Logging

Efficiently capture telemetry from your app for debugging and performance analysis.

Framework
os

On This Page
Overview
Topics

Overview

When debugging problems in your app, it's helpful to record the exact sequence of events that occurred while your app was running, along with supplemental data about those events. With this data, you can reconstruct the sequence and determine where problems occurred. This style debugging is often necessary when you are unable to attach a debugger to the app, such as when you're diagnosing problems that occurred in the past on a user's machine.

In addition, you often want to get a general sense of how your app executes, when certain events occur, and the start and end times of specific tasks. Such data, when imported into tools like Instruments, can help you visualize exactly what your app does over a period of time. For example, you might want to see how your app performs when it executes multiple overlapping tasks.

The unified logging system provides a comprehensive and performant API to capture telemetry across all levels of the system. This unified system centralizes the storage of log data in memory and in a data store on disk. You get fine-grained control at runtime over what kind of data your app logs and where that data is stored. You can automatically correlate related log entries to specific parts of your app or specific actions taken by the user.

After you log data, you retrieve it using system tools or by accessing it within your own app.

Important

Unified logging is available in iOS 10.0 and later, macOS 10.12 and later, tvOS 10.0 and later, and watchOS 3.0 and later, and supersedes ASL (Apple System Logger) and the Syslog APIs. Historically, log messages were written to specific locations on disk, such as `/etc/system.log`. The unified logging system stores messages in memory and in a data store, rather than writing to text-based log files.

Understand the Unified Logging System

Unified logging is built around some key concepts:

- You define *subsystems* to organize large topic areas within your app or apps. For example, you might define a subsystem for each process that you create. Within a subsystem, you define *categories* to further distinguish parts of that subsystem. For example, if you used a single subsystem for your app, you might create separate categories for model code and user-interface code. In a game, you might use categories to distinguish between physics, AI, world simulation, and rendering.

- To record data, you log *messages* to a particular subsystem and category. Each message also has a log level, which describes what kind of information is recorded in the message. For example, an error message states that a significant error occurred, so the logging system always logs error messages to the persistent data store. In contrast, you use debug messages to record verbose data about the inner workings of your app; those verbose messages are recorded only when you specifically request it.
- You can correlate all messages associated with a single *activity*. An activity is a request made to or by some subsystem in your app. Activities are usually triggered by a user action. For example, when the user selects a menu item, your app might perform multiple actions to resolve that user's request. Your app might log messages across different subsystems or categories. Other processes or the system may also log events based on requests your code makes. By creating an activity, you correlate all of these messages together, so you can trace everything that happened as a result of that user request. You can also create activites for other app behaviors, such as a database updates.
- To visualize performance data in Instruments, use the same subsystems to record *signposts*. Signposts represent critical one-time events or the start or end times of a time interval. For example, when you process data, you might capture the start time and end time for those calculations. Inside Instruments, you can see this information visually and track what happens over time. You can also create your own custom instruments to automatically analyze this data.

To use the unified logging system, perform these actions:

- Create unique subsystems for interesting parts of your applications that you want to be able to analyze without seeing unrelated log messages. If you want more granularity when filtering messages within a subsystem, define separate message categories within that subsystem.
- Log messages for interesting events, choosing an appropriate log level for each message. At a minimum, log messages related to significant or catastrophic errors in your app.
- Wrap code triggered by user actions and other major events into activities.
- Create signposts for critical events and record timestamps for major tasks in your app.
- Determine which tools are best suited to help you record and analyze the log data that your app generates.

Topics

Essentials

Choosing the Log Level for a Message
Determine when and how a message is logged.

Logging Messages
Record useful debugging and analysis information.

Formatting Log Messages
Use a format string to provide additional data in a log message.

Collecting Log Messages in Activities

Find messages related to a specific user action or application event.

Recording Performance Data

Add signposts to record interesting time-based events.

Determining Which Data to Log

Record data that is valuable for debugging and performance tuning.

Viewing Log Messages

Use various tools to retrieve log information.

Customizing Logging Behavior While Debugging

Control which log events are recorded.

Logs

os_log_t

A log object that you pass to logging functions to send messages to that log.

`os_log_type_t`

Logging levels supported by the system.

Generating Log Messages

`os_log_with_type`

Sends a message at a specific logging level, such as default, info, debug, error, or fault, to the logging system.

`os_log`

Sends a default-level message to the logging system.

`os_log_info`

Sends an info-level message to the logging system.

`os_log_debug`

Sends a debug-level message to the logging system.

`os_log_error`

Sends an error-level message to the logging system.

`os_log_fault`

Sends a fault-level message to the logging system.

Activity Tracing

os_activity_t

An object that represents an activity triggered by the user.

Logging Signpost Events

`os_signpost_emit_with_type`

Logs a point of interest in your code as a time interval or as an event for debugging performance in Instruments.

`os_signpost_interval_begin`

Marks the start of a time interval in your code using a signpost.

`os_signpost_interval_end`

Marks the end of a time interval in your code using a signpost.

`os_signpost_event_emit`

Marks an event in your code using a signpost.

Signpost Types

Values that determine the kind of signpost the system logs.

os_signpost_id_t

An identifier you use to distinguish between signposts that have the same name and destination log.