AKADEMY
TWENTYTWENTYONE

The Art of Logging

How to get helpful error reports

Andreas Cord-Landwehr

cordlandwehr@kde.org

Next up...

About me

- I am Andreas (nick: CoLa) and I am with KDE for about a decade
- In my day job, I am working on software for big agriculture vehicles

My Motivation for this Talk

- Getting a bug report without proper logs is frustrating, especially when you cannot reproduce it
- I get annoyed when I start an application and see tons of unuseful (for me in a user perspective) log messages
- Time is valueable and you should not waste it by spending too much time on reading logs to analyze a problem
- . . . and I think that doing logging correctly is easy :D

About me

- I am Andreas (nick: CoLa) and I am with KDE for about a decade
- In my day job, I am working on software for big agriculture vehicles

My Motivation for this Talk

- Getting a bug report without proper logs is frustrating, especially when you cannot reproduce it
- I get annoyed when I start an application and see tons of unuseful (for me in a user perspective) log messages
- Time is valueable and you should not waste it by spending too much time on reading logs to analyze a problem
- . . . and I think that doing logging correctly is easy :D

Developer Logs

Log messages are text lines that are printed by the a program when something "meaningful" happens

Examples:

- something happened that should not happen
- output the helps to follow program logic and where a developer can analyze if steps fit to his thinking
- tracking of user interactions in order to make an error analysis possible
- tracing points (mostly out of scope here)

Usually, logs are printed to stdout – at the end of the talk I will show alternatives

Log Message Severity Levels

Fatal	Something so critical happend that you only print a log message and then die.
Critical	Something bad happend and an operation could not be performed; possibly with data loss.
Warning	Something unexpected/unwanted happened but the program can resume and handle with this situation.
Info	A low frequent change in the programs busines state done by user or by data.
Debug	Messages that print information for "interesting" program locations and help you to see what happes.
Trace	High frequent debug messages that can be activated to deeply analyse specific operations.

Log Message Severity Levels

Fatal	Something so critical happend that you only print a log message and then die.
Critical	Something bad happend and an operation could not be performed; possibly with data loss.
Warning	Something unexpected/unwanted happened but the program can resume and handle with this situation.
Info	A low frequent change in the programs busines state done by user or by data.
Debug	Messages that print information for "interesting" program locations and help you to see what happes.
Trace	High frequent debug messages that can be activated to deeply analyse specific operations.

Log Message Severity Levels

Fatal	Something so critical happend that you only print a log message and then die.
Critical	Something bad happend and an operation could not be performed; possibly with data loss.
Warning	Something unexpected/unwanted happened but the program can resume and handle with this situation.
Info	A low frequent change in the programs busines state done by user or by data.
Debug	Messages that print information for “interesting” program locations and help you to see what happes.
Trace	High frequent debug messages that can be activated to deeply analyse specific operations.

Log Message Severity Levels

Fatal	Something so critical happend that you only print a log message and then die.
Critical	Something bad happend and an operation could not be performed; possibly with data loss.
Warning	Something unexpected/unwanted happened but the program can resume and handle with this situation.
Info	A low frequent change in the programs busines state done by user or by data.
Debug	Messages that print information for “interesting” program locations and help you to see what happes.
Trace	High frequent debug messages that can be activated to deeply analyse specific operations.

Log Message Severity Levels

Fatal	Something so critical happend that you only print a log message and then die.
Critical	Something bad happend and an operation could not be performed; possibly with data loss.
Warning	Something unexpected/unwanted happened but the program can resume and handle with this situation.
Info	A low frequent change in the programs busines state done by user or by data.
Debug	Messages that print information for “interesting” program locations and help you to see what happes.
Trace	High frequent debug messages that can be activated to deeply analyse specific operations.

Log Message Severity Levels

Fatal	Something so critical happend that you only print a log message and then die.
Critical	Something bad happend and an operation could not be performed; possibly with data loss.
Warning	Something unexpected/unwanted happened but the program can resume and handle with this situation.
Info	A low frequent change in the programs busines state done by user or by data.
Debug	Messages that print information for “interesting” program locations and help you to see what happes.
Trace	High frequent debug messages that can be activated to deeply analyse specific operations.

Next up...

1 Part 1: Introduction

2 Part 2: The Qt Logging Framework in KDE

3 Part 3: Log Sinks

4 The End

QDebug

Stream-API (there is also printf API, but I will only use stream API in the following)

```
1   #include <QDebug>
2   qDebug() << "welcome!";
3   // welcome!
4   qWarning() << QDate::currentDate();
5   // QDate("2021-06-20")
6   qCritical() << QRect(1, 2, 3, 4);
7   // QRect(1,2 3x4)
8   int year = 2021;
9   qDebug() << "Akademy" << year;
10  // Akademy 2021
11  qDebug().nospace().noquote()
12      << "Akademy" << year;
13  // Akademy2021
14  qDebug() << Qt::hex
15      << Qt::uppercasedigits << year;
16  // 7E5
```

- log severity: `qDebug`, `qWarning`, `qInfo`, `qCritical`, `qFatal`
- support for **many** Qt data types with pretty printing
- you can also register your own debug printers for your data type
- simple output assembly
- `QDebugStateSaver` for resetting stream format state

→ `https://doc.qt.io/qt-5/qdebug.html`

QDebug

Stream-API (there is also printf API, but I will only use stream API in the following)

```
1   #include <QDebug>
2   qDebug() << "welcome!";
3   // welcome!
4   qWarning() << QDate::currentDate();
5   // QDate("2021-06-20")
6   qCritical() << QRect(1, 2, 3, 4);
7   // QRect(1,2 3x4)
8   int year = 2021;
9   qDebug() << "Akademy" << year;
10  // Akademy 2021
11  qDebug().nospace().noquote()
12      << "Akademy" << year;
13  // Akademy2021
14  qDebug() << Qt::hex
15      << Qt::uppercasedigits << year;
16  // 7E5
```

- log severity: `qDebug`, `qWarning`, `qInfo`, `qCritical`, `qFatal`
- support for **many** Qt data types with pretty printing
- you can also register your own debug printers for your data type
- simple output assembly
- `QDebugStateSaver` for resetting stream format state

→ `https://doc.qt.io/qt-5/qdebug.html`

QDebug

Stream-API (there is also printf API, but I will only use stream API in the following)

```
1   #include <QDebug>
2   qDebug() << "welcome!";
3   // welcome!
4   qWarning() << QDate::currentDate();
5   // QDate("2021-06-20")
6   qCritical() << QRect(1, 2, 3, 4);
7   // QRect(1,2 3x4)
8   int year = 2021;
9   qDebug() << "Akademy" << year;
10  // Akademy 2021
11  qDebug().nospace().noquote()
12      << "Akademy" << year;
13  // Akademy2021
14  qDebug() << Qt::hex
15      << Qt::uppercasedigits << year;
16  // 7E5
```

- log severity: `qDebug`, `qWarning`, `qInfo`, `qCritical`, `qFatal`
- support for **many** Qt data types with pretty printing
- you can also register your own debug printers for your data type
- simple output assembly
- `QDebugStateSaver` for resetting stream format state

→ `https://doc.qt.io/qt-5/qdebug.html`

QDebug

Stream-API (there is also printf API, but I will only use stream API in the following)

```
1   #include <QDebug>
2   qDebug() << "welcome!";
3   // welcome!
4   qWarning() << QDate::currentDate();
5   // QDate("2021-06-20")
6   qCritical() << QRect(1, 2, 3, 4);
7   // QRect(1,2 3x4)
8   int year = 2021;
9   qDebug() << "Akademy" << year;
10  // Akademy 2021
11  qDebug().nospace().noquote()
12      << "Akademy" << year;
13  // Akademy2021
14  qDebug() << Qt::hex
15      << Qt::uppercasedigits << year;
16  // 7E5
```

- log severity: `qDebug`, `qWarning`, `qInfo`, `qCritical`, `qFatal`
- support for **many** Qt data types with pretty printing
- you can also register your own debug printers for your data type
- simple output assembly
- `QDebugStateSaver` for resetting stream format state

→ `https://doc.qt.io/qt-5/qdebug.html`

QDebug

Stream-API (there is also printf API, but I will only use stream API in the following)

```
1   #include <QDebug>
2   qDebug() << "welcome!";
3   // welcome!
4   qWarning() << QDate::currentDate();
5   // QDate("2021-06-20")
6   qCritical() << QRect(1, 2, 3, 4);
7   // QRect(1,2 3x4)
8   int year = 2021;
9   qDebug() << "Akademy" << year;
10  // Akademy 2021
11  qDebug().nospace().noquote()
12      << "Akademy" << year;
13  // Akademy2021
14  qDebug() << Qt::hex
15      << Qt::uppercasedigits << year;
16  // 7E5
```

- log severity: `qDebug`, `qWarning`, `qInfo`, `qCritical`, `qFatal`
- support for **many** Qt data types with pretty printing
- you can also register your own debug printers for your data type
- simple output assembly
- `QDebugStateSaver` for resetting stream format state

→ https://doc.qt.io/qt-5/qdebug.html

QDebug

Stream-API (there is also printf API, but I will only use stream API in the following)

```
1   #include <QDebug>
2   qDebug() << "welcome!";
3   // welcome!
4   qWarning() << QDate::currentDate();
5   // QDate("2021-06-20")
6   qCritical() << QRect(1, 2, 3, 4);
7   // QRect(1,2 3x4)
8   int year = 2021;
9   qDebug() << "Akademy" << year;
10  // Akademy 2021
11  qDebug().nospace().noquote()
12      << "Akademy" << year;
13  // Akademy2021
14  qDebug() << Qt::hex
15      << Qt::uppercasedigits << year;
16  // 7E5
```

- log severity: `qDebug`, `qWarning`, `qInfo`, `qCritical`, `qFatal`
- support for **many** Qt data types with pretty printing
- you can also register your own debug printers for your data type
- simple output assembly
- `QDebugStateSaver` for resetting stream format state

→ `https://doc.qt.io/qt-5/qdebug.html`

Categorized Logs

The better way

Group log messages by hierarchical categories:

```
category.subcategory.subsubcategory.[...]
```

Log outputs can be enabled/disabled at runtime by category and message type.

Define Global Logging Categories

```
Q_DECLARE_LOGGING_CATEGORY(name) // declare it in header

// define category and make it usable with string identifier
Q_LOGGING_CATEGORY(name, string)
// optional msgType sets enabled minimal severity
Q_LOGGING_CATEGORY(name, string, msgType)
```

Categorized Logs

The better way

Group log messages by hierarchical categories:

```
category.subcategory.subsubcategory.[...]
```

Log outputs can be enabled/disabled at runtime by category and message type.

Define Global Logging Categories

```
Q_DECLARE_LOGGING_CATEGORY(name) // declare it in header

// define category and make it usable with string identifier
Q_LOGGING_CATEGORY(name, string)
// optional msgType sets enabled minimal severity
Q_LOGGING_CATEGORY(name, string, msgType)
```

Categorized Logs

Example

loggingcategories.h

```
1    #pragma once
2    #include <QLoggingCategory>
3    Q_DECLARE_LOGGING_CATEGORY(mycoolcategory);
```

loggingcategories.cpp

```
1    #include "loggingcategories.h"
2    Q_LOGGING_CATEGORY(mycoolcategory, "myapp.foo", QtWarningMsg);
```

Just using it:

```
1    #include "loggingcategories.h"
2    qCDebug(mycoolcategory) << "log message in category myapp.foo";
```

Log Filtering

Assume we have an app `myapp` with logging categories:

1. `<QLibraryInfo::DataPath>/qtlogging.ini`, `[Rules]` section
2. `.config/QtProject/qtlogging.ini`, `[Rules]` section
3. `setFilterRules(const QString &rules)`
4. `[Rules]` section of file set in `QT_LOGGING_CONF`
5. `QT_LOGGING_RULES` environment variable

Output filtering with `QT_LOGGING_RULES` variable:

```
1 QT_LOGGING_RULES=*=true ./myapp # we see everything
2 QT_LOGGING_RULES=myapp.lib=true,myapp.backend=false ./myapp # only lib, no
      backend
```

Logging Rules Format

```
category[.type] = true|false
```

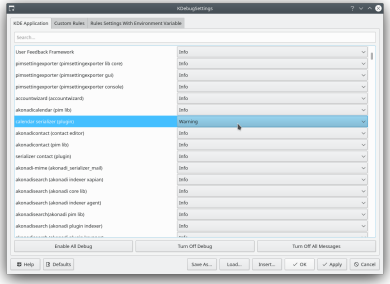
```
1    *=true
2    kate.*=false
3    kwrite.debug=true
4    *.critical=true
5    *.strangethings.*=true
```

Rule Formats

- type can be any of: debug, info, warning, critical
- ∗ wildcard only allowed as first and/or last character
- rules evaluated from first to last

User Friendly Log Filtering

Use `kdebugsettings` to edit `.config/QtProject/qtlogging.ini`:



Note: To allow configuration, you have to install the logging category names though! (`/usr/share/qlogging-categories5/`)

Using Extra-CMake-Modules

api.kde.org/ecm/module/ECMQtDeclareLoggingCategory.html

```
ecm_qt_declare_logging_category( MYPROJECT_SRCS
    HEADER "myproject_debug.h"
    IDENTIFIER "MYPROJECT_DEBUG"
    CATEGORY_NAME "myproject"
    OLD_CATEGORY_NAMES "myprojectlog"
    DESCRIPTION "My project"
    EXPORT MyProject
)
ecm_qt_export_logging_category( IDENTIFIER "MYPROJECT_SUBMODULE_DEBUG"
    CATEGORY_NAME "myproject.submodule"
    DESCRIPTION "My project - submodule"
    EXPORT MyProject
)
ecm_qt_install_logging_categories( EXPORT MyProject
    FILE myproject.categories
    DESTINATION "${KDE_INSTALL_LOGGINGCATEGORIESDIR}"
)
```

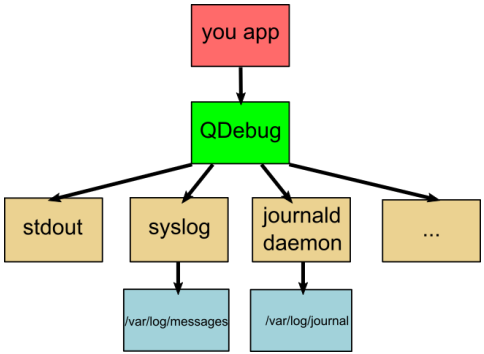
Next up...

Logging Sinks

Perspective = we look at a full system with many apps

Question: How to store logs and serialize async log outputs?

- Qt supports several log backend integrations apart from stdout
- backend selected at Qt configure time (eg. Fedora enables journald)
- when enabled, such log databases can be super helpful
- . . . unless there are too many logs



Note: Per default journald forwards system session logs to syslog

journald

I like it

- Internal log rotation and file consistency checks
- Pretty robost database files with internal index handling for fast usage
- Awesome CLI tool: `journalctl`:
  1. `journalctl -b 1` : only last boot's logs
  2. `journalctl -u sddm` : only log of `sddm` systemd service
  3. `journalctl -u sddm -p 1..4` only `sddm` output up to warning
  4. ... and much more

Remarks for Integration

- for Qt only one backend must be configured at once (default = none)
- when backend is configured, you do not see console output unless: `QT_FORCE_STDERR_LOGGING=1`
- systemd services (thanks @Plasma team!) are nicely integrated

journald

I like it

- Internal log rotation and file consistency checks
- Pretty robost database files with internal index handling for fast usage
- Awesome CLI tool: `journalctl`:
  1. `journalctl -b 1` : only last boot's logs
  2. `journalctl -u sddm` : only log of `sddm` systemd service
  3. `journalctl -u sddm -p 1..4` only `sddm` output up to warning
  4. ... and much more

Remarks for Integration

- for Qt only one backend must be configured at once (default = none)
- when backend is configured, you do not see console output unless: `QT_FORCE_STDERR_LOGGING=1`
- systemd services (thanks @Plasma team!) are nicely integrated

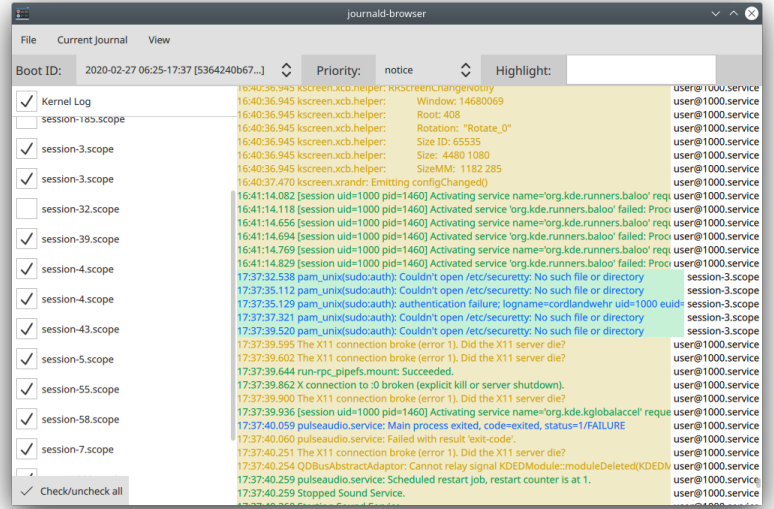
journald with colors and filters

Suggestion for nicer integration

`https://invent.kde.org/libraries/kjournald`

- goal: QAbstractItemModel abstraction of journald's C-API
- allows trivial integration into applications: I am looking for a nice use case in KDE :)
- ships a reference implementation for a QtQuick based journald browser
  1. filter by boots, systemd services and severity
  2. rainbow colors for services
- my main use case right now: offline analysis of journald databases from embedded devices → could this be helpful for Plasma mobile, too?

kjournald-browser



Next up...

1 Part 1: Introduction

2 Part 2: The Qt Logging Framework in KDE

3 Part 3: Log Sinks

4 The End

Key Messages

Please take this home

- Only use categorized logging
- Add all meaningful log messages you want
- Disable verbose logs per default (set level to eg. Warning or Info)
- For bug reports you can tell used to enable more logs
- Start `journalctl` or kjournald-browser and look at your own system

Skipped due to time: you can format your output messages as you like
→ `https://doc.qt.io/qt-5/debug.html`

The End

Question Time

Contact

Contact mail: cordlandwehr@kde.org
irc: CoLa