Engine::Logger Documentation

Table of Contents

Overview

Engine::Logger is a high-performance, thread-safe logging system for C++23 applications. It features configurable worker threads, lock-free queues, file rotation, structured logging, and comprehensive error handling.

Key Features

- Seven log levels (Trace to Critical)
- Thread-safe singleton with configurable worker threads
- Type-safe configuration with compile-time validation
- Structured logging with JSON output support
- Pre-compiled color DSL for terminal output
- Comprehensive error handling with Result<T,E>
- Automatic file rotation with disk space monitoring
- Lock-free queue implementation for multi-worker mode
- Performance benchmarking support

Performance Characteristics

- Single worker mode: ~1M messages/sec on modern hardware
- Multi-worker mode: ~3M messages/sec with 4 workers
- Lock-free queue overhead: ~50ns per operation
- File write throughput: Limited by disk I/O

Log Levels

The logger supports seven severity levels, from most to least verbose:

LogLevel::Trace

The most verbose logging level for detailed debugging information.

```cpp
Engine::Logger::trace("Entering function processData");
// Output: TRACE: Entering function processData
```

LogLevel::Debug

Debugging information useful during development.

```cpp
Engine::Logger::debug("Variable x = 42");
// Output: DEBUG: Variable x = 42
```

LogLevel::Info

General informational messages about program flow.

```cpp
Engine::Logger::info("Server started on port 8080");
// Output: INFO: Server started on port 8080
```

LogLevel::Success

Positive confirmation messages for successful operations.

cpp

```cpp
Engine::Logger::success("Database connection established");
// Output: SUCCESS: Database connection established
```

LogLevel::Warning

Warning messages for potentially problematic situations.

cpp

```cpp
Engine::Logger::warning("Cache size exceeding 80% capacity");
// Output: WARNING: Cache size exceeding 80% capacity
```

LogLevel::Error

Error messages for recoverable failures.

cpp

```cpp
Engine::Logger::error("Failed to load configuration file");
// Output: ERROR: Failed to load configuration file
```

LogLevel::Critical

Critical errors that may cause system failure.

cpp

```cpp
Engine::Logger::critical("Out of memory - shutting down");
// Output: CRITICAL: Out of memory - shutting down
```

Basic Logging Functions

Each log level has three overloaded methods for different use cases:

trace()

Logs a trace-level message.

cpp

```cpp
Engine::Logger::trace(message)
Engine::Logger::trace(message, context)
Engine::Logger::trace(message, context, handler)
```

- `message`: The log message string
- `context`: Optional key-value pairs for additional data
- `handler`: Optional specific handler name

Example:

```cpp
// Simple message
Engine::Logger::trace("Processing item");

// With context
Engine::Logger::trace("Processing item", {{"item_id", 123}, {"size", 1024}});

// With specific handler
Engine::Logger::trace("Processing item", {{"item_id", 123}}, "file_logger");
```

debug()

Logs a debug-level message.

```cpp
Engine::Logger::debug(message)
Engine::Logger::debug(message, context)
Engine::Logger::debug(message, context, handler)
```

Example:

```cpp
Engine::Logger::debug("Cache hit rate: 85%", {{"hits", 850}, {"total", 1000}});
```

info()

Logs an info-level message.

```cpp
Engine::Logger::info(message)
Engine::Logger::info(message, context)
Engine::Logger::info(message, context, handler)
```

Example:

cpp

```cpp
Engine::Logger::info("User logged in", {
    {"username", "john_doe"},
    {"ip_address", "192.168.1.100"},
    {"session_id", "abc123"}
});
```

success()

Logs a success-level message.

cpp

```cpp
Engine::Logger::success(message)
Engine::Logger::success(message, context)
Engine::Logger::success(message, context, handler)
```

Example:

cpp

```cpp
Engine::Logger::success("Payment processed", {
    {"transaction_id", "TXN-12345"},
    {"amount", 99.99},
    {"currency", "USD"}
});
```

warning()

Logs a warning-level message.

cpp

```cpp
Engine::Logger::warning(message)
Engine::Logger::warning(message, context)
Engine::Logger::warning(message, context, handler)
```

Example:

cpp

```cpp
Engine::Logger::warning("Disk space low", {
    {"available_gb", 10},
    {"threshold_gb", 20},
    {"partition", "/var/log"}
});
```

error()

Logs an error-level message.

cpp

```cpp
Engine::Logger::error(message)
Engine::Logger::error(message, context)
Engine::Logger::error(message, context, handler)
```

Example:

cpp

```cpp
Engine::Logger::error("API request failed", {
    {"status_code", 500},
    {"endpoint", "/api/users"},
    {"retry_count", 3}
});
```

critical()

Logs a critical-level message.

cpp

```cpp
Engine::Logger::critical(message)
Engine::Logger::critical(message, context)
Engine::Logger::critical(message, context, handler)
```

Example:

```cpp
Engine::Logger::critical("Database connection lost", {
    {"host", "db.example.com"},
    {"port", 5432},
    {"last_error", "Connection timeout"}
});
```

Configuration System

The logger uses a type-safe builder pattern for configuration:

ConfigTemplate::builder()

Creates a type-safe configuration builder.

```cpp
auto config = Engine::ConfigTemplate::builder()
    .name("my_handler")
    .level(Engine::LogLevel::Info)
    .format("simple", "%(levelname): %(message)")
    .output("simple", Engine::StreamTarget::cout())
    .build();
```

Builder Methods

name()

Sets the handler name (required).

```cpp
.name(handler_name)
```

- `handler_name`: Unique identifier for this handler

Example:

```cpp
.name("console_handler")
```

level()

Sets the minimum log level (required).

```cpp
.level(log_level)
```

- `log_level`: Minimum LogLevel to process

Example:

```cpp
.level(Engine::LogLevel::Warning)  // Only WARNING and above
```

format()

Defines a format pattern (required, can be called multiple times).

```cpp
.format(format_name, pattern)
```

- `format_name`: Name for this format
- `pattern`: Format string with tokens

Format tokens:

- `%(levelname)`: Log level name (TRACE, DEBUG, etc.)
- `%(message)`: Log message
- `%(date)`: Current date (YYYY-MM-DD)
- `%(time)`: Current time (HH:MM:SS)
- `%(thread)`: Thread ID
- `%(file)`: Source file name
- `%(line)`: Line number
- `%(function)`: Function name
- `%(context[key])`: Context value by key

Examples:

cpp

```cpp
// Simple format
.format("simple", "%(levelname): %(message)")

// Detailed format
.format("detailed", "%(date) %(time) [%(levelname)] %(file):%(line) - %(message)")

// With context
.format("context", "[%(context[request_id])] %(levelname): %(message)")
```

output()

Adds an output target (required, can be called multiple times).

cpp

```cpp
.output(format_name, target)
```

- `format_name`: Format to use for this output
- `target`: StreamTarget or file path

Examples:

cpp

```cpp
// Console output
.output("simple", Engine::StreamTarget::cout())
.output("errors", Engine::StreamTarget::cerr())

// File output
.output("detailed", std::filesystem::path("app.log"))
.output("errors", std::filesystem::path("errors.log"))
```

filter()

Adds a filter function (optional).

cpp

```cpp
.filter(filter_function)
```

- `filter_function`: Function that returns true to accept record

Examples:

cpp

```cpp
// Only warnings and above
.filter([](const Engine::LogRecord& record) {
    return record.level >= Engine::LogLevel::Warning;
})

// Only specific component
.filter([](const Engine::LogRecord& record) {
    auto it = record.context.find("component");
    return it != record.context.end() &&
           std::any_cast<std::string>(it->second) == "auth";
})
```

context()

Sets base context data (optional).

cpp

```cpp
.context(context_map)
```

- `context_map`: Default key-value pairs added to all records

Example:

cpp

```cpp
.context({
    {"app_version", "1.2.3"},
    {"environment", "production"},
    {"server_id", "web-01"}
})
```

structured()

Enables JSON output format (optional).

cpp

```cpp
.structured(enable)
```

- `enable`: true for JSON output, false for formatted text

Example:

```cpp
.structured(true)  // Outputs JSON instead of formatted text
```

Handler Management

add_handler()

Adds a configuration handler to the logger.

```cpp
Engine::Logger::instance().add_handler(config)
```

Returns `Result<void, ConfigError>`.

Example:

```cpp
auto result = Engine::Logger::instance().add_handler(
    Engine::ConfigTemplate::builder()
        .name("console")
        .level(Engine::LogLevel::Debug)
        .format("simple", "[%(levelname)] %(message)")
        .format("detailed", "%(date) %(time) [%(levelname)] %(message)")
        .output("simple", Engine::StreamTarget::cout())
        .output("detailed", std::filesystem::path("debug.log"))
        .build()
);

result.if_ok([]() {
    std::cout << "Handler added successfully\n";
});

result.if_err([](Engine::ConfigError err) {
    std::cerr << "Failed to add handler\n";
});
```

remove_handler()

Removes a handler by name.

```cpp
Engine::Logger::instance().remove_handler(handler_name)
```

- `handler_name`: Name of handler to remove

Example:

```cpp
auto result = Engine::Logger::instance().remove_handler("console");
result.if_ok([]() {
    std::cout << "Handler removed\n";
});
```

Queue Configuration

set_queue_capacity()

Sets the maximum queue size.

```cpp
Engine::Logger::instance().set_queue_capacity(capacity)
```

- `capacity`: Maximum number of queued records (default: 8192)

Example:

```cpp
// Increase queue size for high-volume logging
Engine::Logger::instance().set_queue_capacity(32768);
```

set_overflow_policy()

Sets behavior when queue is full.

```cpp
Engine::Logger::instance().set_overflow_policy(policy)
```

- `policy`: OverflowPolicy enum value

Policies:

- `OverflowPolicy::Block`: Wait for space (default)
- `OverflowPolicy::DropOldest`: Remove oldest record
- `OverflowPolicy::DropNewest`: Drop new record

Example:

```cpp
// Don't block on full queue, drop old messages
Engine::Logger::instance().set_overflow_policy(Engine::OverflowPolicy::DropOldest);
```

Statistics and Monitoring

get_stats()

Returns logger statistics.

```cpp
Engine::Logger::instance().get_stats()
```

Returns struct with:

- `queued_records`: Current queue size
- `dropped_records`: Total dropped records
- `processed_records`: Total processed records
- `handler_count`: Number of handlers
- `queue_saturated`: Queue near capacity (>90%)

Example:

```cpp
auto stats = Engine::Logger::instance().get_stats();
std::cout << "Logger Statistics:\n";
std::cout << "  Queued: " << stats.queued_records << "\n";
std::cout << "  Processed: " << stats.processed_records << "\n";
std::cout << "  Dropped: " << stats.dropped_records << "\n";
std::cout << "  Handlers: " << stats.handler_count << "\n";
std::cout << "  Saturated: " << (stats.queue_saturated ? "Yes" : "No") << "\n";
```

benchmark()

Runs performance benchmark.

```cpp
Engine::Logger::instance().benchmark(num_messages)
```

- `num_messages`: Number of test messages (default: 100000)

Returns struct with:

- `messages_per_second`: Throughput rate
- `avg_latency`: Average message latency
- `min_latency`: Minimum latency
- `max_latency`: Maximum latency

Example:

```cpp
std::cout << "Running benchmark...\n";
auto result = Engine::Logger::instance().benchmark(1000000);
std::cout << "Performance Results:\n";
std::cout << "  Throughput: " << result.messages_per_second << " msg/sec\n";
std::cout << "  Avg latency: " << result.avg_latency.count() << " ns\n";
std::cout << "  Min latency: " << result.min_latency.count() << " ns\n";
std::cout << "  Max latency: " << result.max_latency.count() << " ns\n";
```

Multi-Worker Logger

MultiWorkerLogger

High-performance logger with multiple worker threads and lock-free queues.

```cpp
Engine::MultiWorkerLogger<N>::instance()
```

- `N`: Number of worker threads (1-16)

Example:

cpp

```cpp
// Use 4 worker threads for high throughput
Engine::MultiWorkerLogger<4>::instance().add_handler(
    Engine::ConfigTemplate::builder()
        .name("high_perf")
        .level(Engine::LogLevel::Info)
        .format("simple", "%(message)")
        .output("simple", Engine::StreamTarget::cout())
        .build()
);

// Log using the multi-worker instance
Engine::MultiWorkerLogger<4>::info("High performance logging");
```

Performance comparison:

cpp

```cpp
// Benchmark single worker
auto single_result = Engine::Logger::instance().benchmark(100000);
std::cout << "Single worker: " << single_result.messages_per_second << " msg/sec\n";

// Benchmark 4 workers
auto multi_result = Engine::MultiWorkerLogger<4>::instance().benchmark(100000);
std::cout << "Four workers: " << multi_result.messages_per_second << " msg/sec\n";
```

Context Data

Using Context

Add key-value pairs to log records for structured data.

cpp

```cpp
Engine::Logger::info("User action", {
    {"user_id", 12345},
    {"action", "purchase"},
    {"amount", 99.99},
    {"items", 3}
});
```

Supported Types

- Strings: `std::string`, `const char*`
- Integers: `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`
- Floating point: `float`, `double`
- Boolean: `bool`
- Any other type (stored but displayed as type name)

Example with various types:

cpp

```cpp
Engine::Logger::info("Complex event", {
    {"string_val", "hello"},
    {"int_val", 42},
    {"double_val", 3.14159},
    {"bool_val", true},
    {"char_ptr", "C-string"},
    {"custom_type", std::vector<int>{1, 2, 3}}  // Displayed as [vector<int>]
});
```

Accessing Context in Formats

cpp

```cpp
.format("ctx", "[%(context[request_id])] %(levelname): %(message)")

// Usage
Engine::Logger::info("Request processed", {{"request_id", "REQ-12345"}});
// Output: [REQ-12345] INFO: Request processed
```

Color Output

Color Tags

Use tags in format strings for colored terminal output.

cpp

```cpp
.format("colored", "<red>ERROR:</red> %(message)")
```

Available Tags

Colors:

- `<red>...</red>`: Red text
- `<green>...</green>`: Green text
- `<yellow>...</yellow>`: Yellow text
- `<blue>...</blue>`: Blue text
- `<magenta>...</magenta>`: Magenta text
- `<cyan>...</cyan>`: Cyan text
- `<white>...</white>`: White text

Styles:

- `<bold>...</bold>`: Bold text
- `<dim>...</dim>`: Dim text
- `<italic>...</italic>`: Italic text
- `<underline>...</underline>`: Underlined text

Examples

cpp

```cpp
// Colored by level
auto config = Engine::ConfigTemplate::builder()
    .name("colored_console")
    .level(Engine::LogLevel::Debug)
    .format("debug", "<dim>%(levelname): %(message)</dim>")
    .format("info", "<green>%(levelname)</green>: %(message)")
    .format("warning", "<yellow>%(levelname): %(message)</yellow>")
    .format("error", "<bold><red>%(levelname): %(message)</red></bold>")
    .output("debug", Engine::StreamTarget::cout())
    .filter([](const Engine::LogRecord& r) {
        // Use different formats based on level
        return true;
    })
    .build();

// Complex coloring
.format("fancy", "<bold>%(date) %(time)</bold> [<cyan>%(levelname)</cyan>] <italic>%(message)</
```

File Output

Basic File Logging

```cpp
.output("format_name", std::filesystem::path("path/to/file.log"))
```

File Rotation

Automatic file rotation based on size.

File Cache Configuration:

- `max_file_size`: 100MB (default)
- `idle_timeout`: 30 seconds
- `auto_flush`: true
- `enable_rotation`: true
- `monitor_disk_space`: true

Example:

```cpp
// Files automatically rotate at 100MB
auto config = Engine::ConfigTemplate::builder()
    .name("rotating_file")
    .level(Engine::LogLevel::Info)
    .format("log", "%(date) %(time) [%(levelname)] %(message)")
    .output("log", std::filesystem::path("logs/app.log"))
    .build();
```

Rotated file naming:

- Original: `app.log`
- Rotated: `app_20240115_143022.log`

Multiple File Outputs

cpp

```cpp
auto config = Engine::ConfigTemplate::builder()
    .name("multi_file")
    .level(Engine::LogLevel::Debug)
    .format("full", "%(date) %(time) [%(levelname)] %(file):%(line) - %(message)")
    .format("errors", "%(date) %(time) [%(levelname)] %(message)")
    .output("full", std::filesystem::path("logs/full.log"))
    .output("errors", std::filesystem::path("logs/errors.log"))
    .filter([](const Engine::LogRecord& r) {
        // Errors go to both files, others only to full.log
        return true;
    })
    .build();
```

Macros

Convenience Macros

Simplified logging macros for common use cases.

cpp

```cpp
LOG_TRACE(message, ...)
LOG_DEBUG(message, ...)
LOG_INFO(message, ...)
LOG_SUCCESS(message, ...)
LOG_WARNING(message, ...)
LOG_ERROR(message, ...)
LOG_CRITICAL(message, ...)
```

Examples:

cpp

```cpp
// Simple message
LOG_INFO("Server started");

// With context
LOG_ERROR("Connection failed", {{"host", "192.168.1.1"}, {"port", 8080}});

// With context and handler
LOG_WARNING("Low memory", {{"available_mb", 512}}, "system_monitor");
```

Setup Macros

LOGGER_SETUP_DEV()

Quick development configuration.

```cpp
LOGGER_SETUP_DEV();
// Creates console handler with:
// - Name: "console"
// - Level: Debug
// - Format: "%(levelname): %(message)"
// - Output: stdout
```

LOGGER_SETUP_PROD()

Production configuration.

```cpp
LOGGER_SETUP_PROD();
// Creates file handler with:
// - Name: "file"
// - Level: Info
// - Format: "%(date) %(time) [%(levelname)] %(message)"
// - Output: "app.log"
// - Structured: JSON format
```

LOGGER_SETUP_MULTI_WORKER(N)

Multi-worker configuration.

```cpp
LOGGER_SETUP_MULTI_WORKER(4);
// Creates high-performance handler with:
// - 4 worker threads
// - Lock-free queues
// - Console output
```

Error Handling

Result Type

All fallible operations return `Result<T, E>` for safe error handling.

cpp

```cpp
template<typename T, typename E>
class Result {
    bool is_ok() const;
    bool is_err() const;
    const T& value() const;
    E error() const;
    void if_ok(function<void(const T&)>);
    void if_err(function<void(E)>);
};
```

Error Types

ConfigError

Configuration-related errors:

cpp

```cpp
enum class ConfigError {
    None,               // No error
    InvalidName,        // Handler name invalid
    InvalidFormat,      // Format pattern invalid
    MissingFormat,      // No format specified
    NoOutputs,          // No outputs specified
    TooManyHandlers,    // Exceeded 32 handler limit
    Unknown            // Unknown error
};
```

Example handling:

cpp

```cpp
auto result = Engine::Logger::instance().add_handler(config);
result.if_err([](Engine::ConfigError err) {
    switch (err) {
        case Engine::ConfigError::TooManyHandlers:
            std::cerr << "Too many handlers registered\n";
            break;
        case Engine::ConfigError::InvalidFormat:
            std::cerr << "Invalid format pattern\n";
            break;
        default:
            std::cerr << "Configuration error\n";
    }
});
```

FileError

File operation errors:

cpp

```cpp
enum class FileError {
    None,                       // No error
    DirectoryCreationFailed,    // Cannot create directory
    FileOpenFailed,             // Cannot open file
    WriteFailed,                // Write operation failed
    FlushFailed,                // Flush operation failed
    RotationFailed,             // File rotation failed
    PermissionDenied,           // Insufficient permissions
    DiskFull,                   // Insufficient disk space
    Unknown                     // Unknown error
};
```

Example handling:

cpp

```cpp
// File operations return Result<void, FileError>
auto& cache = Engine::get_file_cache();
auto result = cache.write(path, "log message");
result.if_err([](Engine::FileError err) {
    std::cerr << "File error: " << Engine::file_error_to_string(err) << '\n';
});
```

Complete Examples

Basic Console Logger

cpp

```cpp
#include "logger.h"

int main() {
    // Setup console logger
    auto config = Engine::ConfigTemplate::builder()
        .name("console")
        .level(Engine::LogLevel::Info)
        .format("simple", "[%(levelname)] %(message)")
        .output("simple", Engine::StreamTarget::cout())
        .build();

    auto result = Engine::Logger::instance().add_handler(std::move(config));
    if (result.is_err()) {
        std::cerr << "Failed to setup logger\n";
        return 1;
    }

    // Use logger
    LOG_INFO("Application started");
    LOG_DEBUG("This won't show (below Info level)");
    LOG_WARNING("This is a warning");
    LOG_ERROR("This is an error");

    // With context
    LOG_INFO("User logged in", {{"user_id", 123}, {"username", "john"}});

    return 0;
}
```

Production Logger Setup

cpp

```cpp
#include "logger.h"

void setup_production_logging() {
    // Console output for important messages
    auto console_config = Engine::ConfigTemplate::builder()
        .name("console")
        .level(Engine::LogLevel::Warning)
        .format("colored", "<yellow>%(levelname)</yellow>: %(message)")
        .output("colored", Engine::StreamTarget::cout())
        .build();

    // File output for everything
    auto file_config = Engine::ConfigTemplate::builder()
        .name("file")
        .level(Engine::LogLevel::Debug)
        .format("detailed", "%(date) %(time) [%(levelname)] %(file):%(line) - %(message)")
        .output("detailed", std::filesystem::path("logs/app.log"))
        .context({
            {"app_version", "2.1.0"},
            {"environment", "production"}
        })
        .build();

    // JSON output for analysis
    auto json_config = Engine::ConfigTemplate::builder()
        .name("json")
        .level(Engine::LogLevel::Info)
        .format("json", "%(message)")
        .output("json", std::filesystem::path("logs/app.json"))
        .structured(true)
        .build();

    // Add all handlers
    Engine::Logger::instance().add_handler(std::move(console_config));
    Engine::Logger::instance().add_handler(std::move(file_config));
    Engine::Logger::instance().add_handler(std::move(json_config));
}
```

Filtered Component Logger

cpp

```cpp
// Logger that only logs messages from specific components
auto auth_logger = Engine::ConfigTemplate::builder()
    .name("auth")
    .level(Engine::LogLevel::Trace)
    .format("auth", "[AUTH] %(time) %(levelname): %(message)")
    .output("auth", std::filesystem::path("logs/auth.log"))
    .filter([](const Engine::LogRecord& record) {
        auto it = record.context.find("component");
        return it != record.context.end() &&
               std::any_cast<std::string>(it->second) == "auth";
    })
    .build();

Engine::Logger::instance().add_handler(std::move(auth_logger));

// Usage
LOG_INFO("Login attempt", {{"component", "auth"}, {"user", "admin"}});
LOG_INFO("Database query", {{"component", "db"}});  // Won't be logged by auth logger
```

High-Performance Multi-Worker Logger

cpp

```cpp
#include "logger.h"

int main() {
    // Use 8 worker threads for maximum performance
    using HighPerfLogger = Engine::MultiWorkerLogger<8>;

    // Configure for high-volume logging
    HighPerfLogger::instance().set_queue_capacity(65536);
    HighPerfLogger::instance().set_overflow_policy(Engine::OverflowPolicy::DropOldest);

    // Add handler
    auto config = Engine::ConfigTemplate::builder()
        .name("perf")
        .level(Engine::LogLevel::Info)
        .format("fast", "%(message)")
        .output("fast", std::filesystem::path("perf.log"))
        .build();

    HighPerfLogger::instance().add_handler(std::move(config));

    // Benchmark
    std::cout << "Running performance test...\n";
    auto result = HighPerfLogger::instance().benchmark(1000000);
    std::cout << "Throughput: " << result.messages_per_second << " msg/sec\n";

    // High-speed logging
    for (int i = 0; i < 10000000; ++i) {
        HighPerfLogger::info("High speed message", {{"index", i}});
    }

    // Check statistics
    auto stats = HighPerfLogger::instance().get_stats();
    std::cout << "Dropped messages: " << stats.dropped_records << "\n";

    return 0;
}
```

Custom Format with Colors

cpp

```cpp
// Create a beautiful colored format
auto pretty_config = Engine::ConfigTemplate::builder()
    .name("pretty")
    .level(Engine::LogLevel::Trace)
    .format("trace", "<dim>%(time) [TRACE] %(message)</dim>")
    .format("debug", "<cyan>%(time) [DEBUG] %(message)</cyan>")
    .format("info", "<green>%(time) [INFO] %(message)</green>")
    .format("success", "<bold><green>%(time) [SUCCESS] %(message)</green></bold>")
    .format("warning", "<yellow>%(time) [WARN] %(message)</yellow>")
    .format("error", "<red>%(time) [ERROR] %(message) (%(file):%(line))</red>")
    .format("critical", "<bold><red>%(time) [CRITICAL] %(message) (%(file):%(line))</red></bold
    .output("trace", Engine::StreamTarget::cout())
    .output("debug", Engine::StreamTarget::cout())
    .output("info", Engine::StreamTarget::cout())
    .output("success", Engine::StreamTarget::cout())
    .output("warning", Engine::StreamTarget::cout())
    .output("error", Engine::StreamTarget::cerr())
    .output("critical", Engine::StreamTarget::cerr())
    .build();
```

Error Recovery Logger

cpp

```cpp
// Logger with comprehensive error handling
void safe_log(const std::string& message) {
    static bool fallback_mode = false;

    if (!fallback_mode) {
        try {
            Engine::Logger::info(message);
        } catch (const std::exception& e) {
            fallback_mode = true;
            std::cerr << "Logger failed, switching to fallback: " << e.what() << "\n";
            std::cerr << "Original message: " << message << "\n";
        }
    } else {
        // Direct console output as fallback
        std::cerr << "[FALLBACK] " << message << "\n";
    }
}
```

Rotating File Logger with Monitoring

cpp

```cpp
// Setup rotating file logger with monitoring
void setup_monitored_file_logger() {
    auto config = Engine::ConfigTemplate::builder()
        .name("monitored")
        .level(Engine::LogLevel::Info)
        .format("log", "%(date) %(time) [%(levelname)] %(message)")
        .output("log", std::filesystem::path("logs/app.log"))
        .build();

    Engine::Logger::instance().add_handler(std::move(config));

    // Monitor file cache statistics
    std::thread monitor([]{
        while (true) {
            auto stats = Engine::get_file_cache().get_stats();
            if (stats.available_disk_space < 1024 * 1024 * 1024) { // Less than 1GB
                Engine::Logger::critical("Low disk space", {
                    {"available_bytes", stats.available_disk_space}
                });
            }
            std::this_thread::sleep_for(std::chrono::minutes(5));
        }
    });
    monitor.detach();
}
```

Best Practices

1. **Choose appropriate log levels**: Use TRACE/DEBUG for development, INFO for normal flow, WARNING for issues, ERROR for failures, CRITICAL for system problems.
2. **Use context data**: Add structured data instead of formatting it into messages.

   ```cpp
   // Good
   LOG_INFO("User login", {{"user_id", 123}, {"ip", "192.168.1.1"}});

   // Avoid
   LOG_INFO("User 123 logged in from 192.168.1.1");
   ```

3. **Configure for your environment**: Use different configurations for development and production.
4. **Monitor performance**: Use `get_stats()` to check for dropped messages and queue saturation.
5. **Handle errors gracefully**: Always check Result types from configuration operations.
6. **Use multi-worker logger for high throughput**: When logging millions of messages per second.
7. **Set up log rotation**: Prevent disk space issues with automatic rotation.
8. **Use filters wisely**: Filter at the handler level to reduce processing overhead.
9. **Benchmark your configuration**: Test performance with your specific use case.
10. **Clean shutdown**: The logger automatically flushes on destruction, but explicit shutdown can be useful.