality.

asynch-child

Description

Component creates new asynchronous message.

New messages can be created from asynchronous and even from synchronous messages (binding type will be *SOFT* for this case).

Current message body will be body of new asynch. message.

URI format

```
asynch-child:service:operation[?options]
```

where *service *is service name (e.g. *customer*) and *operation *is operation name (e.g. *createCustomer*).

Options

Parameter	**Default**	**Description**
correlationId	null	Unique message ID. If not defined then ID is generated with *UUID.randomUUID().*
sourceSystem	null	Source system (e.g. CRM). If not defined then default value *IP* (internal integration platform) is used.
bindingType	HARD	Binding type defines how tightly child message does influence parent message: ● *HARD*: result of child message influences result of parent message (for example when child message ends in *FAILED* state then parent message will in *FAILED* state too) ● *SOFT*: result of child message has no effect to parent message
objectId	null	Object ID which is impacted by this message. For example when we change customer object then this is customer ID.
funnelValue	null	This value is used in detection of concurrent messages which impact identical target object.

Example usage

```
asynch-child:customer:createCustomer
```

```
asynch-child:customer:createCustomer?bindingType=HARD&correlationId=566&sourceSystem=CRM
```

extcall

Description

Apache Camel component for wrapping external calls with checks for duplicate and outdated calls.

URI format

```
extcall:[keyType]:[targetURI]
```

where *keyType* can be one of:

- **message -** to generate a key based on message source system and correlation ID, effectively providing duplicate call protection, but not obsolete call protection
- **entity -** to generate a key based on message objectId property, providing both duplicate call protection and obsolete call protection
- **custom -** to use a custom key provided in the *ExtCallComponentParams.EXTERNAL_CALL_KEY* exchange property

In the first two cases (message and entity), if the *ExtCallComponentParams.EXTERNAL_CALL_KEY* exchange property is provided, it will be appended to the generated key.

By default, the *targetURI* is used as the operation. This can be changed by providing an optional *ExtCallComponentParams.EXTERNAL_CALL_OPERATION* exchange property. The *targetURI* will still be the URI that is called, if the external call is not skipped, but the duplicate/obsolete protection logic will use the *ExtCallComponentParams.EXTERNAL_CALL_OPERATION* value for checking, if the call should be made or skipped.

msg-funnel

`Since version 0.2`

Description

This component is for asynchronous messages only.

Component "funnel" is for filtering concurrent messages at specific integration point. This filtering ensures that only one message at one moment will be processed, even in garanteed order (optional choice).

Usage example: if external system can process only one request for specific customer then you need to filter outgoing requests to this system.

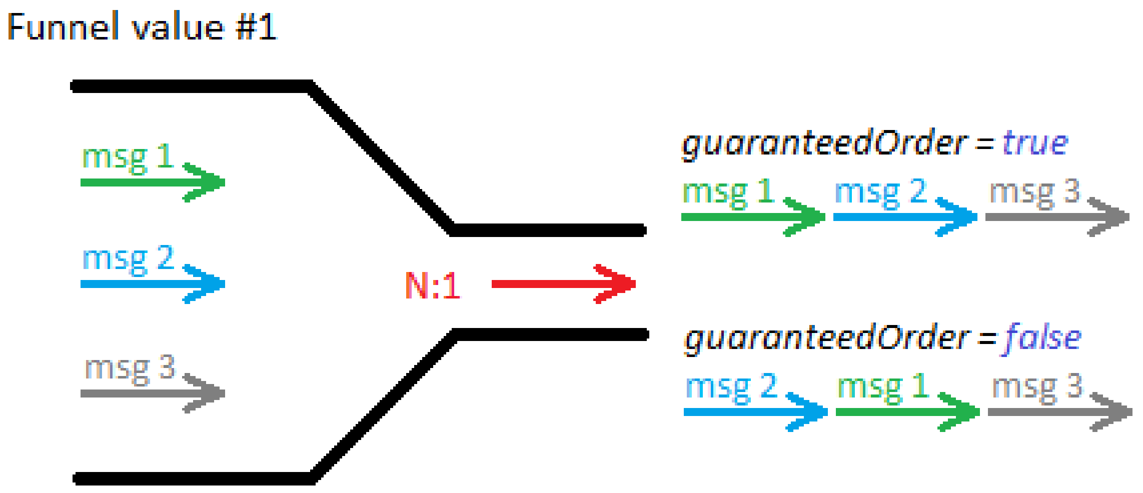
Msg-funnel component uses *funnel value* for finding out if two or more concurrent messages impact identical object. Funnel value can be anything, e.g. customer ID, subscriber mobile number, custom string etc.

Funnel value is taken from *AsynchConstants.FUNNEL_VALUE_HEADER* header value - can be set explicitly or use *org.cleverbus.api.asynch.AsynchRouteBuilder* class for simpler route implementation. Example of *AsynchRouteBuilder* usage is shown on Guaranteed message processing order page.

If there is processing message and this component detects different concurrent message(s) with the same funnel value that is also processed (message in states *PROCESSING*, *WAITING* and *WAITING_FOR_RES*) then processing of current message is postponed (changed state to *POSTPONED*). This message will stay in this state for certain moment (configuration parameter *asynch.postponedInterval*) and then will try it to process again.

Component is working in two modes (according to the parameter *guaranteedOrder*):

- **classic funnel** - component checks concurrent messages and ensures that only one message at one time will be processed. This is default mode, parameter *guaranteedOrder *is *false.*
- **guaranteed funnel** - if parameter *guaranteedOrder *is true, then component garantees processing order of message (by msgTimestamp) on top of that. Component in this mode takes into consideration messages in other states (*PARTLY_FAILED*, *FAILED* and *POSTPONED*) because it's necessary to guarantee processing order among all messages with same funnel value.



URI format

```
msg-funnel:default[?options]
```

Options

Parameter	Type	Default	Description
idleInterval	number	600	Interval in seconds how long can be message be processing (= be in state *PROCESSING*). In other words during this interval will be message considered as being processed. This is for cases when processing message can be locked during processing, forexample in communication with external system that doesn't respond (message will stay in *WAITING_FOR_RES* state)
guaranteedOrder	boolean	false	*True* if funnel component should guaranteed order of processing messages. By default funnel works with running messages (PROCESSING, WAITING, WAITING_FOR_RES* only and if it's necessary to guarantee processing order then also *PARTLY_FAILED* , *FAILED* and *POSTPONED* messages should be involved.
excludeFailedState	boolean	false	Messages in *FAILED* state are taking into consideration for guaranteed order by default. If you want to exclude this state then set this parameter to *true* . Value of this parameter has sense only when *guaranteedOrder* is *true*.

id	string		Funnel identifier that specifies unique point during message processing. Messages are checked and filtered out to this point. If not defined then identifier is derived from route ID (that is unique in one Camel context) where this component is placed. Set this parameter is necessary in practice only when more funnel components are used in one route.
funnelValue	string		Msg-funnel component uses *funnel value* for finding out if two or more concurrent messages impact identical object. Funnel value can be anything, e.g. customer ID, subscriber mobile number, custom string etc.

Example usage

```
msg-funnel:default?idleInterval=1000

msg-funnel:default?guaranteedOrder=true&id=myEndpointName

msg-funnel:default?idleInterval=100&excludeFailedState=true&id=myEndpointName
```

Example implementations

Funnel

Funnel example can be found in *org.cleverbus.modules.in.funnel.FunnelRoute*.

Input XML contains *funnelValue* and *contentValue* After XML is validated and callback response is sent back, funnel value of the route is set according to the *funnelValue* xml parameter. *contentValue* represents identification of the request.

MultiFunnel

MultiFunnel example can be found in *org.cleverbus.modules.in.funnel.MultiFunnelRoute*.

Difference between Funnel and MultiFunnel example is that MultiFunnel can utilize more than one *funnelValue*. In this example it's limited to 3, but can be changed to any positive number.

throttling

`Since version 0.3`

Description

Component for [throttling](http://en.wikipedia.org/wiki/Throttling_process_(computing)) functionality.

Throttling is functionality that checks count of input requests to integration platform and if this count exceeds defined limit then new requests are restricted.

Main goal of throttling is to limit disproportionate (and often wilful) overload of integration platform with huge amount of input requests for processing. This would have bad impact to application performance and even it can stop processing of current messages.

Throttling component counts input requests from specified source (external) system and for specific operation. If this count exceeds defined limit in defined time interval then CleverBus will start to reject new input requests (only requests from specified source system and for specified operation) - exception is thrown.

Throttling can be disabled at all by setting *disable.throttling* parameter in *application.cfg* to value *true*.

Throttling configuration

Default configuration is in *throttlingCore.cfg* file placed in *core* module. Configuration file *throttling.properties* placed in *web-admin* module has higher priority and therefore parameters in this file can override parameters from *throttlingCore.cfg* file.

throttlingCore.cfg content:

```
throttling.defaultInterval = 60
throttling.defaultLimit = 60
```

Use the following parameters for throttling configuration:

- throttling.defaultInterval: default time interval in seconds. Default value is 60.
- throttling.defaultLimit: default max. count of requests for specified time interval. Default value is 60.
- throttling.sourceSystem.operationName, where

- *sourceSystem* identifies source (callee) system. Can be used specific system names (defined with *org.cleverbus.api.entity.ExternalSystemExtEnum*) or use asterisk (*) that represents any source system. Then only requests with specific operations are checked.
- *operationName* identifies operation. Can be also used asterisk (*) that represents any operation. Then only requests from source systems are checked.

Parameter *throttling.sourceSystem.operationName* has the format *limit[/interval]*:

- *limit* specifies max. count of requests for specific combination source system vs. operation
- *interval* specifies time interval (in seconds). If not defined then value from *throttling.defaultLimit* parameter is used.

System level vs. operation level

Let's say we allow 30 messages per minute to flow through on system level, like this

```
throttling.SAP = 30
```

for input, or

```
throttling.out.SAP = 30
```

for output. If we're more restrictive on operation level output and allow only 20 messages calling operation *asyncOutage* per minute,

```
throttling.out.*.asyncOutage = 20
```

after calling operation *asyncOutage* 20 times through SAP system, this limitation will kick in first. If, on the other hand, we were to set this limit to 40 messages per minute,

```
throttling.out.*.asyncOutage = 40
```

messages would be stopped after 30th one.

Limiting **only on operation level** will leave unincluded operations unhindered.

Limiting **only on system level** will take total number of operations going through that system.

Examples:

- throttling.crm.op1 = 10 (restriction to calls of operation *op1* from *CRM* system to 10 requests for 60 seconds)
- throttling.crm.* = 10/40 (restriction to calls of any operation from *CRM* system to 10 requests for 40 seconds)
- throttling.*.sendSms = 60/30 (restriction to calls of operation *sendSms* from any system to 60 requests for 30 seconds)

Configuration via JMX

Throttling parameters is allowed to configured via JMX, for example with *jconsole* tool:

URI format

```
throttling:requestType[:operationName]
```

Options

Parameter	Description
requestType	Specifies request type • *SYNC*: synchronnous request • *ASYNC*: asynchronnous request Throttling component for asynchronnous request is used in route *AsynchInMessageRoute* for processing all input asynch. requests.
operationName	operation name, e.g. "createCustomer" (mandatory for *SYNC* request type only)

Prerequisites

Throttling component needs reference to *org.cleverbus.spi.throttling.ThrottlingProcessor* that needs (in default implementation) reference to *org.cleverbus.spi.throttling.ThrottlingConfiguration* and *org.cleverbus.spi.throttling.ThrottleCounter*.

Common Spring configuration (*sp_camel_services.xml*):

```
    <bean id="throttlingConfiguration"
class="org.cleverbus.core.throttling.ThrottlingPropertiesConfiguration">
       <constructor-arg ref="confProperties"/>
       <property name="throttlingDisabled" value="${disable.throttling}"/>
   </bean>
   <bean class="org.cleverbus.core.throttling.ThrottleCounterMemoryImpl" />
   <bean class="org.cleverbus.core.throttling.ThrottleProcessorImpl" />
```

where *confProperties* is reference to *PropertiesFactoryBean* with throttling configuration file (*spring-ws-servlet.xml*).

```
    <bean id="confProperties"
class="org.springframework.beans.factory.config.PropertiesFactoryBean">
       <property name="ignoreResourceNotFound" value="true"/>
       <property name="locations">
           <list>
               <value>classpath:applicationCore.cfg</value>
               ...
               <value>classpath:throttlingCore.cfg</value>
               <value>classpath:throttling.cfg</value>
```

```
                <value>classpath:throttling0.cfg</value>
            </list>
        </property>
    </bean>
```

Example usage

```
throttling:sync:sendSms
```

```
throttling:async:
```

Example implementations

Purpose of these two examples is to show how throttling component behaves in case of message overflow.

Request contains single parameter in its body - requestAmount. Request is then duplicated and sent to throttling component this many times.

Synchronous throttling example

Synchronous throttling example can be found in *org.cleverbus.modules.in.throttling.SyncThrottlingRoute*.

When request is received, it is unmarshaled (xml -> *SyncThrottlingRequest* class type), then goes through validation, which checks whether *requestAmount* parameter is within range of *0* and *SyncThrottlingRoute.REQUEST_AMOUNT_LIMIT*, then request is duplicated *requestAmount* of times and sent to throttling:sync: endpoint. After that response xml is formulated in *SyncThrottlingResponse* format.

Response contains original *requestAmount* if successful, *ValidationException* is thrown if request fails at validation, finally *ThrottlingExceededException* is thrown if throttling component limit is exceeded.

Asynchronous throttling example

Asynchronous throttling example can be found in *org.cleverbus.modules.in.throttling.AsyncThrottlingRoute*.

Before request in unmarshaled, it goes through xpath validation to formulate synchronous callback response. This validation is same as with synchronous example, i.e. checks whether *requestAmount* parameter is within range of *0* and *SyncThrottlingRoute.REQUEST_AMOUNT_LIMIT*. Caller then receives *OK* response on success or *FAIL* with respective exception (*AsyncThrottlingResponse* xml format).

ThrottlingExceededException is **not** thrown during this phase.

On successful validation body is unmarshaled to *AsyncThrottlingRequest* class type, request is duplicated *requestAmount* of times and sent to throttling:async: endpoind. Success or failure of this operation is logged.

Web Services

Description

CleverBus provides several tools and components for communication with external systems.

Components

- CloseableHttpComponentsMessageSender

SOAP 1.2

Since version 0.4

CleverBus supports SOAP 1.1 and 1.2 for outgoing web service communication. There are *getOutWsUri* and *getOutWsSoap12Uri* methods in *org.cleverbus.api.route.AbstractBasicRoute*.

There is the following configuration prerequisite (*sp_ws.xml*):

```
    <bean id="messageFactorySOAP11"
class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory">
       <property name="soapVersion">
           <util:constant static-field="org.springframework.ws.soap.SoapVersion.SOAP_11"/>
       </property>
   </bean>

   <bean id="messageFactorySOAP12"
class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory">
       <property name="soapVersion">
           <util:constant static-field="org.springframework.ws.soap.SoapVersion.SOAP_12"/>
       </property>
   </bean>
```

HTTP Message Sender

`From version 0.2`

Description

CleverBus provides own customized solution as *CloseableHttpComponentsMessageSender*, which uses the [Apache HttpComponents HttpClient](). Use that if you need more advanced and easy-to-use functionality (such as authentication, HTTP connection pooling, and so forth).

The following example shows how to configure this component, and to use Apache HttpClient to authenticate using HTTP authentication with connection pooling per host:

```
<bean id="billingSender"
class="org.cleverbus.core.common.ws.transport.http.CloseableHttpComponentsMessageSender">
<constructor-arg index="0" value="true"/> <!-- use Preemptive Auth-->
<property name="credentials">
 <bean class="org.apache.http.auth.UsernamePasswordCredentials">
  <constructor-arg index="0" value="${billing.user}"/>
  <constructor-arg index="1" value="${billing.password}"/>
 </bean>
</property>
<property name="maxTotalConnections" value="${billing.maxTotalConnections}"/>
<property name="defaultMaxPerHost" value="${billing.maxTotalConnections}"/>
<property name="maxConnectionsPerHost">
 <props>
  <prop key="${billing.uri}">${billing.maxTotalConnections}</prop>
 </props>
</property>
<property name="connectionTimeout" value="${billing.connectionTimeout}"/>
<property name="readTimeout" value="${billing.readTimeout}"/>
</bean>
```

CloseableHttpComponentsMessageSender has several type of implementation:

- *NtlmCloseableHttpComponentsMessageSender* for NTLM authentication support

Camel events

Description

CleverBus uses Apache Camel events concept:

- all Camel events extend *java.util.EventObject* resp. *org.apache.camel.management.event.AbstractExchangeEvent*. There are many events already defined in Camel, e.g. *ExchangeCompletedEvent*, *ExchangeFailedEvent*, *ExchangeSendingEvent* etc. These events are generated automatically by Camel.
- if you want to catch these events then you have to implement *EventNotifier *interface and this notifier add to Camel context. It's possible to use method *addEventNotifier *in class *AbstractBasicRoute *for our routes.

Usage example: http://camel.apache.org/eventnotifier-to-log-details-about-all-sent-exchanges.html

There are new events defined in CleverBus which can be caught anywhere by *EventNotifier:*

- *ProcessingMsgAsynchEvent*
- *WaitingMsgAsynchEvent*
- *PartlyFailedMsgAsynchEvent*
- *FailedMsgAsynchEvent*
- *CompletedMsgAsynchEvent*

There was a discussion about whether to use Spring or Camel events. Finally we use Camel events because we use native support in Camel with lot of predefined events and Spring framework is only one implementation of bean registry and therefore we didn't want to be dependant on specific implementation.

Usage

There is *@EventNotifier* annotation that marks classes which implement *org.apache.camel.spi.EventNotifier *interface for listening to Camel events.

EventNotifier implementations have special behaviour in Camel, most of them are Camel services with well-defined lifecycle. Use parent class *EventNotifierBase* to meet these requirements.

If event notifier extends *EventNotifierBase* class then it's not necessary to initialize them in Camel context, it's automatically by default.

Custom event

```
/**
* Event for failed VF call. This event occurs when asynch. message with calling VF failed.
*/
public class FailedFromVfEvent extends FailedMsgAsynchEvent {
   private static final long serialVersionUID = 8376176024372186281L;

   /**
    * Creates new event.
    *
    * @param exchange the exchange
    * @param message  the message
    */
   public FailedFromVfEvent(Exchange exchange, Message message) {
       super(exchange, message);
   }
}
```

Event throwing, method *notifyFailedFromVf*

```
/**
* Helper class for creating events.
*/
public final class TutanEventHelper {
   private TutanEventHelper() {
   }

   /**
    * Throws {@link FailedFromVfEvent} and notifies all registered event notifiers.
    *
    * @param exchange the exchange
    */
   public static void notifyFailedFromVf(Exchange exchange) {
       AsynchEventHelper.notifyMsg(exchange, new EventNotifierCallback() {
           @Override
           public boolean ignore(EventNotifier notifier) {
               return notifier.isIgnoreExchangeEvents() ||
notifier.isIgnoreExchangeCompletedEvent();
           }
           @Override
           public AbstractAsynchEvent createEvent(Exchange exchange) {
               return new FailedFromVfEvent(exchange, getMsgFromExchange(exchange));
           }
       });
   }

   private static Message getMsgFromExchange(Exchange exchange) {
       return (Message) exchange.getIn().getHeader(AsynchConstants.MSG_HEADER);
   }
}
```

Notifier implementation

```
/**
* Event notifier for listening to {@link FailedFromVfEvent} events.
*/
@EventNotifier
public class VfTopUpFailedEventNotifier extends EventNotifierBase<FailedFromVfEvent> {

    @Override
    public void doNotify(FailedFromVfEvent event) throws Exception {
        Message message = event.getExchange().getIn().getHeader(AsynchConstants.MSG_HEADER,
Message.class);
        if ("createTopUp".equals(message.getOperationName())) {
            getProducerTemplate().send(CreateTopUpRoute.URI_CREATE_TOP_UP_VF_FAILED,
event.getExchange());
        }
    }
}
```

Development

The whole development was moved to GitHub.

Main resources:

- Wiki: https://cleverbus.atlassian.net/wiki/display/CLVBUS
- Issues: https://github.com/integram/cleverbus/issues
- GitHub: https://github.com/integram/cleverbus
- Build server: https://travis-ci.org/integram/cleverbus
- Forum: https://groups.google.com/d/forum/cleverbus

Development tips

Restrict Spring bean inicialization

If there are many Spring beans with route definitions then startup time for loading ESB application can be quite long. This functionality enables to **include** only those Spring beans which should be initialized or **exclude** those Spring beans which we don't want to initialize.

It can be handy during development because it's possible via system properties to restrict set of Spring beans (=Camel routes) which will be initialized.

More info in Configuration.

Calling services between Spring contexts

Since version 0.4

Common scenario for solving: there is root Spring context root and two child contexts for admin GUI (web MVC) and for web services (spring WS). And we need to call services in WS context from admin GUI controllers.

There is package of handful classes for solving this problem, see *com.cleverlance.cleverbus.core.common.contextcall.ContextCall* as main interface for use.

Routes initialization test

Since version 0.4

If you want to test that all routes will be successfully initialized in Camel then use the following test. This test mainly checks that all routes have unique ID and URI.

```
/**
* Test that verifies if all Camel routes are correctly initialized - if there are unique route
IDs and unique URIs.
*
* @author <a href="mailto:petr.juza@cleverlance.com">Petr Juza</a>
*/
public class RoutesInitTest extends AbstractModulesDbTest {
   @BeforeClass
   public static void setInitAllRoutes() {
       setInitAllRoutes(true);
   }
   @Test
   public void testInit() {
       // nothing to do - if all routes are successfully initialized then test is OK
       System.out.println("All routes were successfully initialized");
   }
}
```

Best practices

Dependencies between routes

There is used IoC container by Spring framework for CleverBus routes configuration and initialization. Therefore it's not problem to use everything what Spring framework offers in this area (for example auto-wiring). Nevertheless, it's very useful to initialize or activate only those routes which there are necessary for unit tests and then it's good practice to leave dependency management on Apache Camel. In other words solve dependencies on the fly, not during application initialization as auto-wiring.

Camel uses registry abstraction for dependency management and most often implementation is Spring framework.

More information: http://camel.apache.org/how-does-camel-look-up-beans-and-endpoints.html

Error codes for external calls

It's good practice to define error code, that extends *org.cleverbus.core.common.exceptions.ErrorExtEnum interface,* for each external call failure and this error code propagates to source system.

Integration platform is central point and if there is any error then it's first point where somebody try to find out reasons of failure, where did error occur. Therefore is very useful to define unique error codes for all external call failures to have immediately information where is the problem.

There is error catalogue presentation in Admin GUI.

Use type converters

There is one very often EIP integration pattern - message translation. Often is necessary to convert one data structure to another data structure. You can use lot of approaches directly in route implementation but it's often related to specific route only. Therefore it's good practice to use Type Converters in Camel.

What are common goals?

- encapsulate transformation logic to one place
- limit to bad practices

    - OUT modules can't depend on IN modules (for example billing routes shoudn't be dependant on customer routes)
    - common project code can't depend on module specifics

IN routes are generally specific for the project but OUT routes are specific for calling system. And then if external system is used in more projects then it would be nice to reuse it or move it to CleverBus extensions for further use.

How to make it?

- use interfaces to solve dependency between common and specific code - common code defines interface and specific code implements it.
- module dependency
    - use converters and use *.convertBodyTo(class)* in route implementation
    - place converters to IN modules because there should be no dependency to IN module in OUT module

Interface versioning

If there is any public API (e.g. WSDL for Web Services) then it's necessary to handle backward compatibility of this interface.

Backward compatible changes:

- adding new operation
- adding new (XML) type to schema

Non-backward compatible changes:

- removing operation
- renaming operation
- changes in (XML) types or message attributes
- changes in namespace

Versioning

Use versions in format *<major>.<minor>.*

Minor version is for compatible changes, *major* version indicates non-compatible changes.

Versioning should be explicit - it means to present version number in elements, URLs etc.:

- add *major* and *minor* version to WSDL name: *MyService-v1.2.wsdl*
- add *major* version to *targetNamespace* of WSDL: *<definition targetNamespace="*

http://cleverbss.org/ws/MyService-v1" xmlns="http://schemas.xmlsoap.org/wsdl/">

- add *major* and *minor* version to *portType* element: *<portType name="MyServicePort-v1.2">*
- add *major* and *minor* version to *service* element: *<service name="MyService-v1.2">*
- add *major* and *minor* version to *endpoint*: *<soap:address location="http://cleverbss.org/myService/v1"/>*

If there is change in WSDL's version then change versions of XSDs as well.

Contribution

Give us your feedback

Cleverbus mission is to build up the ESB exactly with the behaviour and functionality which developer needs for the project. With minimum expenses, with maximally effective and quick process of development, with automatic tools for everything what can be reasonably automated to solve time and effort, but still with the full access to the code for tuning, debugging or when requirements are somehow special. CleverBus is ESB dedicated to developers to make their work easy and interesting. Your live feedback from your projects and your experience with CleverBus helps us to keep the right direction. If you have any comments, send it to us via email (development@cleverbus.org) or join CleverBus forum.

Contribute to the code

We welcome any new member of the community, who volunteers to add new piece of code to CleverBus or fix the existing one, to write documentation or help others on forum. How to contribute is described on GitHub guides.

Generally it's very simple: you don't need to ask if you can help, you don't need to apply to join the team - all you do is you fork the repository of your choice. In your own fork you can implement and test any enhancements or fixes you would like to contribute. Once you feel your code is ready, just let us know by starting a pull request. The pull request informs the team that there is a new contribution ready to get merged into the main line of code.

Contribute to forum and documentation

We welcome any new volunteers who help us create comprehensive documentation and knowledge base on forum.

Be patient

CleverBus has been developed for one year as proprietary software in Cleverlance company. We decided to make it as open-source few months ago. We need to move everything from private to public systems (issues, wiki, build server, Maven repository etc.), translate everything to English and change our internal processes how we will use CleverBus. **Please be patiant, it's time consuming to make all these changes.**

Source code conventions

Classes and interfaces name conventions

Base package for all classes is *org.cleverbus*.

There are the following name conventions for classes and interfaces:

Suffix	**Meaning**
...*Route*	Route implementation
...*Converter*	Converter implementation
...*Service*	Service implementation
...*DAO*	DAO interfaces
...*Exception*	Exception
...*Controller*	MVC controller
...*Factory*	Class that creates another classes
...*Test*	Test class
...*Tools*	Class with useful static methods
...*Helper*	Helper class, usually for local use
...*Enum*	Enumeration

Method name conventions

There are the following name conventions for methods:

Prefix	**Meaning**	**Return value**
set ...	sets value or reference	
get ...	gets value or reference	value/reference or throws exception
add ...	adds value/reference	
update ...	data/state change	
delete ...	data remova	

create ...	new object creation	
exists ...	existence check	It returns true or false.
check ...	common check	true or false
validate ...	validates input values or internal object state	
findAll ...	finds all possible values	collection or empty collection
findBy /*findXyzBy*...	finds value(s) by specified input parameters	value/reference or null
test ...	test method, usually annotated by *@Test*	

Demarcation of transactions - there is expected readonly transaction for methods which have the following prefixes: *get, exists, check, findAll, find*.

Syntax rules

Adhere to basic Java Code Conventions from Sun/Oracle.

Common syntax rules:

- Basic unit of indentation is 4 spaces. Tab should be converted to spaces.
- Line lenght is 120 characters.
- Maximum size of one class shoudn't exceed 700 lines.
- Use English language exclusively - for class, interface and method names, in comments, GUI, etc.
- Use UTF-8 character set in all source code files.

How to use *TODO*?

TODO uses developer for himself, it's not tool for assigning work to anybody else. Never use *TODO* without username.

Use *TODO* in the following format where *TO_WHOM* is developer's username, for example PJUZA, THANUS.

```
TODO TO_WHOM
```

JavaDoc

JavaDoc and comments in general are very useful - it's not easy to find balance between many comments without meaningful information and no comments at all. There are few rules which are good to adhere: How to Write Doc Comments for the Javadoc Tool

Use *package-info.java* file for every package.

@deprecated

We have to adhere backward compatibility and sometimes there is no other possibility then leave old code for deprecation and create new one for next releases.

If any class or method is deprecated then it's necessary to:

- use *@Deprecated* annotation
- add *@deprecated* into JavaDoc with information what to use instead

```
    /**
    * Helper method which creates input route for asynch. messages.
    *
    * @param route                the route where we want to create asynch. input routing
    * @param routeId              the input route ID
    * @param inUri                the from URI of this route
    *
    * @deprecated Use {@link AsynchRouteBuilder} instead.
    */
   @Deprecated
   public static RouteDefinition createAsynchInRoute(RouteBuilder route, String routeId, String
inUri)
```

Methods or classes marked as deprecated will be removed in next major version.

License header

Use the following Apache license header in all Java classes.

```
/*
* Copyright 2014 the original author or authors.
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*      http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/
package org.cleverbus.spi.alerts;
```

Running CleverBus

Description

Information about running CleverBus.

- H2 database
- Maintenance
- Monitoring
- Reference installations

H2 database

Description

Default database (mainly for development and testing purposes) is H2 database.

H2 database is running in embedded (local)in-memory mode with database name *cleverBusDB*.

```
db.driver=org.h2.Driver
db.url=jdbc:h2:mem:cleverBusDB
db.username=sa
db.password=
```

If there's need to see records in in-memory database during CleverBus running then there are the following possibilites.

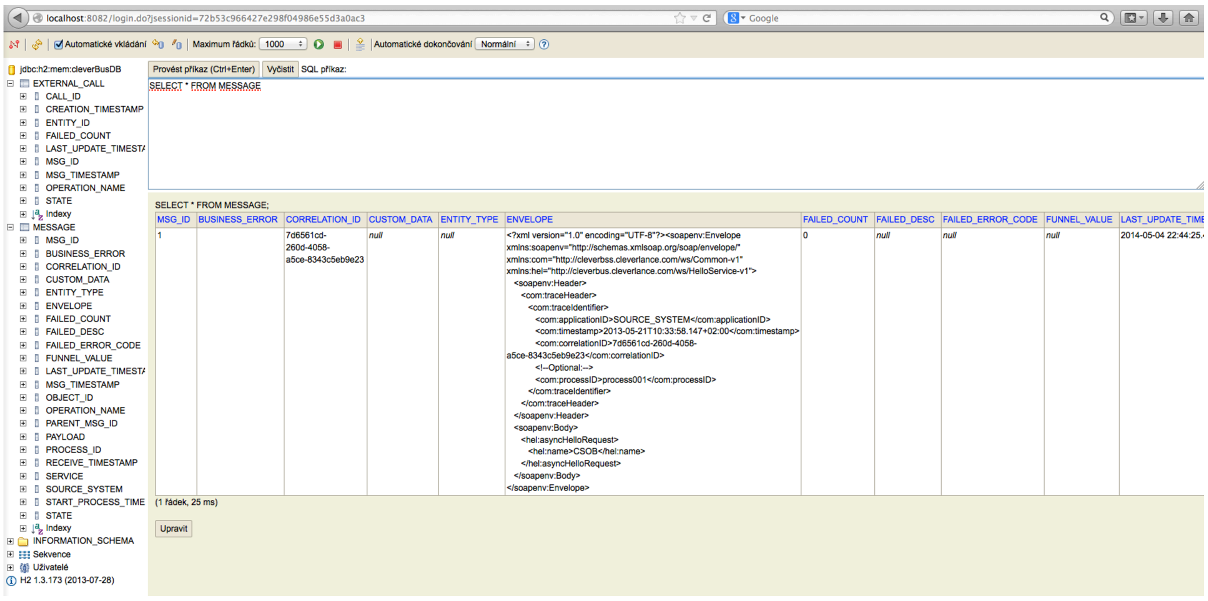
H2 console

H2 console is web browser based application that is configured on 8082 port.

configuration in sp_dataSources.xml

```
<bean id="h2WebServer" class="org.h2.tools.Server"factory-method="createWebServer" init-
method="start" destroy-method="stop">
<constructor-arg value="-web,-webAllowOthers,-webPort,8082"/>
</bean>
```

Login page


localhost:8082/login.do?jsessionid=72b53c966427e298f04986e55d3a0ac3

Automatické vkládání | Maximum řádků: 1000 | Automatické dokončování: Normální

jdbc:h2:mem:cleverBusDB
- EXTERNAL_CALL
  - CALL_ID
  - CREATION_TIMESTAMP
  - ENTITY_ID
  - FAILED_COUNT
  - LAST_UPDATE_TIMEST
  - MSG_ID
  - MSG_TIMESTAMP
  - OPERATION_NAME
  - STATE
  - Indexy
- MESSAGE
  - MSG_ID
  - BUSINESS_ERROR
  - CORRELATION_ID
  - CUSTOM_DATA
  - ENTITY_TYPE
  - ENVELOPE
  - FAILED_COUNT
  - FAILED_DESC
  - FAILED_ERROR_CODE
  - FUNNEL_VALUE
  - LAST_UPDATE_TIMEST
  - MSG_TIMESTAMP
  - OBJECT_ID
  - OPERATION_NAME
  - PARENT_MSG_ID
  - PAYLOAD
  - PROCESS_ID
  - RECEIVE_TIMESTAMP
  - SERVICE
  - SOURCE_SYSTEM
  - START_PROCESS_TIME
  - STATE
  - Indexy
- INFORMATION_SCHEMA
- Sekvence
- Uživatelé

H2 1.3.173 (2013-07-28)

Provést příkaz (Ctrl+Enter) | Vyčistit | SQL příkaz:

```
SELECT * FROM MESSAGE
```

SELECT * FROM MESSAGE;

MSG_ID	BUSINESS_ERROR	CORRELATION_ID	CUSTOM_DATA	ENTITY_TYPE	ENVELOPE	FAILED_COUNT	FAILED_DESC	FAILED_ERROR_CODE	FUNNEL_VALUE	LAST_UPDATE_TIME
1		7d6561cd-260d-4058-a5ce-8343c5eb9e23	*null*	*null*	<?xml version="1.0" encoding="UTF-8"?><soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:com="http://cleverbss.cleverlance.com/ws/Common-v1" xmlns:hel="http://cleverbus.cleverlance.com/ws/HelloService-v1"> <soapenv:Header> <com:traceHeader> <com:traceIdentifier> <com:applicationID>SOURCE_SYSTEM</com:applicationID> <com:timestamp>2013-05-21T10:33:58.147+02:00</com:timestamp> <com:correlationID>7d6561cd-260d-4058-a5ce-8343c5eb9e23</com:correlationID> <!--Optional:--> <com:processID>process001</com:processID> </com:traceIdentifier> </com:traceHeader> </soapenv:Header> <soapenv:Body> <hel:asyncHelloRequest> <hel:name>CSOB</hel:name> </hel:asyncHelloRequest> </soapenv:Body> </soapenv:Envelope>	0	*null*	*null*	*null*	2014-05-04 22:44:25.

(1 řádek, 25 ms)

Upravit

JDBC via TCP

It's possible to connect H2 CleverBus in-memory DB if H2 server is configured.

configuration in sp_dataSources.xml

```
<bean id="h2Server" class="org.h2.tools.Server" factory-method="createTcpServer" init-
method="start"
destroy-method="stop" depends-on="h2WebServer">
<constructor-arg value="-tcp,-tcpAllowOthers,-tcpPort,9092"/>
</bean>
```

Login in Squirell:

Maintenance

Description

Integration platform generates application logs and writes records to database.

Application logs

This chapter is specific for CleverBus admin GUI use.

Application logs are generated via logback library and are configured in *logback.xml*. There is Maven property *log.folder* that specifies target folder for application logs.

Log names are in format *logFile_RRRR-MM-DD_N.log*, where *N* is order number unique for each day. If the log file is greater than 10 MB then new log file with next order number is created. It's best practice to backup these application logs, at least for 2 months.

Logback offers lot of possibilities where and how log items save to - to file system, to database, send it by emails etc.

CleverBus provides simple tool for application log searching (see Admin GUI - Search in log by date). This tool must be correctly configured (*log.folder.path* - path to application logs, *log.file.pattern* - format of log file names) and then allows to go through all application logs and find searched information.

When we use standard installation with Apache Tomcat server then we have all logs in */srv/cbssesb/logs/* folder with the the following sub-folders:

- *apache*: logs generated by Apache HTTP server
- *j2ee*: logs from integration platform
- *tomcat*: logs generated by Apache Tomcat

Database

CleverBus stores lot of records to database, namely to these tables (see Data model for more information):

- *message*
- *external_call*
- *request* (it depends on configuration)
- *response* (it depends on configuration)

Also database records should be backuped. Remove old records to keep good performance - we remove records older then 2 months usually on our production installations.

There is functionality for archiving database tables.

Stop ESB

CleverBus allows to switch to stopping mode where no new requests will be processed, only current asynchronous messages will be finished (messages in states *PROCESSING* and *WAITING_FOR_RES*).

You will find more information in admin GUI documentation.

Monitoring

Description

During maintenance and monitoring each integration platform is very important to rely mainly on application logs, where there are a substantial and detailed information about running platform - CleverBus.

Application log

The main task for application log is hold the information from running of ESB which are required to solve any problems.

Sample of one logging line in following format:

%d{ISO8601} [${serverId}, ${MACHINE}, %thread, %X{REQUEST_URI}, %X{REQUEST_ID}, %X{SESSION_ID}, %X{SOURCE_SYSTEM}, %X{CORRELATION_ID}, %X{PROCESS_ID}] %-5level %logger{36} - %msg%n

One log line contains standard parameters which can configure in file *logback.xml* (see documentation for Logback lib that is used as logging platform. In addition the following custom logging parameters are used:

- LOG_SESSION_ID: unique HTTP session identifier. Thanks to this parameter can be grouped records from one user (system). This parameter is not filled if process is asynchronous and postponed (processed in the future).
- REQUEST_URI: URI of incoming request to integration platform. This parameter is not filled if process is asynchronous and postponed (processed in the future).
- REQUEST_ID: unique identifier of single request. Thanks to this parameter can grouped records from one request (HTTP request). This parameter is not filled if process is asynchronous and postponed (processed in the future).
- SOURCE_SYSTEM: identifier of source system which send inbound request
- CORRELATION_ID: unique identifier of single asynchronous message. Thanks to this parameter it can search all correlated records in log file for specific message. Note: this parameter is used by Log Viewer in admin gui to show correlated records.
- PROCESS_ID: optional identifier of process which identifies records correlated to one business process e.g. creating new user which is composed by two subprocess: create new user and activate it.

Example

*2011-04-05 08:07:07,964 [server1, http-8080-Processor16, /sc-web-console/sc/sn/mock_demo_service/on/Demo, 127.0.1.1:4c8ed99e:12f244556e1:-7fdd, 6B6*****C6C2, deih3u36bh] INFO c.c.s.f.w.f.screen.ScreenFormPanel - the message for key 'widget16_label_key' was not found: form='demo_form1', locale='cs'*

2013-08-07 02:01:34,542 [MACHINE_IS_UNDEFINED, Camel (camelContext) thread #25 - <seda://asynch_message_route>, , 192.168.198.100:-7d3dda4f:1405477688f:-60c4, , a2e7cf84-f4fe-e211-b400-005056bc0011] DEBUG

Detection	Severity	Recommended action	Description of problem
Search string "] ERROR " in log files /srv/cleverbus/logs/j2ee/ (or where files are stored)	ERROR	Forward to resolve by support team	Fault status in the ESB application
Search string "] FATAL " in log files /srv/cleverbus/logs/j2ee/ (or where files are stored)	FATAL	Forward to resolve by support team	Critical status in the ESB application
Search string "was changed to FAILED"	WARN	Forward to resolving to administrator, who should look at where the problem is and why the message could not be processed.	Asynchronous message is in FAILED status (some business or technical error occurs)

Database

In database CleverBus stores records to support asynchronous processing of messages. If integration platform is processing synchronous request than information about that can be search in log file, no in database. Exception is logging and monitoring all request/responses sent by integration platform into external systems.

From monitoring of database of view is recommended to observe followings:

- statuses of asynchronous *messages* in table message, column *state* (status) stavy - either directly by select into database or observe reports, see Admin GUI - Admin - Message report service.
- when any information in message is changed, column *last_update_timestamp* is updated by actual time
- statuses of external calls in table *external_call*, column *state*.

Detection	Severity	Recommended action	Description of problem
Message in FAILED state	WARN	Using the application log determine the causes of errors.	Asynchronous message ended up in the final status FAILED, which means that during the processing occurred an error (business or technical error).

Message is long time in the state WAITING_FOR_RES	WARN	Using the application log to make sure that we sent a message to the external system properly and correctly and, if necessary, we even received an acknowledgment of receipt of our report from external system.	Asynchronous message is waiting to response from external system. If there are messages in this status for long time (normal response is received during a few of milliseconds) it means that the problem is on external system side.
Message is long time in the state PROCESSING	WARN	Checks whether repair mechanisms works as expected and why messages are still in PROCESSING status.	Asynchronous message is in PROCESSING state. If some error occurs during processing a message and the message remains in this status, CleverBus has corrigible mechanism which changes after configurable time (default value is 300s) status of message to PARTLY_FAILED without changes of failed count (failed_count).
External call is in FAILED_END status	WARN	Using the application log determine the causes of errors.	Only external call - confirmations can be in this status and it means the problem where CleverBus could not confirm processing of asynchronous message. Confirmation of result of processing asynchronous messages is joined only with some external systems. If the confirmation fails a workflow process is stopped because external system can have some dependencies to next processing of it.
Message is long time in the state POSTPONED	WARN	Check configurable parameters asynch.postponedInterval, which stands for after how long becomes inactive message in processing state (PROCESSING, WAITING, WAITING_FOR_RES) marked as "not processing".	Messages in status POSTPONED are processed with the same mechanism as messages in PARTLY_FAILED status. Only one difference is that postponed messages are processed preferably. Note: if the behavior of this mechanism would be unexpected (around PARTLY_FAILED messages processing) so it will be the same unexpected behaviour during POSTPONED processing.

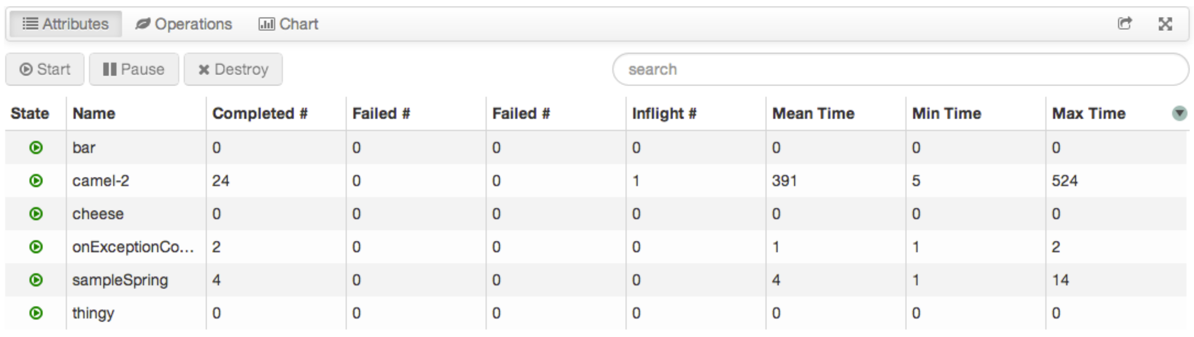
Hawtio

Monitoring of Apache Camel with hawtio

The core of CleverBus integration platform is Apache Camel framework To monitor Apache Camel CleverBus provides a lightweight and modular HTML5 web console hawtio tool. Lets you browse CamelContexts, routes, endpoints. Visualise running routes and their metrics. Create endpoints. Send messages. Trace message flows, as well profile routes to identifiy which parts runs fast or slow.

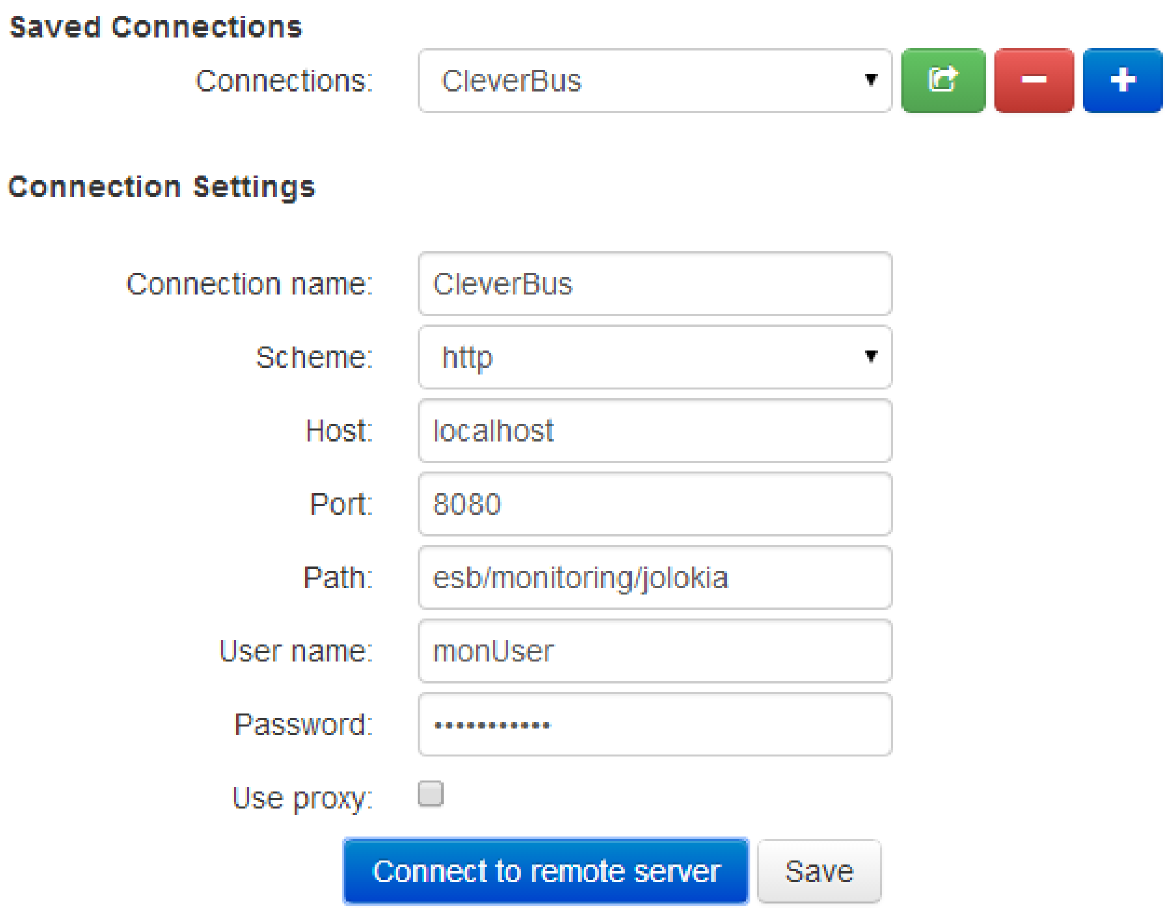
You can view all the running Camel applications in the current JVM. You can among others see the following details:

- Lists of all running Camel applications
- Detailed information of each Camel Context such as Camel version number, runtime statics
- Lists of all routes in each Camel applications and their runtime statistics
- Manage the lifecycle of all Camel applications and their routes, so you can restart / stop / pause / resume, etc.
- Graphical representation of the running routes along with real time metrics
- Live tracing and debugging of running routes
- Profile the running routes with real time runtime statics; detailed specified per processor
- Browsing and sending messages to Camel endpoint



State	Name	Completed #	Failed #	Failed #	Inflight #	Mean Time	Min Time	Max Time
▶	bar	0	0	0	0	0	0	0
▶	camel-2	24	0	0	1	391	5	524
▶	cheese	0	0	0	0	0	0	0
▶	onExceptionCo...	2	0	0	0	1	1	2
▶	sampleSpring	4	0	0	0	4	1	14
▶	thingy	0	0	0	0	0	0	0

You can also configure JMX properties, for example for Throttling Configuration:



Saved Connections

Connections:	CleverBus

Connection Settings

Connection name:	CleverBus
Scheme:	http
Host:	localhost
Port:	8080
Path:	esb/monitoring/jolokia
User name:	monUser
Password:	••••••••••••
Use proxy:	☐

JavaMelody

Monitoring of CleverBus integration platform with JavaMelody

CleverBus as integration plaftorm provides tools to measure and calculate statistics on real operation of CleverBus depending on the usage of the application by users thanks to JavaMelody - tool to monitor Java or Java EE applications servers in QA and production environments.

It allows to improve applications in QA and production and helps to:

- give facts about the average response times and number of executions
- make decisions when trends are bad, before problems become too serious
- optimize based on the more limiting response times
- find the root causes of response times
- verify the real improvement after optimizations

It includes summary charts showing the evolution over time of the following indicators:

- Number of executions, mean execution times and percentage of errors of http requests, sql requests, jsf actions, struts actions, jsp pages or methods of business façades (if EJB3, Spring or Guice)
- Java memory
- Java CPU
- Number of user sessions
- Number of jdbc connections

These charts can be viewed on the current day, week, month, year or custom period.

By default javamelody is turn off. To enable you have to configure JAVA_OPTS argument this way: **-Djavamelody.disabled=false**.

JMX

Description

JMX (Java Management Extension) allows to monitor statistics about asynchronnous messages or starts administration operations.

JVM and Camel attributes

See Camel JMX for more info about Camel attributes and operations - "By default, JMX instrumentation agent is enabled in Camel, which means that Camel runtime creates and registers MBean management objects with a MBeanServer instance in the VM. This allows Camel users to instantly obtain insights into how Camel routes perform down to the individual processor level. The supported types of management objects are endpoint, route, service, and processor."

Parameter	**Description**	**Note**
Current heap size	Current heap memory size.	If there is no enough memory then garbage collector needs more often to free memory and the performance is suffering.
Maximum heap size	Maximum heap memory size.	Related to the previous parameter.
Live threads	Current count of running threads	If there is trend with increasing number of threads then it seems to be a problem.
Apache Camel parametry - *org.apache.camel*	Running statistics from Apache Camel	

MessageStatus - statistics about asynchronnous messages

Parameter	Description

countOfWaitingAfterInterval	Count of messages in state WAITING and after interval
countOfOkAfterInterval	Count of messages in state OK and after interval
countOfNewAfterInterval	Count of messages in state NEW and after interval
countOfPartlyFailedAfterInterval	Count of messages in state PARTLY_FAILED and after interval
countOfCancelAfterInterval	Count of messages in state CANCEL and after interval
countOfWaitingForResponseAfterInterval	Count of messages in state WAITING_FOR_RES and after interval
countOfPostponedAfterInterval	Count of messages in state POSTPONED and after interval

MessageAdmin - message administration

Since 0.2 version

restartMessage

Restarts message for next processing.

totalRestart parameter determines if message should start from scratch again (*true*) or if message should continue when it failed (*false*).

cancelMessage

Cancels next message processing, sets message to CANCEL state.

startNextProcessing

Starts next processing of PARTLY_FAILED and POSTPONED messages. Can run one process at one time only.

repairProcessingMessages

Starts repairing processing messages.

Reference installations

- Centropol Telecom application stack
- SazkaMobil performance

Centropol Telecom application stack

Description

CleverBus platform is light-weight application that can be installed on arbitrary web Java application server.

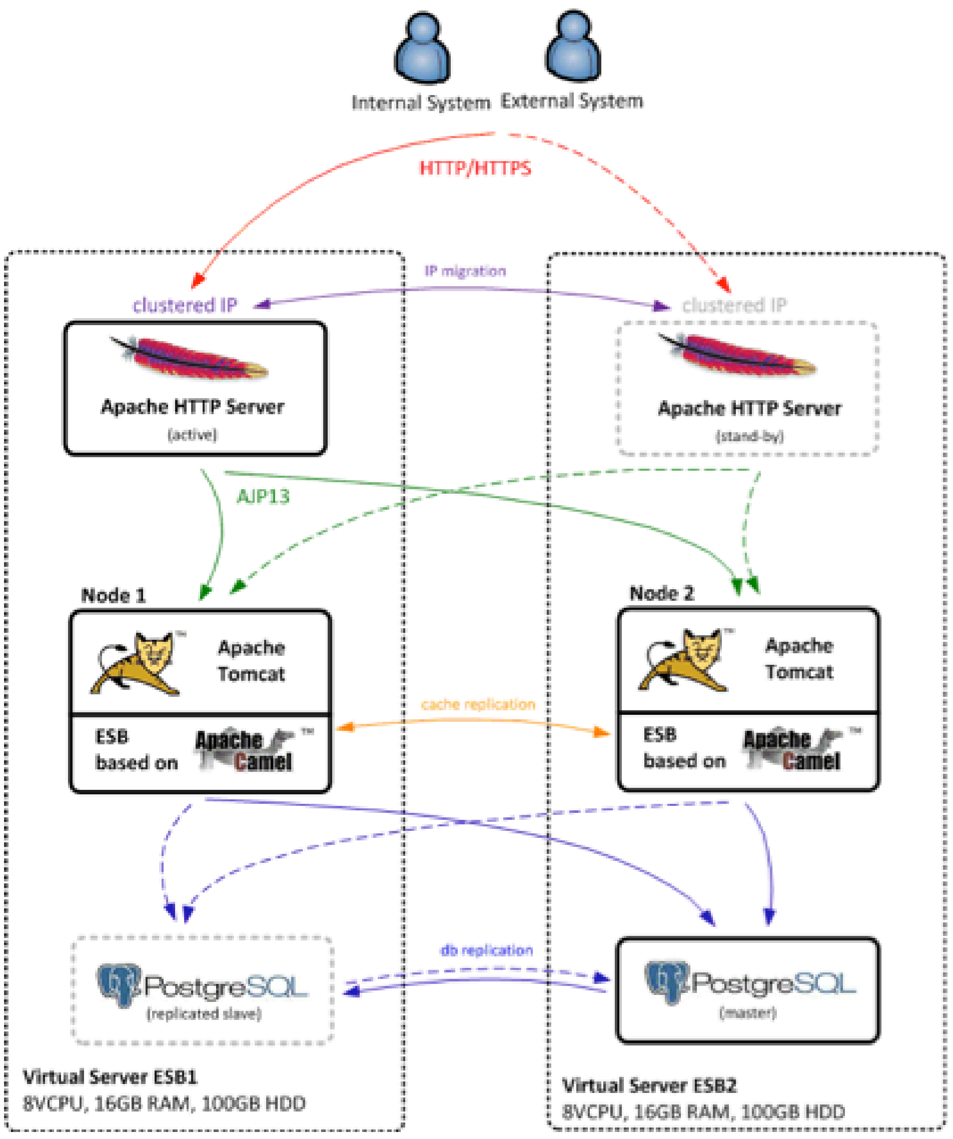
General requirements:

- application web server (Apache Tomcat 7 and higher is best solution)
- SQL database (PostgreSQL is recommened, but there is no dependency to specific database)
- Java version 6 and higher
- at least 256MB heap memory (JVM parameters: *-Xms192m -Xmx256m*)
- at least 256MB permgen memory (JVM parameters: -*XX:MaxPermSize=256m*)
- at least 5GB free space on filesystem for storing application logs

Centropol Telecom configuration

Centropol Telecom uses the following technology stack:

- CentOS 6.4 64bit
- Apache Tomcat 7
- PostgreSQL 9 (open source version)
- Apache HTTP server - mod_proxy_balancer is used for load balancing (round robbin algorhitm).
- HTTP proxy terminates SSL communication with external systems


Internal System
External System
HTTP/HTTPS
IP migration
clustered IP
clustered IP
Apache HTTP Server
(active)
Apache HTTP Server
(stand-by)
AJP13
Node 1
Node 2
Apache Tomcat
Apache Tomcat
ESB based on Apache Camel
ESB based on Apache Camel
cache replication
db replication
PostgreSQL
(replicated slave)
PostgreSQL
(master)
Virtual Server ESB1
8VCPU, 16GB RAM, 100GB HDD
Virtual Server ESB2
8VCPU, 16GB RAM, 100GB HDD

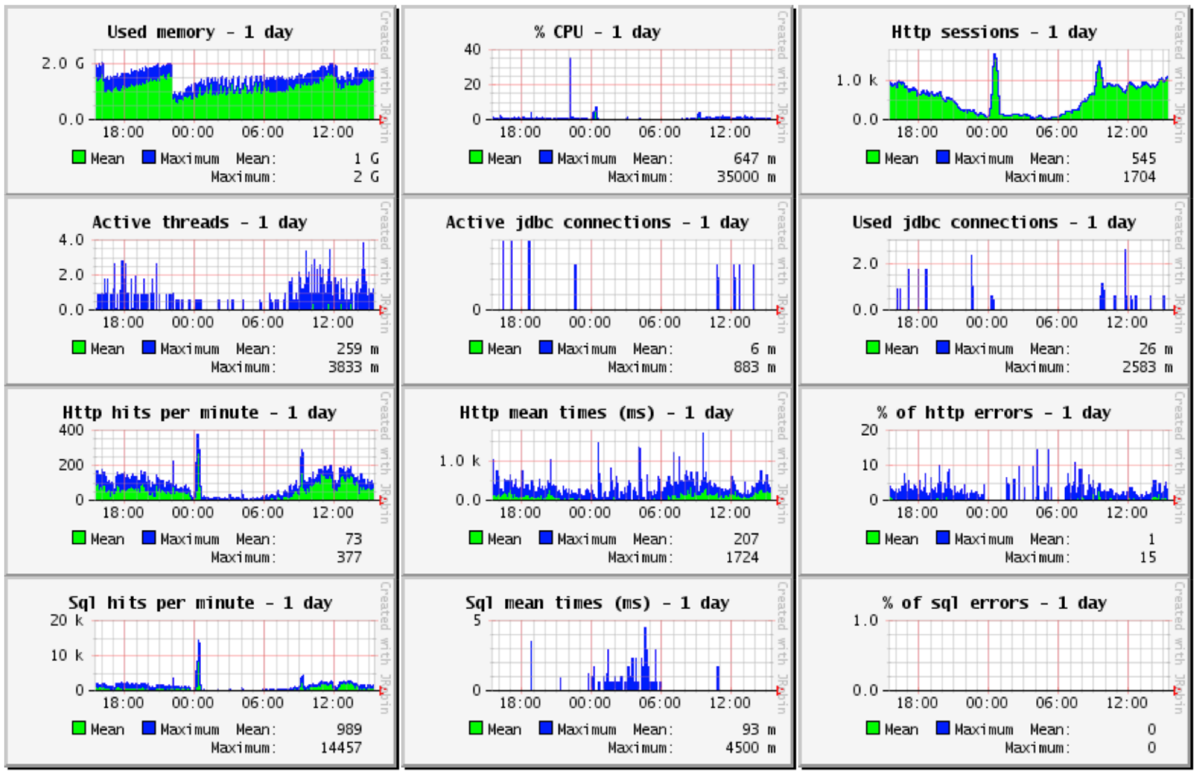
SazkaMobil performance

Performance statistics from production

Hardware

Server infrastructure is composed of blade server Cisco UCS B200 with chassis Cisco 5108, Intel xeon 4CPU + 8GB RAM + 96GB HDD. We need more memory and space on filesystem because there are two other running applications on the same server.

Request count (12.9.2014)



Performance test reports from test environment

Test 1

Overall count of requests during 10.9.2014: **91611**

Asynchronous requests from overall count: **78576** (**607** in state *FAILED*)

Average processing time of asynchronous message (*last_update_timestamp* **-** *receive_timestamp*): **0,003941**

Average response time during the whole day: **470ms**, min is not measured, max 02s 31,000ms

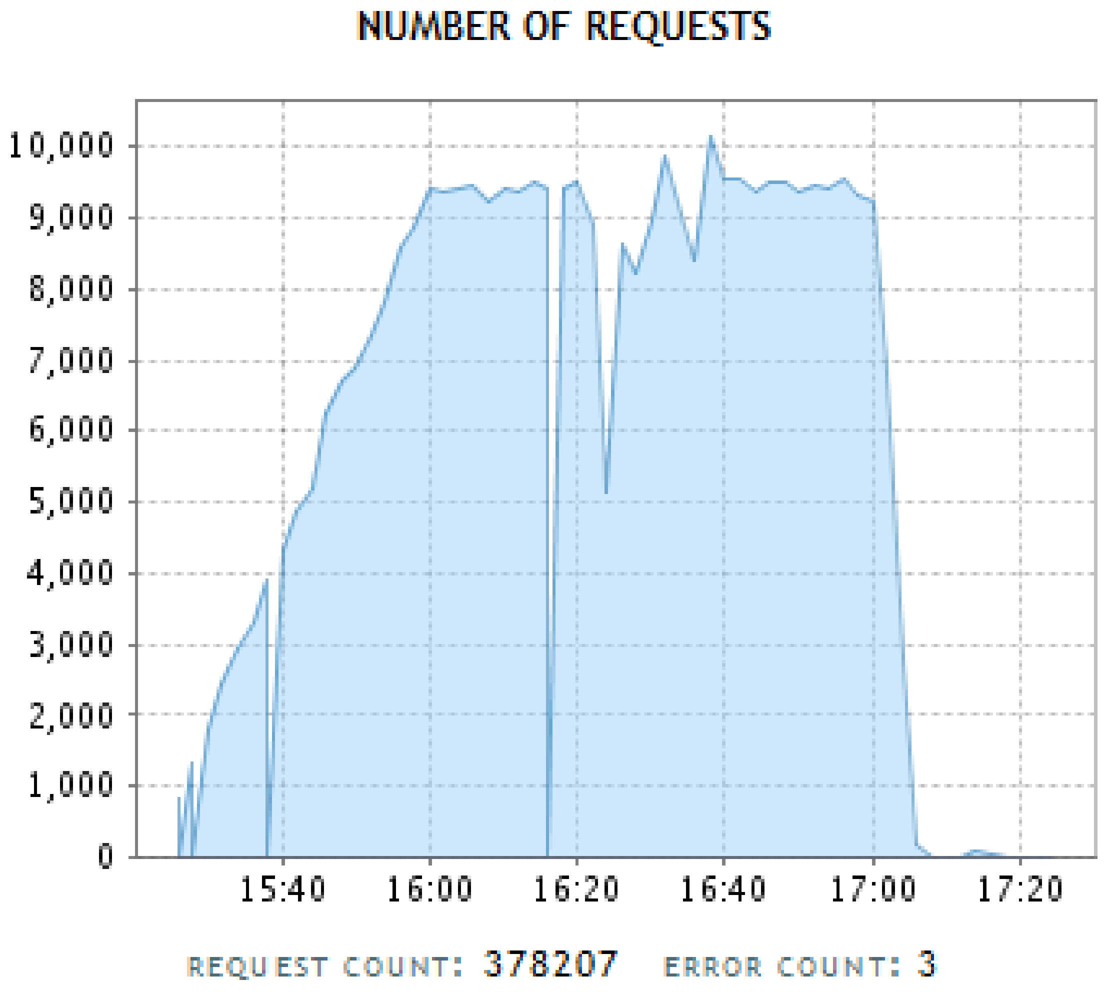
Request sizes:

- *createOrderExt* (order creation) – cca 2,89KB
- *createTopUpExt* (charging) - 1,36 KB
- *updateSubscriberAssignedProducts* - 1,44 KB
- *getSubscriber* - 276 bytes

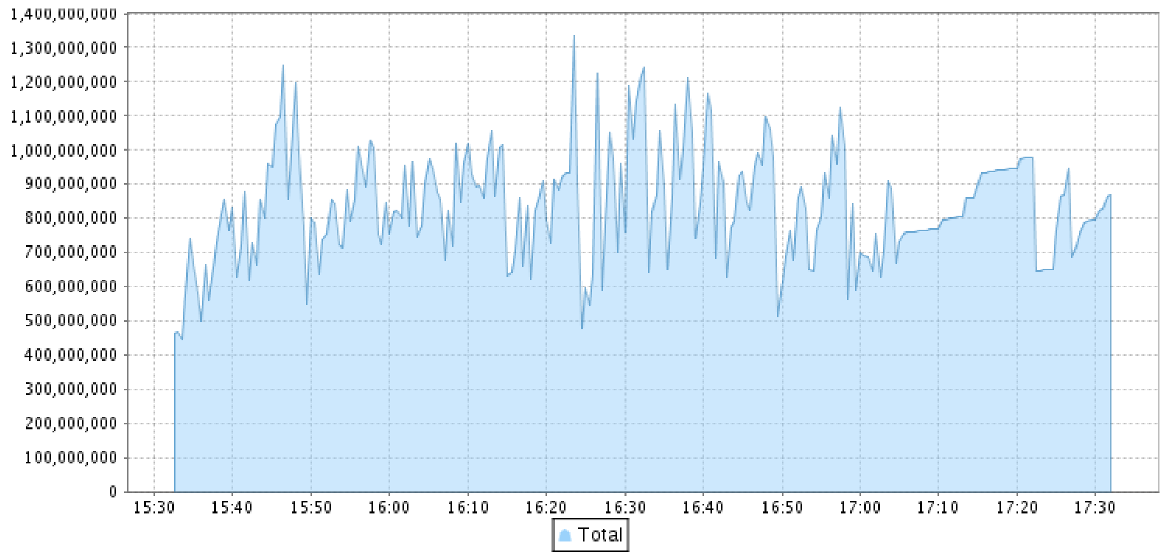
Average size is 0,3KB, max. size is 3KB.

Test 2

- load was **56** http requests / second
- overall count of requests: **378207**
- average response time was 250ms, 2s during stress test because of external systems
- memory was not higher than 1,3GB



MEMORY USAGE HISTORY



Hardware

	Proxy	**Selfcare**	**ESB**	**Billing**
# Cores	2	8	**2**	10
RAM [GB]	3	10	**3**	64

System

	Proxy	**Selfcare**	**ESB**	**Billing**
CPU %	7	8	**43**	8
I/O disk xfers	38	650	**600**	38
Disk Read KB/s	20	0	**500**	10
Disk Write KB/s	350	5000	**6500**	380
IO/s	35	650	**580**	38
Net I/O in MB/s	2,8	0,7	**1,3**	2,5
Net I/O out MB/s	2,8	0,5	**1,4**	2,8
pgpgin MB/s	0,5	0	**10**	0,3
pgpgout MB/s	7	100	**130**	7
pswpin MB/s	0	0	**0,6**	0
pswpout MB/s	0	0	**0,7**	0

Apache

	Proxy	**ESB**
Total Accesses	345415	**364918**
Total Traffic [GB]	1,1	**2,1**
requests/s	56,6	**60,6**
kB/second	193	**372**

JVM

	ESB	**Billing**
Sessions avg	4500	**30**
Requests total	363551	**392980**
Requests avg	9000	**10000**

Avg Response [ms]	60	**40**
Heap avg [GB]	1	**1**

CleverBus extensions

Descriptions

CleverBus extensions is standalone project that contains already implemented routes for possible use in CleverBus application.

CleverBus extensions are not open-sourced, it's proprietary code of Cleverlance company. If you would like to use them then contact CleverBus team.

If you want to write new extensions then see How to implement new extensions

Extensions overview

Extension	Description
ARES	checking if specified company ID is in ARES system and verifies if the company isn't in the registry of debtors
MVCR	checking identity card state, if exists and if yes if it's not stolen
File upload	uploading files with use *FileRepository* implementation

Development info

Extensions are located in separated project and Maven modules:

```
<groupId>com.cleverlance.cleverbus.extensions</groupId>
<artifactId>extensions</artifactId>
```

Stable version: 0.1

Development version: 0.3-SNAPSHOT

Nightly builds: https://hudson2.clance.local/view/CleverBus/job/CleverBus%20extensions/ (Cleverlance proprietary)

See development info for where to find CleverBus extensions.

How to implement new extensions

When you want to create new extension then follow these steps:

1) locate CleverBus extensions project

See Development info for more details.

2) create new module

- create new Maven module for new extension (pom.xml, README.txt)
- create new packages which start by *org.cleverbus.extension.[extension name]*

3) create extension configuration

Each extension must have Spring XML configuration file that completelly initializes specific extension.

- /META-INF/sp_ext_[ext_name].xml - this configuration most often loads property files and initializes routes.
- ext_[ext_name].cfg - file with configuration parameters

Each extension is initialized in independent Spring context, see dynamic extension loading for more details.

4) implement routes

Use parent class *org.cleverbus.api.route.AbstractExtRoute* for imlementation of extension routes.

If there are specific error codes then implement *ErrorExtEnum *with your codes.

Enumeration of external systems (*ExternalSystemExtEnum*) or enumeration of services (*ServiceExtensionEnum*) is in *common* module of CleverBus extension project.

Each extension can have own XSD/WSDL resources for converting to Java - use *jaxb_global_bindings.xjb* file from *common* module. See MVCR module for more details.

5) unit tests and wiki

Don't forget to implement unit tests and write comprehensive info about extension to wiki. Add new page under CleverBus extensions page.

Dynamic extension loading

Description

Dynamic extensions loading allows to add selected extension to CleverBus application independently by each other.

There is *extensions.cfg* configuration file (in *web-admin* module by default) with extension configuration parameters.

If you want to add one specific extension to CleverBus application then follow the following steps:

0) Prerequisite - extension configuration loader must be initialized

```
    <bean class="org.cleverbus.core.common.extension.PropertiesExtensionConfigurationLoader"
depends-on="camelContext">
       <constructor-arg ref="confProperties"/>
   </bean>
```

1) add Maven dependency to specific extension

For example add dependency to ARES extenion:

```
        <groupId>org.cleverbus.extensions</groupId>
       <artifactId>ares</artifactId>
```

2) add Spring configuration of specific extension

*extensions.cfg *(defined directly in CleverBus *sc-web-admin* module) or *extensions0.cfg* (has higher priority) defines references to Spring root configuration files for each extension. Each property has to starts with *context.ext* prefix.

Example

```
context.ext1 = classpath:/META-INF/sp_ext_ares.xml
```

sp_ext_ares.xml file can contain "unlimited" another Spring configuration specific for ARES extension, imports other configurations for the extension, choose between XML, annotation or Java config style etc.

For each item in the previous configuration file is new Spring child context created. This context is child context of Spring Camel (Web Service) context.

Child Spring context is type of ClassPathXmlApplicationContext

See *PropertiesExtensionConfigurationLoader* for more details.

There is the following Spring context hierarchy:

- root context
  - Camel (Spring Web Service) context
    - extension1 context
    - extension2 context
  - Spring MVC (admin) context

ARES

**Description: **extension for checking if specified company ID is in ARES system and verifies if the company isn't in the registry of debtors.

Maven module:

```
<groupId>com.cleverlance.cleverbus.extensions</groupId>
<artifactId>ares</artifactId>
```

Package: *com.cleverlance.cleverbus.extension.ares*

Route	*CompanyIDValidationRoute*
IN/OUT	OUT
URL	
Description	Route definition for calling the external registry ARES, based on the response a Company (Company ID) is listed as debtor. The only required parameter for the core functionality is the *COMPANY_ID_KEY*, it's necessary for the ext. query.
Configuration	Parameter Default value Description *ares.uri* http4://wwwinfo.mfcr.cz/cgi-bin/ares/darv_bas.cgi ARES service URI
Error codes	no error codes
Notes	ARES service description

File upload

Description

Extension for uploading files with use *FileRepository* implementation

Maven module:

```
<groupId>com.cleverlance.cleverbus.extensions</groupId>
<artifactId>file-upload</artifactId>
```

Package: *com.cleverlance.cleverbus.extension.fileupload*

Route	*UploadFileRoute*
IN/OUT	IN
URL	.../http/upload
Description	File is uploaded via PUT HTTP method. Input file is uploaded to temporary directory with unique file identifier (fileId) that is returned back. When this fileId comes with other data then the file is moved and renamed to right target folder. Uploading files can be tested with curl command tool: `curl -i -X PUT -T "/Volumes/Obelix/context.xml" http://localhost:8080/esb/http/...`

Configuration	Parameter Default value Description *file.maxUploadedFileSize* 5000 Maximum size (in kB) for uploaded files *file.upload.servletName* CamelServlet Servlet name defined in web.xml for uploading files
	web.xml:
	
	`    <!-- Camel servlet-->`
	`    <servlet>`
	`        <servlet-name>CamelServlet</servlet-name>`
	`        <servlet-`
	`class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-class>`
	`        <load-on-startup>1</load-on-startup>`
	`    </servlet>`
	`    <servlet-mapping>`
	`        <servlet-name>CamelServlet</servlet-name>`
	`        <url-pattern>/http/*</url-pattern>`
	`    </servlet-mapping>`

Error codes	See *ErrorFileUploadEnum* Error code Description E100 it's possible to use HTTP PUT method only E101 uploaded file exceeded maximum possible size

Prerequisites	File upload uses *FileRepository* contract and therefore it's necessary to inicialize *FileRepository's* implementations. `<bean id="fileRepository" class="com.cleverlance.cleverbus.core.common.file.DefaultFileRepository"/>` *DefaultFileRepository* uses *dir.temp* and *dir.fileRepository* configuration properties, see Configuration page for more details.
Notes	

Extension for checking identity card state, if exists and if yes if it's not stolen

Maven module:

```
<groupId>com.cleverlance.cleverbus.extensions</groupId>
<artifactId>mvcr</artifactId>
```

Package: *com.cleverlance.cleverbus.extension.mvcr*

Route	IdentityCardValidationRoute
IN/OUT	OUT
URL	
Description	Route definition for calling MVCR identity card validation service. When some error occurs, route acts like identity card is not stolen. User must set the *IDENTITY_CARD_KEY* header with identityCard number. Response is saved like boolean value in *IDENTITY_CARD_STOLEN_KEY* header attribute.
Configuration	Parameter Default value Description *mvcr.uri* http4://aplikace.mvcr.cz/neplatne-doklady/doklady.aspx MVCR service URI
Error codes	no error codes
Notes	MVCR service description

Release notes of extensions

0.3-snapshot

- CLVBUS-167 - Upgrade Extensions to last version of CleverBusu

0.1 (30.10.2014)

- CLVBUS-81 - Vydeleni addons (nove extensions) do separatniho projektu

Release notes

Release notes are published for the following products - CleverBus Community Edition - CleverBus Enterprise Edition

Enterprise Edition

v2.2.0 (20.1.2016)

Fixed

- [CLVBUS-234] - Even if the ESB is stopped the sync services works

New Features

- [CLVBUS-192] - Redesign of sync/async message logging and log searching
- [CLVBUS-225] - Throttling - per system limit added
- [CLVBUS-238] - Sync/async messages - unified message format
- [CLVBUS-245] - Web Admin GUI - modularization
- [CLVBUS-266] - Web Admin GUI - REST Management API
- [CLVBUS-259] - Funnel - performance improvement

v2.1.3 (21.12.2015)

Fixed

- [CLVBUS-265] - Calling throttling component produces StackOverflowError

New Features

- [CLVBUS-262] - Alerting - multitasking for threshold calculation added

v2.1.2 (11.12.2015)

Fixed

- [CLVBUS-261] - Throttling - produce ConcurrentModificationException

v2.1.1 (9.12.2015)

Fixed

- [CLVBUS-253] - Funnel - failes when multifunnel option is used and quaranteed order option is not used
- [CLVBUS-260] - Async message - error occures while message detail is displayed

New Features

- [CLVBUS-233] - REST services - Added basic authentication support

v2.1.0 (31.7.2015)

Fixed

- [CLVBUS-191] - ESB startup - random occurence of NullPointerException

New Features

- [CLVBUS-223] - Monitoring - zabbix integration added
- [CLVBUS-224] - Monitoring - introduced threshold calculation optimization
- [CLVBUS-229] - Throttling - explicit throtling - no defaults now
- [CLVBUS-169] - Throttling - output throtling added
- [CLVBUS-171] - Planned outages - management by REST API
- [CLVBUS-193] - Planned outages - WEB GUI
- [CLVBUS-207] - Migrated to camel 2.15.2
- [CLVBUS-208] - Async message - archivation optimalization
- [CLVBUS-213] - Async message - schedulling support added
- [CLVBUS-212] - CI - automatic performance tests support added

v2.0.4 (30.10.2015)

New Features

- [CLVBUS-222] - Async message - message search web service added
- [CLVBUS-216] - Funnel - multifunnel option support added
- [CLVBUS-228] - Alerting - global switch off configuration option added
- [CLVBUS-232] - Messages - Custom namespace can be used for traceHeader
- [CLVBUS-249] - EmailService - Default UTF-8 encoding for outgoing emails

v2.0.3 (16.6.2015)

Fixed

- [CLVBUS-226] - ESB startup - Timers delay (atleast 1 minute) is must

New Features

- [CLVBUS-208] - Async message - archivation optimalization
- [CLVBUS-214] - Funnel - funnel value in URI support added

v2.0.2 (4.5.2015)

Fixed

- [CLVBUS-191] - ESB startup - random occurence of NullPointerException while external system is called after ESB startup

v2.0.1 (17.4.2015)

Fixed

- [CLVBUS-179] - Async message - serialization error under stress

v2.0.0 (13.4.2015)

New Features

- [CLVBUS-170] - Runtime - Java 1.8 support

v1.3 (29.7.2015)

New Features

- [CLVBUS-176] - DB Archivation - with custom job implementation
- [CLVBUS-178] - EmailService - attachement support introduced
- [CLVBUS-208] - DB Archivation - big data optimalization

v1.2 (1.4.2015)

GitHub verze (org.cleverbus) Java 1.6 suppport

v1.1 (24.11.2014)

Fixed

- [CLVBUS-166] - Funnel - check of postponedIntervalWhenFailed not working

New Features

- [CLVBUS-117] - Automatic bean name generation - ROUTE BEAN
- [CLVBUS-132] - Custom check support added at the startup of ESB
- [CLVBUS-162] - Multiple entity managers support
- [CLVBUS-167] - CleverBus Extensions upgrade to CleverBus latest version

Community Edition

CleverBus can be downloaded from GitHub releases

v2.0 (22.1.2016)

Fixed

- [CLVBUS-191] - ESB startup - random occurence of NullPointerException while external system is called after ESB startup
- [CLVBUS-226] - ESB startup - Timers delay (atleast 1 minute) is must
- [CLVBUS-179] - Async message - serialization error under stress

New Features

- [CLVBUS-170] - Runtime - Java 1.8 support
- [CLVBUS-208] - Async message - archivation optimalization
- [CLVBUS-214] - Funnel - funnel value in URI support added
- [CLVBUS-176] - DB Archivation - with custom job implementation
- [CLVBUS-178] - EmailService - attachement support introduced
- [CLVBUS-208] - DB Archivation - big data optimalization
- [CLVBUS-222] - Async message - message search web service added
- [CLVBUS-216] - Funnel - multifunnel option support added
- [CLVBUS-228] - Alerting - global switch off configuration option added
- [CLVBUS-232] - Messages - Custom namespace can be used for traceHeader
- [CLVBUS-249] - EmailService - Default UTF-8 encoding for outgoing emails

1.1 (24.11.2014)

Repository

Javadoc for Cleverbus-1.1

Incompatible changes:

- Replace Cleverlance namespaces

Highlights:

Other New Features:

- Repackage web admin console in web-admin
- Rename Maven modules with prefix cleverbus

Bugs:

- Mismatch of JAXB classes and real package structure

Release notes old versions

Description

This page contains old releases to keep history of CleverBus development.

Look at GitHub for new releases

1.1-snapshot

Incompatible changes

- [CLVBUS-155] Odmazani @Deprecated veci
- [CLVBUS-164] Prejmenovat packages na org.cleverbus a zmenit licenci
  - packages *com.cleverlance -> org.cleverbus*
  - prejmenovani Maven modulu
    - groupId: *com.cleverlance -> org.cleverbus* (see Maven and Spring)

~~~
 ```xml
<dependency>
 <groupId>org.cleverbus</groupId>
 <artifactId>integration</artifactId>
</dependency>
```
~~~

```
-   artefactIds: *sc-core -\> core, sc-test -\> test, ...*
```

  - core-api: */com/cleverlance/cleverbus/core/modules/in/common/commonTypes-v1.0.xsd -> /org/cleverbus/api/modules/in/common/commonTypes-v1.0.xsd*
  - configuration parameter renamed: *directCall.localhostUri => contextCall.localhostUri*

Highlights

Other tasks

- [CLVBUS-162] Vice entity manageru na projektu s CleverBusem
- [CLVBUS-117] Automaticke generovani jmena beany - ROUTE BEAN
- [CLVBUS-161] Zavislost v jinych Maven projektech na stejne verzi Camelu
- [CLVBUS-132] Moznost vlastni implementace kontroly pri inicializaci aplikace
- [CLVBUS-166] Garantovane poradi nechodi jak by melo - kontrola postponedIntervalWhenFailed

1.0.1-snapshot

Bug fixes

- [CLVBUS-166] Garantovane poradi nechodi jak by melo - kontrola postponedIntervalWhenFailed

0.4 (=1.0 release, 15.10.2014)

Incompatible changes

- API changes
  - [CLVBUS-115] - zptn nekompatibilní *TraceHeaderProcessor*
- DB changes - necessary to run SQL script (*db_schema_postgreSql_0_4.sql*) for upgrading table structures
  - [CLVBUS-52] - new tables *request* and *response*
  - [CLVBUS-105] - changes in *message* table, added new column *parent_binding_type*
  - [CLVBUS-87] - changes in *message* table, added new columns *guaranteed_order* and *exclude_failed_state*
  - [CLVBUS-152] - changes in *message *table, added new column *funnel_component_id*
  - [CLVBUS-113] Pridat DB proceduru na archivaci zaznamu - *db_schema_postgreSql_archive_0_4.sql*

Highlights

- [CLVBUS-131] Zprávy ve stavu PARTLY_FAILED se nezpracovaji i kdyz ESB neni ve stop modu
- [CLVBUS-106] Pidat podporu pro volani WS pomoci SOAP 1.2
- [CLVBUS-127] Pridat kontrolni Spring beanu pro overovani konfigurace
- [CLVBUS-105] Evidence type vazby parent-child
- [CLVBUS-93] Prehled vstupnich endpointu a WSDL
- [CLVBUS-89] Cekajici PROCESSING zprava ve fronte
- [CLVBUS-88] Vylepseni trychtyre - zohledneni casu
- [CLVBUS-66] legalni stop ESB
- [CLVBUS-52] Ukladani volani externich systemu
- [CLVBUS-134] Kontrola stavu zprav a odeslani notifikace pri jejich prekroceni
- [CLVBUS-87] Garantované poadí zpracování více asynchronních zpráv v rámci jednoho business procesu
- [CLVBUS-50] Integraní logy - high overview

    - new configuration parameter added (l*og.file.pattern*)
    - logs can be compressed into gzip format now
    - MDC context was changed - added source system of caller (*SOURCE_SYSTEM*) if process is asynchronous and therefore changed *logback.xml*

Other tasks:

- [CLVBUS-128] Nespravne vyhodnocovani podporovaneho externiho systemu v DelegateConfirmationCallback
- [CLVBUS-139] Pridani copyright
- [CLVBUS-137] Chybne nacitani sloupce failedErrorCode pres failedErrorCodeInternal v entite Message
- [CLVBUS-118] Konsolidace web provided knihoven
- [CLVBUS-119] Zamenit basic auth ve web konzoli na form based (session) auth
- [CLVBUS-123] Presun GUI popisku do properties
- [CLVBUS-91] Prechod plne na JPA
- [CLVBUS-90] Upgrade na Camel 2.13
- [CLVBUS-124] Vytvorit "deployable" test pro overeni nasaditelnosti na server
- [CLVBUS-110] pridani indexu na funnel_value v tabulce messages
- [CLVBUS-113] Pridat DB proceduru na archivaci zaznamu
    - *db_schema_postgreSql_archive_0_4.sql*
- [CLVBUS-152] Oprava trychtyre - nutne zohlednit aktualni stav zpracovani zprav
- [CLVBUS-115] Validace applicationID dle zadaneho enumu (zptn nekompatibilní použítí *TraceHeaderProcessor*, nutné implementovat parametrický konstruktor)
- [CLVBUS-153] Odmazani mock serveru
- [CLVBUS-154] Upgrade archetype
- [CLVBUS-141] Umozneni zretezeni validace pomoci vice validatoru AsynchRouteBuilder#withValidator()

0.3.1-snapshot

Bug fixes

- [CLVBUS-116] TraceHeaderProcessor, ktery neni mandatory.

0.3 (25.6.2014)

Incompatible changes

- [CLVBUS-65] sp_monitoring.xml renamed to sp_jmx.xml + moved from sc-core module to sc-web-admin

- [CLVBUS-95] changes in sc-web-admin module - new default Spring profiles are "dev, h2" + "db_schema_test.sql" removal
- [CLVBUS-81] removed sc-addons module, new project was created for CleverBus extensions, see CleverBus extensions
- [CLVBUS-84] new API and SPI were created
    - **There is really lot of incompatible changes, the following list presents major ones only. Version 0.3 is not backward compatible to version 0.1 but it was necessary to made these changes before more projects will use CleverBus.**
    - Maven modules were reorganized, see Maven and Spring
    - new API (module sc-core-api) was created, package com.cleverlance.cleverbus.api
        - *AsynchConstants* - new class for route URIs

```java
   AsynchMessageRoute.URI_ERROR_HANDLING >> AsynchConstants.URI_ERROR_HANDLING
   AsynchMessageRoute.MSG_HEADER >> AsynchConstants.MSG_HEADER
   AsynchMessageRoute.URI_ERROR_FATAL >> AsynchConstants.URI_ERROR_FATAL
   AsynchMessageRoute.BUSINESS_ERROR_PROP_SUFFIX >>
AsynchConstants.BUSINESS_ERROR_PROP_SUFFIX
   AsynchMessageRoute.ASYNCH_MSG_HEADER >> AsynchConstants.ASYNCH_MSG_HEADER
   AsynchInMessageRoute.OBJECT_ID_HEADER >> AsynchConstants.OBJECT_ID_HEADER
   AsynchInMessageRoute.SERVICE_HEADER >> AsynchConstants.SERVICE_HEADER
   AsynchInMessageRoute.OPERATION_HEADER >> AsynchConstants.OPERATION_HEADER
   AsynchInMessageRoute.URI_ASYNCH_IN_MSG >> AsynchConstants.URI_ASYNCH_IN_MSG
   ...
```

-   change in package name of CleverBus entities for persistence - *com.cleverlance.cleverbus.api.entity*
-   *com.cleverlance.cleverbus.api.extcall.ExtCallComponentParams* - new class with params for extcall component, before *ExternalCallComponent*

    - new SPI (module sc-core-spi) was created, package *com.cleverlance.cleverbus.spi*
    - changes in class names in *sc-core-test* module - new class names *AbstractTest* and *AbstractDbTest*
    - several classes were moved from *sc-common* module to *sc-core*, such as the following

```java
com.cleverlance.cleverbus.common.HumanReadable ->
com.cleverlance.cleverbus.core.HumanReadable;
com.cleverlance.cleverbus.common.ws.component (sc-common) ->
com.cleverlance.cleverbus.core.common.ws.component (sc-core)
com.cleverlance.cleverbus.common.spring (sc-common) ->
com.cleverlance.cleverbus.core.common.spring (sc-core)
```

~~~~
Event:
com.cleverlance.cleverbus.common.event.EventNotifier (sc-common) ->
com.cleverlance.cleverbus.core.common.events.EventNotifier (sc-api)
com.cleverlance.cleverbus.common.event.EventNotifierBase (sc-common) ->
com.cleverlance.cleverbus.core.common.events.EventNotifierBase (sc-api)
com.cleverlance.cleverbus.common.event.EventNotifierAutoRegistry (sc-common) ->
com.cleverlance.cleverbus.core.common.events.EventNotifierAutoRegistry (sc-core)

Direct call:
com.cleverlance.cleverbus.common.directcall (sc-common) ->
com.cleverlance.cleverbus.core.common.directcall (sc-core)
com.cleverlance.cleverbus.core.common.route.DirectCallHttpImpl ->
com.cleverlance.cleverbus.core.common.directcall.DirectCallHttpImpl
com.cleverlance.cleverbus.core.common.route.DirectCallWsRoute ->
com.cleverlance.cleverbus.core.common.directcall.DirectCallWsRoute

Version:
com.cleverlance.cleverbus.common.version (sc-common) ->
com.cleverlance.cleverbus.core.common.version (sc-core)
```
~~~~

- [CLVBUS-81] - changes in main implementation classes for throttling

Highlights

- [CLVBUS-81] - Vydeleni addons (nove extensions) do separatniho projektu
- [CLVBUS-84] - Vytvoreni CleverBus API
- [CLVBUS-109] - Vytvoreni komponenty pro throttling

- [CLVBUS-61] changes in pooling incoming asynchronous messages
- [CLVBUS-57] [new monitoring possibilites with JavaMelody and hawtio](Monitoring)
- [CLVBUS-63] better and nicer admin GUI
- [CLVBUS-36] authorization in routes
- [CLVBUS-56] AsynchRouteBuilder - new builder class for asynchronous routes with fluent API
- [CLVBUS-47] run server with selected set of routes
- [CLVBUS-48] dynamic routes loading from JAR files

Other tasks

- [CLVBUS-69] - Chyba behem restartu zprav - dead lock

- [CLVBUS-80] - Nefunkcni autorizace na adrese .../ws/.../v1
- [CLVBUS-5] - Moznost reloadu konfigurace bez nutnosti restartovani serveru

- [CLVBUS-15] - Parent-child koncept
- [CLVBUS-64] - Problem s poolingem u Http clienta
- [CLVBUS-67] - Upgrade Apache Camel na 2.11.3
- [CLVBUS-76] - Visi zprava ve stavu WAITING
- [CLVBUS-95] - Moznost prohlizeni zaznamu pri pouziti H2 DB
- [CLVBUS-71] - Oprava warnings z kompilace - deprecated API
- [CLVBUS-99] - Pri zapnutem JavaMelody je problem s vypnutim aplikace v Tomcatu
- [CLVBUS-29] - Integrace CleverBus komponent jako Addon
- [CLVBUS-63] - Vylepseni admin GUI
- [CLVBUS-72] - Souhrnny bod - zlepseni monitoringu a logovani zprav
- [CLVBUS-97] - Direct WS call - moznost poslani SOAP header
- [CLVBUS-100] - Moznost manualniho spusteni akce pro zpracovani PARTLY_FAILED zprav a pro jejich opravu
- [CLVBUS-101] - Umoznit vyhledavani zprav jen pres correlation_id (bez nutnosti vyberu zdrojoveho systemu)
- [CLVBUS-68] - Prezentace katalogu chyb
- [CLVBUS-65] - dynamicke nastaveni throttlingu pomoci JMX

0.2 (15.6.2014)

Technický release z dvodu interních projektových záležitostí, na tu verzi se nepojit.

Všechny tasky udlané do této verze jsou uvedeny v pehledu verze 0.3.

0.1.1-snapshot

Bug fixes

- [CLVBUS-70] Throttling - Array index out of range: 0
- [CLVBUS-69] Chyba behem restartu zprav - dead lock
- [CLVBUS-76] Visi zprava ve stavu WAITING
- [CLVBUS-80] Nefunkcni autorizace na adrese .../ws/.../v1
- [CLVBUS-61] Overeni spravne funkcnosti prijmu asynchronnich zprav (thread-polls)
- [CLVBUS-97] Direct WS call - moznost poslani SOAP header

0.1 (4.2.2014)

- [CLVBUS-1] - Vytvoreni zakladu pro CleverBus
- [CLVBUS-2] - Stav CANCEL - manualni zruseni zpracovani zpravy

- [CLVBUS-3] - Podpora pro manualni zruseni zpravy (CANCEL) v admin gui
- [CLVBUS-6] - Upgrade na Camel 2.12
- [CLVBUS-8] - Online sledovani pres JMX
- [CLVBUS-14] - Pridat processID do hlavicky asynchronnich zprav
- [CLVBUS-16] - Prime volani externich systemu
- [CLVBUS-20] - FileRepository presunout z projektu Centropol
- [CLVBUS-21] - Pridani ukazkove routy pro volani pres WS
- [CLVBUS-22] - Podpora eventu
- [CLVBUS-30] - Pridat tlacitko na odhlaseni z admin casti
- [CLVBUS-31] - Upgrade na quartz2
- [CLVBUS-33] - Pi chyb throttling exception se zpráva stejn propíše do db
- [CLVBUS-35] - Pri throttlingu vyhazovat HTTP 503
- [CLVBUS-37] - Provedeni vlastni akce pokud zpracovani zpravy skonci (OK, FAILED)
- [CLVBUS-38] - Automaticke mapovani zakladnich vyjimek na jejich error kody
- [CLVBUS-41] - Obecna architektura
- [CLVBUS-42] - Upgrade Camel na 2.11.2
- [CLVBUS-45] - ExceptionTranslator - vracet pouze root exception
- [CLVBUS-46] - Uprava SOAP fault odpovedi - pridat explicitne element pro kod chyby
- [CLVBUS-49] - proverit indexy nad tabulkami CleverBUS (message, external_call)
- [CLVBUS-50] - Integraní logy - high overview
- [CLVBUS-51] - Kontrola poctu selhani pokud zprava zustava ve stavu PROCESSING
- [CLVBUS-53] - Rozsireni NotificationService o formatovani zprav a vyberu vlastniho adresata
- [CLVBUS-54] - Lepsi podpora pro zakladani internich asynchr. zprav
- [CLVBUS-55] - Implementace "omezovatka"