ActivityLoggingService

OMiLAB Technical Documentation

Software version: v0.2.3

Contents

Revision History

Revision	Date	Author(s)	Description
1.0	27.04.16	D Goetzinger	Created Document

1 Service Details

Prerequisites for the user

- None

Prerequisites for administrative user

- None

Prerequisites for the service operator

- Tomcat server administration skills
- Familiarity with the concepts of the OMiLAB Microservice Infrastructure

Dependencies

- MySQL

Frameworks

- Spring Boot 1.2.5.RELEASE and related

Summary
The ActivityLoggingService provides functionality to log certain events from other services. CRUD actions on important data sets should be especially considered. This enables on the one hand a certain amount of accountability to answer questions like "Who deleted what on which day?". On the other hand the ActivityLoggingService itself or other services can access the related data and visualize current activities, performed by OMiLAB users. Currently this kind of visualization is not implemented, but planned.

Access to the ActivityLoggingService is facilitate through a few classes present in all services, based on the ServiceTemplate

2 User Manual

The ActivityLoggingService currently does not provide any user interface. Although there are plans to change this[1].

[1] https://gitlab.dke.univie.ac.at/omilab-core-infrastructure/OMiLAB_Issues/issues/51

3 DevOps Manual

3.1 Installation

The installation of the ActivityLoggingService may be performed similar to all other services. It can be built from source using maven as build system. The according target to create a deployable package, is *"package"*. The profile *"distribution"* provides a well-documented configuration with sane default values. So the command to build the ActivityLoggingService using the command line interface can be *"mvn -Pdistribution package"*. This will create a deployable war archive in a new generated *"target"* folder.

Before a deployment on the application server is possible, the service has to be configured for the specific environment, what is performed in the file *"application.properties"*. If the configuration-section of the application-distribution.properties has not already been edited before building, the file can be found in the war archive at *"/WEB-INF/classes/application.properties"*.

```
1 # Credentials to access the database
2 spring.datasource.url = jdbc:mysql://localhost:3306/logging
3 spring.datasource.username = logging
4 spring.datasource.password = password
```

What is left to do after the configuration, is the preparation of the database. When the option *"spring.jpa.hibernate.ddl-auto"* is set to *"validate"* the service expects to find valid schema at the database given and will exit, if it is not present. In order to create the necessary schema the file *"schema.sql"* has to be imported to the configured database. After the service is configured and the database schema is imported, the service may be deployed on the application server. The service has been tested extensively with Apache Tomcat 7 and JRE 7, as well as MySQL 5.5.

3.2 Configuration

The ActivityLoggingService provides no further possibility for configuration. It just takes messages from the type OMiLAB Generic Logging Interface and persists these to the database. Queries on these information have to be done directly via SQL.

The core of the RoleService is the table *"security_association"* which depicts the the ternary relation between the project, represented through the service instance, the role and the LDAP username. This is the table that is edited through the user interface.

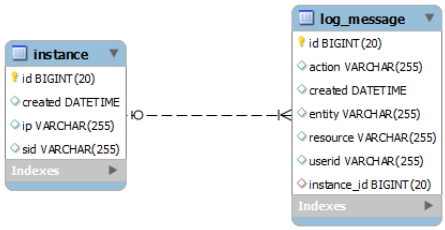

Figure 3.1: ActivityLoggingService database scheme

The column *"sid"* in the table *"instance"* is used to uniquely identify a service, and set at the respective instantiation. For the instantiation the ActivityLoggingService uses its own format, that differs from the OMiLAB Generic Interface. Also the initial registration of a new service at the ActivityLoggingService happens automatically, when the first message is logged.

The other table *"log_message"* stores the actual log messages. The different columns mean the following:

- *action:* The kind of action that has been performed. Typically values like "create", "update" and "delete" can be found here.
- *created:* The date and time, when the action has been performed.
- *entity:* The name of the entity that has been affected by a change.
- *resource:* The id of the resource the action has been performed on.
- *userid:* The username of the user, who performed the action.

id	action	entity	created	resource	userid	instance_id
958	update	Element	2016-02-19 12:52:48	1092	dopplers	2
959	rename	nt:file	2016-02-19 12:57:06	/omilab/JCS/model.png	dopplers	4
960	upload	nt:file	2016-02-19 12:57:55	/omilab/JCS/mc_donalds.png	dopplers	4
961	upload	nt:file	2016-02-19 12:58:02	/omilab/JCS/overall_structure.png	dopplers	4

Table 3.1: Excerpt of table *"log_message"*

id	created	ip	sid
2	2015-09-21 13:53:10	127.0.0.1	TS-VIE-01
4	2015-09-21 14:22:19	127.0.0.1	REPO-01

Table 3.2: Excerpt of table *"instance"*

Table 3.1 and table 3.2 show excerpts of example content on the database of the ActivityLoggingService.

One can see here, that first the first entry references an action that has happened at the *"TextService"*. The user *"dopplers"* updated the text of the *"Element"* with the id *"1092"*. Right after this he renamed a file in the Repository and uploaded two new files for the project *"JCS"*.

If further information on the different resources is required, one should use the id in the column *"resource"* to obtain it from the TextService itself.

3.3 Interface and Implementation details

OMiLAB Generic Interface

No implementation present in the ActivityLoggingService.

OMiLAB Logging Interface

The OMiLAB Logging Interface provides a common interface for all OMiLAB services to send their log messages to. The ActivityLoggingService will then take care of all further processing.

The schema of the format looks like the following:

```
{
 "id": "http://www.omilab.org/LogMessage.json#",
 "$schema": "http://json-schema.org/draft-04/schema#",
 "title": "OMiLAB LogMessage",
 "description": "Represents a LogMessage",
 "type": "object",
 "properties": {
  "userid": {
   "description": "username of the user who caused the
      action",
   "type": "string"
  },
  "resource": {
   "description": "id of the affected resource",
   "type": "integer"
  },
  "sid": {
   "description": "Unique id of the service",
   "type": "string"
  },
  "entity": {
```

```
    "description": "the entity type of the affected resource",
    "type": "string"
  },
  "action": {
    "description": "action that has been performed",
    "type": "string"
  },

 },
 "required": ["userid", "resource", "sid", "entity", "action"]
}
```

An example request may look like the following:

```
{
 "userid": "dgoetzinger",
 "resource" : "4",
 "sid" : "TextService-VIE-01",
 "entity" : "TextElement",
 "action" : "update"
}
```

These messages may be directed to the *"log"* endpoint of ActivityLoggingService, e.g. *"http://localhost:8080/logging/log"*. Please make sure, that the HTTP headers are set correctly. The "Accept" and "Content-Type" values have to be set to *"application/json"*.

Before one can start sending these kind of messages, the *"SID"* has to be registered though. In order to do so, a message only containing the SID has to be sent to *"http://localhost:8080/logging/instanceMgmt"*. Removal is not possible.

Usage in the ServiceTemplate

In the package *"org.omilab.services.template.service.logging"* some classes are present that facilitate the asynchronous access to the ActivityLoggingService. In order to use it in the service, first the setting *"omilab.activity"* has to be set to the full URL (including context, without trailing slash) of the ActivityLoggingService. Furthermore the setting *"omilab.sid"* has to be set to a unique value, in order to facilitate the distinction between the different services. SID stands for "Service Identifier".

In order to use log messsages programmatically one has to inject the class *"LoggingService"* into the class, where it should be used. This may look like the following:

```
@Autowired
private LoggingService logService;
```

From now on, it is possible to access the ActivityLoggingService through the *"logService"* field. The important method here is *"logMessage"*, which expects a new class instance of *"LogMessage"*. This class should be constructed based on the schema described above. An exemplary call looks like the following:

```
1 logService.logMessage(new LogMessage(request.getUsername(),
    "create", item.getClass().getSimpleName(), Long.toString(
    item.getId())));
```

4 Contact

4.1 Service Developer

David Götzinger

Research Group Knowledge Engineering
Faculty of Computer Science
University of Vienna

Währinger Straße 29
1090 Vienna
AUSTRIA

Email: dgoetzinger@dke.univie.ac.at