Ibeo SDK – Logging – Cheat sheet

Basic usage

Includes	To use the logging system the file `"ibeo/common/logging/logging.hpp"` should be included.
Namespace	All logging classes are in the namespace `ibeo::common::logging`.
Create a logger	`LogManager::getInstance().createLogger("ibeo::common::app::Foo")`
Use the global logger	`LogManager::globalLogger()` *Note: use the global logger with care. The more the global logger is used the less you can control the logging output!*
Log Levels	From most to less important: • Critical • Error • Warning • Info • Trace • Debug
Send a log message	Using macros: • `LOGCRITICAL(logger, "Log text.")` • `LOGERROR(logger, "Log text.")` • `LOGWARNING(logger, "Log text.")` • `LOGINFO(logger, "Log text.")` • `LOGTRACE(logger, "Log text.")` • `LOGDEBUG(logger, "Log text.")` Using logger methods: • `logger->critical(LOGMSG << "Log text.");` • `logger->error(LOGMSG << "Log text.");` • `logger->warning(LOGMSG << "Log text.");` • `logger->info(LOGMSG << "Log text.");` • `logger->trace(LOGMSG << "Log text.");` • `logger->debug(LOGMSG << "Log text.");` The log text when using macros or logger methods can be constructed using the stream output operator `"<<"` and IO manipulators: `"Magic number: 0x" << std::hex << std::setw(4) << magicNumber` To add a newline, use `ibeo::common::logging::endl` instead of `std::endl`.

Configuration

Configuration	The configuration of the logging system is done with XML files formatted similar to the log4j2 configuration.
	
	`<?xml version='1.0' encoding='UTF-8'?>`
	`<Configuration>`
	`  <Backends>`
	`  <Backend`
	`  id="ibeo::common::logging::ConsoleLoggerBackend"`
	`  format="%d [%t] %-5level %.30logger - %msg%n" />`
	`  <Backend`
	`  id="ibeo::common::logging::FileLoggerBackend"`
	`  format="%d [%t] %-5level %.30logger - %msg%n">`
	`  <Path>ibeo.log</Path>`
	`  </Backend>`
	`  </Backends>`
	
	`  <Loggers>`
	`  <Root level="debug">`
	`  <BackendRef`
	`  id="ibeo::common::logging::ConsoleLoggerBackend" />`
	`  <BackendRef`
	`  id="ibeo::common::logging::FileLoggerBackend" />`
	`  </Root>`
	`  <Logger`
	`  id="ibeo::common::app::Foo"`
	`  level="critical">`
	`  <BackendRef`
	`  id="ibeo::common::logging::ConsoleLoggerBackend" />`
	`  </Logger>`
	`  </Loggers>`
	`</Configuration>`

Load configuration from file	`LogManager::getInstance().loadConfig("mysubfolder/logconfig.xml")` *Note: if a file named "`logconfig.xml`" exists in the current working directory, it is loaded automatically during startup!*
Set configuration from string	If the configuration is not stored in a file it can be set directly with: `LogManager::getInstance().parseConfig(stringBufferWithXmlFormattedConfiguration)`
Set default log level	In section `<Configuration><Loggers><Root>` set the attribute `level` to one of critical, error, warning, info, trace, or debug: `  <Root level="debug" />` *Note: if no configuration is given the default log level is set to error.*

Set log level for specific logger	In section `<Configuration><Loggers>` locate or create a section `<Logger>` with the id attribute equal to the logger's ID and set the attribute `level` to one of critical, error, warning, info, trace, or debug:
	
	`  <Logger`
	`  id="ibeo::common::app::Foo"`
	`  level="critical">`
	`  </Logger>`

Set default	In section `<Configuration><Loggers><Root>` create a section `<BackendRef>`

backends	with the attribute equal to the backend's ID for each backend that should be used as default.
	
	`<Root level="debug">`
	`  <BackendRef`
	`  id="ibeo::common::logging::ConsoleLoggerBackend" />`
	`</Root>`

Set backends for specific logger	In section `<Configuration><Loggers>` locate or create a section `<Logger>` with the id attribute equal to the logger's ID. In this section create another section `<BackendRef>` with the attribute equal to the backend's ID for each backend that should be used by this logger.
	
	`<Logger id="ibeo::common::app::Foo">`
	`  <BackendRef`
	`  id="ibeo::common::logging::ConsoleLoggerBackend" />`
	`</Logger>`

Backends	The logging system itself provides two backends which can be used by loggers: • `ibeo::common::logging::ConsoleLoggerBackend` for logging to the console (stdout or stderr according to the log level) • `ibeo::common::logging::FileLoggerBackend` for logging to a file Custom logger backends can be added to the system (see Custom Backends). If no configuration is given, all loggers use the `ConsoleLoggerBackend` only.

Set log message format	Formatting the log messages is done in the backends. To set a custom format locate or create a section `<Backend>` with the id attribute equal to the backend's ID in the section `<Configuration><Backends>` and set the `format` attribute (see Formatting).
	
	`<Backend`
	`  id="ibeo::common::logging::ConsoleLoggerBackend"`
	`  format="%d [%t] %-5level %.30logger - %msg%n" />`

Set log file path	The path to the log file is configured in the `FileLoggerBackend`. In section `<Configuration><Backends>` locate or create a section `<Backend>` with the id attribute `ibeo::common::logging::FileLoggerBackend`. In this section create another section `<Path>` with the value set to the path of the log file.
	
	`<Backend`
	`  id="ibeo::common::logging::FileLoggerBackend">`
	`  <Path>ibeo.log</Path>`
	`</Backend>`

Formatting

The conversion of a log message into a textual output is determined by a format string that is set in the configuration. This string consists of conversion patterns which are replaced by the corresponding log message field and plain text in any order and multiplicity. Each conversion pattern can have alignment and length parameters, some can be extended with optional arguments enclosed in brackets “{}” (see below table).

Print the log level	The level of the log message is printed using the `level` conversion pattern which has two optional arguments: • `length` is used to limit the length of the output (default is no limit). • `lowerCase` is used to convert the output to lower case, if set to `true`, or upper case otherwise (default is upper case). `“%level{length=1}{lowerCase=true}”`
Print date/time	To print the date and/or time when the log message was created, use the date conversion pattern which has one optional argument determining the format of the date/time string. Possible values are: • `DEFAULT`: use default format (`“%Y-%m-%d %H:%M:%S,%s”`) • `UNIX`: print the number of seconds since start of epoch (1970-01-01 00:00:00). • `UNIX_MILLIS`: print the number of milliseconds since start of epoch (1970-01-01 00:00:00). • Any valid format string that can be interpreted by the `strftime` function (see here). As a special extension the parameter `%s` (lower case character) can be used here to print the milliseconds. `“%date{“Now it's %T,%s.”}”`
Print file name	The `file` conversion pattern is used to print path and name of the source file containing the code that sent the log message. This pattern has a single optional argument that limits the number of elements to be printed. Truncation is done from the beginning of the file path. `“%file{2}”` `if` used with a file path `“foo1/foo2/bar.cpp”` it will print `“foo2/bar.cpp”`.
Print function name	The `func` or `function` conversion pattern is used to print the name of the function containing the code that sent the log message. `“%func”`
Print line number	The `line` conversion pattern is used to print the line number within the source file containing the code that sent the log message. `“%line”`
Print the log message text	The `msg` or `message` conversion pattern is used to print the log message text. `“%msg”`
Print the logger ID	The `logger` conversion pattern is used to print the ID of the logger that sent the message. This pattern has a single optional argument that limits the number of elements to be printed. Truncation is done from the beginning of the logger ID.

	`"%logger{2}"` `if` used with a logger ID `"ibeo::ref::tools::app:Foo"` it will print `"app::Foo"`.
Print the sequence number	The `seqNo` or `sequenceNumber` conversion pattern is used to print the sequence number of log message. This global number is incremented every time a new log message is created . Thus, it can be used to uniquely identify a log message int the output and for sorting. `"%seqNo"`
Print thread ID	The `thread` conversion pattern prints the ID of the thread that executes the code that sent the log message. `"%thread"` *Note: the output may vary across different operating systems.*

Alignment and/or length modifier	Each conversion pattern can have an optional alignment and/or length modifier between the percent sign and the conversion pattern name. Possible formats are:
	
	`"%15msg"` the output is at least 15 characters wide and right aligned, e.g. `"·····Log text."`
	`"%-15msg"` the output is at least 15 characters wide and left aligned, e.g. `"Log text....."`
	`"%.5msg"` the output is at most 5 characters wide and truncated from beginning, e.g. `"text."`
	`"%.-5msg"` the output is at most 5 characters wide and truncated from end, e.g. `"Log t"`
	`"%10.15msg"` the output is at least 10 characters and at most 15 characters wide, right aligned if shorter and truncated from beginning if longer
	`"%-10.15msg"` the output is at least 10 characters and at most 15 characters wide, left aligned if shorter and truncated from beginning if longer
	`"%10.-15msg"` the output is at least 10 characters and at most 15 characters wide, right aligned if shorter and truncated from end if longer
	`"%-10.-15msg"` the output is at least 10 characters and at most 15 characters wide, left aligned if shorter and truncated from end if longer

Custom Backends

Sometimes the provided backends are not sufficient, e.g. when you want to log into a database or a message window of a GUI application. Therefore, the logging system provides an interface for writing custom backends.

Create a synchronous backend	A synchronous backend handles the log messages immediately after they were sent to the logging system. As this task runs in the same thread, the application is typically blocked during this time. Thus, this method is good when you need immediate output, but should not be used for longer processing times. To implement a synchronous backend create a class that derives from `LoggerBackend` and override the `log` method.
Create an asynchronous backend	An asynchronous backend uses a queue internally to decouple the processing of the log messages in the backend. Thus, the application is not blocked when the log message is sent and the processing is done in a separate thread. This is the preferred method when the processing takes a noticable time, e.g. when sending log messages to am remote database. To implement an asynchronous backend create a class that derives from `AsyncLoggerBackend` and override the `logAsync` method.
Register a custom backend	To let the logging system know about an additional backend the method `LogManager::registerBackend()` should be called with an instance of the new backend.

Configuring a custom backend	The configuration of a custom backend is placed in the configuration file in the same way as for the provided backends (see Configuration). The logging system takes care that the corresponding section in the XML file is given to the `configure()` method in the custom backend. The content of this section is transparent to the logging system. So, any parameter definition can be placed here as long it adheres to the XML formatting rules.
	E.g.:
	`<Backend`
	`id="ibeo::common::database::DbLoggerBackend">`
	`<ServerName>db.ibeo.as</ServerName>`
	`<UserName>user</UserName>`
	`<Password>password</Password>`
	`<Instance>myDatabase</Instance>`
	`</Backend>`
	
	*Note: all custom backends must be registered **before** the configuration is loaded! Otherwise, the custom backends might not work as expected.*