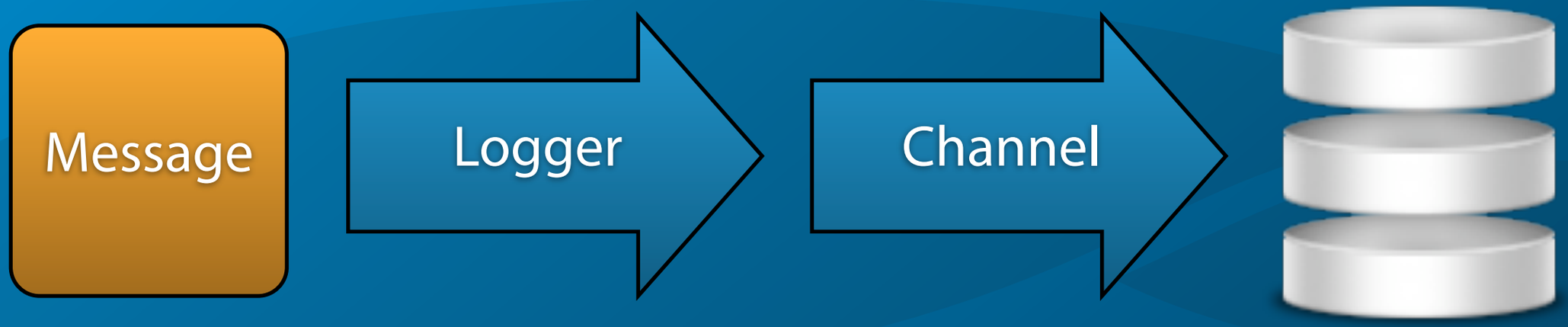
Logging

Working with the POCO logging framework.

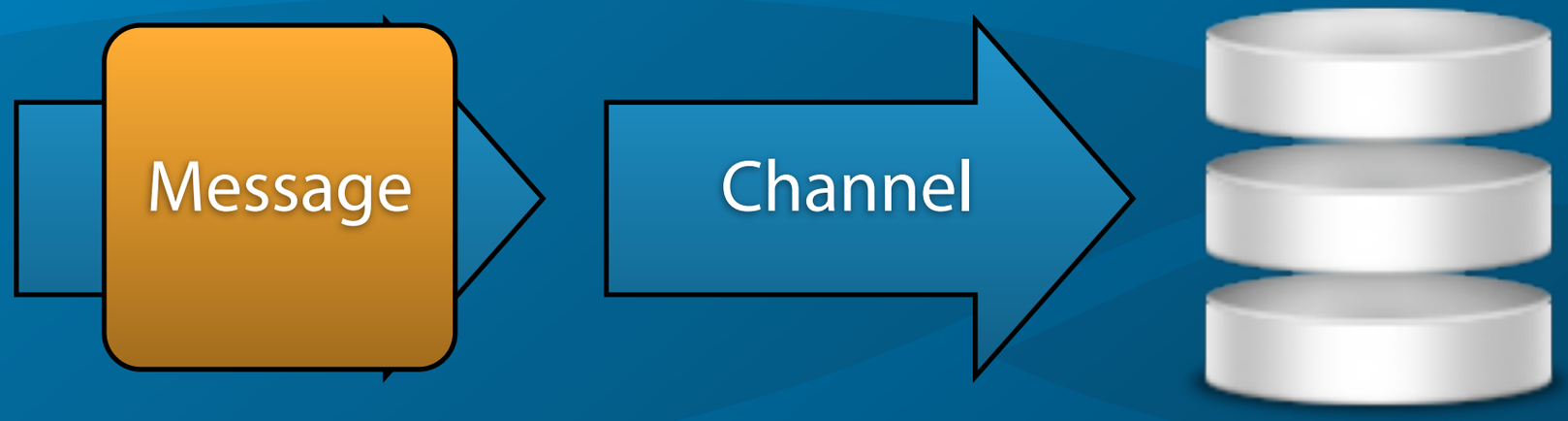
Overview

- Messages, Loggers and Channels
- Formatting
- Performance Considerations

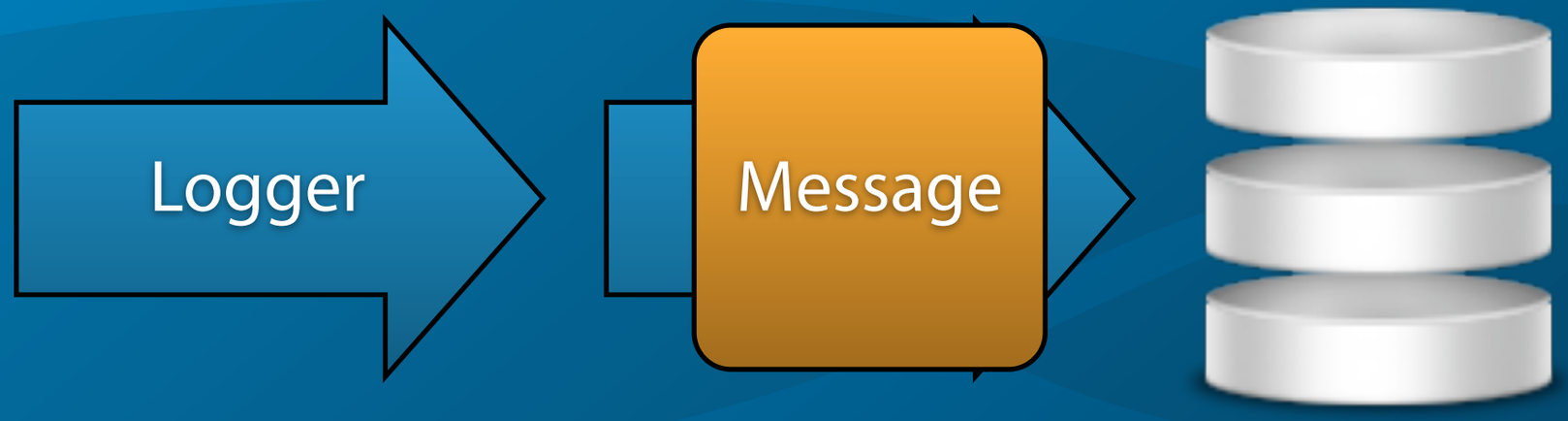
Logging Architecture



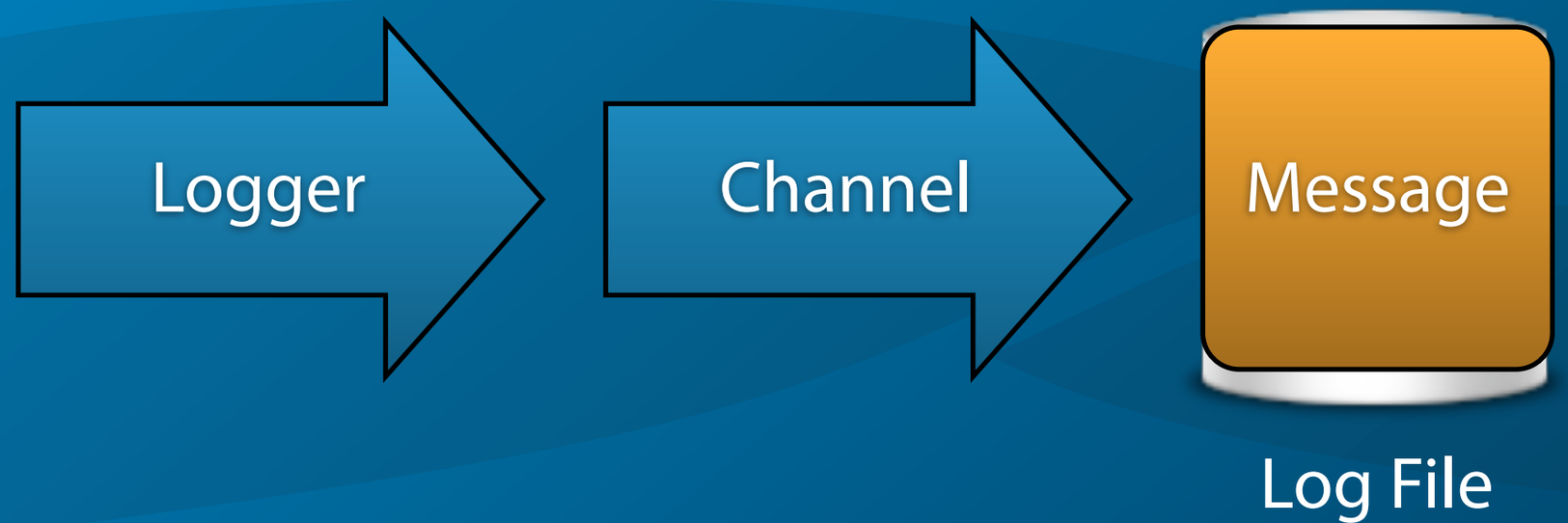
Logging Architecture



Logging Architecture



Logging Architecture



Log Messages

- All log messages are stored and transported in Poco::Message objects.
- #include "Poco/Message.h"
- A message has
  - a priority
  - a source
  - a text
  - a timestamp
  - a process and thread identifier
  - optional parameters (name-value pairs)

Message Priority

- POCO defines eight message priorities:
  - PRIO_FATAL (highest priority)
  - PRIO_CRITICAL
  - PRIO_ERROR
  - PRIO_WARNING
  - PRIO_NOTICE
  - PRIO_INFORMATION
  - PRIO_DEBUG
  - PRIO_TRACE (lowest priority)
- void setPriority(Priority prio)
  Priority getPriority() const

Message Source

- Describes the source of a log message.
- Usually set by a Logger to its name.
- Thus the name for a logger should be chosen wisely:
  - the name of the class producing the message, or
  - the name of the subsystem producing the message.
- void setSource(const std::string& source)
  const std::string& getSource() const

Message Text

- The actual message to be logged.
- No requirements regarding format, length, etc.
- May be modified by a formatter before it appears in log destination.

- void setText(const std::string& text)
  const std::string& getText() const

Message Timestamp

- The date and time the message was created, with up to microsecond precision.
- Automatically initialized by the constructor of Poco::Message to the current date and time.
- void setTime(const Timestamp& time)
  const Timestamp& getTime() const

Process and Thread Identifier

- The Process Identifier (PID) is a long int value, storing the system's process ID.
- The Thread Identifier (TID) is also a long int value storing the serial number of the current thread.
- Additionally, the name of the current thread is stored.
- Process Identifier, Thread Identifier and Thread Name are initialized in the constructor of Poco::Message.

Process and Thread Identifier (cont'd)

- void setThread(const std::string& threadName)
  const std::string& getThread() const
- void setTid(long tid)
  long getTid() const
- void setPid(long pid)
  long getPid() const

Message Parameters

- A message can store an arbitrary number of name-value pairs.
- Names and values can be arbitrary strings.
- Message parameters can be referenced in a formatter.
- Message parameters are accessed using the index operator.

Logger

- Poco::Logger is the main entry point into the logging framework.
- #include "Poco/Logger.h"
- An application uses instances of the Poco::Logger class to generate log messages.
- Every logger has an attached Channel, which delivers the messages to their destination (possibly, via other channels).
- Every logger has a name, which becomes the source of all messages generated by that logger. The name is set upon creation and cannot be changed later.

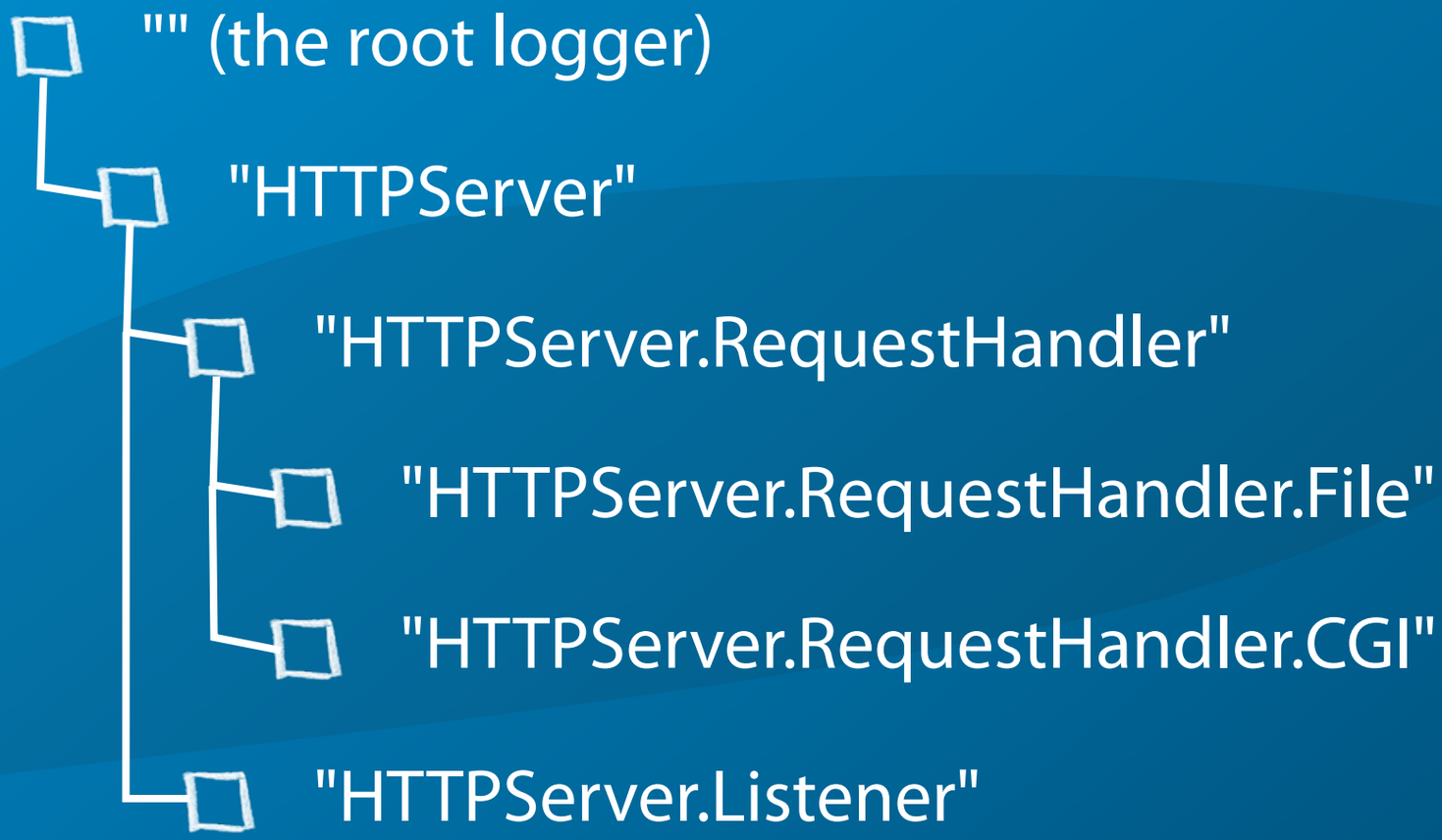
Logger (cont'd)

- A logger filters messages based on their priority.
  - Only messages with a priority equal or higher than the logger's log level are propagated.
  - Example: A logger with the log level set to PRIO_ERROR will only propagate messages with priority PRIO_ERROR, PRIO_CRITICAL or PRIO_FATAL. Messages with priority PRIO_WARNING or lower will be discarded.

Logger Hierarchy

- Based on their name, loggers form a tree-like hierarchy.
- A logger's name consists of one or more components, separated by a period. Each component corresponds to a namespace within the namespace of the previous component.
- A special logger, the root logger, is a logger with an empty string as its name. It forms the root of the logger hierarchy.
- There is no limit on the maximum depth of the logger hierarchy.

Logger Hierarchy Example



Logger Hierarchy Inheritance

- A newly created logger inherits its log level and attached channel from its ancestor in the logger hierarchy.
- Example: on creation, the logger "HTTPServer.RequestHandler.CGI" will inherit its log level and attached channel from the logger named "HTTPServer.RequestHandler".
- Once a logger has been fully created, it is no longer related to its ancestor. In other words, changes to log level or channel of a logger have no effect on its already existing descendants.
- It is possible to set log level and attached channel for a complete sub hierarchy at once.

Logging Messages

- void log(const Message& msg)
  if the message's priority is greater than or equal to the logger's log level, propagates the message to the attached channel. The message is left unchanged.

- void log(const Exception& exc)
  creates and logs a message with priority PRIO_ERROR and the exception's display text

Logging Messages (cont'd)

- void fatal(const std::string& text)
  void critical(const std::string& text)
  void error(const std::string& text)
  void warning(const std::string& text)
  void notice(const std::string& text)
  void information(const std::string& text)
  void debug(const std::string& text)
  void trace(const std::string& text)

  creates and logs a message with the respective priority and the given text.

Logging Messages (cont'd)

- void dump(const std::string& text,
  const void* buffer, int length,
  Message::Priority prio = Message::PRIO_DEBUG)
  logs the given message with the given priority. The text is followed by a hex dump of the given buffer.

Determining Log Levels

- bool is(int level) const
  returns true if the logger's log level is at least level.

- bool fatal() const
  bool critical() const
  bool error() const
  bool warning() const
  bool notice() const
  bool information() const
  bool debug() const
  bool trace() const
  return true if the logger's log level is at least the respective level

Accessing Loggers

- POCO manages a global map of loggers.
- You do not create a logger yourself. Instead, you ask POCO to give you a reference to a logger.
- POCO creates new loggers dynamically on demand.
- `static Logger& get(const std::string& name)`
  returns a reference to the logger with the given name. Creates the logger if necessary.
- It is safe (and recommended, for performance reasons), to store references to loggers once they have been obtained.

```
#include "Poco/Logger.h"

using Poco::Logger;

int main(int argc, char** argv)
{
    Logger& logger = Logger::get("TestLogger");

    logger.information("This is an informational message");
    logger.warning("This is a warning message");

    return 0;
}
```

Channels

- A subclass of Poco::Channel is responsible for delivering messages (Poco::Message) to their destination (e.g., the console or a log file).
- Every Poco::Logger (which itself is a subclass of Poco::Channel) is connected to a Poco::Channel.
- POCO provides various subclasses of Poco::Channel that deliver messages to the console, log files or the system's logging facility.
- You can define your own channel classes.
- Channels use reference counting for memory management.

Channel Properties

- A channel can support an arbitrary number of properties (name-value pairs), which are used to configure it.
- Properties are set with the setProperty() member function:
  void setProperty(const std::string& name, const std::string& value)
- The value of a property can be obtained with getProperty():
  std::string getProperty(const sdt::string& name)
- These two functions are defined in the Poco::Configurable class, which is a super class of Poco::Channel.

ConsoleChannel

- Poco::ConsoleChannel is the most basic channel implementation.
- #include "Poco/ConsoleChannel.h"
- It simply writes the text of any message it receives to the standard output (std::clog).
- It has no configurable properties.
- It is the default channel for the root logger.

WindowsConsoleChannel

- Poco::WindowsConsoleChannel is similar to ConsoleChannel, but writes directly to the Windows console instead of std::clog.
- #include "Poco/WindowsConsoleChannel.h"
- It simply writes the text of any message it receives to the Windows console.
- It has no configurable properties.
- UTF-8 encoded text is supported.

NullChannel

- Poco::NullChannel discards all messages sent to it.
- #include "Poco/NullChannel.h"
- It also ignores all properties set with setProperty().

SimpleFileChannel

- Poco::SimpleFileChannel is the simplest way to write a log file.
- The message's text is appended to a file, followed by a newline.
- Optional simple log file rotation is supported: once the primary log file exceeds a certain size, a secondary log file is created (or truncated, if it already exists). If the secondary log file exceeds the maximum size, the primary log file is truncated, and logging continues with the primary log file, and so on.

SimpleFileChannel Properties

Property	Description
path	The path of the primary log file.
secondaryPath	The path of the secondary log file. Defaults to <path>.1
rotation	The optional log file rotation mode: never: no rotation (default) <n>: rotate if file size exceeds <n> bytes <n> K: rotate if file size exceeds <n> Kilobytes <n> M: rotate if file size exceeds <n> Megabytes

```
#include "Poco/Logger.h"
#include "Poco/SimpleFileChannel.h"
#include "Poco/AutoPtr.h"

using Poco::Logger;
using Poco::SimpleFileChannel;
using Poco::AutoPtr;

int main(int argc, char** argv)
{
    AutoPtr<SimpleFileChannel> pChannel(new SimpleFileChannel);
    pChannel->setProperty("path", "sample.log");
    pChannel->setProperty("rotation", "2 K");

    Logger::root().setChannel(pChannel);

    Logger& logger = Logger::get("TestLogger"); // inherits root channel

    for (int i = 0; i < 100; ++i)
        logger.information("Testing SimpleFileChannel");

    return 0;
}
```

FileChannel

- Poco::FileChannel provides full-blown log file support.
- #include "Poco/FileChannel.h"
- The message's text is appended to a file, followed by a newline.
- Supports log file rotation based on file sizes or time intervals.
- Supports automatic archiving (with different file naming strategies), compression (gzip) and purging (based on age or number of archived files) of archived log files.

FileChannel Properties

Property	Description
path	The path of the log file
rotation	The log file rotation mode: never: no rotation <n>: rotate if file size exceeds <n> bytes <n> K: rotate if file size exceeds <n> Kilobytes <n> M: rotate if file size exceeds <n> Megabytes [day,][hh:][mm]: rotated on specified weekday/time daily/weekly/monthly: every day/seven days/thirty days <n> hours/weeks/months: every <n> hours/weeks/months

FileChannel Properties (cont'd)

Property	Description
archive	The naming of archived log files: number: a automatically incremented number, starting with 0, is appended to the log file name. The newest archived file always has 0. timestamp: a timestamp in the form YYYYMMDDHHMMSS is appended to the log file.
times	Specifies whether times for rotation are treated as local or UTC. Valid values are local and utc.
compress	Automatically compress archived files. Specify true or false.

FileChannel Properties (cont'd)

Property	Description
purgeAge	Specify a maximum age for archived log files. Files older than this age will be purged. <n> [seconds]/minutes/hours/days/weeks/months
purgeCount	Specify a maximum number of archived log files. If that number is reached, the oldest archived log file will be purged.

```
#include "Poco/Logger.h"
#include "Poco/FileChannel.h"
#include "Poco/AutoPtr.h"

using Poco::Logger;
using Poco::FileChannel;
using Poco::AutoPtr;

int main(int argc, char** argv)
{
    AutoPtr<FileChannel> pChannel(new FileChannel);
    pChannel->setProperty("path", "sample.log");
    pChannel->setProperty("rotation", "2 K");
    pChannel->setProperty("archive", "timestamp");

    Logger::root().setChannel(pChannel);

    Logger& logger = Logger::get("TestLogger"); // inherits root channel

    for (int i = 0; i < 100; ++i)
        logger.information("Testing FileChannel");

    return 0;
}
```

EventLogChannel

- Poco::EventLogChannel, available on Windows NT platforms only, logs to the Windows Event Log.
- #include "Poco/EventLogChannel.h"
- Poco::EventLogChannel registers PocoFoundation.dll as message definition resource DLL with the Windows Event Log.
- When viewing the Windows Event Log, the Event Viewer application must be able to find PocoFoundation.dll, otherwise log messages will not be displayed as expected.

EventLogChannel Properties

Property	Description
name	The name of the event source. Usually the application name.
loghost, host	The name of the host where the Event Log service is running. Defaults to localhost.
logfile	The name of the log file. Defaults to "Application".

SyslogChannel

- Poco::SyslogChannel, available on Unix platforms only, logs to the local Syslog daemon.
- #include "Poco/SyslogChannel.h"
- The Net library contains a RemoteSyslogChannel class that works with remote Syslog daemons, using the UDP-based Syslog protocol.
- See the reference documentation for supported properties.

AsyncChannel

- Poco::AsyncChannel allows to run a channel in a separate thread. This decouples the thread producing the log messages from the thread delivering the log messages.
- All log messages are stored in a FIFO queue.
- A separate thread extracts messages from the queue and sends them to another channel.

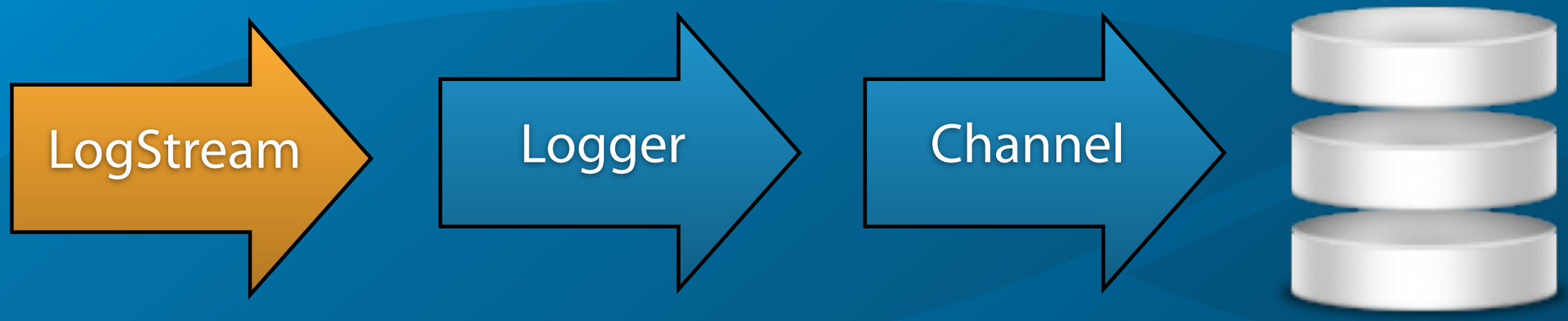
```
#include "Poco/Logger.h"
#include "Poco/AsyncChannel.h"
#include "Poco/ConsoleChannel.h"
#include "Poco/AutoPtr.h"

using Poco::Logger;
using Poco::AsyncChannel;
using Poco::ConsoleChannel;
using Poco::AutoPtr;

int main(int argc, char** argv)
{
    AutoPtr<ConsoleChannel> pCons(new ConsoleChannel);
    AutoPtr<AsyncChannel> pAsync(new AsyncChannel(pCons));

    Logger::root().setChannel(pAsync);

    Logger& logger = Logger::get("TestLogger");

    for (int i = 0; i < 10; ++i)
        logger.information("This is a test");

    return 0;
}
```

SplitterChannel

- Poco::SplitterChannel forwards a message to one or more other channels.
- #include "Poco/SplitterChannel.h"
- void addChannel(Channel* pChannel)
  adds a new channel to the Poco::SplitterChannel

```
#include "Poco/Logger.h"
#include "Poco/SplitterChannel.h"
#include "Poco/ConsoleChannel.h"
#include "Poco/SimpleFileChannel.h"
#include "Poco/AutoPtr.h"

using Poco::Logger;
using Poco::SplitterChannel;
using Poco::ConsoleChannel;
using Poco::SimpleFileChannel;
using Poco::AutoPtr;

int main(int argc, char** argv)
{
	AutoPtr<ConsoleChannel> pCons(new ConsoleChannel);
	AutoPtr<SimpleFileChannel> pFile(new SimpleFileChannel("test.log"));
	AutoPtr<SplitterChannel> pSplitter(new SplitterChannel);
	pSplitter->addChannel(pCons);
	pSplitter->addChannel(pFile);

	Logger::root().setChannel(pSplitter);
	Logger::root().information("This is a test");
	return 0;
}
```

Logging Streams



LogStream

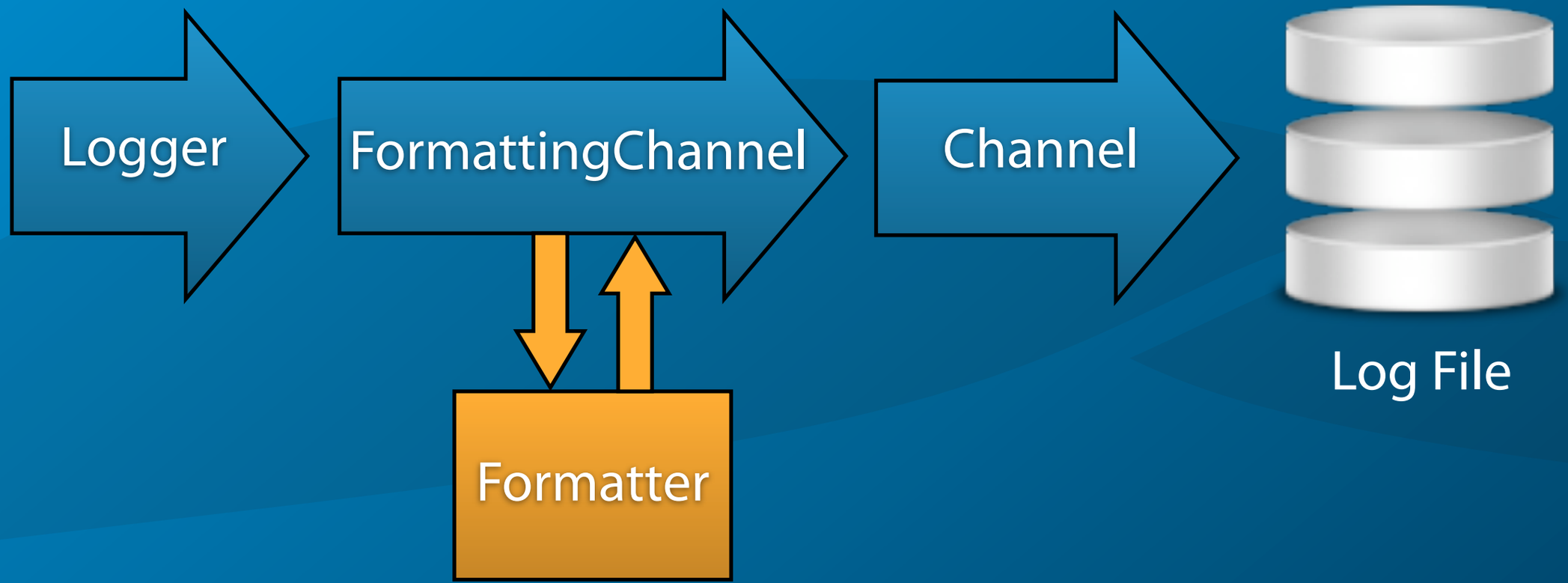
- Poco::LogStream provides an ostream interface to a Logger.
- #include "Poco/LogStream.h"
- All features of a stream can be used to format logging messages.
- A log message must be terminated with std::endl (or a CR or LF character).

LogStream

- The priority of the messages can be set with:
  LogStream& priority(Message::Priority prio)
  LogStream& fatal()
  LogStream& critical()
  LogStream& error()
  LogStream& warning()
  LogStream& notice()
  LogStream& information()
  LogStream& debug()
  LogStream& trace

```
#include "Poco/LogStream.h"
#include "Poco/Logger.h"

using Poco::Logger;
using Poco::LogStream;

int main(int argc, char** argv)
{
    Logger& logger = Logger::get("TestLogger");
    LogStream lstr(logger);

    lstr << "This is a test" << std::endl;

    return 0;
}
```

Message Formatting



FormattingChannel and Formatter

- Poco::FormattingChannel and Poco::Formatter are responsible for formatting log messages.
- Poco::FormattingChannel passes each message it receives through a Poco::Formatter, before propagating the message to the next channel.

- #include "Poco/FormattingChannel.h"
- #include "Poco/Formatter.h"
- Poco::Formatter is the base class for all formatter classes.
- Like channels, formatters can be configured using properties.

PatternFormatter

- Poco::PatternFormatter formats messages according to a printf-style pattern.
- #include "Poco/PatternFormatter.h"
- For details, please see the reference documentation.

```
#include "Poco/ConsoleChannel.h"
#include "Poco/FormattingChannel.h"
#include "Poco/PatternFormatter.h"
#include "Poco/Logger.h"
#include "Poco/AutoPtr.h"

using Poco::ConsoleChannel;
using Poco::FormattingChannel;
using Poco::PatternFormatter;
using Poco::Logger;
using Poco::AutoPtr;

int main(int argc, char** argv)
{
    AutoPtr<ConsoleChannel> pCons(new ConsoleChannel);
    AutoPtr<PatternFormatter> pPF(new PatternFormatter);
    pPF->setProperty("pattern", "%Y-%m-%d %H:%M:%S %s: %t");
    AutoPtr<FormattingChannel> pFC(new FormattingChannel(pPF, pCons));

    Logger::root().setChannel(pFC);
    Logger::get("TestChannel").information("This is a test");

    return 0;
}
```

Performance Considerations

- Creating a message takes a bit of time (the current time, current process ID and current thread ID must be determined).
- Creating a meaningful message may need more time, because string concatenations and number formatting, etc. are necessary.
- Messages are usually passed down the channel chain by reference.
- Exceptions: FormattingChannel and AsyncChannel create a copy of the message.

Performance Considerations (cont'd)

- Logging is always enabled or disabled (or, more correctly speaking, the log level is set) for each logger separately, so determining which log level is set for a specific logger is a constant time operation (a simple inlined integer comparison), once you have a reference to a logger.
- Obtaining a reference to a logger is an operation with logarithmic complexity (basically a std::map lookup). The logger's name is used as the key in the lookup, so the length of the logger's name affects the lookup time linearly (string comparison). This, however, can probably be neglected.

Performance Considerations (cont'd)

- Usually, a reference to a logger (Logger::get()) is only obtained once in an application. For example, in a class you'll obtain the reference to its logger in the class' constructor, and from then on use only the reference.
- You should avoid calling Logger::get() frequently. It's much better to call it once for every logger you're going to use, and then store the reference to the logger for later use.
- Logging performance depends on the channel(s) you're going to use. Actual channel performance is highly system dependent.

Performance Considerations (cont'd)

- Constructing log messages is often a time consuming operation consisting of string creations, string concatenations, number formatting, etc.
- In such a case it's a good idea to check whether the message will actually be logged before constructing it, using is(), fatal(), critical(), etc.
- There are also macros that do the check before constructing the message:
  poco_fatal(msg), poco_critical(msg), poco_error(msg), etc.

```
// ...

if (logger.warning())
{
     std::string msg("This is a warning");
     logger.warning(msg);
}

// is equivalent to

poco_warning(logger, "This is a warning");
```

applied informatics

www.appinf.com | info@appinf.com
T +43 4253 32596 | F +43 4253 32096