Logging / I18N

Frameworks used for logging and i18n

- SLF4J API
- FR I18N framework (i18n-core component)

  The I18N framework provides in the i18n-slf4j module, a `LocalizedLogger` class, which should be used everywhere in the code to log messages and/or exceptions.
  The `LocalizedLogger` class delegates to SLF4J API.

  An SLF4J implementation can be chosen at runtime by providing the appropriate jar in the classpath. In particular, two implementations are used:
- OpenDJ implementation, used when the `opendj-slf4j-adapter` jar is included. It delegate to `ErrorLogger` and `DebugLogger` classes from OpenDJ server.
  - this implementation is used in the server.
- java.util.logging implementation, used when the `slf4j-jdk14` jar is included.
  - this implementation is used in the tools.

Log levels

A message can be logged using the following levels (provided by SLF4J API and `LocalizedLogger` class):

- **error**: signals a fatal or non fatal error that requires an action from an administrator
  - corresponds to *fatal* / *error* in OpenDJ server 2
- **warning**: signals a potential or real issue that does not require immediate action but may need an administrative action later
  - corresponds to *warning* in OpenDJ server 2
- **info**: a high-level notice
  - corresponds to *notice* in OpenDJ server 2
- **debug**: informational message
  - corresponds to *info* in OpenDJ server 2
- **trace**: information needed to debug
  - corresponds to *debug* in OpenDJ server 2

Log levels and i18n:

- Messages logged at error, warning, info and debug level **must be** internationalized.
- Messages logged at trace level are not internationalized.

Logger and category and message ID

Each logged message has a category. The category is given by the classname where the message is logged.

This allows to define a single logger per class, with the classname.

However, in order to have a higher kind of grouping, some pre-defined categories with simple names are also used (eg, `CORE`, `SYNC`, ...)

There is a mapping between packages (information which is included in classnames) and these simple categories. If a class does not map to a pre-defined package, then its category is its fully qualified class name.

For example, all classes in the `org.forgerock.opendj.server.core` package or a sub-package are mapped to the `CORE` category.

Note that for debug logging in the server (messages logged by `DebugLogger` class), the pre-defined categories are NOT used.

I18N messages identification

For i18n messages, there is a way to uniquely identify the messages with two properties:

- the resource name of the `LocalizableMessage`.
- the ordinal of the `LocalizableMessage`.

Ordinal is extracted from the message name suffix, while resource name corresponds to the resource file containing the messages.

```
ERR_ADMIN_CANNOT_GET_LISTENER_BASE_1=some message
ERR_ADMIN_CANNOT_GET_MANAGED_OBJECT_2=another message
...
```

Code examples

Loggers

Each class should declare its own logger using the following code. This is valid for server classes as well as client tools classes.

```
private static final LocalizedLogger logger = LocalizedLogger.getLoggerForThisClass();
```

Should you exceptionally need to log to a specific category that does not correspond to the class, you can create another logger to log this specific category:

```
private static final LocalizedLogger extensionLogger = LocalizedLogger.getLocalizedLogger("org.opends.server.
extensions"); // will log to category EXTENSION according to pre-defined mapping
```

Non-debug (I18N) messages

The logging methods accept a localizable message descriptor and its arguments. It is the preferred way of logging:

```
logger.error(ERR_ADMIN_CANNOT_GET_LISTENER_BASE, 123, "some string"); // message with a numeric argument and a
string argument
```

Any exception is passed as the last argument:

```
logger.error(ERR_ADMIN_CANNOT_GET_LISTENER_BASE, 123, "some string", anException);
```

Debug messages

The `LocalizedLogger#trace` method accepts non localized arguments as an alternative to localizable argument:

```
logger.trace("a debug message with arg1: %s and arg2: %s", arg1, arg2)
```

There is also a specific method to trace an exception in addition to the message:

```
logger.traceException(anException, "a debug message with arg1: %s and arg2: %s", arg1, arg2)
```