

Cardano.BM - logging, benchmarking and monitoring

Alexander Diemand

Denis Shevchenko

Andreas Triantafyllos

April 2019

Abstract

This framework combines logging, benchmarking and monitoring. Complex evaluations of STM or monadic actions can be observed from outside while reading operating system counters before and after, and calculating their differences, thus relating resource usage to such actions.

Through interactive configuration, the runtime behaviour of logging or the measurement of resource usage can be altered.

Further reduction in logging can be achieved by redirecting log messages to an aggregation function which will output the running statistics with less frequency than the original message.

Contents

1	Logging, benchmarking and monitoring	3
1.1	Main concepts	3
1.1.1	LogObject	3
1.1.2	Trace	3
1.1.3	Backend	4
1.1.4	Configuration	4
1.2	Overview	5
1.2.1	Backends	5
1.2.2	Trace	5
1.2.3	Monitoring	5
1.2.4	IMPORTANT!	5
1.3	Requirements	6
1.3.1	Observables	6
1.3.2	Traces	7
1.3.3	Aggregation	8
1.3.4	Monitoring	8
1.3.5	Reporting	8
1.3.6	Visualisation	8
1.4	Description	9
1.4.1	Contravariant Functors Explanation	9
1.4.2	Logging with Trace	9
1.4.3	Micro-benchmarks record observables	9
1.4.4	Configuration	11
1.4.5	Information reduction in Aggregation	12
1.4.6	Output selection	12
1.4.7	Monitoring	12
1.5	Examples	12
1.5.1	Simple example showing plain logging	12
1.5.2	Complex example showing logging, aggregation, and observing <i>IO</i> actions	15
1.5.3	Performance example for time measurements	23
1.6	Code listings - contra-tracer package	25
1.6.1	Examples	25
1.6.2	Contravariant Tracer	25
1.6.3	Transformers	26
1.6.4	Output	26
1.7	Code listings - iohk-monitoring package	27
1.7.1	Cardano.BM.Observer.STM	27
1.7.2	Cardano.BM.Observer.Monad	28
1.7.3	Cardano.BM.Trace	31
1.7.4	Cardano.BM.Setup	34
1.7.5	Cardano.BM.Counters	35

1.7.6	Cardano.BM.Counters.Common	35
1.7.7	Cardano.BM.Counters.Dummy	36
1.7.8	Cardano.BM.Counters.Linux	37
1.7.9	Cardano.BM.Data.Aggregated	44
1.7.10	Cardano.BM.Data.AggregatedKind	50
1.7.11	Cardano.BM.Data.Backend	50
1.7.12	Cardano.BM.Data.BackendKind	51
1.7.13	Cardano.BM.Data.Configuration	52
1.7.14	Cardano.BM.Data.Counter	53
1.7.15	Cardano.BM.Data.LogItem	54
1.7.16	Cardano.BM.Data.Observable	59
1.7.17	Cardano.BM.Data.Output	59
1.7.18	Cardano.BM.Data.Rotation	61
1.7.19	Cardano.BM.Data.Severity	61
1.7.20	Cardano.BM.Data.SubTrace	62
1.7.21	Cardano.BM.Data.Trace	63
1.7.22	Cardano.BM.Data.Tracer	63
1.7.23	Cardano.BM.Configuration	65
1.7.24	Cardano.BM.Configuration.Model	65
1.7.25	Cardano.BM.Configuration.Static	76
1.7.26	Cardano.BM.Backend.Switchboard	77
1.7.27	Cardano.BM.Backend.Log	84
1.7.28	Cardano.BM.Backend.LogBuffer	94
1.7.29	Cardano.BM.Backend.EKGView	95
1.7.30	Cardano.BM.Backend.Editor	100
1.7.31	Cardano.BM.Backend.Graylog	107
1.7.32	Cardano.BM.Backend.Aggregation	110
1.7.33	Cardano.BM.Backend.Monitoring	116
1.7.34	Cardano.BM.Backend.Prometheus	122
1.7.35	Cardano.BM.Backend.ExternalAbstraction	122
1.7.36	Cardano.BM.Backend.TraceAcceptor	123
1.7.37	Cardano.BM.Backend.TraceForwarder	124
2	Testing	126
2.1	Test main entry point	126
2.2	Test case generation	127
2.2.1	instance Arbitrary Aggregated	127
2.3	Tests	128
2.3.1	Testing aggregation	128
2.3.2	Cardano.BM.Test.STM	132
2.3.3	Cardano.BM.Test.Trace	132
2.3.4	Testing configuration	144
2.3.5	Rotator	153
2.3.6	Cardano.BM.Test.Structured	155
2.3.7	Cardano.BM.Test.Tracer	155

Chapter 1

Logging, benchmarking and monitoring

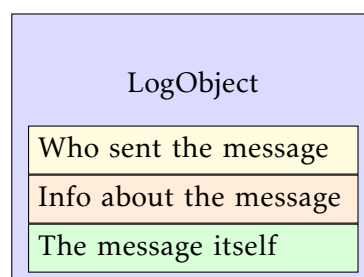
1.1 Main concepts

The main concepts of the framework:

1. `LogObject` - captures the observable information
2. `Trace` - transforms and delivers the observables
3. `Backend` - receives and outputs observables
4. `Configuration` - defines behaviour of traces, routing of observables

1.1.1 `LogObject`

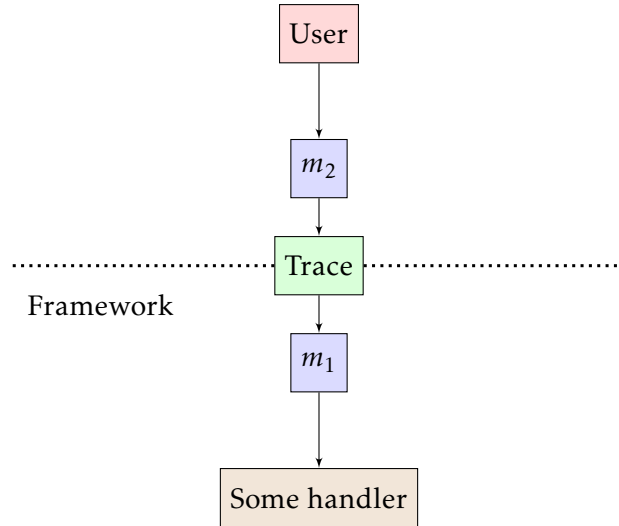
`LogObject` represents an observation to be logged or otherwise further processed. It is annotated with a logger name, meta information (timestamp and severity level), and some particular message:



Please see `Cardano.BM.Data.LogItem` for more details.

1.1.2 `Trace`

You can think of `Trace` as a pipeline for messages. It is a *consumer* of messages from a user's point of view, but a *source* of messages from the framework's point of view. A user traces an observable to a `Trace`, which ends in the framework that further processes the message.



Please see the section 1.4.1 for more details about the ideas behind **Trace**.

1.1.3 Backend

A **Backend** must implement functions to process incoming messages of type **LogObject**. It is an instance of **IsEffectuator**. Moreover, a backend is also life-cycle managed. The class **IsBackend** ensures that every backend implements the *realize* and *unrealize* functions.

The central backend in the framework is the **Switchboard**. It sets up all the other backends and redirects incoming messages to these backends according to configuration:



1.1.4 Configuration

Configuration defines how the message flow in the framework is routed and the behaviour of distinct **Traces**. It can be parsed from a file in YAML format, or it can explicitly be defined in code.

Please note that **Configuration** can be changed at runtime using the interactive editor (see *Cardano.BM.Configuration.Editor* for more details).

1.2 Overview

Figure 1.1 displays the relationships among modules in *Cardano.BM*.

1.2.1 Backends

As was mentioned above, the central backend is the **Switchboard** that redirects incoming log messages to selected backends according to **Configuration**.

The backend **EKGView** displays runtime counters and user-defined values in a browser.

The **Log** backend makes use of the **katip** package to output log items to files or the console. The format can be chosen to be textual or JSON representation.

The **Aggregation** backend computes simple statistics over incoming log items (e.g. last, min, max, mean) (see **Cardano.BM.Data.Aggregated**). Alternatively, **Aggregation** can also estimate the average of the values passed in using *EWMA*, the exponentially weighted moving average. This works for numerical values, that is if the content of a **LogObject** is a **LogValue**.

The backend **LogBuffer** keeps the latest message per context name and shows these collected messages in the GUI (**Editor**), or outputs them to the switchboard.

Output selection determines which log items of a named context are routed to which backend. In the case of the **Log** output, this includes a configured output sink, *scribe* in *katip* parlance.

Items that are aggregated lead to the creation of an output of their current statistics. To prevent a potential infinite loop these aggregated statistics cannot be routed again back into **Aggregation**.

1.2.2 Trace

Log items are created in the application's context and passed in via a hierarchy of **Traces**. Such a hierarchy of named traces can be built with the function **appendName**. The newly added child **Trace** will add its name to the logging context and behave as configured. Among the different kinds of **Traces** implemented are:

1. **NoTrace** which suppresses all log items,
2. **SetSeverity** which sets a specific severity to all log items,
3. **FilterTrace** which filters the log items passing through it,
4. **ObservableTrace** which allows capturing of operating system counters.

(further behaviour types are implemented in **Cardano.BM.Data.SubTrace**)

1.2.3 Monitoring

With *Monitoring* we aim to shortcut the logging-analysis cycle and immediately evaluate monitors on logged values when they become available. In case a monitor is triggered a number of actions can be run: either internal actions that can alter the **Configuration**, or actions that can lead to alerting in external systems.

1.2.4 IMPORTANT!

It is not the intention that this framework should (as part of normal use) record sufficient information so as to make the sequence of events reproducible, i.e. it is not an audit or transaction log.

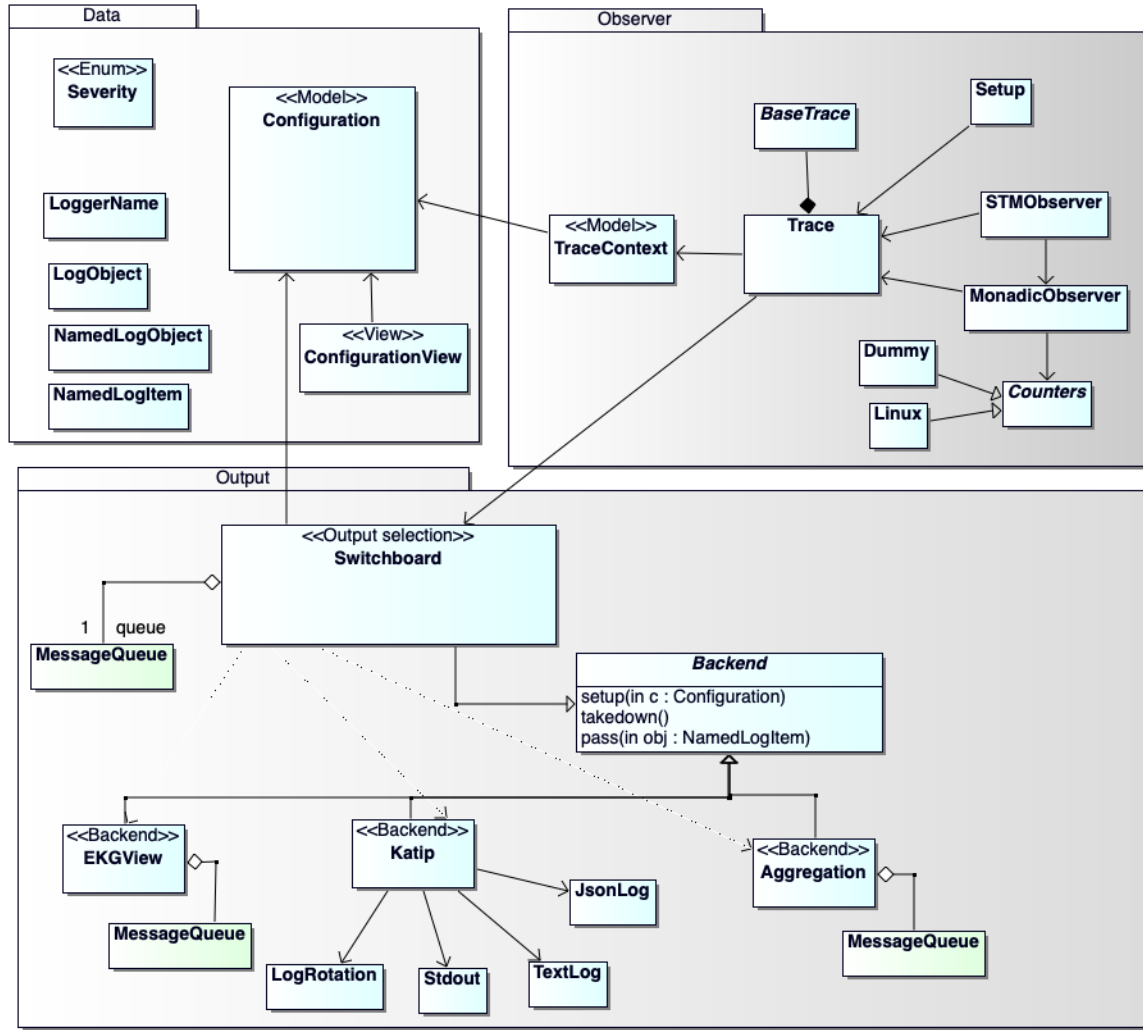


Figure 1.1: Overview of module relationships. The arrows indicate import of a module. The arrows with a triangle at one end would signify “inheritance” in object-oriented programming, but we use it to show that one module replaces the other in the namespace, thus specializes its interface.

1.3 Requirements

1.3.1 Observables

We can observe the passage of the flow of execution through particular points in the code (really the points at which the graph is reduced). Typically observables would be part of an outcome (which has a start and an end). Where the environment permits these outcomes could also gather additional environmental context (e.g read system counters, know the time). The proposed framework would be able to aggregate, filter such outcome measures so as to calculate things (where appropriate) such as:

- min/max/mean/variance of the resource costs of achieving an outcome
- elapsed wall-clock time

- CPU cycles
- memory allocations, etc
- exponentially weighted moving average of outcomes, events
- min/max/mean/variance of inter-arrival times of demand for service (the arrival pattern)
- measuring offered load against the system (e.g rate/distribution of requests against the wallet by an exchange, transactions being forwarded between nodes)

STM evaluation

We treat STM evaluation as a black box and register measurables (counters) before entering, and report the difference at exit together with the result. Logging in an STM will keep a list of log items which at the exit of the evaluation will be passed to the logging subsystem. Since we do not know the exact time an event occurred in the STM action, we annotate the event afterwards with the time interval of the STM action.

Function evaluation

We treat a function call as a black box and register measurables (counters) before entering, and report the difference at exit together with the result. The function is expected to accept a 'Trace' argument which receives the events.

QuickCheck properties *tentatively*

The function

```
quickCheckResult :: Testable prop => prop -> IO Result
```

will return a *Result* data structure from which we can extract the number of tests performed. Recording the start and end times allows us to derive the time spent for a single test. (although this measurement is wrong as it includes the time spent in QuickCheck setting up the test case (and shrinking?))

1.3.2 Traces

Log items are sent as streams of events to the logging system for processing (aggregation, ..) before output. Functions that need to log events must accept a *Trace* argument. There is no monad related to logging in the monad stack, thus this can work in any monadic environment.

Trace Context

A Trace maintains a named context stack. A new name can be put onto it, and all subsequent log messages are labeled with this named context. This is also true to all downstream functions which receive the modified Trace. We thus can see the call tree and how the evaluation entered the context where a logging function was called. The context also maintains a mapping from name to Severity: this way a logging function call can early end and not produce a log item when the minimum severity is not reached.

SubTrace

A Trace is created in *IO* within `setupTrace` with the intent to pass the traced items to a downstream logging framework for outputting to various destinations in different formats. Apart from adding a name to the naming stack we can also alter the behaviour of the Trace. The newly created Trace with a specific function to process the recorded items will forward these to the upstream Trace. This way we can, for example, locally turn on aggregation of observables and only report a summary to the logs.

1.3.3 Aggregation

Log items contain a named context, severity and a payload (message, structured value). Thinking of a relation

`(name, severity) -> value`

, folding a summarizing function over it outputs

`(name, severity) -> Summary`

. Depending on the type of *value*, the summary could provide for example:

- `*` : first, last, count, the time between events (mean, sigma)
- `Num` : min, max, median, quartiles, mean, sigma, the delta between events (mean, sigma)

Other possible aggregations:

- exponentially weighted moving average
- histograms

1.3.4 Monitoring

- Enable (or disable) measuring events and performance at runtime (e.g. measure how block holding time has changed).
- Send alarms when observables give evidence for abnormalities
- Observe actions in progress, i.e. have started and not yet finished
- Bridge to *Datadog*?

1.3.5 Reporting

We might want to buffer events in case an exception is detected. This FIFO queue could then be output to the log for post-factum inspection.

1.3.6 Visualisation

EKG

<https://hackage.haskell.org/package/ekg>

This library allows live monitor a running instance over HTTP. There is a way we can add our own metrics to it and update them.

Log files

The output of observables immediately or aggregated to log files. The format is chosen to be JSON for easier post-processing.

Web app

Could combine EKG, log files and parameterization into one GUI.
(e.g. <https://github.com/HeinrichApfelmus/threepenny-gui>)

1.4 Description

1.4.1 Contravariant Functors Explanation

Tracer's implementations is based on a **contravariant** package.

Please see the presentation in docs/pres-20190409/contravariant-idea to understand the core idea of the contravariant functor.

1.4.2 Logging with **Trace**

Setup procedure

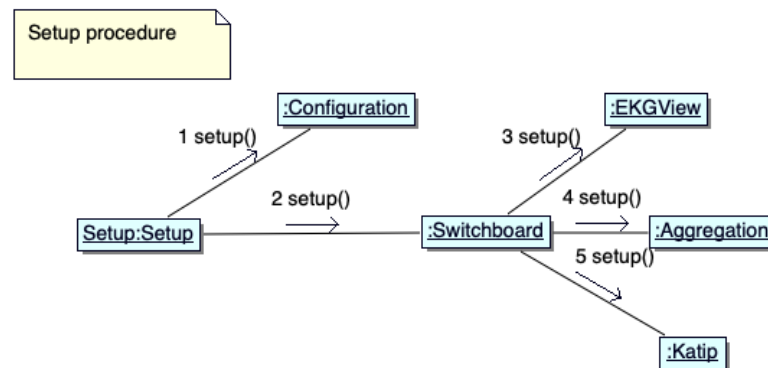


Figure 1.2: Setup procedure

Hierarchy of **Traces**

1.4.3 Micro-benchmarks record observables

Micro-benchmarks are recording observables that measure resource usage of the whole program for a specific time. These measurements are then associated with the subsystem that was observed at that time. Caveat: if the executable under observation runs on a multiprocessor computer where more than one parallel thread executes at the same time, it becomes difficult to associate resource usage to a single function. Even more so, as Haskell's thread do not map directly to operating system threads. So the expressiveness of our approach is only valid statistically when a large number of observables have been captured.

Counters

The framework provides access to the following O/S counters (defined in **ObservableInstance**) on *Linux*:

- monotonic clock (see **MonotonicClock**)
- CPU or total time (*/proc/<pid>/stat*) (see **ProcessStats**)
- memory allocation (*/proc/<pid>/statm*) (see **MemoryStats**)
- network bytes received/sent (*/proc/<pid>/net/netstat*) (see **NetStats**)
- disk input/output (*/proc/<pid>/io*) (see **IOStats**)

On all platforms, access is provided to the RTS counters (see **GhcRtsStats**).

Implementing micro-benchmarks

In a micro-benchmark we capture operating system counters over an STM evaluation or a function, before and afterwards. Then, we compute the difference between the two and report all three measurements via a *Trace* to the logging system. Here we refer to the example that can be found in **complex example**.

```
STM.bracketObserveIO trace "observeSTM" (stmAction args)
```

The capturing of STM actions is defined in **Cardano.BM.Observer.STM** and the function *STM.bracketObserveIO* has type:

```
bracketObserveIO
  :: Configuration
  → Trace IO a
  → Severity
  → Text
  → STM.STM t
  → IO t
```

It accepts a *Trace* to which it logs, adds a name to the context name and enters this with a *SubTrace*, and finally the STM action which will be evaluated. Because this evaluation can be retried, we cannot pass to it a *Trace* to which it could log directly. A variant of this function **bracketObserveLogIO** also captures log items in its result, which then are threaded through the *Trace*.

Capturing observables for a function evaluation in *IO*, the type of *bracketObserveIO* (defined in **Cardano.BM.Observer.Monad**) is:

```
bracketObserveIO
  :: Configuration
  → Trace IO a
  → Severity
  → Text
  → IO t
  → IO t
```

It accepts a *Trace* to which it logs items, adds a name to the context name and enters this with a *SubTrace*, and then the IO action which will be evaluated.

```

bracketObserveIO trace "observeDownload" $ do
  license ← openURI "http://www.gnu.org/licenses/gpl.txt"
  case license of
    Right bs → logInfo trace $ pack $ BS8.unpack bs
    Left e → logError trace $ "failed to download; error: " ++ (show e)
  threadDelay 50000 -- .05 second
  pure ()

```

Counters are evaluated before the evaluation and afterwards. We trace these as log items **ObserveOpen** and **ObserveClose**, as well as the difference with type **ObserveDiff**.

Configuration of mu-benchmarks

Observed STM actions or functions enter a new named context with a **SubTrace**. Thus, they need a configuration of the behaviour of this **SubTrace** in the new context. We can define this in the configuration for our example:

```
CM.setSubTrace c "complex.observeDownload" (Just $ ObservableTrace [NetStats,IOStats])
```

This enables the capturing of network and I/O stats from the operating system. Other Observables are implemented in **Cardano.BM.Data.Observable**. Captured observables need to be routed to backends. In our example we configure:

```
CM.setBackends c "complex.observeIO" (Just [AggregationBK])
```

to direct observables from named context *complex.observeIO* to the Aggregation backend.

1.4.4 Configuration

Format

The configuration is parsed from a file in *Yaml* format (see <https://en.wikipedia.org/wiki/YAML>) on startup. In a first parsing step the file is loaded into an internal *Representation*. This structure is then further processed and validated before copied into the runtime **Configuration**.

Configuration editor

The configuration editor (figure 1.3) provides a minimalistic GUI accessible through a browser that directly modifies the runtime configuration of the logging system. Most importantly, the global minimum severity filter can be set. This will suppress all log messages that have a severity assigned that is lower than this setting. Moreover, the following behaviours of the logging system can be changed through the GUI:

- *Backends*: relates the named logging context to a **BackendKind**
- *Scribes*: if the backend is **KatipBK**, defines to which outputs the messages are directed (see **ScribeId**)
- *Severities* a local minimum severity filter for just the named context (see **Severity**)
- *SubTrace* entering a new named context can create a new **Trace** with a specific behaviour (see **SubTrace**)
- *Aggregation* if the backend is *AggregationBK*, defines which aggregation method to use (see **AggregatedKind**)



Figure 1.3: The configuration editor is listening on *localhost* and can be accessed through a browser. At the top is the setting for the global minimum severity filter, that drops all messages that have a severity lower than this setting. Below are the settings for various behaviours of the logging system.

1.4.5 Information reduction in **Aggregation**

Statistics

Configuration

1.4.6 Output selection

Configuration

1.4.7 Monitoring

Configuration

Evaluation of monitors

Actions fired

1.5 Examples

1.5.1 Simple example showing plain logging

```
{-# LANGUAGE CPP #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE ScopedTypeVariables #-}
# if defined (linux_HOST_OS)
# define LINUX
# endif
module Main
```

```

    (main)
    where
import Control.Concurrent (threadDelay)
import Control.Concurrent.MVar (MVar, newMVar, modifyMVar_, withMVar)
import Data.Aeson (FromJSON)

import Cardano.BM.Backend.Switchboard (addExternalBackend)
import Cardano.BM.Data.Backend
import qualified Cardano.BM.Configuration.Model as CM
import Cardano.BM.Configuration.Static (defaultConfigStdout)
#ifdef LINUX
import Cardano.BM.Data.Output (ScribeDefinition (..),
    ScribePrivacy (..), ScribeKind (..), ScribeFormat (..))
#endif
import Cardano.BM.Setup (setupTrace_)
import Cardano.BM.Trace (Trace, appendName, logDebug, logError,
    logInfo, logNotice, logWarning)

```

a simple backend

```

type MyBackendMVar a = MVar (MyBackendInternal a)
newtype MyBackend a = MyBackend { myBE :: MyBackendMVar a }
data MyBackendInternal a = MyBackendInternal {
    counter :: Int
}

instance (FromJSON a) => IsBackend MyBackend a where
    typeOf _ = UserDefinedBK "MyBackend"
    realize _ = MyBackend <$> newMVar (MyBackendInternal 0)
    unrealize be = putStrLn $ "unrealize " <> show (typeOf be)

instance IsEffectuator MyBackend a where
    effectuate be _item = do
        modifyMVar_ (myBE be) $ \mybe ->
            return $ mybe { counter = counter mybe + 1 }
    handleOverflow _ = putStrLn "Error: MyBackend's queue full!"

```

Entry procedure

```

main :: IO ()
main = do
    c ← defaultConfigStdout
    CM.setDefaultBackends c [KatipBK, UserDefinedBK "MyBackend"]
#ifdef LINUX
    CM.setSetupBackends c [KatipBK, GraylogBK]
    CM.setDefaultBackends c [KatipBK, GraylogBK, UserDefinedBK "MyBackend"]
    CM.setGraylogPort c 3456
    CM.setSetupScribes c [ScribeDefinition {
        scName = "text"
        ,scFormat = ScText
        ,scKind = StdoutSK
    }

```

```

        ,scPrivacy = ScPublic
        ,scRotation = Nothing
    }
    ,ScribeDefinition {
        scName = "json"
        ,scFormat = ScJson
        ,scKind = StdoutSK
        ,scPrivacy = ScPublic
        ,scRotation = Nothing
    }
    ,ScribeDefinition {
        scName = "systemd"
        ,scFormat = ScText
        ,scKind = JournalSK
        ,scPrivacy = ScPublic
        ,scRotation = Nothing
    }
}

CM.setScribes c "simple.systemd" (Just [ "JournalSK::systemd" ])
# endif
CM.setScribes c "simple.json" (Just [ "StdoutSK::json" ])
(tr :: Trace IO String, sb) ← setupTrace_c "simple"
be :: MyBackend String ← realize c
let mybe = MkBackend { bEffectuate = effectuate be, bUnrealize = unrealize be }
addExternalBackend sb mybe "MyBackend"
let trText = appendName "text" tr
    trJson = appendName "json" tr
# ifdef LINUX
    trSystemd = appendName "systemd" tr
# endif
logDebug trText "this is a debug message\nwith a second line"
logDebug trJson "this is a debug message\nwith a second line"
logInfo trText "this is an information."
logInfo trJson "this is an information."
logNotice trText "this is a notice!"
logNotice trJson "this is a notice!"
logWarning trText "this is a warning!"
logWarning trJson "this is a warning!"
logError trText "this is an error!"
logError trJson "this is an error!"
# ifdef LINUX
logError trSystemd "this is an error!"
# endif
threadDelay 80000
withMVar (myBE be) $ \backend →
    putStrLn $ "read in total " ++ (show $ counter backend) ++ " messages."
return ()

```


1.5.2 Complex example showing logging, aggregation, and observing IO actions

Module header and import directives

```

{-# LANGUAGE CPP #-}
{-# LANGUAGE ScopedTypeVariables #-}
# if defined (linux_HOST_OS)
# define LINUX
# endif

{-define the parallel procedures that create messages -}
# define RUN_ProcMessageOutput
# define RUN_ProcObserveIO
# define RUN_ProcObserveSTM
# define RUN_ProcObserveDownload
# define RUN_ProcRandom
# define RUN_ProcMonitoring
# undef RUN_ProcBufferDump

module Main
  (main)
  where

import Control.Concurrent (threadDelay)
import qualified Control.Concurrent.Async as Async
import Control.Monad (forM_)
# ifdef ENABLE_OBSERVABLES
import Control.Monad (forM)
import GHC.Conc.Sync (atomically, STM, TVar, newTVar, readTVar, writeTVar)
# ifdef LINUX
import qualified Data.ByteString.Char8 as BS8
import Network.Download (openURI)
# endif
# endif
import qualified Data.HashMap.Strict as HM
import Data.Text (Text, pack)
import System.Random

import Cardano.BM.Configuration (Configuration)
import qualified Cardano.BM.Configuration.Model as CM
import Cardano.BM.Data.Aggregated (Measurable (...))
import Cardano.BM.Data.AggregatedKind
import Cardano.BM.Data.BackendKind
import Cardano.BM.Data.LogItem
import Cardano.BM.Data.MonitoringEval
import Cardano.BM.Data.Output
import Cardano.BM.Data.Rotation
import Cardano.BM.Data.Severity
import Cardano.BM.Data.SubTrace
# ifdef ENABLE_OBSERVABLES
import Cardano.BM.Data.Observable
import Cardano.BM.Observer.Monad (bracketObserveIO)
import qualified Cardano.BM.Observer.STM as STM
# endif
import Cardano.BM.Setup

```

```
import Cardano.BM.Trace
```

Define configuration

Selected values can be viewed in EKG on <http://localhost:12789>.

The configuration editor listens on <http://localhost:13789>.

```
prepare_configuration :: IO CM.Configuration
prepare_configuration = do
  c ← CM.empty
  CM.setMinSeverity c Warning
  CM.setSetupBackends c [KatipBK
# ifdef ENABLE_AGGREGATION
    ,AggregationBK
# endif
# ifdef ENABLE_EKG
    ,EKGViewBK
# endif
# ifdef ENABLE_GUI
    ,EditorBK
# endif
    ,MonitoringBK
    ,TraceForwarderBK
  ]
  CM.setDefaultBackends c [KatipBK, TraceForwarderBK]
  CM.setSetupScribes c [ScribeDefinition {
    scName = "stdout"
    ,scKind = StdoutSK
    ,scFormat = ScText
    ,scPrivacy = ScPublic
    ,scRotation = Nothing
  }
    ,ScribeDefinition {
    scName = "logs/out.odd.json"
    ,scKind = FileSK
    ,scFormat = ScJson
    ,scPrivacy = ScPublic
    ,scRotation = Nothing
  }
    ,ScribeDefinition {
    scName = "logs/out.even.json"
    ,scKind = FileSK
    ,scFormat = ScJson
    ,scPrivacy = ScPublic
    ,scRotation = Nothing
  }
    ,ScribeDefinition {
    scName = "logs/downloading.json"
    ,scKind = FileSK
    ,scFormat = ScJson
    ,scPrivacy = ScPublic
```

```

    ,scRotation = Nothing
  }
  ,ScribeDefinition {
    scName = "logs/out.txt"
    ,scKind = FileSK
    ,scFormat = ScText
    ,scPrivacy = ScPublic
    ,scRotation = Just $ RotationParameters
      {rpLogLimitBytes = 5000-- 5kB
      ,rpMaxAgeHours = 24
      ,rpKeepFilesNum = 3
      }
    }
  ]
CM.setDefaultScribes c [ "StdoutSK::stdout" ]
CM.setScribes c "complex.random" (Just [ "StdoutSK::stdout", "FileSK::logs/out.txt" ])
forM_ [(1 :: Int)..10] $ \x →
  if odd x
  then
    CM.setScribes c ("#aggregation.complex.observeSTM." <> (pack $ show x)) $ Just [ "FileSK::logs/out.txt" ]
  else
    CM.setScribes c ("#aggregation.complex.observeSTM." <> (pack $ show x)) $ Just [ "FileSK::logs/out.txt" ]
# ifdef LINUX
# ifdef ENABLE_OBSERVABLES
  CM.setSubTrace c "complex.observeDownload" (Just $ ObservableTraceSelf [IOStats,NetStats])
# endif
  CM.setBackends c "complex.observeDownload" (Just [KatipBK])
  CM.setScribes c "complex.observeDownload" (Just [ "StdoutSK::stdout", "FileSK::logs/download" ])
# endif
  CM.setSubTrace c "#messagecounters.switchboard" $ Just NoTrace
  CM.setSubTrace c "#messagecounters.katip" $ Just NoTrace
  CM.setSubTrace c "complex.random" (Just $ TeeTrace "ewma")
  CM.setSubTrace c "#ekgview"
    (Just $ FilterTrace [(Drop (StartsWith "#ekgview.#aggregation.complex.random"),
      Unhide [(EndsWith ".count"),
        (EndsWith ".avg"),
        (EndsWith ".mean")]),
      (Drop (StartsWith "#ekgview.#aggregation.complex.observeIO"),
        Unhide [(Contains "diff.RTS.cpuNs.timed.")]),
      (Drop (StartsWith "#ekgview.#aggregation.complex.observeSTM"),
        Unhide [(Contains "diff.RTS.gcNum.timed.")]),
      (Drop (StartsWith "#ekgview.#aggregation.complex.message"),
        Unhide [(Contains ".timed.m")])
    ])
  CM.setSubTrace c "#messagecounters.ekgview" $ Just NoTrace
# ifdef ENABLE_OBSERVABLES
  CM.setSubTrace c "complex.observeIO" (Just $ ObservableTraceSelf [GhcRtsStats,MemoryStats])
forM_ [(1 :: Int)..10] $ \x →
  CM.setSubTrace
    c
    ("complex.observeSTM." <> (pack $ show x))

```

```

    (Just $ ObservableTraceSelf [GhcRtsStats,MemoryStats])
# endif
# ifdef ENABLE_AGGREGATION
    CM.setBackends c "complex.message" (Just [AggregationBK,KatipBK,TraceForwarderBK])
    CM.setBackends c "complex.random" (Just [KatipBK,EKGViewBK])
    CM.setBackends c "complex.random.ewma" (Just [KatipBK])
    CM.setBackends c "complex.observeIO" (Just [AggregationBK,MonitoringBK])
    CM.setSubTrace c "#messagecounters.aggregation" $ Just NoTrace
# endif
    forM_ [(1::Int)..10] $ \x → do
# ifdef ENABLE_AGGREGATION
    CM.setBackends c
        ("complex.observeSTM." <> (pack $ show x))
        (Just [AggregationBK])
# endif
    CM.setBackends c
        ("#aggregation.complex.observeSTM." <> (pack $ show x))
        (Just [KatipBK])

    CM.setAggregatedKind c "complex.random.rr" (Just StatsAK)
    CM.setAggregatedKind c "complex.random.ewma.rr" (Just (EwmaAK 0.42))
# ifdef ENABLE_GUI
    CM.setBackends c "#aggregation.complex.random" (Just [EditorBK])
    CM.setBackends c "#aggregation.complex.random.ewma" (Just [EditorBK])
    CM.setBackends c "#messagecounters.switchboard" (Just [EditorBK,KatipBK])
# endif
# ifdef ENABLE_EKG
    CM.setSubTrace c "#messagecounters.monitoring" $ (Just Neutral)
    CM.setBackends c "#aggregation.complex.message" (Just [EKGViewBK,MonitoringBK])
    CM.setBackends c "#aggregation.complex.monitoring" (Just [MonitoringBK])
    CM.setBackends c "#aggregation.complex.observeIO" (Just [EKGViewBK])
    CM.setEKGport c 12790
    CM.setLogOutput c "iohk-monitoring/log-pipe"
# ifdef ENABLE_PROMETHEUS
    CM.setPrometheusPort c 12800
# endif
# endif
# ifdef ENABLE_GUI
    CM.setGUIport c 13790
# endif
    CM.setMonitors c $ HM.fromList
        [( "complex.monitoring"
          , (Just (Compare "monitMe" (GE,(OpMeasurable 10)))
            , Compare "monitMe" (GE,(OpMeasurable 42))
            , [CreateMessage Warning "MonitMe is greater than 42!"]
            )
          )
        , ( "#aggregation.complex.monitoring"
          , (Just (Compare "monitMe.fcount" (GE,(OpMeasurable 8)))
            , Compare "monitMe.mean" (GE,(OpMeasurable 25))
            , [CreateMessage Warning "MonitMe.mean is greater than 25!"]
            )
          )
        ]

```

```

    )
  )
  ,("complex.observeIO.close"
    , (Nothing
      , Compare "complex.observeIO.close.Mem.size" (GE, (OpMeasurable 25))
      , [CreateMessage Warning "closing mem size is greater than 25!"]
    )
  )
]
CM.setBackends c "complex.monitoring" (Just [AggregationBK, KatipBK, MonitoringBK])
return c

```

Dump the log buffer periodically

```

dumpBuffer :: Switchboard Text → Trace IO Text → IO (Async.Async ())
dumpBuffer sb trace = do
  logInfo trace "starting buffer dump"
  proc ← Async.async (loop trace)
  return proc
where
  loop tr = do
    threadDelay 25000000 -- 25 seconds
    buf ← readLogBuffer sb
    forM_ buf $ \ (logname, LogObject _ lometa locontent) → do
      let tr' = modifyName (\n → "#buffer. " <> n <> logname) tr
      traceNamedObject tr' (lometa, locontent)
    loop tr

```

Thread that outputs a random number to a Trace

```

randomThr :: Trace IO Text → IO (Async.Async ())
randomThr trace = do
  logInfo trace "starting random generator"
  let trace' = appendName "random" trace
  proc ← Async.async (loop trace')
  return proc
where
  loop tr = do
    threadDelay 500000 -- 0.5 second
    num ← randomRIO (42 - 42, 42 + 42) :: IO Double
    lo ← (,) <$> (mkLOMeta Debug Public) <*> pure (LogValue "rr" (PureD num))
    traceNamedObject tr lo
  loop tr

```

Thread that outputs a random number to monitoring Trace

```

# ifdef RUN_ProcMonitoring
monitoringThr :: Trace IO Text → IO (Async.Async ())

```

```

monitoringThr trace = do
  logInfo trace "starting numbers for monitoring..."
  let trace' = appendName "monitoring" trace
  proc ← Async.async (loop trace')
  return proc
where
  loop tr = do
    threadDelay 500000 -- 0.5 second
    num ← randomRIO (42 - 42, 42 + 42) :: IO Double
    lo ← (,) < $ > (mkLOMeta Warning Public) < * > pure (LogValue "monitMe" (PureD num))
    traceNamedObject tr lo
  loop tr
# endif

```

Thread that observes an IO action

```

# ifdef ENABLE_OBSERVABLES
observeIO :: Configuration → Trace IO Text → IO (Async.Async ())
observeIO config trace = do
  logInfo trace "starting observer"
  proc ← Async.async (loop trace)
  return proc
where
  loop tr = do
    threadDelay 5000000 -- 5 seconds
    let tr' = appendName "observeIO" tr
    _ ← bracketObserveIO config tr' Warning "complex.observeIO" $ do
      num ← randomRIO (100000, 200000) :: IO Int
      ls ← return $ reverse $ init $ reverse $ 42 : [1..num]
      pure $ const ls ()
    loop tr
# endif

```

Threads that observe STM actions on the same TVar

```

# ifdef ENABLE_OBSERVABLES
observeSTM :: Configuration → Trace IO Text → IO [Async.Async ()]
observeSTM config trace = do
  logInfo trace "starting STM observer"
  tvar ← atomically $ newTVar ([1..1000] :: [Int])
  -- spawn 10 threads
  proc ← forM ([1 :: Int]..10) $ \x → Async.async (loop trace tvar (pack $ show x))
  return proc
where
  loop tr tvarlist name = do
    threadDelay 10000000 -- 10 seconds
    STM.bracketObserveIO config tr Warning ("observeSTM." <> name) (stmAction tvarlist)
    loop tr tvarlist name
stmAction :: TVar [Int] → STM ()

```

```

stmAction tvarlist = do
  list ← readTVar tvarlist
  writeTVar tvarlist $ reverse $ init $ reverse $ list
  pure ()
# endif

```

Thread that observes an IO action which downloads a text in order to observe the I/O statistics

```

# ifdef LINUX
# ifdef ENABLE_OBSERVABLES
observeDownload :: Configuration → Trace IO Text → IO (Async.Async ())
observeDownload config trace = do
  proc ← Async.async (loop trace)
  return proc
where
  loop tr = do
    threadDelay 1000000 -- 1 second
    let tr' = appendName "observeDownload" tr
    bracketObserveIO config tr' Warning "complex.observeDownload" $ do
      license ← openURI "http://www.gnu.org/licenses/gpl.txt"
      case license of
        Right bs → logNotice tr' $ pack $ BS8.unpack bs
        Left _ → return ()
      threadDelay 50000 -- .05 second
      pure ()
    loop tr
# endif
# endif

```

Thread that periodically outputs a message

```

msgThr :: Trace IO Text → IO (Async.Async ())
msgThr trace = do
  logInfo trace "start messaging .."
  let trace' = appendName "message" trace
  Async.async (loop trace')
where
  loop tr = do
    threadDelay 3000000 -- 3 seconds
    logNotice tr "N O T I F I C A T I O N ! ! !"
    logDebug tr "a detailed debug message."
    logError tr "Boooooomm .."
    loop tr

```

Main entry point

```

main :: IO ()
main = do

```

```

-- create configuration
c ← prepare_configuration

-- create initial top-level Trace
(tr :: Trace IO Text, sb) ← setupTrace_c "complex"
logNotice tr "starting program; hit CTRL-C to terminate"
-- user can watch the progress only if EKG is enabled.
#ifdef ENABLE_EKG
logInfo tr "watch its progress on http://localhost:12789"
#endif
#ifdef RUN_ProcBufferDump
procDump ← dumpBuffer sb tr
#endif
#ifdef RUN_ProcRandom
{-start thread sending unbounded sequence of random numbers to a trace which aggregates them in
procRandom ← randomThr tr
#endif
#ifdef RUN_ProcMonitoring
procMonitoring ← monitoringThr tr
#endif
#ifdef RUN_ProcObserveIO
-- start thread endlessly reversing lists of random length
#ifdef ENABLE_OBSERVABLES
procObsvIO ← observeIO c tr
#endif
#endif
#ifdef RUN_ProcObserveSTM
-- start threads endlessly observing STM actions operating on the same TVar
#ifdef ENABLE_OBSERVABLES
procObsvSTMs ← observeSTM c tr
#endif
#endif
#ifdef LINUX
#ifdef RUN_ProcObserveDownload
-- start thread endlessly which downloads sth in order to check the I/O usage
#ifdef ENABLE_OBSERVABLES
procObsvDownload ← observeDownload c tr
#endif
#endif
#endif
#ifdef RUN_ProcMessageOutput
-- start a thread to output a text messages every n seconds
procMsg ← msgThr tr
-- wait for message thread to finish, ignoring any exception
_ ← Async.waitCatch procMsg
#endif
#ifdef LINUX
#ifdef RUN_ProcObserveDownload
-- wait for download thread to finish, ignoring any exception
#ifdef ENABLE_OBSERVABLES
_ ← Async.waitCatch procObsvDownload

```



```

# endif
# endif
# endif
# ifdef RUN_ProcObseverSTM
    -- wait for observer thread to finish, ignoring any exception
# ifdef ENABLE_OBSERVABLES
    _ ← forM procObsvSTMs Async.waitCatch
# endif
# endif
# ifdef RUN_ProcObserveIO
    -- wait for observer thread to finish, ignoring any exception
# ifdef ENABLE_OBSERVABLES
    _ ← Async.waitCatch procObsvIO
# endif
# endif
# ifdef RUN_ProcRandom
    -- wait for random thread to finish, ignoring any exception
    _ ← Async.waitCatch procRandom
# endif
# ifdef RUN_ProcMonitoring
    _ ← Async.waitCatch procMonitoring
# endif
# ifdef RUN_ProcBufferDump
    _ ← Async.waitCatch procDump
# endif
    return ()

```

1.5.3 Performance example for time measurements

Module header and import directives

```

{-# LANGUAGE ScopedTypeVariables #-}
module Main
    (main)
    where

    import qualified Control.Concurrent.Async as Async
    import Control.Monad (forM_)
    import qualified Data.HashMap.Strict as HM
    import Data.Text (Text)
    import Criterion (Benchmark, bench, nfIO)
    import Criterion.Main (defaultMain)

    import Cardano.BM.Backend.Switchboard
    import qualified Cardano.BM.Configuration.Model as CM
    import Cardano.BM.Data.Aggregated (Measurable (..))
    import Cardano.BM.Data.BackendKind
    import Cardano.BM.Data.LogItem
    import Cardano.BM.Data.MonitoringEval
    import Cardano.BM.Data.Severity
    import Cardano.BM.Setup
    import Cardano.BM.Trace

```

Define configuration

```

prepare_configuration :: IO CM.Configuration
prepare_configuration = do
  c ← CM.empty
  CM.setMinSeverity c Warning
  CM.setSetupBackends c [MonitoringBK]
  CM.setDefaultBackends c [MonitoringBK]
  CM.setMonitors c $ HM.fromList
    [ ("performance.monitoring"
      , (Nothing
        , Compare "monitMe" (GE, (OpMeasurable 42))
        , [SetGlobalMinimalSeverity Debug]
        )
      )
    ]
  CM.setBackends c "performance.monitoring" (Just [MonitoringBK])
  return c

```

Thread that outputs a value to monitoring Trace

```

monitoringThr :: Trace IO Text → Int → IO (Async.Async ())
monitoringThr trace objNumber = do
  trace' ← appendName "monitoring" trace
  obj ← (,) < $ > (mkLOMeta Warning Public) < * > pure (LogValue "monitMe" (PureD 123.45))
  proc ← Async.async (loop trace' obj)
  return proc
where
  loop tr lo = do
    forM_ [1..objNumber] $ \_ → traceNamedObject tr lo
    -- terminate Switchboard
    killPill ← (,) < $ > (mkLOMeta Warning Public) < * > pure KillPill
    traceNamedObject tr killPill

```

Main entry point

```

main :: IO ()
main = defaultMain
  [ benchMain 1000
  , benchMain 10000
  , benchMain 100000
  , benchMain 1000000
  ]
benchMain :: Int → Benchmark
benchMain objNumber = bench (show objNumber ++ " objects") $ nfIO $ do
  c ← prepare_configuration
  (tr :: Trace IO Text, sb) ← setupTrace_c "performance"
  procMonitoring ← monitoringThr tr objNumber

```

```

_ ← Async.wait procMonitoring
_ ← waitForTermination sb
return ()

```

1.6 Code listings - contra-tracer package

1.6.1 Examples

Tracing using the contravariant **Tracer** naturally reads:

```

let logTrace = traceWith $ showTracing $ stdoutTracer
in logTrace "hello world"

```

1.6.2 Contravariant **Tracer**

The notion of a **Tracer** is an action that can be used to observe information of interest during evaluation. **Tracers** can capture (and annotate) such observations with additional information from their execution context.

```

newtype Tracer m a = Tracer {runTracer :: a → m ()}

```

A **Tracer** is an instance of *Contravariant*, which permits new **Tracers** to be constructed that feed into the existing Tracer by use of *contramap*.

```

instance Contravariant (Tracer m) where
  contramap f (Tracer t) = Tracer (t ∘ f)

```

Although a **Tracer** is invoked in a monadic context (which may be *Identity*), the construction of a new **Tracer** is a pure function. This brings with it the constraint that the derived **Tracers** form a hierarchy which has its root at the top level tracer.

In principle a **Tracer** is an instance of *Semigroup* and *Monoid*, by sequential composition of the tracing actions.

```

instance Applicative m ⇒ Semigroup (Tracer m s) where
  Tracer a1 <> Tracer a2 = Tracer $ λs → a1 s * > a2 s
instance Applicative m ⇒ Monoid (Tracer m s) where
  mappend = (<>)
  mempty = nullTracer

```

nullTracer

The simplest tracer - one that suppresses all output.

```

nullTracer :: Applicative m ⇒ Tracer m a
nullTracer = Tracer $ \_ → pure ()

```

traceWith

```

traceWith :: Tracer m a → a → m ()
traceWith = runTracer

```

1.6.3 Transformers

Applying *show* on a **Tracer**'s messages

The Tracer transformer exploiting Show.

```
showTracing :: (Show a) => Tracer m String -> Tracer m a
showTracing = contramap show
```

Conditional tracing - statically defined

The Tracer transformer that allows for on/off control of tracing at trace creation time.

```
condTracing :: (Monad m) => (a -> Bool) -> Tracer m a -> Tracer m a
condTracing active tr = Tracer $ \s ->
  when (active s) (traceWith tr s)
```

Conditional tracing - dynamically evaluated

The tracer transformer that can exercise dynamic control over tracing, the dynamic decision being made using the context accessible in the monadic context.

```
condTracingM :: (Monad m) => m (a -> Bool) -> Tracer m a -> Tracer m a
condTracingM activeP tr = Tracer $ \s -> do
  active <- activeP
  when (active s) (traceWith tr s)
```

natTrace

Natural transformation from monad *m* to monad *n*.

```
natTracer :: (forall x . m x -> n x) -> Tracer m s -> Tracer n s
natTracer nat (Tracer tr) = Tracer (nat o tr)
```

1.6.4 Output

Directing a **Tracer**'s output to stdout

The Tracer that prints a string (as a line) to stdout (usual caveats about interleaving should be heeded).

```
stdoutTracer :: (MonadIO m) => Tracer m String
stdoutTracer = Tracer $ liftIO o putStrLn
```

Outputting a **Tracer** with *Debug.Trace*

A Tracer that uses *TraceM* (from **Debug.Trace**) as its output mechanism.

```
debugTracer :: (Applicative m) => Tracer m String
debugTracer = Tracer Debug.Trace.traceM
```

1.7 Code listings - iohk-monitoring package

1.7.1 Cardano.BM.Observer.STM

```
stmWithLog :: STM.STM (t, [(LOMeta, LOContent a)]) → STM.STM (t, [(LOMeta, LOContent a)])
stmWithLog action = action
```

Observe STM action in a named context

With given name, create a **SubTrace** according to **Configuration** and run the passed STM action on it.

```
bracketObserveIO :: Config.Configuration → Trace IO a → Severity → Text → STM.STM t → IO t
bracketObserveIO config trace severity name action = do
  subTrace ← fromMaybe Neutral < $ > Config.findSubTrace config name
  bracketObserveIO' subTrace severity trace action
where
  bracketObserveIO' :: SubTrace → Severity → Trace IO a → STM.STM t → IO t
  bracketObserveIO' NoTrace _ _ act =
    STM.atomically act
  bracketObserveIO' subtrace sev logTrace act = do
    mCountersid ← observeOpen subtrace sev logTrace
    -- run action; if an exception is caught, then it will be logged and rethrown.
    t ← (STM.atomically act) 'catch' (λ(e :: SomeException) → (TIO.hPutStrLn stderr (pack (show e)) >> throw e))
    case mCountersid of
      Left openException →
        -- since observeOpen faced an exception there is no reason to call observeClose
        -- however the result of the action is returned
        TIO.hPutStrLn stderr ("ObserveOpen: " <> pack (show openException))
      Right countersid → do
        res ← observeClose subtrace sev logTrace countersid [ ]
        case res of
          Left ex → TIO.hPutStrLn stderr ("ObserveClose: " <> pack (show ex))
          _ → pure ()
    pure t
```

Observe STM action in a named context and output captured log items

The STM action might output messages, which after "success" will be forwarded to the logging trace. Otherwise, this function behaves the same as **bracketObserveIO**.

```
bracketObserveLogIO :: Config.Configuration → Trace IO a → Severity → Text → STM.STM (t, [(LOMeta, LOContent a)])
bracketObserveLogIO config trace severity name action = do
  subTrace ← fromMaybe Neutral < $ > Config.findSubTrace config name
  bracketObserveLogIO' subTrace severity trace action
where
  bracketObserveLogIO' :: SubTrace → Severity → Trace IO a → STM.STM (t, [(LOMeta, LOContent a)])
  bracketObserveLogIO' NoTrace _ _ act = do
    (t, _) ← STM.atomically $ stmWithLog act
    pure t
  bracketObserveLogIO' subtrace sev logTrace act = do
```

```

mCountersid ← observeOpen subtrace sev logTrace
-- run action, return result and log items; if an exception is
-- caught, then it will be logged and rethrown.
(t,as) ← (STM.atomically $ stmWithLog act) 'catch'
      (λ(e :: SomeException) → (TIO.hPutStrLn stderr (pack (show e)) >> throwM e))
case mCountersid of
  Left openException →
    -- since observeOpen faced an exception there is no reason to call observeClose
    -- however the result of the action is returned
    TIO.hPutStrLn stderr ("ObserveOpen: " <> pack (show openException))
  Right countersid → do
    res ← observeClose subtrace sev logTrace countersid as
    case res of
      Left ex → TIO.hPutStrLn stderr ("ObserveClose: " <> pack (show ex))
      _ → pure ()
  pure t

```

1.7.2 Cardano.BM.Observer.Monadic

Monadic.bracketObserverIO

Observes an *IO* action. The subtrace type is found in the configuration with the passed-in name.

Microbenchmarking steps:

1. Create a *trace* which will have been configured to observe things besides logging.

```

import qualified Cardano.BM.Configuration.Model as CM
ooo
c ← config
trace ← setupTrace (Right c) "demo-playground"
where
  config :: IO CM.Configuration
  config = do
    c ← CM.empty
    CM.setMinSeverity c Debug
    CM.setSetupBackends c [KatipBK, AggregationBK]
    CM.setDefaultBackends c [KatipBK, AggregationBK]
    CM.setSetupScribes c [ScribeDefinition {
      scName = "stdout"
      ,scKind = StdoutSK
      ,scRotation = Nothing
    }
    ]
    CM.setDefaultScribes c ["StdoutSK::stdout"]
  return c

```

2. *c* is the **Configuration** of *trace*. In order to enable the collection and processing of measurements (min, max, mean, std-dev) *AggregationBK* is needed.

```

CM.setDefaultBackends c [KatipBK, AggregationBK]

```

in a configuration file (YAML) means

```
defaultBackends:
  - KatipBK
  - AggregationBK
```

3. Set the measurements that you want to take by changing the configuration of the *trace* using **setSubTrace**, in order to declare the namespace where we want to enable the particular measurements and the list with the kind of measurements.

```
CM.setSubTrace
  config
    "submit-tx"
  (Just $ ObservableTraceSelf observablesSet)
where
  observablesSet = [MonotonicClock, MemoryStats]
```

4. Find an action to measure. e.g.:

```
runProtocolWithPipe x hdl proto 'catch' (λProtocolStopped → return ())
```

and use **bracketObserveIO**. e.g.:

```
bracketObserveIO trace "submit-tx" $
  runProtocolWithPipe x hdl proto 'catch' (λProtocolStopped → return ())
```

```
bracketObserveIO :: Config.Configuration → Trace IO a → Severity → Text → IO t → IO t
bracketObserveIO config trace severity name action = do
  subTrace ← fromMaybe Neutral < $ > Config.findSubTrace config name
  bracketObserveIO' subTrace severity trace action
where
  bracketObserveIO' :: SubTrace → Severity → Trace IO a → IO t → IO t
  bracketObserveIO' NoTrace _ _ act = act
  bracketObserveIO' subtrace sev logTrace act = do
    mCountersid ← observeOpen subtrace sev logTrace
    -- run action; if an exception is caught it will be logged and rethrown.
    t ← act 'catch' (λ(e :: SomeException) → (TIO.hPutStrLn stderr (pack (show e)) >> throwM e))
    case mCountersid of
      Left openException →
        -- since observeOpen faced an exception there is no reason to call observeClose
        -- however the result of the action is returned
        TIO.hPutStrLn stderr ("ObserveOpen: " <> pack (show openException))
      Right countersid → do
        res ← observeClose subtrace sev logTrace countersid []
        case res of
          Left ex → TIO.hPutStrLn stderr ("ObserveClose: " <> pack (show ex))
          _ → pure ()
    pure t
```

Monadic.bracketObserverM

Observes a *MonadIO m* \Rightarrow *m* action.

```

bracketObserveM :: (MonadCatch m, MonadIO m)  $\Rightarrow$  Config.Configuration  $\rightarrow$  Trace m a  $\rightarrow$  Severity  $\rightarrow$  Text
bracketObserveM config trace severity name action = do
  subTrace  $\leftarrow$  liftIO $ fromMaybe Neutral < $ > Config.findSubTrace config name
  bracketObserveM' subTrace severity trace action
where
  bracketObserveM' :: (MonadCatch m, MonadIO m)  $\Rightarrow$  SubTrace  $\rightarrow$  Severity  $\rightarrow$  Trace m a  $\rightarrow$  m t  $\rightarrow$  m t
  bracketObserveM' NoTrace _ _ act = act
  bracketObserveM' subtrace sev logTrace act = do
    mCountersid  $\leftarrow$  observeOpen subtrace sev logTrace
    -- run action; if an exception is caught it will be logged and rethrown.
    t  $\leftarrow$  act 'catch' ( $\lambda$ (e :: SomeException)  $\rightarrow$  liftIO (TIO.hPutStrLn stderr (pack (show e))  $\gg$  throwM e))
    case mCountersid of
      Left openException  $\rightarrow$ 
        -- since observeOpen faced an exception there is no reason to call observeClose
        -- however the result of the action is returned
        liftIO $ TIO.hPutStrLn stderr ("ObserveOpen: " <> pack (show openException))
      Right countersid  $\rightarrow$  do
        res  $\leftarrow$  observeClose subtrace sev logTrace countersid [ ]
        case res of
          Left ex  $\rightarrow$  liftIO (TIO.hPutStrLn stderr ("ObserveClose: " <> pack (show ex)))
          _  $\rightarrow$  pure ()
    pure t

```

Monadic.bracketObserver

Observes a *MonadIO m* \Rightarrow *m* action. This observer bracket does not interfere on exceptions.

```

bracketObserveX :: (MonadIO m)  $\Rightarrow$  Config.Configuration  $\rightarrow$  Trace m a  $\rightarrow$  Severity  $\rightarrow$  Text  $\rightarrow$  m t  $\rightarrow$  m t
bracketObserveX config trace severity name action = do
  subTrace  $\leftarrow$  liftIO $ fromMaybe Neutral < $ > Config.findSubTrace config name
  bracketObserveX' subTrace severity trace action
where
  bracketObserveX' :: (MonadIO m)  $\Rightarrow$  SubTrace  $\rightarrow$  Severity  $\rightarrow$  Trace m a  $\rightarrow$  m t  $\rightarrow$  m t
  bracketObserveX' NoTrace _ _ act = act
  bracketObserveX' subtrace sev logTrace act = do
    countersid  $\leftarrow$  observeOpen0 subtrace sev logTrace
    -- run action
    t  $\leftarrow$  act
    observeClose0 subtrace sev logTrace countersid [ ]
    pure t

```

observerOpen

```

observeOpen :: (MonadCatch m, MonadIO m)  $\Rightarrow$  SubTrace  $\rightarrow$  Severity  $\rightarrow$  Trace m a  $\rightarrow$  m (Either SomeException
observeOpen subtrace severity logTrace = (do
  state  $\leftarrow$  observeOpen0 subtrace severity logTrace

```



```

    return (Right state)) 'catch' (return ◦ Left)
observeOpen0 :: (MonadIO m) ⇒ SubTrace → Severity → Trace m a → m CounterState
observeOpen0 subtrace severity logTrace = do
  -- take measurement
  counters ← liftIO $ readCounters subtrace
  let state = CounterState counters
  if counters ≡ []
  then return ()
  else do
    -- send opening message to Trace
    meta ← mkLOMeta severity Confidential
    traceNamedObject logTrace (meta, ObserveOpen state)
  return state

```

observeClose

```

observeClose
  :: (MonadCatch m, MonadIO m) ⇒ SubTrace → Severity → Trace m a
  → CounterState → [(LOMeta, LOContent a)]
  → m (Either SomeException ())
observeClose subtrace sev logTrace initState logObjects = (do
  observeClose0 subtrace sev logTrace initState logObjects
  return (Right ())) 'catch' (return ◦ Left)
observeClose0 :: (MonadIO m) ⇒ SubTrace → Severity → Trace m a
  → CounterState → [(LOMeta, LOContent a)]
  → m ()
observeClose0 subtrace sev logTrace initState logObjects = do
  let initialCounters = csCounters initState
  -- take measurement
  counters ← liftIO $ readCounters subtrace
  if counters ≡ []
  then return ()
  else do
    mle ← mkLOMeta sev Confidential
    -- send closing message to Trace
    traceNamedObject logTrace $
      (mle, ObserveClose (CounterState counters))
    -- send diff message to Trace
    traceNamedObject logTrace $
      (mle, ObserveDiff (CounterState (diffCounters initialCounters counters)))
    -- trace the messages gathered from inside the action
    forM_ logObjects $ traceNamedObject logTrace
  return ()

```

1.7.3 Cardano.BM.Trace

Utilities

Natural transformation from monad m to monad n .

```

natTrace :: (forall x o m x → n x) → Trace m a → Trace n a
natTrace nat basetrace = natTracer nat basetrace

```

Enter new named context

A new context name is added.

```

appendName :: LoggerName → Trace m a → Trace m a
appendName name =
  modifyName (λprevLoggerName → appendWithDot name prevLoggerName)
appendWithDot :: LoggerName → LoggerName → LoggerName
appendWithDot "" newName = newName
appendWithDot xs "" = xs
appendWithDot xs newName = xs <> "." <> newName

```

Change named context

The context name is overwritten.

```

modifyName
  :: (LoggerName → LoggerName)
  → Tracer m (LogObject a)
  → Tracer m (LogObject a)
modifyName k = contramap f
  where
    f (LogObject name meta item) = LogObject (k name) meta item

```

Contramap a trace and produce the naming context

```

named :: Tracer m (LogObject a) → Tracer m (LOMeta, LOContent a)
named = contramap $ uncurry (LogObject mempty)

```

Trace a LogObject through

```

traceNamedObject
  :: MonadIO m
  ⇒ Trace m a
  → (LOMeta, LOContent a)
  → m ()
traceNamedObject logTrace lo =
  traceWith (named logTrace) lo

```

Concrete Trace on stdout

This function returns a trace with an action of type "`LogObject a → IO ()`" which will output a text message as text and all others as JSON encoded representation to the console.

TODO remove *locallock*

```

locallock :: MVar ()
locallock = unsafePerformIO $ newMVar ()

stdoutTrace :: Tracer IO (LogObject T.Text)
stdoutTrace = Tracer $ \ (LogObject logname _ lc) →
  withMVar locallock $ \ _ →
    case lc of
      (LogMessage logItem) →
        output logname $ logItem
      obj →
        output logname $ toStrict (encodeToLazyText obj)
where
  output nm msg = TIO.putStrLn $ nm <> " :: " <> msg

```

Concrete Trace into a TVar

```

traceInTVar :: STM.TVar [a] → Tracer STM.STM a
traceInTVar tvar = Tracer $ \a → STM.modifyTVar tvar ((:) a)

traceInTVarIO :: STM.TVar [a] → Tracer IO a
traceInTVarIO tvar = Tracer $ \a →
  STM.atomically $ STM.modifyTVar tvar ((:) a)

```

Enter message into a trace

The function `traceNamedItem` creates a `LogObject` and threads this through the action defined in the `Trace`.

```

traceNamedItem
  :: MonadIO m
  ⇒ Trace m a
  → PrivacyAnnotation
  → Severity
  → a
  → m ()
traceNamedItem logTrace p s m =
  traceNamedObject logTrace ≪
    (,) <$> liftIO (mkLOMeta s p)
    <*> pure (LogMessage m)

```

Logging functions

```

logDebug, logInfo, logNotice, logWarning, logError, logCritical, logAlert, logEmergency
  :: MonadIO m ⇒ Trace m a → a → m ()
logDebug   logTrace = traceNamedItem logTrace Public Debug
logInfo    logTrace = traceNamedItem logTrace Public Info
logNotice  logTrace = traceNamedItem logTrace Public Notice
logWarning logTrace = traceNamedItem logTrace Public Warning
logError   logTrace = traceNamedItem logTrace Public Error

```

```

logCritical logTrace = traceNamedItem logTrace Public Critical
logAlert    logTrace = traceNamedItem logTrace Public Alert
logEmergency logTrace = traceNamedItem logTrace Public Emergency
logDebugS, logInfoS, logNoticeS, logWarningS, logErrorS, logCriticalS, logAlertS, logEmergencyS
  :: MonadIO m => Trace m a -> a -> m ()
logDebugS logTrace = traceNamedItem logTrace Confidential Debug
logInfoS logTrace = traceNamedItem logTrace Confidential Info
logNoticeS logTrace = traceNamedItem logTrace Confidential Notice
logWarningS logTrace = traceNamedItem logTrace Confidential Warning
logErrorS logTrace = traceNamedItem logTrace Confidential Error
logCriticalS logTrace = traceNamedItem logTrace Confidential Critical
logAlertS logTrace = traceNamedItem logTrace Confidential Alert
logEmergencyS logTrace = traceNamedItem logTrace Confidential Emergency

```

1.7.4 Cardano.BM.Setup

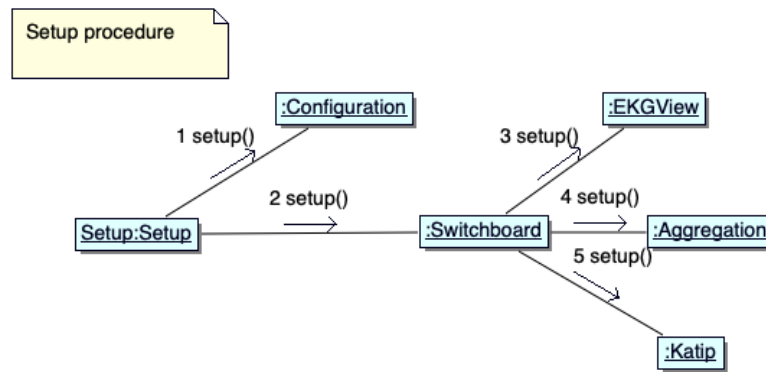


Figure 1.4: Setup procedure

setupTrace

Setup a new **Trace** with either a given **Configuration** or a **FilePath** to a configuration file. After all tracing operations have ended; `shutdownTrace` must be called.

```

setupTrace :: (MonadIO m, FromJSON a, ToObject a) => Either FilePath Config.Configuration -> Text -> m ()
setupTrace (Left cfgFile) name = do
  c <- liftIO $ Config.setup cfgFile
  fst <$> setupTrace_ c name
setupTrace (Right c) name = fst <$> setupTrace_ c name
setupTrace_ :: (MonadIO m, FromJSON a, ToObject a) => Config.Configuration -> Text -> m (Trace m a, Switchboard)
setupTrace_ c name = do
  sb <- liftIO $ Switchboard.realize c
  let tr = appendName name $ natTrace liftIO (Switchboard.mainTraceConditionally c sb)
  return (tr, sb)

```

shutdown

Shut down the Switchboard and all the **Traces** related to it.

```
shutdown :: (FromJSON a, ToObject a) => Switchboard.Switchboard a → IO ()
shutdown = Switchboard.unrealize
```

withTrace

Setup a **Trace** from **Configuration** and pass it to the action. At the end, shutdown all the components and close the trace.

```
withTrace :: (MonadIO m, MonadMask m, FromJSON a, ToObject a) => Config.Configuration → Text → (Trace m a) → m a
withTrace cfg name action =
  bracket
    (setupTrace_cfg name)           -- acquire
    (\(tr, sb) → liftIO $ shutdown sb) -- release
    (\(tr, _) → action tr)         -- action
```

1.7.5 Cardano.BM.Counters

Here the platform is chosen on which we compile this program.

Currently, we mainly support *Linux* with its 'proc' filesystem.

```
{-# LANGUAGE CPP #-}
# if defined (linux_HOST_OS)
# define LINUX
# endif
module Cardano.BM.Counters
(
  Platform.readCounters
  , getMonoClock
) where
# ifdef LINUX
import qualified Cardano.BM.Counters.Linux as Platform
# else
import qualified Cardano.BM.Counters.Dummy as Platform
# endif
import Cardano.BM.Counters.Common (getMonoClock)
```

1.7.6 Cardano.BM.Counters.Common

Common functions that serve *readCounters* on all platforms.

```
nominalTimeToMicroseconds :: Word64 → Microsecond
nominalTimeToMicroseconds = fromMicroseconds ∘ toInteger ∘ ('div' 1000)
```

Read monotonic clock

```
getMonoClock :: IO [Counter]
getMonoClock = do
  t ← getMonotonicTimeNSec
  return [Counter MonotonicClockTime "monoclock" $ Microseconds (t `div` 1000)]
```

Read GHC RTS statistics

Read counters from GHC's RTS (runtime system). The values returned are as per the last GC (garbage collection) run.

```
readRTSStats :: IO [Counter]
readRTSStats = do
  iscollected ← GhcStats.getRTSStatsEnabled
  if iscollected
  then ghcstats
  else return []
where
  ghcstats :: IO [Counter]
  ghcstats = do
    -- need to run GC?
    rts ← GhcStats.getRTSStats
    let getrts = ghcval rts
    return [getrts (Bytes ∘ fromIntegral ∘ GhcStats.allocated_bytes, "bytesAllocated")
          ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_live_bytes, "liveBytes")
          ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_large_objects_bytes, "largeBytes")
          ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_compact_bytes, "compactBytes")
          ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_slop_bytes, "slopBytes")
          ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_mem_in_use_bytes, "usedMemBytes")
          ,getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.gc_cpu_ns, "gcCpuNs")
          ,getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.gc_elapsed_ns, "gcElapsedNs")
          ,getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.cpu_ns, "cpuNs")
          ,getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.elapsed_ns, "elapsedNs")
          ,getrts (PureI ∘ toInteger ∘ GhcStats.gcs, "gcNum")
          ,getrts (PureI ∘ toInteger ∘ GhcStats.major_gcs, "gcMajorNum")
          ]
  ghcval :: GhcStats.RTSStats → ((GhcStats.RTSStats → Measurable), Text) → Counter
  ghcval s (f, n) = Counter RTSStats n $ (f s)
```

1.7.7 Cardano.BM.Counters.Dummy

This is a dummy definition of `readCounters` on platforms that do not support the 'proc' filesystem from which we would read the counters.

The only supported measurements are monotonic clock time and RTS statistics for now.

```
readCounters :: SubTrace → IO [Counter]
readCounters NoTrace           = return []
readCounters Neutral           = return []
readCounters (TeeTrace _)      = return []
readCounters (FilterTrace _)   = return []
```

```

readCounters UntimedTrace           = return [ ]
readCounters DropOpening           = return [ ]
readCounters (SetSeverity _)        = return [ ]
# ifdef ENABLE_OBSERVABLES
readCounters (ObservableTraceSelf tts) = readCounters' tts [ ]
readCounters (ObservableTrace _ tts) = readCounters' tts [ ]
readCounters' :: [ObservableInstance] → [Counter] → IO [Counter]
readCounters' [ ] acc = return acc
readCounters' (MonotonicClock : r) acc = getMonoClock >>= λxs → readCounters' r $ acc ++ xs
readCounters' (GhcRtsStats : r) acc = readRTSStats >>= λxs → readCounters' r $ acc ++ xs
readCounters' ( _ : r) acc = readCounters' r acc
# else
readCounters (ObservableTraceSelf _) = return [ ]
readCounters (ObservableTrace _ _) = return [ ]
# endif

```

1.7.8 Cardano.BM.Counters.Linux

we have to expand the `readMemStats` function
to read full data from `proc`

```

readCounters :: SubTrace → IO [Counter]
readCounters NoTrace           = return [ ]
readCounters Neutral           = return [ ]
readCounters (TeeTrace _)      = return [ ]
readCounters (FilterTrace _)  = return [ ]
readCounters UntimedTrace     = return [ ]
readCounters DropOpening     = return [ ]
readCounters (SetSeverity _)  = return [ ]
# ifdef ENABLE_OBSERVABLES
readCounters (ObservableTraceSelf tts) = do
    pid ← getProcessID
    takeMeasurements pid tts
readCounters (ObservableTrace pid tts) =
    takeMeasurements pid tts
takeMeasurements :: ProcessID → [ObservableInstance] → IO [Counter]
takeMeasurements pid tts =
    foldrM (λ(sel, fun) a →
        if any (≡ sel) tts
        then (fun >>= λxs → return $ a ++ xs)
        else return a) [ ] selectors
where
    selectors = [ (MonotonicClock, getMonoClock)
                , (MemoryStats, readProcStatM pid)
                , (ProcessStats, readProcStats pid)
                , (NetStats, readProcNet pid)
                , (IOStats, readProcIO pid)
                , (GhcRtsStats, readRTSStats)
                ]
# else

```

```

readCounters (ObservableTraceSelf _) = return []
readCounters (ObservableTrace _ _) = return []
# endif

# ifdef ENABLE_OBSERVABLES
pathProc :: FilePath
pathProc = "/proc/"
pathProcStat :: ProcessID → FilePath
pathProcStat pid = pathProc </> (show pid) </> "stat"
pathProcStatM :: ProcessID → FilePath
pathProcStatM pid = pathProc </> (show pid) </> "statm"
pathProcIO :: ProcessID → FilePath
pathProcIO pid = pathProc </> (show pid) </> "io"
pathProcNet :: ProcessID → FilePath
pathProcNet pid = pathProc </> (show pid) </> "net" </> "netstat"
# endif

```

Reading from a file in /proc/<pid>

```

# ifdef ENABLE_OBSERVABLES
readProcList :: FilePath → IO [Integer]
readProcList fp = do
  fs ← getFileStatus fp
  if readable fs
  then do
    cs ← readFile fp
    return $ map (\s → maybe 0 id $ (readMaybe s :: Maybe Integer)) (words cs)
  else
    return []
  where
    readable fs = intersectFileModes (fileMode fs) ownerReadMode ≡ ownerReadMode
# endif

```

readProcStatM - /proc/<pid>/statm

```

/proc/[pid]/statm
  Provides information about memory usage, measured in pages. The columns are:
    size      (1) total program size
               (same as VmSize in /proc/[pid]/status)
    resident  (2) resident set size
               (same as VmRSS in /proc/[pid]/status)
    shared    (3) number of resident shared pages (i.e., backed by a file)
               (same as RssFile+RssShmem in /proc/[pid]/status)
    text      (4) text (code)
    lib       (5) library (unused since Linux 2.6; always 0)
    data      (6) data + stack
    dt        (7) dirty pages (unused since Linux 2.6; always 0)

```

```

# ifdef ENABLE_OBSERVABLES
readProcStatM :: ProcessID → IO [Counter]
readProcStatM pid = do
  ps0 ← readProcList (pathProcStatM pid)

```



```

let ps = zip colnames ps0
    psUseful = filter (("unused"  $\neq$ )  $\circ$  fst) ps
return $ map (\(n,i)  $\rightarrow$  Counter MemoryCounter n (PureI i)) psUseful
where
    colnames :: [Text]
    colnames = ["size", "resident", "shared", "text", "unused", "data", "unused"]
# endif

```

readProcStats - //proc//<pid >//stat

```
/proc/[pid]/stat
```

Status information about the process. This is used by ps(1). It is defined in the kernel source file fs/proc/array.c.

The fields, in order, with their proper scanf(3) format specifiers, are listed below. Whether or not certain of these fields display valid information is governed by a ptrace access mode PTRACE_MODE_READ_FSCREDS | PTRACE_MODE_NOAUDIT check (refer to ptrace(2)). If the check denies access, then the field value is displayed as 0. The affected fields are indicated with the marking [PT].

- (1) pid %d
The process ID.
- (2) comm %s
The filename of the executable, in parentheses. This is visible whether or not the executable is swapped out.
- (3) state %c
One of the following characters, indicating process state:
 - R Running
 - S Sleeping in an interruptible wait
 - D Waiting in uninterruptible disk sleep
 - Z Zombie
 - T Stopped (on a signal) or (before Linux 2.6.33) trace stopped
 - t Tracing stop (Linux 2.6.33 onward)
 - W Paging (only before Linux 2.6.0)
 - X Dead (from Linux 2.6.0 onward)
 - x Dead (Linux 2.6.33 to 3.13 only)
 - K Wakekill (Linux 2.6.33 to 3.13 only)
 - W Waking (Linux 2.6.33 to 3.13 only)
 - P Parked (Linux 3.9 to 3.13 only)
- (4) ppid %d
The PID of the parent of this process.
- (5) pgrp %d
The process group ID of the process.
- (6) session %d
The session ID of the process.
- (7) tty_nr %d
The controlling terminal of the process. (The minor device number is contained in the combination of bits 31 to 20 and 7 to 0; the major device number is in bits 15 to 8.)
- (8) tpgid %d
The ID of the foreground process group of the controlling terminal of the process.

- (9) `flags` %u
The kernel `flags` word of the process. For bit meanings, see the `PF_*` defines in the Linux kernel source file `include/linux/sched.h`. Details depend on the kernel version.
- The format for this field was %lu before Linux 2.6.
- (10) `minflt` %lu
The number of minor faults the process has made which have not required loading a memory page from disk.
- (11) `cminflt` %lu
The number of minor faults that the process's waited-for children have made.
- (12) `majflt` %lu
The number of major faults the process has made which have required loading a memory page from disk.
- (13) `cmajflt` %lu
The number of major faults that the process's waited-for children have made.
- (14) `utime` %lu
Amount of time that this process has been scheduled in user mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`). This includes guest time, `guest_time` (time spent running a virtual CPU, see below), so that applications that are not aware of the guest time field do not lose that time from their calculations.
- (15) `stime` %lu
Amount of time that this process has been scheduled in kernel mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).
- (16) `cutime` %ld
Amount of time that this process's waited-for children have been scheduled in user mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`). (See also `times(2)`.) This includes guest time, `cguest_time` (time spent running a virtual CPU, see below).
- (17) `cstime` %ld
Amount of time that this process's waited-for children have been scheduled in kernel mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).
- (18) `priority` %ld
(Explanation for Linux 2.6) For processes running a real-time scheduling policy (policy below; see `sched_setscheduler(2)`), this is the negated scheduling priority, minus one; that is, a number in the range -2 to -100, corresponding to real-time priorities 1 to 99. For processes running under a non-real-time scheduling policy, this is the raw nice value (`setpriority(2)`) as represented in the kernel. The kernel stores nice values as numbers in the range 0 (high) to 39 (low), corresponding to the user-visible nice range of -20 to 19.
- (19) `nice` %ld
The nice value (see `setpriority(2)`), a value in the range 19 (low priority) to -20 (high priority).
- (20) `num_threads` %ld
Number of threads in this process (since Linux 2.6). Before kernel 2.6, this field was hard coded to 0 as a placeholder for an earlier removed field.
- (21) `itrealvalue` %ld
The time in jiffies before the next `SIGALRM` is sent to the process due to an interval timer. Since kernel 2.6.17, this field is no longer maintained, and is hard coded as 0.
- (22) `starttime` %llu
The time the process started after system boot. In kernels before Linux 2.6, this value was expressed in jiffies. Since Linux 2.6, the value is expressed in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).
- The format for this field was %lu before Linux 2.6.
- (23) `vsize` %lu
Virtual memory size in bytes.
- (24) `rss` %ld
Resident Set Size: number of pages the process has in real memory. This is just the pages which count toward text, data, or stack space. This does not include pages which have not been demand-loaded in, or which are swapped out.

- (25) `rsslim %lu`
Current soft limit in bytes on the rss of the process; see the description of `RLIMIT_RSS` in `getrlimit(2)`.
- (26) `startcode %lu [PT]`
The address above which program text can run.
- (27) `endcode %lu [PT]`
The address below which program text can run.
- (28) `startstack %lu [PT]`
The address of the start (i.e., bottom) of the stack.
- (29) `kstkesp %lu [PT]`
The current value of ESP (stack pointer), as found in the kernel stack page for the process.
- (30) `kstkeip %lu [PT]`
The current EIP (instruction pointer).
- (31) `signal %lu`
The bitmap of pending signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use `/proc/[pid]/status` instead.
- (32) `blocked %lu`
The bitmap of blocked signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use `/proc/[pid]/status` instead.
- (33) `sigignore %lu`
The bitmap of ignored signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use `/proc/[pid]/status` instead.
- (34) `sigcatch %lu`
The bitmap of caught signals, displayed as a decimal number. Obsolete, because it does not provide information on real-time signals; use `/proc/[pid]/status` instead.
- (35) `wchan %lu [PT]`
This is the "channel" in which the process is waiting. It is the address of a location in the kernel where the process is sleeping. The corresponding symbolic name can be found in `/proc/[pid]/wchan`.
- (36) `nswap %lu`
Number of pages swapped (not maintained).
- (37) `cswap %lu`
Cumulative nswap for child processes (not maintained).
- (38) `exit_signal %d (since Linux 2.1.22)`
Signal to be sent to parent when we die.
- (39) `processor %d (since Linux 2.2.8)`
CPU number last executed on.
- (40) `rt_priority %u (since Linux 2.5.19)`
Real-time scheduling priority, a number in the range 1 to 99 for processes scheduled under a real-time policy, or 0, for non-real-time processes (see `sched_setscheduler(2)`).
- (41) `policy %u (since Linux 2.5.19)`
Scheduling policy (see `sched_setscheduler(2)`). Decode using the `SCHED_*` constants in `linux/sched.h`.

The format for this field was `%lu` before Linux 2.6.22.
- (42) `delayacct_blkio_ticks %llu (since Linux 2.6.18)`
Aggregated block I/O delays, measured in clock ticks (centiseconds).
- (43) `guest_time %lu (since Linux 2.6.24)`
Guest time of the process (time spent running a virtual CPU for a guest operating system), measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).
- (44) `cguest_time %ld (since Linux 2.6.24)`
Guest time of the process's children, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).

- (45) `start_data %lu` (since Linux 3.3) [PT]
Address above which program initialized and uninitialized (BSS) data are placed.
- (46) `end_data %lu` (since Linux 3.3) [PT]
Address below which program initialized and uninitialized (BSS) data are placed.
- (47) `start_brk %lu` (since Linux 3.3) [PT]
Address above which program heap can be expanded with `brk(2)`.
- (48) `arg_start %lu` (since Linux 3.5) [PT]
Address above which program command-line arguments (`argv`) are placed.
- (49) `arg_end %lu` (since Linux 3.5) [PT]
Address below program command-line arguments (`argv`) are placed.
- (50) `env_start %lu` (since Linux 3.5) [PT]
Address above which program environment is placed.
- (51) `env_end %lu` (since Linux 3.5) [PT]
Address below which program environment is placed.
- (52) `exit_code %d` (since Linux 3.5) [PT]
The thread's exit status in the form reported by `waitpid(2)`.

```
#ifdef ENABLE_OBSERVABLES
```

```
readProcStats :: ProcessID → IO [Counter]
```

```
readProcStats pid = do
```

```
    ps0 ← readProcList (pathProcStat pid)
```

```
    let ps = zip colnames ps0
```

```
    psUseful = filter (("unused" ≠) ∘ fst) ps
```

```
    return $ map (λ(n,i) → Counter StatInfo n (PureI i)) psUseful
```

```
where
```

```
colnames :: [Text]
```

```
colnames = [ "pid", "unused", "unused", "ppid", "pgrp", "session", "ttynr", "tpgid", "flags", "minflt", "majflt", "cmajflt", "utime", "stime", "cutime", "cstime", "priority", "nice", "rtime", "itrealvalue", "starttime", "vsize", "rss", "rsslim", "startcode", "endcode", "startstack", "signal", "blocked", "sigignore", "sigcatch", "wchan", "nswap", "cswap", "exitcode", "policy", "blkio", "guesttime", "cguesttime", "startdata", "enddata", "startbrk", "argstart", "envend", "exitcode"
```

```
    ]
```

```
#endif
```

readProcIO - `//proc/<pid>/io`

```
/proc/[pid]/io (since kernel 2.6.20)
```

This file contains I/O statistics for the process, for example:

```
# cat /proc/3828/io
rchar: 323934931
wchar: 323929600
syscr: 632687
syscw: 632675
read_bytes: 0
write_bytes: 323932160
cancelled_write_bytes: 0
```

The fields are as follows:

`rchar`: characters read

The number of bytes which this task has caused to be read from storage. This is simply the sum of bytes which this process passed to `read(2)` and similar system calls. It includes things such as terminal I/O and is unaffected by whether or not actual physical disk I/O was required (the

```

    read might have been satisfied from pagecache).

wchar: characters written
    The number of bytes which this task has caused, or shall cause to be written to disk. Similar
    caveats apply here as with rchar.

syscr: read syscalls
    Attempt to count the number of read I/O operations—that is, system calls such as read(2) and
    pread(2).

syscw: write syscalls
    Attempt to count the number of write I/O operations—that is, system calls such as write(2) and
    pwrite(2).

read_bytes: bytes read
    Attempt to count the number of bytes which this process really did cause to be fetched from the
    storage layer. This is accurate for block-backed filesystems.

write_bytes: bytes written
    Attempt to count the number of bytes which this process caused to be sent to the storage layer.

cancelled_write_bytes:
    The big inaccuracy here is truncate. If a process writes 1MB to a file and then deletes the
    file, it will in fact perform no writeout. But it will have been accounted as having caused 1MB
    of write. In other words: this field represents the number of bytes which this process caused
    to not happen, by truncating pagecache. A task can cause "negative" I/O too. If this task
    truncates some dirty pagecache, some I/O which another task has been accounted for (in its
    write_bytes) will not be happening.

Note: In the current implementation, things are a bit racy on 32-bit systems: if process A reads
process B's /proc/[pid]/io while process B is updating one of these 64-bit counters, process A could
see an intermediate result.

Permission to access this file is governed by a ptrace access mode PTRACE_MODE_READ_FSCREDS check; see
ptrace(2).

```

```

# ifdef ENABLE_OBSERVABLES
readProcIO :: ProcessID → IO [Counter]
readProcIO pid = do
    ps0 ← readProcList (pathProcIO pid)
    let ps = zip3 colnames ps0 units
    return $ map (λ(n,i,u) → Counter IOCounter n (u i)) ps
where
    colnames :: [Text]
    colnames = [ "rchar", "wchar", "syscr", "syscw", "rbytes", "wbytes", "cxwbytes" ]
    units = [ Bytes ∘ fromInteger, Bytes ∘ fromInteger, PureI, PureI, Bytes ∘ fromInteger, Bytes ∘ fromInteger, Bytes ∘ fromInteger ]
# endif

```

Network TCP/IP counters

example:

```

\\
cat /proc/<pid>/net/netstat
\\

```

```

TcpExt: SyncookiesSent SyncookiesRecv SyncookiesFailed EmbryonicRsts PruneCalled RcvPruned OfoPruned OutOfWindowIcmps Lo!
!ckDroppedIcmps ArpFilter TW TwRecycled TWKilled PAWSActive PAWSEstab DelayedACKs DelayedACKLocked DelayedACKLost Listen0!
!verflows ListenDrops TCPHPHits TCPPureAcks TCPHPAcks TCPRecovery TCPSackRecovery TCPSACKReneging TCPSACKReorder TCPRe!
!enoReorder TCPTSReorder TCPFullUndo TCPPartialUndo TCPDSACKUndo TCPLossUndo TCPLostRetransmit TCPRecoveryFailures TCPSackFai!
!lures TCPLossFailures TCPFastRetrans TCPSlowStartRetrans TCPTimeouts TCPLossProbes TCPLossProbeRecovery TCPRecoveryF!
!ail TCPSackRecoveryFail TCPRecvCollapsed TCPDSACKOldSent TCPDSACKOfoSent TCPDSACKRecv TCPDSACKOfoRecv TCPAbortOnData TCPA!
!bortOnClose TCPAbortOnMemory TCPAbortOnTimeout TCPAbortOnLinger TCPAbortFailed TCPMemoryPressures TCPMemoryPressuresChro!
!no TCPSACKDiscard TCPDSACKIgnoredOld TCPDSACKIgnoredNoUndo TCPSpuriousRTOs TCPMD5NotFound TCPMD5Unexpected TCPMD5Failure!
! TCPSackShifted TCPSackMerged TCPSackShiftFallback TCPBacklogDrop PFMemallocDrop TCPMinTTLDrop TCPDeferAcceptDrop IPReve!
!rsePathFilter TCPTimeWaitOverflow TCPReqQFullDoCookies TCPReqQFullDrop TCPRetransFail TCPRecvCoalesce TCPOfOQueue TCPOFOD!
!rop TCPOFOMerge TCPChallengeACK TCPSYNChallenge TCPFastOpenActive TCPFastOpenActiveFail TCPFastOpenPassive TCPFastOpenPas!

```

```

!siveFail TCPFastOpenListenOverflow TCPFastOpenCookieReqd TCPFastOpenBlackhole TCPSpuriousRtxHostQueues BusyPollRxPackets!
! TCPAutoCorking TCPFromZeroWindowAdv TCPToZeroWindowAdv TCPWantZeroWindowAdv TCPSynRetrans TCPOrigDataSent TCPHystartTra!
!inDetect TCPHystartTrainCwnd TCPHystartDelayDetect TCPHystartDelayCwnd TCPACKSkippedSynRecv TCPACKSkippedPAWS TCPACKSkip!
!pedSeq TCPACKSkippedFinWait2 TCPACKSkippedTimeWait TCPACKSkippedChallenge TCPWinProbe TCPKeepAlive TCPMTUPFail TCPMTUPSu!
!ccess TCPDelivered TCPDeliveredCE TCPAckCompressed
TcpExt: 0 0 0 0 28 0 0 0 0 1670 1 0 0 6 6029 1 1766 0 0 384612 66799 105553 0 21 0 638 0 1 7 1 1 32 128 0 1 0 22 0 116!
! 383 19 0 0 0 1788 224 178 0 435 224 0 13 0 0 0 0 0 67 0 0 0 0 3 1 668 0 0 0 4 0 0 0 0 91870 4468 0 224 22 23 0 0 0 !
!0 0 0 0 6 0 21492 0 0 11 188 188680 6 145 13 425 0 3 4 0 0 1 117 22984 0 0 192495 0 4500
IpExt: InNoRoutes InTruncatedPkts InMcastPkts OutMcastPkts InBcastPkts OutBcastPkts InOctets OutOctets InMcastOctets Out!
!McastOctets InBcastOctets OutBcastOctets InCsumErrors InNoECTPkts InECT1Pkts InECT0Pkts InCEPkts
IpExt: 0 0 20053 8977 2437 23 3163525943 196480057 2426648 1491754 394285 5523 0 3513269 0 217426 0

```

```

# ifdef ENABLE_OBSERVABLES
readProcNet :: ProcessID → IO [Counter]
readProcNet pid = do
  ls0 ← lines < $ > readFile (pathProcNet pid)
  let ps0 = readinfo ls0
  let ps1 = map (λ(n,c) → (n,readMaybe c :: Maybe Integer)) ps0
  return $ mapCounters $ filter selcolumns ps1
where
  construct "IpExt:OutOctets" i = Bytes $ fromInteger i
  construct "IpExt:InOctets" i = Bytes $ fromInteger i
  construct _ i = PureI i
  -- only a few selected columns will be returned
  selcolumns (n,_) = n ∈ [ "IpExt:OutOctets", "IpExt:InOctets" ]
  mapCounters [] = []
  mapCounters ((n,c):r) = case c of
    Nothing → mapCounters r
    Just i → mapCounters r <> [ Counter NetCounter (pack n) (construct n i) ]
  readinfo :: [String] → [(String,String)]
  readinfo [] = []
  readinfo (_:[]) = []
  readinfo (l1:l2:r) =
    let col0 = words l1
        cols = tail col0
        vals = tail $ words l2
        pref = head col0
    in
      readinfo r <> zip (map (λn → pref ++ n) cols) vals
# endif

```

1.7.9 Cardano.BM.Data.Aggregated

Measurable

A **Measurable** may consist of different types of values. Time measurements are strict, so are *Bytes* which are externally measured. The real or integral numeric values are lazily linked, so we can decide later to drop them.

```

data Measurable = Microseconds {-# UNPACK #-} ! Word64
  | Nanoseconds {-# UNPACK #-} ! Word64
  | Seconds {-# UNPACK #-} ! Word64
  | Bytes {-# UNPACK #-} ! Word64
  | PureD ! Double

```

```

| PureI      ! Integer
| Severity   S.Severity
deriving (Eq, Read, Generic, ToJSON, FromJSON)

```

Measurable can be transformed to an integral value.

```

instance Ord Measurable where
  compare (Seconds a) (Seconds b)           = compare a b
  compare (Microseconds a) (Microseconds b) = compare a b
  compare (Nanoseconds a) (Nanoseconds b)   = compare a b
  compare (Seconds a) (Microseconds b)      = compare (a * 1000 * 1000) b
  compare (Nanoseconds a) (Microseconds b)  = compare a (b * 1000)
  compare (Seconds a) (Nanoseconds b)       = compare (a * 1000 * 1000 * 1000) b
  compare (Microseconds a) (Nanoseconds b)  = compare (a * 1000) b
  compare (Microseconds a) (Seconds b)      = compare a (b * 1000 * 1000)
  compare (Nanoseconds a) (Seconds b)       = compare a (b * 1000 * 1000 * 1000)
  compare (Bytes a) (Bytes b)               = compare a b
  compare (PureD a) (PureD b)               = compare a b
  compare (PureI a) (PureI b)               = compare a b
  compare (Severity a) (Severity b)         = compare a b
  compare (PureI a) (Seconds b) | a ≥ 0      = compare a (toInteger b)
  compare (PureI a) (Microseconds b) | a ≥ 0 = compare a (toInteger b)
  compare (PureI a) (Nanoseconds b) | a ≥ 0  = compare a (toInteger b)
  compare (PureI a) (Bytes b) | a ≥ 0        = compare a (toInteger b)
  compare (Seconds a) (PureI b) | b ≥ 0      = compare (toInteger a) b
  compare (Microseconds a) (PureI b) | b ≥ 0 = compare (toInteger a) b
  compare (Nanoseconds a) (PureI b) | b ≥ 0  = compare (toInteger a) b
  compare (Bytes a) (PureI b) | b ≥ 0        = compare (toInteger a) b
  compare a@(PureD _) (PureI b)             = compare (getInteger a) b
  compare (PureI a) b@(PureD _)             = compare a (getInteger b)
  compare _a _b                             = LT

```

Measurable can be transformed to an integral value.

```

getInteger :: Measurable → Integer
getInteger (Microseconds a) = toInteger a
getInteger (Nanoseconds a) = toInteger a
getInteger (Seconds a)     = toInteger a
getInteger (Bytes a)       = toInteger a
getInteger (PureI a)       = a
getInteger (PureD a)       = round a
getInteger (Severity a)    = toInteger (fromEnum a)

```

Measurable can be transformed to a rational value.

```

getDouble :: Measurable → Double
getDouble (Microseconds a) = fromIntegral a
getDouble (Nanoseconds a) = fromIntegral a
getDouble (Seconds a)     = fromIntegral a
getDouble (Bytes a)       = fromIntegral a
getDouble (PureI a)       = fromInteger a
getDouble (PureD a)       = a
getDouble (Severity a)    = fromIntegral (fromEnum a)

```

It is a numerical value, thus supports functions to operate on numbers.

instance Num Measurable where

(+) (Microseconds a) (Microseconds b) = Microseconds (a + b)
 (+) (Nanoseconds a) (Nanoseconds b) = Nanoseconds (a + b)
 (+) (Seconds a) (Seconds b) = Seconds (a + b)
 (+) (Bytes a) (Bytes b) = Bytes (a + b)
 (+) (PureI a) (PureI b) = PureI (a + b)
 (+) (PureD a) (PureD b) = PureD (a + b)
 (+) a = a

(*) (Microseconds a) (Microseconds b) = Microseconds (a * b)
 (*) (Nanoseconds a) (Nanoseconds b) = Nanoseconds (a * b)
 (*) (Seconds a) (Seconds b) = Seconds (a * b)
 (*) (Bytes a) (Bytes b) = Bytes (a * b)
 (*) (PureI a) (PureI b) = PureI (a * b)
 (*) (PureD a) (PureD b) = PureD (a * b)
 (*) a = a

abs (Microseconds a) = Microseconds (abs a)
 abs (Nanoseconds a) = Nanoseconds (abs a)
 abs (Seconds a) = Seconds (abs a)
 abs (Bytes a) = Bytes (abs a)
 abs (PureI a) = PureI (abs a)
 abs (PureD a) = PureD (abs a)
 abs a = a

signum (Microseconds a) = Microseconds (signum a)
 signum (Nanoseconds a) = Nanoseconds (signum a)
 signum (Seconds a) = Seconds (signum a)
 signum (Bytes a) = Bytes (signum a)
 signum (PureI a) = PureI (signum a)
 signum (PureD a) = PureD (signum a)
 signum a = a

negate (Microseconds a) = Microseconds (negate a)
 negate (Nanoseconds a) = Nanoseconds (negate a)
 negate (Seconds a) = Seconds (negate a)
 negate (Bytes a) = Bytes (negate a)
 negate (PureI a) = PureI (negate a)
 negate (PureD a) = PureD (negate a)
 negate a = a

fromInteger = PureI

subtractMeasurable :: Measurable → Measurable → Measurable

subtractMeasurable (Microseconds a) (Microseconds b) = Microseconds (a - b)
 subtractMeasurable (Nanoseconds a) (Nanoseconds b) = Nanoseconds (a - b)
 subtractMeasurable (Seconds a) (Seconds b) = Seconds (a - b)
 subtractMeasurable (Bytes a) (Bytes b) = Bytes (a - b)
 subtractMeasurable (PureI a) (PureI b) = PureI (a - b)
 subtractMeasurable (PureD a) (PureD b) = PureD (a - b)
 subtractMeasurable a = a

Pretty printing of **Measurable**.

instance Show Measurable where

show v@(Microseconds a) = show a ++ showUnits v


```

show v@(Nanoseconds a) = show a ++ showUnits v
show v@(Seconds a)     = show a ++ showUnits v
show v@(Bytes a)       = show a ++ showUnits v
show v@(PureI a)       = show a ++ showUnits v
show v@(PureD a)       = show a ++ showUnits v
show v@(Severity a)    = show a ++ showUnits v

showUnits :: Measurable → String
showUnits (Microseconds _) = " s"
showUnits (Nanoseconds _) = " ns"
showUnits (Seconds _)     = " s"
showUnits (Bytes _)       = " B"
showUnits (PureI _)       = ""
showUnits (PureD _)       = ""
showUnits (Severity _)    = ""

-- show in S.I. units
showSI :: Measurable → String
showSI (Microseconds a) = show (fromFloatDigits ((fromIntegral a) / (1000 :: Float) / (1000 :: Float))) ++
                             showUnits (Seconds a)
showSI (Nanoseconds a) = show (fromFloatDigits ((fromIntegral a) / (1000 :: Float) / (1000 :: Float) / (1000 :: Float))) ++
                             showUnits (Seconds a)
showSI v@(Seconds a)   = show a ++ showUnits v
showSI v@(Bytes a)     = show a ++ showUnits v
showSI v@(PureI a)     = show a ++ showUnits v
showSI v@(PureD a)     = show a ++ showUnits v
showSI v@(Severity a) = show a ++ showUnits v

```

Stats

A **Stats** statistics is strictly computed.

```

data BaseStats = BaseStats {
  fmin :: !Measurable,
  fmax :: !Measurable,
  fcount :: {-# UNPACK #-} !Word64,
  fsum_A :: {-# UNPACK #-} !Double,
  fsum_B :: {-# UNPACK #-} !Double
} deriving (Show, Generic, ToJSON, FromJSON)

instance Eq BaseStats where
  (BaseStats mina maxa counta sumAa sumBa) ≡ (BaseStats minb maxb countb sumAb sumBb) =
    mina ≡ minb ∧ maxa ≡ maxb ∧ counta ≡ countb ∧
    abs (sumAa - sumAb) < 1.0e-4 ∧
    abs (sumBa - sumBb) < 1.0e-4

data Stats = Stats {
  flast :: !Measurable,
  fold :: !Measurable,
  fbasic :: !BaseStats,
  fdelta :: !BaseStats,
  ftimed :: !BaseStats
} deriving (Show, Eq, Generic, ToJSON, FromJSON)

```

```
meanOfStats :: BaseStats → Double
meanOfStats = fsum_A
```

```
stdevOfStats :: BaseStats → Double
stdevOfStats s =
    calculate (fcount s)
  where
    calculate :: Word64 → Double
    calculate n =
      if n ≥ 2
      then sqrt $ (fsum_B s) / (fromInteger $ fromIntegral (n - 1))
      else 0
```

instance Semigroup Stats disabled for the moment, because not needed.

We use a parallel algorithm to update the estimation of mean and variance from two sample statistics. (see https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Parallel_algorithm)

```
instance Semigroup Stats where
  (<>) a b = let counta = fcount a
              countb = fcount b
              newcount = counta + countb
              delta = fsum_A b - fsum_A a
            in
      Stats {flast = flast b -- right associative
            ,fmin   = min (fmin a) (fmin b)
            ,fmax   = max (fmax a) (fmax b)
            ,fcount = newcount
            ,fsum_A = fsum_A a + (delta / fromInteger newcount)
            ,fsum_B = fsum_B a + fsum_B b + (delta * delta) * (fromInteger (counta * countb) / fromInteger newcount)
            }
```

```
stats2Text :: Stats → Text
stats2Text (Stats slast _ sbasic sdelta stimed) =
  pack $
    "{ last=" ++ show slast ++
    ", basic-stats=" ++ showStats' (sbasic) ++
    ", delta-stats=" ++ showStats' (sdelta) ++
    ", timed-stats=" ++ showStats' (stimed) ++
    " }"
```

```
where
  showStats' :: BaseStats → String
  showStats' s =
    ", { min=" ++ show (fmin s) ++
    ", max=" ++ show (fmax s) ++
    ", mean=" ++ show (meanOfStats s) ++ showUnits (fmin s) ++
    ", std-dev=" ++ show (stdevOfStats s) ++
    ", count=" ++ show (fcount s) ++
    " }"
```

Exponentially Weighted Moving Average (EWMA)

Following https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average we calculate the exponential moving average for a series of values Y_t according to:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

```
data EWMA = EmptyEWMA {alpha :: Double}
  | EWMA {alpha :: Double
    , avg :: Measurable
    } deriving (Show, Eq, Generic, ToJSON, FromJSON)
```

Aggregated

```
data Aggregated = AggregatedStats Stats
  | AggregatedEWMA EWMA
  deriving (Eq, Generic, ToJSON, FromJSON)
```

instance Semigroup Aggregated disabled for the moment, because not needed.

```
instance Semigroup Aggregated where
  (<>) (AggregatedStats a) (AggregatedStats b) =
    AggregatedStats (a <> b)
  (<>) a _ = a
```

```
singletonStats :: Measurable → Aggregated
```

```
singletonStats a =
  let stats = Stats {flast = a
    , fold          = Nanoseconds 0
    , fbasic = BaseStats
      {fmin = a
      , fmax = a
      , fcount = 1
      , fsum_A = getDouble a
      , fsum_B = 0}
    , fdelta = BaseStats
      {fmin = 0
      , fmax = 0
      , fcount = 1
      , fsum_A = 0
      , fsum_B = 0}
    , ftimed = BaseStats
      {fmin = Nanoseconds 0
      , fmax = Nanoseconds 0
      , fcount = 1
      , fsum_A = 0
      , fsum_B = 0}
  }
```

```

in
  AggregatedStats stats

instance Show Aggregated where
  show (AggregatedStats astats) =
    "{ stats = " ++ show astats ++ " }"
  show (AggregatedEWMA a) = show a

```

1.7.10 Cardano.BM.Data.AggregatedKind

AggregatedKind

This identifies the type of Aggregated.

```

data AggregatedKind = StatsAK
  | EwmaAK {alpha :: Double}
  deriving (Generic, Eq, Show, FromJSON, ToJSON, Read)

```

1.7.11 Cardano.BM.Data.Backend

Accepts a LogObject

Instances of this type class accept a LogObject and deal with it.

```

class IsEffectuator t a where
  effectuate :: t a → LogObject a → IO ()
  effectuatefrom :: forall s o (IsEffectuator s a) ⇒ t a → LogObject a → s a → IO ()
  default effectuatefrom :: forall s o (IsEffectuator s a) ⇒ t a → LogObject a → s a → IO ()
  effectuatefrom t nli _ = effectuate t nli
  handleOverflow :: t a → IO ()

```

Declaration of a Backend

A backend is life-cycle managed, thus can be *realized* and *unrealized*.

```

class (IsEffectuator t a, FromJSON a) ⇒ IsBackend t a where
  typeOf :: t a → BackendKind
  realize :: Configuration → IO (t a)
  realizefrom :: forall s o (IsEffectuator s a) ⇒ Configuration → Trace IO a → s a → IO (t a)
  default realizefrom :: forall s o (IsEffectuator s a) ⇒ Configuration → Trace IO a → s a → IO (t a)
  realizefrom cfg _ = realize cfg
  unrealize :: t a → IO ()

```

Backend

This data structure for a backend defines its behaviour as an IsEffectuator when processing an incoming message, and as an IsBackend for unrealizing the backend.

```

data Backend a = MkBackend
  { bEffectuate :: LogObject a → IO ()
  , bUnrealize :: IO ()
  }

```

1.7.12 Cardano.BM.Data.BackendKind

BackendKind

This identifies the backends that can be attached to the **Switchboard**.

```

data BackendKind =
  AggregationBK
  | EditorBK
  | EKGViewBK
  | GraylogBK
  | KatipBK
  | LogBufferBK
  | MonitoringBK
  | TraceAcceptorBK FilePath
  | TraceForwarderBK
  | UserDefinedBK Text
  | SwitchboardBK
deriving (Eq, Ord, Show, Read)

instance ToJSON BackendKind where
  toJSON AggregationBK      = String "AggregationBK"
  toJSON EditorBK           = String "EditorBK"
  toJSON EKGViewBK          = String "EKGViewBK"
  toJSON GraylogBK          = String "GraylogBK"
  toJSON KatipBK            = String "KatipBK"
  toJSON LogBufferBK        = String "LogBufferBK"
  toJSON MonitoringBK       = String "MonitoringBK"
  toJSON TraceForwarderBK   = String "TraceForwarderBK"
  toJSON (TraceAcceptorBK file) = object [ "kind" . = String "TraceAcceptorBK"
                                           , "path" . = toJSON file
                                           ]
  toJSON (UserDefinedBK name) = object [ "kind" . = String "UserDefinedBK"
                                           , "name" . = toJSON name
                                           ]
  toJSON SwitchboardBK      = String "SwitchboardBK"

instance FromJSON BackendKind where
  parseJSON v = withObject
    "BackendKind"
    (\value → do
      c ← value .: "kind" :: Parser Text
      case c of
        "UserDefinedBK" →
          UserDefinedBK <$> value .: "name"
        "TraceAcceptorBK" →
          TraceAcceptorBK <$> value .: "path"
        _ → fail "not expected kind"
    )
    v
  <|> withText
    "BackendKind"
    (\case
      "AggregationBK" → pure AggregationBK

```

```

"EditorBK"           → pure EditorBK
"EKGViewBK"          → pure EKGViewBK
"GraylogBK"          → pure GraylogBK
"KatipBK"            → pure KatipBK
"LogBufferBK"        → pure LogBufferBK
"MonitoringBK"       → pure MonitoringBK
"TraceForwarderBK" → pure TraceForwarderBK
"SwitchboardBK"      → pure SwitchboardBK
-                    → fail "not expected BackendKind"
)
v

```

1.7.13 Cardano.BM.Data.Configuration

Data structure to help parsing configuration files.

Representation

```

type Port = Int
data Representation = Representation
  { minSeverity    :: Severity
  , rotation       :: Maybe RotationParameters
  , setupScribes   :: [ScribeDefinition]
  , defaultScribes :: [(ScribeKind, Text)]
  , setupBackends  :: [BackendKind]
  , defaultBackends :: [BackendKind]
  , hasEKG         :: Maybe Port
  , hasGraylog     :: Maybe Port
  , hasPrometheus  :: Maybe Port
  , hasGUI         :: Maybe Port
  , logOutput      :: Maybe FilePath
  , options        :: HM.HashMap Text Object
  }
deriving (Generic, Show, ToJSON, FromJSON)

```

parseRepresentation

```

parseRepresentation :: FilePath → IO Representation
parseRepresentation fp = do
  repr :: Representation ← decodeFileThrow fp
  return $ implicit_fill_representation repr

```

after parsing the configuration representation we implicitly correct it.

```

implicit_fill_representation :: Representation → Representation
implicit_fill_representation =
  remove_ekgview_if_not_defined ∘
  filter_duplicates_from_backends ∘
  filter_duplicates_from_scribes ∘
  union_setup_and_usage_backends ∘

```

```

    add_ekgview_if_port_defined ◦
    add_katip_if_any_scribes
  where
    filter_duplicates_from_backends r =
      r {setupBackends = mkUniq $ setupBackends r}
    filter_duplicates_from_scribes r =
      r {setupScribes = mkUniq $ setupScribes r}
    union_setup_and_usage_backends r =
      r {setupBackends = setupBackends r <> defaultBackends r}
  # ifdef ENABLE_EKG
    remove_ekgview_if_not_defined r =
      case hasEKG r of
        Nothing → r {defaultBackends = filter (λbk → bk ≠ EKGViewBK) (defaultBackends r)
                     ,setupBackends = filter (λbk → bk ≠ EKGViewBK) (setupBackends r)
                     }
        Just _ → r
    add_ekgview_if_port_defined r =
      case hasEKG r of
        Nothing → r
        Just _ → r {setupBackends = setupBackends r <> [EKGViewBK]}
  # else
    remove_ekgview_if_not_defined = id
    add_ekgview_if_port_defined = id
  # endif
    add_katip_if_any_scribes r =
      if (any ¬ [null $ setupScribes r, null $ defaultScribes r])
      then r {setupBackends = setupBackends r <> [KatipBK]}
      else r
    mkUniq :: Ord a ⇒ [a] → [a]
    mkUniq = Set.toList ◦ Set.fromList

```

1.7.14 Cardano.BM.Data.Counter

Counter

```

data Counter = Counter
  { cType :: CounterType
  , cName :: Text
  , cValue :: Measurable
  }
  deriving (Show, Eq, Generic, ToJSON, FromJSON)

data CounterType = MonotonicClockTime
  | MemoryCounter
  | StatInfo
  | IOCounter
  | NetCounter
  | RTStats
  deriving (Eq, Show, Generic, ToJSON, FromJSON)

instance ToJSON Microsecond where
  toJSON = toJSON ◦ toMicroseconds
  toEncoding = toEncoding ◦ toMicroseconds

```

Names of counters

```

nameCounter :: Counter → Text
nameCounter (Counter MonotonicClockTime _ _) = "Clock"
nameCounter (Counter MemoryCounter _ _) = "Mem"
nameCounter (Counter StatInfo _ _) = "Stat"
nameCounter (Counter IOCounter _ _) = "IO"
nameCounter (Counter NetCounter _ _) = "Net"
nameCounter (Counter RTSStats _ _) = "RTS"

```

CounterState

```

data CounterState = CounterState {
  csCounters :: [Counter]
}
deriving (Show, Eq, Generic, ToJSON, FromJSON)

```

Difference between counters

```

diffCounters :: [Counter] → [Counter] → [Counter]
diffCounters openings closings =
  getCountersDiff openings closings
where
  getCountersDiff :: [Counter]
    → [Counter]
    → [Counter]
  getCountersDiff as bs =
    let
      getName counter = nameCounter counter <> cName counter
      asNames = map getName as
      aPairs = zip asNames as
      bsNames = map getName bs
      bs' = zip bsNames bs
      bPairs = HM.fromList bs'
    in
      catMaybes $ (flip map) aPairs $ \ (name, Counter _ _ startValue) →
        case HM.lookup name bPairs of
          Nothing → Nothing
          Just counter → let endValue = cValue counter
                        in Just counter {cValue = endValue - startValue}

```

1.7.15 Cardano.BM.Data.LogItem

LoggerName

A **LoggerName** has currently type *Text*.

```

type LoggerName = Text

```


Logging of outcomes with `LogObject`

```

data LogObject a = LogObject
  { loName :: LoggerName
  , loMeta :: !LOMeta
  , loContent :: (LOContent a)
  } deriving (Show, Eq)

instance ToJSON a => ToJSON (LogObject a) where
  toJSON (LogObject _loname _lometa _locontent) =
    object [ "loname" . = _loname
            , "lometa"   . = _lometa
            , "locontent" . = _locontent
            ]

instance (FromJSON a) => FromJSON (LogObject a) where
  parseJSON j = withObject "LogObject"
    (\v -> LogObject <$> v . : "loname"
      <*> v . : "lometa"
      <*> v . : "locontent")
    j

```

Meta data for a `LogObject`. Text was selected over `ThreadId` in order to be able to use the logging system under SimM of our `ouroboros-network` because `ThreadId` from `Control.Concurrent` lacks a `Read` instance.

```

data LOMeta = LOMeta {
  tstamp :: {-# UNPACK #-} !UTCTime
  , tid   :: {-# UNPACK #-} !Text
  , severity :: !Severity
  , privacy :: !PrivacyAnnotation
  } deriving (Show, Eq)

instance ToJSON LOMeta where
  toJSON (LOMeta _tstamp _tid _sev _priv) =
    object [ "tstamp" . = _tstamp
            , "tid"     . = _tid
            , "severity" . = show _sev
            , "privacy"  . = show _priv
            ]

instance FromJSON LOMeta where
  parseJSON j = withObject "LOMeta"
    (\v -> LOMeta <$> v . : "tstamp"
      <*> v . : "tid"
      <*> v . : "severity"
      <*> v . : "privacy")
    j

mkLOMeta :: MonadIO m => Severity -> PrivacyAnnotation -> m LOMeta
mkLOMeta sev priv =
  LOMeta <$> (liftIO getCurrentTime)
    <*> (pack o show <$> (liftIO myThreadId))
    <*> pure sev
    <*> pure priv

```

Convert a timestamp to ns since epoch:

```
utc2ns :: UTCTime → Word64
```

```
utc2ns utctime = fromInteger $ round $ 1000 * 1000 * 1000 * (utcTimeToPOSIXSeconds utctime)
```

```
data MonitorAction = MonitorAlert Text
```

```
  | MonitorAlterGlobalSeverity Severity
```

```
  | MonitorAlterSeverity LoggerName Severity
```

```
  deriving (Show, Eq)
```

```
instance ToJSON MonitorAction where
```

```
  toJSON (MonitorAlert m) =
```

```
    object [ "kind" . = String "MonitorAlert"
            , "message" . = toJSON m ]
```

```
  toJSON (MonitorAlterGlobalSeverity s) =
```

```
    object [ "kind" . = String "MonitorAlterGlobalSeverity"
            , "severity" . = toJSON s ]
```

```
  toJSON (MonitorAlterSeverity n s) =
```

```
    object [ "kind" . = String "MonitorAlterSeverity"
            , "name" . = toJSON n
            , "severity" . = toJSON s ]
```

```
instance FromJSON MonitorAction where
```

```
  parseJSON j = withObject "MonitorAction"
```

```
    (λv → (v . "kind" :: Parser Text)
```

```
    >>=
```

```
    λcase "MonitorAlert" →
```

```
      MonitorAlert <$> v . "message"
```

```
    "MonitorAlterGlobalSeverity" →
```

```
      MonitorAlterGlobalSeverity <$> v . "severity"
```

```
    "MonitorAlterSeverity" →
```

```
      MonitorAlterSeverity <$> v . "name" <*> v . "severity"
```

```
    _ → fail "unknown MonitorAction")
```

```
  j
```

LogStructured could also be:

```
forall b ⇒ (ToJSON b) ⇒ LogStructured b
```

Payload of a **LogObject**:

```
data LOContent a = LogMessage a
```

```
  | LogError Text
```

```
  | LogValue Text Measurable
```

```
  | LogStructured BS.ByteString
```

```
  | ObserveOpen CounterState
```

```
  | ObserveDiff CounterState
```

```
  | ObserveClose CounterState
```

```
  | AggregatedMessage [(Text, Aggregated)]
```

```
  | MonitoringEffect MonitorAction
```

```
  | Command CommandValue
```

```
  | KillPill
```

```
  deriving (Show, Eq)
```

```
instance ToJSON a ⇒ ToJSON (LOContent a) where
```

```
  toJSON (LogMessage m) =
```

```
    object [ "kind" . = String "LogMessage"
```

```

    , "message" . = toJSON m]
toJSON (LogError m) =
  object [ "kind" . = String "LogError"
    , "message" . = toJSON m]
toJSON (LogValue n v) =
  object [ "kind" . = String "LogValue"
    , "name" . = toJSON n
    , "value" . = toJSON v]
toJSON (LogStructured m) =
  object [ "kind" . = String "LogStructured"
    , "message" . = (toJSON $ decodeUtf8 $ BS64.encode m)]
toJSON (ObserveOpen c) =
  object [ "kind" . = String "ObserveOpen"
    , "counters" . = toJSON c]
toJSON (ObserveDiff c) =
  object [ "kind" . = String "ObserveDiff"
    , "counters" . = toJSON c]
toJSON (ObserveClose c) =
  object [ "kind" . = String "ObserveClose"
    , "counters" . = toJSON c]
toJSON (AggregatedMessage ps) =
  object [ "kind" . = String "AggregatedMessage"
    , "pairs" . = toJSON ps]
toJSON (MonitoringEffect a) =
  object [ "kind" . = String "MonitoringEffect"
    , "action" . = toJSON a]
toJSON (Command c) =
  object [ "kind" . = String "Command"
    , "command" . = toJSON c]
toJSON KillPill =
  String "KillPill"
instance (FromJSON a) => FromJSON (LOContent a) where
  parseJSON j = withObject "LOContent"
    (\v -> (v .: "kind" :: Parser Text)
      >>=
        λcase "LogMessage" -> LogMessage <$> v .: "message"
          "LogError" -> LogError <$> v .: "message"
          "LogValue" -> LogValue <$> v .: "name" <*> v .: "value"
          "LogStructured" -> LogStructured <$>
            BS64.decodeLenient <$>
              encodeUtf8 <$>
                (v .: "message" :: Parser LT.Text)
          "ObserveOpen" -> ObserveOpen <$> v .: "counters"
          "ObserveDiff" -> ObserveDiff <$> v .: "counters"
          "ObserveClose" -> ObserveClose <$> v .: "counters"
          "AggregatedMessage" -> AggregatedMessage <$> v .: "pairs"
          "MonitoringEffect" -> MonitoringEffect <$> v .: "action"
          "Command" -> Command <$> v .: "command"
          _ -> fail "unknown LOContent")
    j
  <|>

```

```

    withText "LOContent"
    (λcase "KillPill" → pure KillPill
     _ → fail "unknown LOContent (String)")
  j
loType :: LogObject a → Text
loType (LogObject _ _ content) = loType2Name content

```

Name of a message content type

```

loType2Name :: LOContent a → Text
loType2Name = λcase
  LogMessage _      → "LogMessage"
  LogError _        → "LogError"
  LogValue _ _      → "LogValue"
  LogStructured _   → "LogStructured"
  ObserveOpen _     → "ObserveOpen"
  ObserveDiff _     → "ObserveDiff"
  ObserveClose _    → "ObserveClose"
  AggregatedMessage _ → "AggregatedMessage"
  MonitoringEffect _ → "MonitoringEffect"
  Command _         → "Command"
  KillPill          → "KillPill"

```

Backends can enter commands to the trace. Commands will end up in the **Switchboard**, which will interpret them and take action.

```

data CommandValue = DumpBufferedTo BackendKind
  deriving (Show, Eq)
instance ToJSON CommandValue where
  toJSON (DumpBufferedTo be) =
    object [ "kind" . = String "DumpBufferedTo"
            , "backend" . = toJSON be ]
instance FromJSON CommandValue where
  parseJSON v = withObject "CommandValue"
    (λa → (a .: "kind" :: Parser Text)
      >>=
        λcase "DumpBufferedTo" → DumpBufferedTo <$> a .: "backend"
          _ → fail "unknown CommandValue")
  v

```

Privacy annotation

```

data PrivacyAnnotation =
  Confidential -- confidential information - handle with care
  | Public -- indifferent - can be public.
  deriving (Show, Eq)
instance FromJSON PrivacyAnnotation where
  parseJSON v = withText "PrivacyAnnotation"
    (λcase "Confidential" → pure Confidential
      "Public" → pure Public)

```

```

    _ → fail "unknown PrivacyAnnotation")
  v

```

Data structure for annotating the severity and privacy of an object.

```

data PrivacyAndSeverityAnnotated a
  = PSA {psaSeverity :: !Severity
        ,psaPrivacy :: !PrivacyAnnotation
        ,psaPayload :: a
        }
deriving (Show)

```

Mapping Log Objects

This provides a helper function to transform log items. It would often be used with *contramap*.

```

mapLogObject :: (a → b) → LogObject a → LogObject b
mapLogObject f (LogObject nm me loc) = LogObject nm me (mapLOContent f loc)
instance Functor LogObject where
  fmap = mapLogObject
mapLOContent :: (a → b) → LOContent a → LOContent b
mapLOContent f = λcase
  LogMessage msg → LogMessage (f msg)
  LogError a     → LogError a
  LogStructured m → LogStructured m
  LogValue n v   → LogValue n v
  ObserveOpen st → ObserveOpen st
  ObserveDiff st → ObserveDiff st
  ObserveClose st → ObserveClose st
  AggregatedMessage ag → AggregatedMessage ag
  MonitoringEffect act → MonitoringEffect act
  Command v           → Command v
  KillPill            → KillPill

```

1.7.16 Cardano.BM.Data.Observable

ObservableInstance

```

data ObservableInstance = MonotonicClock
  | MemoryStats
  | ProcessStats
  | NetStats
  | IOStats
  | GhcRtsStats
  deriving (Generic, Eq, Show, FromJSON, ToJSON, Read)

```

1.7.17 Cardano.BM.Data.Output

ScribeKind

This identifies katip's scribes by type.

```

data ScribeKind = FileSK
  | StdoutSK
  | StderrSK
# ifdef ENABLE_SYSTEMD
  | JournalSK
# endif
  | DevNullSK
deriving (Generic, Eq, Ord, Show, Read, FromJSON, ToJSON)

```

ScribeFormat

This defines the scribe's output format.

```

data ScribeFormat = ScText
  | ScJson
deriving (Generic, Eq, Ord, Show, Read, FromJSON, ToJSON)

```

ScribeId

A scribe is identified by **ScribeKind** *x* *Filename*

```

type ScribeId = Text -- (ScribeKind : Filename)

```

ScribePrivacy

This declares if a scribe will be public (and must not contain sensitive data) or private.

```

data ScribePrivacy = ScPublic | ScPrivate
deriving (Generic, Eq, Ord, Show, FromJSON, ToJSON)

```

ScribeDefinition

This identifies katip's scribes by type.

```

data ScribeDefinition = ScribeDefinition
  { scKind    :: ScribeKind
  , scFormat  :: ScribeFormat
  , scName    :: Text
  , scPrivacy :: ScribePrivacy
  , scRotation :: Maybe RotationParameters
  }
deriving (Generic, Eq, Ord, Show, ToJSON)
instance FromJSON ScribeDefinition where
  parseJSON (Object o) = do
    kind      ← o .: "scKind"
    name      ← o .: "scName"
    mayFormat ← o .:? "scFormat"
    mayPrivacy ← o .:? "scPrivacy"
    rotation  ← o .:? "scRotation"
    return $ ScribeDefinition

```

```

    {scKind    = kind
    ,scName    = name
    ,scFormat  = fromMaybe ScJson mayFormat
    ,scPrivacy = fromMaybe ScPublic mayPrivacy
    ,scRotation = rotation
    }
  parseJSON invalid = typeMismatch "ScribeDefinition" invalid

```

1.7.18 Cardano.BM.Data.Rotation

RotationParameters

```

data RotationParameters = RotationParameters
  {rpLogLimitBytes :: !Word64 -- max size of file in bytes
  ,rpMaxAgeHours   :: !Word   -- hours
  ,rpKeepFilesNum  :: !Word   -- number of files to keep
  } deriving (Generic, Show, Eq, Ord, FromJSON, ToJSON)

```

1.7.19 Cardano.BM.Data.Severity

Severity

The intended meaning of severity codes:

Debug *detailed information about values and decision flow* Info general information of events; progressing properly Notice *needs attention; something – progressing properly* Warning may continue into an error condition if continued Error *unexpected set of event or condition occurred* Critical error condition causing degrade of operation Alert *a subsystem is no longer operating correctly, likely requires man* at this point, the system can never progress without additional intervention

We were informed by the Syslog taxonomy: https://en.wikipedia.org/wiki/Syslog#Severity_level

```

data Severity = Debug
  | Info
  | Notice
  | Warning
  | Error
  | Critical
  | Alert
  | Emergency
  deriving (Show, Eq, Ord, Bounded, Enum, Generic, ToJSON, Read)

instance FromJSON Severity where
  parseJSON = withText "severity" $ \case
    "Debug"    → pure Debug
    "Info"     → pure Info
    "Notice"   → pure Notice
    "Warning"  → pure Warning
    "Error"    → pure Error
    "Critical" → pure Critical
    "Alert"    → pure Alert
    "Emergency" → pure Emergency
    _          → pure Info -- catch all

```

1.7.20 Cardano.BM.Data.SubTrace

SubTrace

```

data NameSelector = Exact Text | StartsWith Text | EndsWith Text | Contains Text
    deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)
data DropName     = Drop NameSelector
    deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)
data UnhideNames = Unhide [NameSelector]
    deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)

data SubTrace = Neutral
    | UntimedTrace
    | NoTrace
    | TeeTrace LoggerName
    | FilterTrace [(DropName, UnhideNames)]
    | DropOpening
    | ObservableTraceSelf [ObservableInstance]
    | ObservableTrace ProcessID [ObservableInstance]
    | SetSeverity Severity
    deriving (Generic, Show, Read, Eq)

#ifdef POSIX
instance ToJSON ProcessID where
    toJSON (CPid pid) = Number $ fromIntegral pid
instance FromJSON ProcessID where
    parseJSON v = CPid < $ > parseJSON v
# else
-- Wrap the Win32 DWORD type alias so that it can be logged
newtype ProcessID = ProcessID ProcessId
    deriving (Generic, Show, Read, Eq)
instance ToJSON ProcessID where
    toJSON (ProcessID pid) = Number $ fromIntegral pid
instance FromJSON ProcessID where
    parseJSON v = ProcessID < $ > parseJSON v
# endif

instance FromJSON SubTrace where
    parseJSON = withObject "SubTrace" $ \o → do
        subtrace :: Text ← o .: "subtrace"
        case subtrace of
            "Neutral"          → return $ Neutral
            "UntimedTrace"     → return $ UntimedTrace
            "NoTrace"          → return $ NoTrace
            "TeeTrace"         → TeeTrace      < $ > o .: "contents"
            "FilterTrace"      → FilterTrace   < $ > o .: "contents"
            "DropOpening"      → return $ DropOpening
            "ObservableTraceSelf" → ObservableTraceSelf < $ > o .: "contents"
            "ObservableTrace"  → ObservableTrace < $ > o .: "pid"
                                < * > o .: "contents"
            "SetSeverity"      → SetSeverity   < $ > o .: "contents"
            other              → fail $ "unexpected subtrace: " ++ (unpack other)

instance ToJSON SubTrace where

```



```

toJSON Neutral      = object [ "subtrace" . = String "Neutral" ]
toJSON UntimedTrace = object [ "subtrace" . = String "UntimedTrace" ]
toJSON NoTrace      = object [ "subtrace" . = String "NoTrace" ]
toJSON (TeeTrace name) = object [ "subtrace" . = String "TeeTrace" , "contents" . = t
toJSON (FilterTrace dus) = object [ "subtrace" . = String "FilterTrace" , "contents" . = t
toJSON DropOpening    = object [ "subtrace" . = String "DropOpening" ]
toJSON (ObservableTraceSelf os) = object [ "subtrace" . = String "ObservableTraceSelf", "contents"
toJSON (ObservableTrace pid os) = object [ "subtrace" . = String "ObservableTrace", "pid" . = toJSON
    , "contents" . = toJSON os ]
toJSON (SetSeverity sev) = object [ "subtrace" . = String "SetSeverity" , "contents" . = t

```

1.7.21 Cardano.BM.Data.Trace

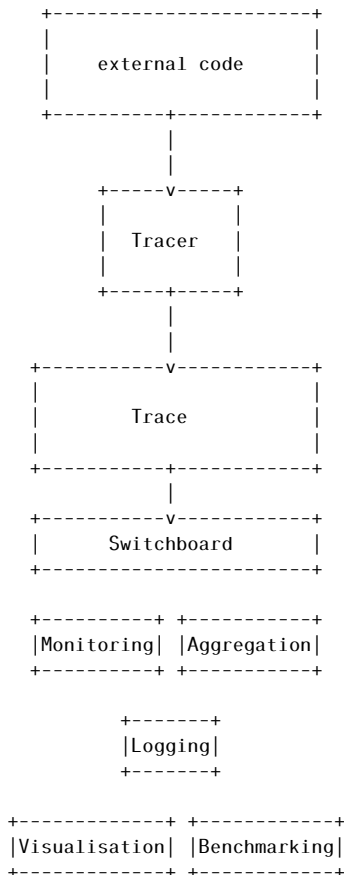
Trace

A **Trace** *m a* is a **Tracer** *m* (**LogObject** *a*).

```
type Trace m a = Tracer m (LogObject a)
```

1.7.22 Cardano.BM.Data.Tracer

This module extends the basic **Tracer** with one that keeps a list of context names to create the basis for **Trace** which accepts messages from a Tracer and ends in the **Switchboard** for further processing of the messages.



ToLogObject - transforms a logged item to LogObject

The function `toLogObject` can be specialized for various environments

```
class Monad m => ToLogObject m where
  toLogObject :: ToObject a => Tracer m (LogObject a) -> Tracer m a
instance ToLogObject IO where
  toLogObject :: ToObject a => Tracer IO (LogObject a) -> Tracer IO a
  toLogObject tr = Tracer $ \a -> do
    lo <- LogObject <$> pure ""
    <*> (mkLOMeta Debug Public)
    <*> pure (LogMessage a)
    traceWith tr lo
```

To be placed in `ouroboros-network` ○

```
instance (MonadFork m, MonadTimer m) => ToLogObject m where
  toLogObject tr = Tracer $ \a -> do
    lo <- LogObject <$> pure ""
    <*> (LOMeta <$> getMonotonicTime -- must be evaluated at the calling site
    <*> (pack ○ show <$> myThreadId)
    <*> pure Debug
    <*> pure Public)
    <*> pure (LogMessage a)
    traceWith tr lo
```

ToObject - transforms a logged item to JSON

Katip requires JSON objects to be logged as context. This typeclass provides a default instance which uses `ToJSON` and produces an empty object if 'toJSON' results in any type other than `Object`. If you have a type you want to log that produces an `Array` or `Number` for example, you'll want to write an explicit instance here. You can trivially add a `ToObject` instance for something with a `ToJSON` instance like:

```
instance ToObject Foo

class ToJSON a => ToObject a where
  toObject :: a -> Object
  default toObject :: a -> Object
  toObject v = case toJSON v of
    Object o -> o
    s@(String _) -> HM.singleton "string" s
    _ -> mempty
instance ToObject () where
  toObject _ = mempty
instance ToObject String
instance ToObject Text
instance ToJSON a => ToObject (LogObject a)
instance ToJSON a => ToObject (LOContent a)
```

1.7.23 Cardano.BM.Configuration

see **Cardano.BM.Configuration.Model** for the implementation.

```
getOptionOrDefault :: CM.Configuration → Text → Text → IO Text
getOptionOrDefault cg name def = do
  opt ← CM.getOption cg name
  case opt of
    Nothing → return def
    Just o → return o
```

Test severities

Test severity of the given **LOMeta** to be greater or equal to those of the specific **LoggerName**.

```
testSeverity :: CM.Configuration → LoggerName → LOMeta → IO Bool
testSeverity config loggename meta = do
  globminsev ← CM.minSeverity config
  globnamesev ← CM.inspectSeverity config loggename
  let minsev = max globminsev $ fromMaybe Debug globnamesev
  return $ (severity meta) ≥ minsev
```

1.7.24 Cardano.BM.Configuration.Model

Configuration.Model

<<Model>>	
Configuration	
cgMinSeverity	: Severity
cgMapSeverity	: Map = LoggerName -> Severity
cgMapSubtrace	: Map = LoggerName -> SubTrace
cgOptions	: Map = Text -> Aeson.Object
cgMapBackend	: Map = LoggerName -> [BackendKind]
cgDefBackends	: BackendKind [*]
cgSetupBackends	: BackendKind [*]
cgMapScribe	: Map = LoggerName -> [Scribeld]
cgDefScribes	: Scribeld [*]
cgSetupScribes	: ScribeDefinition [*]
cbMapAggregatedKind	: Map = LoggerName -> AggregatedKind
cgDefAggregatedKind	: AggregatedKind
cgPortEKG	: Int
cgPortGUI	: Int

Figure 1.5: Configuration model

```
type ConfigurationMVar = MVar ConfigurationInternal
newtype Configuration = Configuration
  { getCG :: ConfigurationMVar }
-- Our internal state; see -"Configuration model"-
data ConfigurationInternal = ConfigurationInternal
  { cgMinSeverity      :: Severity
    -- minimum severity level of every object that will be output
  , cgDefRotation      :: Maybe RotationParameters
```

```

-- default rotation parameters
,cgMapSeverity    :: HM.HashMap LoggerName Severity
-- severity filter per loggername
,cgMapSubtrace    :: HM.HashMap LoggerName SubTrace
-- type of trace per loggername
,cgOptions        :: HM.HashMap Text Object
-- options needed for tracing, logging and monitoring
,cgMapBackend     :: HM.HashMap LoggerName [BackendKind]
-- backends that will be used for the specific loggername
,cgDefBackendKs   :: [BackendKind]
-- backends that will be used if a set of backends for the
-- specific loggername is not set
,cgSetupBackends  :: [BackendKind]
-- backends to setup; every backend to be used must have
-- been declared here
,cgMapScribe      :: HM.HashMap LoggerName [ScribeId]
-- katip scribes that will be used for the specific loggername
,cgMapScribeCache :: HM.HashMap LoggerName [ScribeId]
-- map to cache info of the cgMapScribe
,cgDefScribes     :: [ScribeId]
-- katip scribes that will be used if a set of scribes for the
-- specific loggername is not set
,cgSetupScribes   :: [ScribeDefinition]
-- katip scribes to setup; every scribe to be used must have
-- been declared here
,cgMapAggregatedKind :: HM.HashMap LoggerName AggregatedKind
-- kind of Aggregated that will be used for the specific loggername
,cgDefAggregatedKind :: AggregatedKind
-- kind of Aggregated that will be used if a set of scribes for the
-- specific loggername is not set
,cgMonitors       :: HM.HashMap LoggerName (MEvPreCond, MEvExpr, [MEvAction])
,cgPortEKG        :: Int
-- port for EKG server
,cgPortGraylog    :: Int
-- port to Graylog server
,cgPortPrometheus :: Int
-- port for Prometheus server
,cgPortGUI        :: Int
-- port for changes at runtime
,cgLogOutput      :: Maybe FilePath
-- filepath of pipe or file for forwarding of log objects
} deriving (Show, Eq)

```

Backends configured in the Switchboard

For a given context name return the list of backends configured, or, in case no such configuration exists, return the default backends.

```

getBackends :: Configuration → LoggerName → IO [BackendKind]
getBackends configuration name = do
  cg ← readMVar $ getCG configuration

```

```

-- let outs = HM.lookup name (cgMapBackend cg)
-- case outs of
-- Nothing -> return (cgDefBackendKs cg)
-- Just os -> return os
let defs = cgDefBackendKs cg
let mapbks = cgMapBackend cg
let find_s [] = defs
    find_s lnames = case HM.lookup (T.intercalate "." lnames) mapbks of
        Nothing -> find_s (init lnames)
        Just os -> os
return $ find_s $ T.split (≡ ' . ') name

getDefaultBackends :: Configuration → IO [BackendKind]
getDefaultBackends configuration =
    cgDefBackendKs < $ > (readMVar $ getCG configuration)

setDefaultBackends :: Configuration → [BackendKind] → IO ()
setDefaultBackends configuration bes =
    modifyMVar_ (getCG configuration) $ \cg →
        return cg {cgDefBackendKs = bes}

setBackends :: Configuration → LoggerName → Maybe [BackendKind] → IO ()
setBackends configuration name be =
    modifyMVar_ (getCG configuration) $ \cg →
        return cg {cgMapBackend = HM.alter (\_ → be) name (cgMapBackend cg)}

```

Backends to be setup by the Switchboard

Defines the list of **Backends** that need to be setup by the **Switchboard**.

```

setSetupBackends :: Configuration → [BackendKind] → IO ()
setSetupBackends configuration bes =
    modifyMVar_ (getCG configuration) $ \cg →
        return cg {cgSetupBackends = bes}

getSetupBackends :: Configuration → IO [BackendKind]
getSetupBackends configuration =
    cgSetupBackends < $ > (readMVar $ getCG configuration)

```

Scribes configured in the Log backend

For a given context name return the list of scribes to output to, or, in case no such configuration exists, return the default scribes to use.

```

getScribes :: Configuration → LoggerName → IO [ScribeId]
getScribes configuration name = do
    cg ← readMVar (getCG configuration)
    (updateCache, scribes) ← do
        let defs = cgDefScribes cg
        let mapscribes = cgMapScribe cg
        let find_s [] = defs
            find_s lnames = case HM.lookup (T.intercalate "." lnames) mapscribes of
                Nothing -> find_s (init lnames)
                Just os -> os

```

```

let outs = HM.lookup name (cgMapScribeCache cg)
-- look if scribes are already cached
return $ case outs of
  -- if no cached scribes found; search the appropriate scribes that
  -- they must inherit and update the cached map
  Nothing → (True, find_s $ T.split (≡ ' . ') name)
  Just os → (False, os)

when updateCache $ setCachedScribes configuration name $ Just scribes
return scribes

getCachedScribes :: Configuration → LoggerName → IO (Maybe [ScribeId])
getCachedScribes configuration name = do
  cg ← readMVar $ getCG configuration
  return $ HM.lookup name $ cgMapScribeCache cg

setScribes :: Configuration → LoggerName → Maybe [ScribeId] → IO ()
setScribes configuration name scribes =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgMapScribe = HM.alter (\_ → scribes) name (cgMapScribe cg)}

setCachedScribes :: Configuration → LoggerName → Maybe [ScribeId] → IO ()
setCachedScribes configuration name scribes =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgMapScribeCache = HM.alter (\_ → scribes) name (cgMapScribeCache cg)}

setDefaultScribes :: Configuration → [ScribeId] → IO ()
setDefaultScribes configuration scs =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgDefScribes = scs}

```

Scribes to be setup in the **Log** backend

Defines the list of Scribes that need to be setup in the **Log** backend.

```

setSetupScribes :: Configuration → [ScribeDefinition] → IO ()
setSetupScribes configuration sds =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgSetupScribes = sds}

getSetupScribes :: Configuration → IO [ScribeDefinition]
getSetupScribes configuration =
  cgSetupScribes < $ > readMVar (getCG configuration)

```

AggregatedKind to define the type of measurement

For a given context name return its **AggregatedKind** or in case no such configuration exists, return the default **AggregatedKind** to use.

```

getAggregatedKind :: Configuration → LoggerName → IO AggregatedKind
getAggregatedKind configuration name = do
  cg ← readMVar $ getCG configuration
  let outs = HM.lookup name (cgMapAggregatedKind cg)
  case outs of
    Nothing → return $ cgDefAggregatedKind cg
    Just os → return $ os

```

```

setDefaultAggregatedKind :: Configuration → AggregatedKind → IO ()
setDefaultAggregatedKind configuration defAK =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgDefAggregatedKind = defAK}
setAggregatedKind :: Configuration → LoggerName → Maybe AggregatedKind → IO ()
setAggregatedKind configuration name ak =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgMapAggregatedKind = HM.alter (\_ → ak) name (cgMapAggregatedKind cg)}

```

Access port numbers of EKG, Prometheus, GUI

```

getEKGport :: Configuration → IO Int
getEKGport configuration =
  cgPortEKG < $ > (readMVar $ getCG configuration)
setEKGport :: Configuration → Int → IO ()
setEKGport configuration port =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgPortEKG = port}
getGraylogPort :: Configuration → IO Int
getGraylogPort configuration =
  cgPortGraylog < $ > (readMVar $ getCG configuration)
setGraylogPort :: Configuration → Int → IO ()
setGraylogPort configuration port =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgPortGraylog = port}
getPrometheusPort :: Configuration → IO Int
getPrometheusPort configuration =
  cgPortPrometheus < $ > (readMVar $ getCG configuration)
setPrometheusPort :: Configuration → Int → IO ()
setPrometheusPort configuration port =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgPortPrometheus = port}
getGUIport :: Configuration → IO Int
getGUIport configuration =
  cgPortGUI < $ > (readMVar $ getCG configuration)
setGUIport :: Configuration → Int → IO ()
setGUIport configuration port =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgPortGUI = port}

```

Access port numbers of EKG, Prometheus, GUI

```

getLogOutput :: Configuration → IO (Maybe FilePath)
getLogOutput configuration =
  cgLogOutput < $ > (readMVar $ getCG configuration)
setLogOutput :: Configuration → FilePath → IO ()
setLogOutput configuration path =

```



```
modifyMVar_ (getCG configuration) $ \cg →
  return cg {cgLogOutput = Just path}
```

Options

```
getOption :: Configuration → Text → IO (Maybe Text)
getOption configuration name = do
  cg ← readMVar $ getCG configuration
  case HM.lookup name (cgOptions cg) of
    Nothing → return Nothing
    Just o → return $ Just $ pack $ show o
```

Global setting of minimum severity

```
minSeverity :: Configuration → IO Severity
minSeverity configuration =
  cgMinSeverity < $ > (readMVar $ getCG configuration)
setMinSeverity :: Configuration → Severity → IO ()
setMinSeverity configuration sev =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgMinSeverity = sev}
```

Relation of context name to minimum severity

```
inspectSeverity :: Configuration → Text → IO (Maybe Severity)
inspectSeverity configuration name = do
  cg ← readMVar $ getCG configuration
  return $ HM.lookup name (cgMapSeverity cg)
setSeverity :: Configuration → Text → Maybe Severity → IO ()
setSeverity configuration name sev =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgMapSeverity = HM.alter (\_ → sev) name (cgMapSeverity cg)}
```

Relation of context name to SubTrace

A new context may contain a different type of **Trace**. The function **appendName** will look up the **SubTrace** for the context's name.

```
findSubTrace :: Configuration → Text → IO (Maybe SubTrace)
findSubTrace configuration name =
  HM.lookup name < $ > cgMapSubtrace < $ > (readMVar $ getCG configuration)
setSubTrace :: Configuration → Text → Maybe SubTrace → IO ()
setSubTrace configuration name trafo =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgMapSubtrace = HM.alter (\_ → trafo) name (cgMapSubtrace cg)}
```


Monitors

```

Just (
  fromList [
    ("chain.creation.block", Array [
      Object (fromList [ ("monitor", String "((time > (23 s)) Or (time < (17 s)))")),
      Object (fromList [ ("actions", Array [
        String "AlterMinSeverity \"chain.creation\" Debug"])]))
    ], ("#aggregation.critproc.observable", Array [
      Object (fromList [ ("monitor", String "(mean >= (42))")),
      Object (fromList [ ("actions", Array [
        String "CreateMessage \"exceeded\" \"the observable has been too long too high\"",
        String "AlterGlobalMinSeverity Info"])]))
    ]))

getMonitors :: Configuration → IO (HM.HashMap LoggerName (MEvPreCond, MEvExpr, [MEvAction]))
getMonitors configuration = do
  cg ← readMVar $ getCG configuration
  return (cgMonitors cg)

setMonitors :: Configuration → HM.HashMap LoggerName (MEvPreCond, MEvExpr, [MEvAction]) → IO ()
setMonitors configuration monitors =
  modifyMVar_ (getCG configuration) $ \cg →
    return cg {cgMonitors = monitors}

```

Parse configuration from file

Parse the configuration into an internal representation first. Then, fill in **Configuration** after refinement.

```

setup :: FilePath → IO Configuration
setup fp = do
  r ← R.parseRepresentation fp
  setupFromRepresentation r

parseMonitors :: Maybe (HM.HashMap Text Value) → HM.HashMap LoggerName (MEvPreCond, MEvExpr, [MEvAction])
parseMonitors Nothing = HM.empty
parseMonitors (Just hmv) = HM.mapMaybe mkMonitor hmv
  where
    mkMonitor :: Value → Maybe (MEvPreCond, MEvExpr, [MEvAction])
    mkMonitor = parseMaybe $ \v →
      (withObject "" $ \o →
        (,,) <$> o . ? "monitor-if"
          <*> o . ? "monitor"
          <*> o . ? "actions") v
      <|> parseJSON v

setupFromRepresentation :: R.Representation → IO Configuration
setupFromRepresentation r = do
  let mapseverities0 = HM.lookup "mapSeverity" (R.options r)
      mapbackends = HM.lookup "mapBackends" (R.options r)
      mapsubtrace = HM.lookup "mapSubtrace" (R.options r)
      mapscribes0 = HM.lookup "mapScribes" (R.options r)
      mapaggregatedkinds = HM.lookup "mapAggregatedkinds" (R.options r)

```

```

mapmonitors = HM.lookup "mapMonitors" (R.options r)
mapseverities = parseSeverityMap mapseverities0
mapscribes    = parseScribeMap mapscribes0
defRotation   = R.rotation r

cgref ← newMVar $ ConfigurationInternal
{cgMinSeverity    = R.minSeverity r
, cgDefRotation   = defRotation
, cgMapSeverity   = mapseverities
, cgMapSubtrace   = parseSubtraceMap mapsubtrace
, cgOptions       = R.options r
, cgMapBackend    = parseBackendMap mapbackends
, cgDefBackendKs  = R.defaultBackends r
, cgSetupBackends = R.setupBackends r
, cgMapScribe     = mapscribes
, cgMapScribeCache = mapscribes
, cgDefScribes    = r.defaultScribes r
, cgSetupScribes  = fillRotationParams defRotation (R.setupScribes r)
, cgMapAggregatedKind = parseAggregatedKindMap mapaggregatedkinds
, cgDefAggregatedKind = StatsAK
, cgMonitors      = parseMonitors mapmonitors
, cgPortEKG       = r.hasEKG r
, cgPortGraylog   = r.hasGraylog r
, cgPortPrometheus = r.hasPrometheus r
, cgPortGUI       = r.hasGUI r
, cgLogOutput     = R.logOutput r
}

return $ Configuration cgref
where
  parseSeverityMap :: Maybe (HM.HashMap Text Value) → HM.HashMap Text Severity
  parseSeverityMap Nothing = HM.empty
  parseSeverityMap (Just hmv) = HM.mapMaybe mkSeverity hmv
  where
    mkSeverity (String s) = Just (read (unpack s) :: Severity)
    mkSeverity _ = Nothing

  fillRotationParams :: Maybe RotationParameters → [ScribeDefinition] → [ScribeDefinition]
  fillRotationParams defaultRotation = map $ \sd →
    if (scKind sd ≠ StdoutSK) ∧ (scKind sd ≠ StderrSK) ∧ (scKind sd ≠ DevNullSK)
    # ifdef ENABLE_SYSTEMD
      ∧ (scKind sd ≠ JournalSK)
    # endif
  then
    sd {scRotation = maybe defaultRotation Just (scRotation sd)}
  else
    -- stdout, stderr, /dev/null and systemd cannot be rotated
    sd {scRotation = Nothing}

  parseBackendMap Nothing = HM.empty
  parseBackendMap (Just hmv) = HM.map mkBackends hmv
  where
    mkBackends (Array bes) = catMaybes $ map mkBackend $ Vector.toList bes
    mkBackends _ = []
    mkBackend :: Value → Maybe BackendKind

```

```

    mkBackend = parseMaybe parseJSON
  parseScribeMap Nothing = HM.empty
  parseScribeMap (Just hmv) = HM.map mkScribes hmv
  where
    mkScribes (Array scs) = catMaybes $ map mkScribe $ Vector.toList scs
    mkScribes (String s) = [(s :: ScribeId)]
    mkScribes _ = []
    mkScribe :: Value → Maybe ScribeId
    mkScribe = parseMaybe parseJSON
  parseSubtraceMap :: Maybe (HM.HashMap Text Value) → HM.HashMap Text SubTrace
  parseSubtraceMap Nothing = HM.empty
  parseSubtraceMap (Just hmv) = HM.mapMaybe mkSubtrace hmv
  where
    mkSubtrace :: Value → Maybe SubTrace
    mkSubtrace = parseMaybe parseJSON
  r_hasEKG repr = case (R.hasEKG repr) of
    Nothing → 0
    Just p → p
  r_hasGraylog repr = case (R.hasGraylog repr) of
    Nothing → 0
    Just p → p
  r_hasPrometheus repr = case (R.hasPrometheus repr) of
    Nothing → 12799 -- default port for Prometheus
    Just p → p
  r_hasGUI repr = case (R.hasGUI repr) of
    Nothing → 0
    Just p → p
  r_defaultScribes repr = map (λ(k,n) → pack (show k) <> " :: " <> n) (R.defaultScribes repr)
  parseAggregatedKindMap :: Maybe (HM.HashMap Text Value) → HM.HashMap LoggerName AggregatedKind
  parseAggregatedKindMap Nothing = HM.empty
  parseAggregatedKindMap (Just hmv) = HM.mapMaybe mkAggregatedKind hmv
  where
    mkAggregatedKind :: Value → Maybe AggregatedKind
    mkAggregatedKind (String s) = Just $ read $ unpack s
    mkAggregatedKind v = (parseMaybe parseJSON) v

```

Setup empty configuration

```

empty :: IO Configuration
empty = do
  cgreg ← newMVar $ ConfigurationInternal
  {cgMinSeverity    = Debug
  ,cgDefRotation    = Nothing
  ,cgMapSeverity    = HM.empty
  ,cgMapSubtrace    = HM.fromList [
    ("#messagecounters.ekgview", NoTrace),
    ("#messagecounters.aggregation", NoTrace),
    ("#messagecounters.switchboard", NoTrace),
    ("#messagecounters.monitoring", NoTrace),

```

```

        ("#messagecounters.katip",NoTrace),
        ("#messagecounters.graylog",NoTrace)]
,cgOptions      = HM.empty
,cgMapBackend    = HM.empty
,cgDefBackendKs  = []
,cgSetupBackends = []
,cgMapScribe     = HM.empty
,cgMapScribeCache = HM.empty
,cgDefScribes     = []
,cgSetupScribes  = []
,cgMapAggregatedKind = HM.empty
,cgDefAggregatedKind = StatsAK
,cgMonitors      = HM.empty
,cgPortEKG       = 0
,cgPortGraylog   = 0
,cgPortPrometheus = 12799
,cgPortGUI       = 0
,cgLogOutput     = Nothing
}
return $ Configuration cgregf

```

toRepresentation

```

toRepresentation :: Configuration → IO R.Representation
toRepresentation (Configuration c) = do
  cfg ← readMVar c
  let portEKG = cgPortEKG cfg
      portGraylog = cgPortGraylog cfg
      portPrometheus = cgPortPrometheus cfg
      portGUI = cgPortGUI cfg
      otherOptions = cgOptions cfg
      defScribes = cgDefScribes cfg
      splitScribeId :: ScribeId → (ScribeKind, Text)
      splitScribeId x =
        -- "(ScribeId)" = "(ScribeKind) :: (Filename)"
        let (a,b) = T.breakOn "::" x
        in
          (read $ unpack a, T.drop 2 b)
      createOption name f hashmap = if null hashmap
        then HM.empty
        else HM.singleton name $ HM.map f hashmap
  toString :: Show a ⇒ a → Value
  toString = String ∘ pack ∘ show
  toObject :: (MEvPreCond, MEvExpr, [MEvAction]) → Value
  toObject (Nothing, expr, actions) =
    object [ "monitor" . = expr
            , "actions" . = actions
            ]
  toObject (Just precondition, expr, actions) =
    object [ "monitor-if" . = precondition

```

```

    , "monitor"      . = expr
    , "actions"     . = actions
  ]
toJSON' :: [ScribeId] → Value
toJSON' [sid] = toJSON sid
toJSON' ss    = toJSON ss
mapSeverities = createOption "mapSeverity" toJSON $ cgMapSeverity cfg
mapBackends  = createOption "mapBackends" toJSON $ cgMapBackend cfg
mapAggKinds  = createOption "mapAggregatedkinds" toString $ cgMapAggregatedKind cfg
mapScribes   = createOption "mapScribes" toJSON' $ cgMapScribe cfg
mapSubtrace  = createOption "mapSubtrace" toJSON $ cgMapSubtrace cfg
mapMonitors  = createOption "mapMonitors" toObject $ cgMonitors cfg
return $
  R.Representation
    { R.minSeverity    = cgMinSeverity cfg
    , R.rotation       = cgDefRotation cfg
    , R.setupScribes   = cgSetupScribes cfg
    , R.defaultScribes = map splitScribeId defScribes
    , R.setupBackends  = cgSetupBackends cfg
    , R.defaultBackends = cgDefBackendKs cfg
    , R.hasEKG         = if portEKG == 0 then Nothing else Just portEKG
    , R.hasGraylog     = if portGraylog == 0 then Nothing else Just portGraylog
    , R.hasPrometheus  = if portPrometheus == 0 then Nothing else Just portPrometheus
    , R.hasGUI         = if portGUI == 0 then Nothing else Just portGUI
    , R.logOutput      = cgLogOutput cfg
    , R.options        = mapSeverities 'HM.union'
                        mapBackends 'HM.union'
                        mapAggKinds 'HM.union'
                        mapSubtrace 'HM.union'
                        mapScribes 'HM.union'
                        mapMonitors 'HM.union'
                        otherOptions
    }

```

Export **Configuration** into a file

Converts **Configuration** into the form of *Representation* and writes it to the given file.

```

exportConfiguration :: Configuration → FilePath → IO ()
exportConfiguration cfg file = do
  representation ← toRepresentation cfg
  Yaml.encodeFile file representation

```

Evaluation of **FilterTrace**

A filter consists of a *DropName* and a list of *UnhideNames*. If the context name matches the *DropName* filter, then at least one of the *UnhideNames* must match the name to have the evaluation of the filters return *True*.

```

findRootSubTrace :: Configuration → LoggerName → IO (Maybe SubTrace)
findRootSubTrace config logname =

```

```

-- Try to find SubTrace by provided name.
let find_s :: [Text] → IO (Maybe SubTrace)
  find_s [] = return Nothing
  find_s lnames = findSubTrace config (T.intercalate "." lnames) >> λcase
    Just subtrace → return $ Just subtrace
    Nothing → find_s (init lnames)
in find_s $ T.split (≡ ' . ') loggername

testSubTrace :: Configuration → LoggerName → LogObject a → IO (Maybe (LogObject a))
testSubTrace config loggername lo = do
  subtrace ← fromMaybe Neutral < $ > findRootSubTrace config loggername
  return $ testSubTrace' lo subtrace
where
  testSubTrace' :: LogObject a → SubTrace → Maybe (LogObject a)
  testSubTrace' _ NoTrace = Nothing
  testSubTrace' (LogObject _ _ (ObserveOpen _)) DropOpening = Nothing
  testSubTrace' o@(LogObject _ _ (LogValue vname _)) (FilterTrace filters) =
    if evalFilters filters (loggername <> "." <> vname)
    then Just o
    else Nothing
  testSubTrace' o (FilterTrace filters) =
    if evalFilters filters loggername
    then Just o
    else Nothing
  testSubTrace' o (SetSeverity sev) = Just $ o {loMeta = (loMeta o) {severity = sev}}
  testSubTrace' o _ = Just o -- fallback: all pass

evalFilters :: [(DropName, UnhideNames)] → LoggerName → Bool
evalFilters fs nm =
  all (λ(no, yes) → if (dropFilter nm no) then (unhideFilter nm yes) else True) fs
where
  dropFilter :: LoggerName → DropName → Bool
  dropFilter name (Drop sel) = (matchName name sel)
  unhideFilter :: LoggerName → UnhideNames → Bool
  unhideFilter _ (Unhide []) = False
  unhideFilter name (Unhide us) = any (λsel → matchName name sel) us
  matchName :: LoggerName → NameSelector → Bool
  matchName name (Exact name') = name ≡ name'
  matchName name (StartsWith prefix) = T.isPrefixOf prefix name
  matchName name (EndsWith postfix) = T.isSuffixOf postfix name
  matchName name (Contains name') = T.isInfixOf name' name

```

1.7.25 Cardano.BM.Configuration.Static

Default configuration outputting on *stdout*

```

defaultConfigStdout :: IO CM.Configuration
defaultConfigStdout = do
  c ← CM.empty
  CM.setMinSeverity c Debug
  CM.setSetupBackends c [KatipBK]
  CM.setDefaultBackends c [KatipBK]

```

```

CM.setSetupScribes c [ ScribeDefinition {
    scName = "text"
    ,scFormat = ScText
    ,scKind = StdoutSK
    ,scPrivacy = ScPublic
    ,scRotation = Nothing
  }
, ScribeDefinition {
    scName = "json"
    ,scFormat = ScJson
    ,scKind = StdoutSK
    ,scPrivacy = ScPublic
    ,scRotation = Nothing
  }
]
CM.setDefaultScribes c [ "StdoutSK::text" ]
return c

```

Default configuration for testing

```

defaultConfigTesting :: IO CM.Configuration
defaultConfigTesting = do
  c ← CM.empty
  CM.setMinSeverity c Debug
# ifdef ENABLE_AGGREGATION
  CM.setSetupBackends c [ KatipBK, AggregationBK ]
  CM.setDefaultBackends c [ KatipBK, AggregationBK ]
# else
  CM.setSetupBackends c [ KatipBK ]
  CM.setDefaultBackends c [ KatipBK ]
# endif
  CM.setSetupScribes c [ ScribeDefinition {
    scName = "nooutput"
    ,scFormat = ScText
    ,scKind = DevNullSK
    ,scPrivacy = ScPublic
    ,scRotation = Nothing
  }
  ]
  CM.setDefaultScribes c [ "NullSK::nooutput" ]
  return c

```

1.7.26 Cardano.BM.Backend.Switchboard

Switchboard

We are using an *MVar* because we spawn a set of backends that may try to send messages to the switchboard before it is completely setup.

```

type SwitchboardMVar a = MVar (SwitchboardInternal a)
newtype Switchboard a = Switchboard

```



```

    {getSB :: SwitchboardMVar a}
data SwitchboardInternal a = SwitchboardInternal
    {sbQueue    :: TBQ.TBQueue (LogObject a)
    ,sbDispatch :: Async.Async ()
    ,sbLogBuffer :: Cardano.BM.Backend ◦ LogBuffer.LogBuffer a
    ,sbBackends :: NamedBackends a
    }
type NamedBackends a = [(BackendKind, Backend a)]

```

Trace that forwards to the **Switchboard**

Every **Trace** ends in the **Switchboard** which then takes care of dispatching the messages to the selected backends.

This **Tracer** will forward all messages unconditionally to the **Switchboard**. (currently disabled)

```

mainTrace :: IsEffectuator eff a ⇒ eff a → Tracer IO (LogObject a)
mainTrace = Tracer ◦ effectuate

```

This **Tracer** will apply to every message the severity filter as defined in the **Configuration**.

```

mainTraceConditionally :: IsEffectuator eff a ⇒ Configuration → eff a → Tracer IO (LogObject a)
mainTraceConditionally config eff = Tracer $ λitem → do
    mayItem ← Config.testSubTrace config (loName item) item
    case mayItem of
        Just itemF@(LogObject loggename meta _) → do
            passSevFilter ← Config.testSeverity config loggename meta
            when passSevFilter $
                effectuate eff itemF
        Nothing → pure ()

```

Process incoming messages

Incoming messages are put into the queue, and then processed by the dispatcher. The switchboard will never block when processing incoming messages ("eager receiver").

The queue is initialized and the message dispatcher launched.

```

instance IsEffectuator Switchboard a where
    effectuate switchboard item = do
        let writequeue :: TBQ.TBQueue (LogObject a) → LogObject a → IO ()
        writequeue q i = do
            nocapacity ← atomically $ TBQ.isFullTBQueue q
            if nocapacity
            then handleOverflow switchboard
            else atomically $ TBQ.writeTBQueue q i
        sb ← readMVar (getSB switchboard)
        writequeue (sbQueue sb) item
        handleOverflow _ = TIO.hPutStrLn stderr "Error: Switchboard's queue full, dropping log item"

```


Switchboard implements Backend functions**Switchboard** is an **IsBackend**

```

instance (FromJSON a, ToObject a) => IsBackend Switchboard a where
  typeof _ = SwitchboardBK
  realize cfg = do
    -- we setup LogBuffer explicitly so we can access it as a Backend and as LogBuffer
    logbuf :: Cardano.BM.Backend o LogBuffer.LogBuffer a ← Cardano.BM.Backend o LogBuffer.realize cf
    let spawnDispatcher
      :: Switchboard a
      → TBQ.TBQueue (LogObject a)
      → IO (Async.Async ())
    spawnDispatcher switchboard queue = do
      now ← getCurrentTime
      let messageCounters = resetCounters now
      countersMVar ← newMVar messageCounters
      let traceInQueue q =
        Tracer $ λlognamed → do
          item' ← Config.testSubTrace cfg (loName lognamed) lognamed
          case item' of
            Just obj@(LogObject logername meta _) → do
              passSevFilter ← Config.testSeverity cfg logername meta
              when passSevFilter $ do
                nocapacity ← atomically $ TBQ.isFullTBQueue q
                if nocapacity
                then putStrLn "Error: Switchboard's queue full, dropping log items!"
                else atomically $ TBQ.writeTBQueue q obj
            Nothing → pure ()
      _timer ← Async.async $ sendAndResetAfter
        (traceInQueue queue)
        "#messagecounters.switchboard"
        countersMVar
        60000 -- 60000 ms = 1 min
        Debug
    let sendMessage nli befilter = do
      let name = case nli of
        LogObject loname _ (LogValue valueName _) →
          loname <> "." <> valueName
        LogObject loname _ _ → loname
      selectedBackends ← getBackends cfg name
      let selBEs = befilter selectedBackends
      withMVar (getSB switchboard) $ λsb →
        forM_ (sbBackends sb) $ λ(bek, be) →
          when (bek ∈ selBEs) (bEffectuate be nli)
    qProc counters = do
      -- read complete queue at once and process items
      nlis ← atomically $ do
        r ← TBQ.flushTBQueue queue
        when (null r) retry
      return r

```

```

let processItem nli@(LogObject loname _ loitem) = do
  when (loname ≠ "#messagecounters.switchboard") $
    modifyMVar counters $
      λcnt → return $ updateMessageCounters cnt nli
  Config.findSubTrace cfg loname ≫ λcase
  Just (TeeTrace sndName) →
    atomically $ TBQ.writeTBQueue queue $ nli { loName = loname <> "." <> sndName }
  _ → return ()
case loitem of
  KillPill → do
    -- each of the backends will be terminated sequentially
    withMVar (getSB switchboard) $ λsb →
      forM_ (sbBackends sb) (λ(_, be) → bUnrealize be)
    -- all backends have terminated
    return False
  (AggregatedMessage _) → do
    sendMessage nli (filter (≠ AggregationBK))
    return True
  (MonitoringEffect (MonitorAlert _)) → do
    sendMessage nli (filter (≠ MonitoringBK))
    return True
  (MonitoringEffect (MonitorAlterGlobalSeverity sev)) → do
    setMinSeverity cfg sev
    return True
  (MonitoringEffect (MonitorAlterSeverity loggerName sev)) → do
    setSeverity cfg loggerName (Just sev)
    return True
  (Command (DumpBufferedTo bk)) → do
    msgs ← Cardano.BM.Backend ◦ LogBuffer.readBuffer logbuf
    forM_ msgs (λ(lonm, lobj) → sendMessage (lobj { loName = lonm }) (const [bk]))
    return True
  _ → do
    sendMessage nli id
    return True

res ← mapM processItem nlis
when (and res) $ qProc counters

Async.async $ qProc countersMVar
# ifdef PERFORMANCE_TEST_QUEUE
  let qSize = 1000000
# else
  let qSize = 2048
# endif
q ← atomically $ TBQ.newTBQueue qSize
sbref ← newEmptyMVar
let sb :: Switchboard a = Switchboard sbref
backends ← getSetupBackends cfg
bs0 ← setupBackends backends cfg sb
bs1 ← return (LogBufferBK, MkBackend
  { bEffectuate = Cardano.BM.Backend ◦ LogBuffer.effectuate logbuf
  , bUnrealize = Cardano.BM.Backend ◦ LogBuffer.unrealize logbuf

```

```

    })
  let bs = bs1 : bs0
  dispatcher ← spawnDispatcher sb q
  -- link the given Async to the current thread, such that if the Async
  -- raises an exception, that exception will be re-thrown in the current
  -- thread, wrapped in ExceptionInLinkedThread.
  Async.link dispatcher
  putMVar sbref $ SwitchboardInternal {
    sbQueue = q,
    sbDispatch = dispatcher,
    sbLogBuffer = logbuf,
    sbBackends = bs}

  return sb
unrealize switchboard = do
  let clearMVar :: MVar some → IO ()
  clearMVar = void ∘ tryTakeMVar

  (dispatcher, queue) ← withMVar (getSB switchboard) (λsb → return (sbDispatch sb, sbQueue sb))
  -- send terminating item to the queue
  lo ← LogObject <$> pure "kill.switchboard"
    <*> (mkLOMeta Warning Confidential)
    <*> pure KillPill
  atomically $ TBQ.writeTBQueue queue lo
  -- wait for the dispatcher to exit
  res ← Async.waitCatch dispatcher
  either throwM return res
  (clearMVar ∘ getSB) switchboard

```

Integrate with external backend

```

addExternalBackend :: Switchboard a → Backend a → Text → IO ()
addExternalBackend switchboard be name =
  modifyMVar_ (getSB switchboard) $ λsb →
    return $ sb {sbBackends = (UserDefinedBK name, be) : sbBackends sb}

```

Waiting for the switchboard to terminate

```

waitForTermination :: Switchboard a → IO ()
waitForTermination switchboard =
  tryReadMVar (getSB switchboard) >> λcase
    Nothing → return ()
    Just sb → Async.waitCatch (sbDispatch sb) >> return ()

```

Reading the buffered log messages

```

readLogBuffer :: Switchboard a → IO [(LoggerName, LogObject a)]
readLogBuffer switchboard = do
  sb ← readMVar (getSB switchboard)
  Cardano.BM.Backend ∘ LogBuffer.readBuffer (sbLogBuffer sb)

```

Realizing the backends according to configuration

```

setupBackends :: (FromJSON a, ToObject a)
  ⇒ [ BackendKind ]
  → Configuration
  → Switchboard a
  → IO [ (BackendKind, Backend a) ]
setupBackends bes c sb = setupBackendsAcc bes [ ]
where
  setupBackendsAcc [ ] acc = return acc
  setupBackendsAcc (bk : r) acc = do
    setupBackend' bk c sb >>= λcase
      Nothing → setupBackendsAcc r acc
      Just be → setupBackendsAcc r ((bk, be) : acc)

setupBackend' :: (FromJSON a, ToObject a) ⇒ BackendKind → Configuration → Switchboard a → IO (Maybe (BackendKind, Backend a))
setupBackend' SwitchboardBK _ _ = fail "cannot instantiate a further Switchboard"
setupBackend' (UserDefinedBK _) _ _ = fail "cannot instantiate an user-defined backend"
# ifdef ENABLE_MONITORING
setupBackend' MonitoringBK c sb = do
  let basetrace = mainTraceConditionally c sb
  be :: Cardano.BM.Backend ◦ Monitoring.Monitor a ← Cardano.BM.Backend ◦ Monitoring.realizefrom c sb
  return $ Just MkBackend
    { bEffectuate = Cardano.BM.Backend ◦ Monitoring.effectuate be
    , bUnrealize = Cardano.BM.Backend ◦ Monitoring.unrealize be
    }
# else
setupBackend' MonitoringBK _ _ = do
  TIO.hPutStrLn stderr "disabled! will not setup backend 'Monitoring'"
  return Nothing
# endif
# ifdef ENABLE_EKG
setupBackend' EKGViewBK c sb = do
  let basetrace = mainTraceConditionally c sb
  be :: Cardano.BM.Backend ◦ EKGView.EKGView a ← Cardano.BM.Backend ◦ EKGView.realizefrom c sb
  return $ Just MkBackend
    { bEffectuate = Cardano.BM.Backend ◦ EKGView.effectuate be
    , bUnrealize = Cardano.BM.Backend ◦ EKGView.unrealize be
    }
# else
setupBackend' EKGViewBK _ _ = do
  TIO.hPutStrLn stderr "disabled! will not setup backend 'EKGView'"
  return Nothing
# endif
# ifdef ENABLE_AGGREGATION
setupBackend' AggregationBK c sb = do
  let basetrace = mainTraceConditionally c sb
  be :: Cardano.BM.Backend ◦ Aggregation.Aggregation a ← Cardano.BM.Backend ◦ Aggregation.realizefrom c sb
  return $ Just MkBackend
    { bEffectuate = Cardano.BM.Backend ◦ Aggregation.effectuate be
    , bUnrealize = Cardano.BM.Backend ◦ Aggregation.unrealize be
    }

```

```

    }
# else
setupBackend' AggregationBK _ _ = do
    TIO.hPutStrLn stderr "disabled! will not setup backend 'Aggregation'"
    return Nothing
# endif
# ifdef ENABLE_GUI
setupBackend' EditorBK c sb = do
    port ← Config.getGUIport c
    if port > 0
    then do
        let trace = mainTraceConditionally c sb
        be :: Cardano.BM.Backend ◦ Editor.Editor a ← Cardano.BM.Backend ◦ Editor.realizefrom c trace sb
        return $ Just MkBackend
            { bEffectuate = Cardano.BM.Backend ◦ Editor.effectuate be
            , bUnrealize = Cardano.BM.Backend ◦ Editor.unrealize be
            }
    else
        return Nothing
# else
setupBackend' EditorBK _ _ = do
    TIO.hPutStrLn stderr "disabled! will not setup backend 'Editor'"
    return Nothing
# endif
# ifdef ENABLE_GRAYLOG
setupBackend' GraylogBK c sb = do
    port ← Config.getGraylogPort c
    if port > 0
    then do
        let trace = mainTraceConditionally c sb
        be :: Cardano.BM.Backend ◦ Graylog.Graylog a ← Cardano.BM.Backend ◦ Graylog.realizefrom c trace sb
        return $ Just MkBackend
            { bEffectuate = Cardano.BM.Backend ◦ Graylog.effectuate be
            , bUnrealize = Cardano.BM.Backend ◦ Graylog.unrealize be
            }
    else
        return Nothing
# else
setupBackend' GraylogBK _ _ = do
    TIO.hPutStrLn stderr "disabled! will not setup backend 'Graylog'"
    return Nothing
# endif
setupBackend' KatipBK c _ = do
    be :: Cardano.BM.Backend ◦ Log.Log a ← Cardano.BM.Backend ◦ Log.realize c
    return $ Just MkBackend
        { bEffectuate = Cardano.BM.Backend ◦ Log.effectuate be
        , bUnrealize = Cardano.BM.Backend ◦ Log.unrealize be
        }
setupBackend' LogBufferBK _ _ = return Nothing
setupBackend' (TraceAcceptorBK pipePath) c sb = do
    let basetrace = mainTraceConditionally c sb

```

```

be :: Cardano.BM.Backend ◦ TraceAcceptor.TraceAcceptor PipeType a
  ← Cardano.BM.Backend ◦ TraceAcceptor.realizefrom basetrace pipePath
return $ Just MkBackend
  { bEffectuate = Cardano.BM.Backend ◦ TraceAcceptor.effectuate
  , bUnrealize = Cardano.BM.Backend ◦ TraceAcceptor.unrealize be
  }
setupBackend' TraceForwarderBK c _ = do
  be :: Cardano.BM.Backend ◦ TraceForwarder.TraceForwarder PipeType a
    ← Cardano.BM.Backend ◦ TraceForwarder.realize c
  return $ Just MkBackend
    { bEffectuate = Cardano.BM.Backend ◦ TraceForwarder.effectuate be
    , bUnrealize = Cardano.BM.Backend ◦ TraceForwarder.unrealize be
    }
type PipeType =
# ifdef POSIX
  UnixNamedPipe
# else
  NoPipe
# endif

```

MockSwitchboard

MockSwitchboard is useful for tests since it keeps the **LogObjects** to be output in a list.

```

newtype MockSwitchboard a = MockSB (TVar [LogObject a])
instance IsEffectuator MockSwitchboard a where
  effectuate (MockSB tvar) item = atomically $ modifyTVar tvar ((:) item)
  handleOverflow _ = pure ()

```

traceMock

A **Tracer** which forwards **LogObjects** to **MockSwitchboard** simulating functionality of **mainTraceConditionally**

```

traceMock :: MockSwitchboard a → Config.Configuration → Tracer IO (LogObject a)
traceMock ms config =
  Tracer $ λitem@(LogObject logername _) → do
    traceWith mainTrace item
    subTrace ← fromMaybe Neutral < $ > Config.findSubTrace config logername
    case subTrace of
      TeeTrace secName →
        traceWith mainTrace item {loName = secName}
      _ → return ()
  where
    mainTrace = mainTraceConditionally config ms

```

1.7.27 Cardano.BM.Backend.Log

Internal representation

```

type LogMVar = MVar LogInternal
newtype Log a = Log

```

```

    {getK :: LogMVar}
data LogInternal = LogInternal
    {kLogEnv    :: K.LogEnv
    ,msgCounters :: MessageCounter
    ,configuration :: Config.Configuration}

```

Log implements *effectuate*

```

instance ToObject a => IsEffectuator Log a where
    effectuate katip item = do
        let logMVar = getK katip
        c ← configuration < $ > readMVar logMVar
        setupScribes ← getSetupScribes c
        selscribes ← getScribes c (loName item)
        let selscribesFiltered =
            case item of
                LogObject _ (LOMeta _ _ Confidential) (LogMessage _)
                    → removePublicScribes setupScribes selscribes
                _ → selscribes
        forM_ selscribesFiltered $ λsc → passN sc katip item
        -- increase the counter for the specific severity and message type
        modifyMVar_ logMVar $ λli → return $
            li {msgCounters = updateMessageCounters (msgCounters li) item}
        -- reset message counters afer 60 sec = 1 min
        resetMessageCounters logMVar c 60 Warning selscribesFiltered
    where
        removePublicScribes allScribes = filter $ λsc →
            let (_, nameD) = T.breakOn " :: " sc
            -- drop " :: " from the start of name
            name = T.drop 2 nameD
        in
        case find (λx → scName x ≡ name) allScribes of
            Nothing → False
            Just scribe → scPrivacy scribe ≡ ScPrivate
        resetMessageCounters logMVar cfg interval sev scribes = do
            counters ← msgCounters < $ > readMVar logMVar
            let start = mcStart counters
            now = case item of
                LogObject _ meta _ → tstamp meta
            diffTime = round $ diffUTCTime now start
            when (diffTime > interval) $ do
                let counterName = "#messagecounters.katip"
                countersObjects ← forM (HM.toList $ mcCountersMap counters) $ λ(key, count) →
                    LogObject
                        < $ > pure counterName
                        < * > (mkLOMeta sev Confidential)
                        < * > pure (LogValue key (PureI $ toInteger count))
                intervalObject ←
                    LogObject
                        < $ > pure counterName

```



```

    < * > (mkLOMeta sev Confidential)
    < * > pure (LogValue "time_interval_(s)" (PureI diffTime))
  let namedCounters = countersObjects ++ [intervalObject]
  namedCountersFiltered ← catMaybes < $ > (forM namedCounters $ \obj → do
    mayObj ← Config.testSubTrace cfg counterName obj
    case mayObj of
      Just o → do
        passSevFilter ← Config.testSeverity cfg counterName $ loMeta o
        if passSevFilter
          then return $ Just o
          else return Nothing
      Nothing → return Nothing)
  forM_ scribes $ \sc →
    forM_ namedCountersFiltered $ \namedCounter →
      passN sc katip namedCounter
  modifyMVar_ logMVar $ \li → return $
    li {msgCounters = resetCounters now}
  handleOverflow _ = TIO.hPutStrLn stderr "Notice: Katip's queue full, dropping log items!"

```

Log implements backend functions

```

instance (ToObject a, FromJSON a) ⇒ IsBackend Log a where
  typeOf _ = KatipBK
  realize config = do
    let updateEnv :: K.LogEnv → IO UTCTime → K.LogEnv
        updateEnv le timer =
          le {K._logEnvTimer = timer, K._logEnvHost = "hostname"}
    register :: [ScribeDefinition] → K.LogEnv → IO K.LogEnv
    register [ ] le = return le
    register (defsc : dscs) le = do
      let kind = scKind defsc
          sctype = scFormat defsc
          name = scName defsc
          rotParams = scRotation defsc
          name' = pack (show kind) <> " : " <> name
          scr ← createScribe kind sctype name rotParams
          register dscs ≪ K.registerScribe name' scr scribeSettings le
    scribeSettings :: KC.ScribeSettings
    scribeSettings =
      let bufferSize = 5000 -- size of the queue (in log items)
      in
        KC.ScribeSettings bufferSize
    createScribe FileSK ScText name rotParams = mkTextFileScribe
      rotParams
      (FileDescription $ unpack name)
      False
    createScribe FileSK ScJson name rotParams = mkJsonFileScribe
      rotParams
      (FileDescription $ unpack name)
      False

```



```

# if defined (ENABLE_SYSTEMD)
  createScribe JournalSK _ _ = mkJournalScribe
# endif
  createScribe StdoutSK sctype _ _ = mkStdoutScribe sctype
  createScribe StderrSK sctype _ _ = mkStderrScribe sctype
  createScribe DevNullSK _ _ = mkDevNullScribe

cfoKey ← Config.getOptionOrDefault config (pack "cfokey") (pack "<unknown>")
le0 ← K.initLogEnv
      (K.Namespace [ "iohk" ])
      (fromString $ (unpack cfoKey) <> ":" <> showVersion version)
-- request a new time 'getCurrentTime' at most 100 times a second
timer ← mkAutoUpdate defaultUpdateSettings { updateAction = getCurrentTime, updateFreq = 10000 }
let le1 = updateEnv le0 timer
scribes ← getSetupScribes config
le ← register scribes le1

messageCounters ← resetCounters < $ > getCurrentTime
kref ← newMVar $ LogInternal le messageCounters config
return $ Log kref

unrealize katip = do
  le ← withMVar (getK katip) $ \k → return (kLogEnv k)
  void $ K.closeScribes le

example :: IO ()
example = do
  config ← Config.setup "from_some_path.yaml"
  k ← setup config
  meta ← mkLOMeta Info Public
  passN (pack (show StdoutSK)) k $ LogObject
    { loName = "test"
    , loMeta = meta
    , loContent = LogMessage "Hello!"
    }
  meta' ← mkLOMeta Info Public
  passN (pack (show StdoutSK)) k $ LogObject
    { loName = "test"
    , loMeta = meta'
    , loContent = LogValue "cpu-no" 1
    }

```

Needed instances for *katip*:

```

deriving instance ToJSON a ⇒ K.ToObject (LogObject a)
deriving instance K.ToObject Text
deriving instance ToJSON a ⇒ K.ToObject (Maybe (LOContent a))
instance ToJSON a ⇒ KC.LogItem (LogObject a) where
  payloadKeys _ _ = KC.AllKeys
instance KC.LogItem Text where
  payloadKeys _ _ = KC.AllKeys
instance ToJSON a ⇒ KC.LogItem (Maybe (LOContent a)) where
  payloadKeys _ _ = KC.AllKeys

```

Log.passN

The following function copies the **LogObject** to the queues of all scribes that match on their name. Compare start of name of scribe to (*show backend* <> "::<"). This function is non-blocking.

```

passN :: ToObject a => ScribeId -> Log a -> LogObject a -> IO ()
passN backend katip (LogObject loname lometa loitem) = do
  env ← kLogEnv < $ > readMVar (getK katip)
  forM_ (Map.toList $ K._logEnvScribes env) $
    λ(scName, (KC.ScribeHandle _ shChan)) ->
      -- check start of name to match ScribeKind
      if backend `isPrefixOf` scName
      then do
        let (sev, msg, payload) = case loitem of
          (LogMessage logItem) ->
            let loobj = toObject logItem
              (text, maylo) = case (HM.lookup "string" loobj) of
                Just (String m) -> (m, Nothing)
                Just m          -> (TL.toStrict $ encodeToLazyText m, Nothing)
                Nothing         -> (" ", Just loitem)
          in
            (severity lometa, text, maylo)
          (LogError text) ->
            (severity lometa, text, Nothing)
          (LogStructured _) ->
            (severity lometa, "", Just loitem)
          (LogValue name value) ->
            (severity lometa, name <> " = " <> pack (showSI value), Nothing)
          (ObserveDiff _) ->
            let text = TL.toStrict (encodeToLazyText (toObject loitem))
            in
              (severity lometa, text, Just loitem)
          (ObserveOpen _) ->
            let text = TL.toStrict (encodeToLazyText (toObject loitem))
            in
              (severity lometa, text, Just loitem)
          (ObserveClose _) ->
            let text = TL.toStrict (encodeToLazyText (toObject loitem))
            in
              (severity lometa, text, Just loitem)
          (AggregatedMessage aggregated) ->
            let text = T.concat $ (flip map) aggregated $ λ(name, agg) ->
              "\n" <> name <> ": " <> pack (show agg)
            in
              (severity lometa, text, Nothing)
          (MonitoringEffect _) ->
            let text = TL.toStrict (encodeToLazyText (toObject loitem))
            in
              (severity lometa, text, Just loitem)
        KillPill ->
          (severity lometa, "Kill pill received!", Nothing)

```

```

    Command _ →
      (severity lometa, "Command received!", Nothing)
  if (msg == "") ^ (isNothing payload)
  then return ()
  else do
    let threadIdText = KC.ThreadIdText $ tid lometa
    let itemTime = tstamp lometa
    let localname = T.split (≡ ' . ') loname
    let itemKatip = K.Item {
      _itemApp      = env ^. KC.logEnvApp
      , _itemEnv     = env ^. KC.logEnvEnv
      , _itemSeverity = sev2klog sev
      , _itemThread  = threadIdText
      , _itemHost    = env ^. KC.logEnvHost
      , _itemProcess = env ^. KC.logEnvPid
      , _itemPayload = payload
      , _itemMessage = K.logStr msg
      , _itemTime    = itemTime
      , _itemNamespace = (env ^. KC.logEnvApp) <> (K.Namespace localname)
      , _itemLoc     = Nothing
    }
    void $ atomically $ KC.tryWriteTBQueue shChan (KC.NewItem itemKatip)
  else return ()

```

Scribes

The handles to *stdout* and *stderr* will be duplicated because on exit *katip* will close them otherwise.

```

mkStdoutScribe :: ScribeFormat → IO K.Scribe
mkStdoutScribe ScText = do
  stdout' ← hDuplicate stdout
  mkTextFileScribeH stdout' True
mkStdoutScribe ScJson = do
  stdout' ← hDuplicate stdout
  mkJsonFileScribeH stdout' True
mkStderrScribe :: ScribeFormat → IO K.Scribe
mkStderrScribe ScText = do
  stderr' ← hDuplicate stderr
  mkTextFileScribeH stderr' True
mkStderrScribe ScJson = do
  stderr' ← hDuplicate stderr
  mkJsonFileScribeH stderr' True
mkDevNullScribe :: IO K.Scribe
mkDevNullScribe = do
  let logger _ = pure ()
  pure $ K.Scribe logger (pure ())
mkTextFileScribeH :: Handle → Bool → IO K.Scribe
mkTextFileScribeH handler color = do
  mkFileScribeH handler formatter color
where

```

```

    formatter h r =
      let (_,msg) = renderTextMsg r
      in TIO.hPutStrLn h $! msg
mkJsonFileScribeH :: Handle → Bool → IO K.Scribe
mkJsonFileScribeH handler color = do
  mkFileScribeH handler formatter color
where
  formatter h r =
    let (_,msg) = renderJsonMsg r
    in TIO.hPutStrLn h $! msg
mkFileScribeH
  :: Handle
  → (forall a ◦ K.LogItem a ⇒ Handle → Rendering a → IO ())
  → Bool
  → IO K.Scribe
mkFileScribeH h formatter colorize = do
  hSetBuffering h LineBuffering
  locklocal ← newMVar ()
  let logger :: forall a ◦ K.LogItem a ⇒ K.Item a → IO ()
      logger item = withMVar locklocal $ \_ →
        formatter h (Rendering colorize K.V0 item)
  pure $ K.Scribe logger (hClose h)
data Rendering a = Rendering { colorize :: Bool
                              , verbosity :: K.Verbosity
                              , logitem      :: K.Item a
                              }
renderTextMsg :: (K.LogItem a) ⇒ Rendering a → (Int, TL.Text)
renderTextMsg r =
  let m = toLazyText $ formatItem (colorize r) (verbosity r) (logitem r)
  in (fromIntegral $ TL.length m, m)
renderJsonMsg :: (K.LogItem a) ⇒ Rendering a → (Int, TL.Text)
renderJsonMsg r =
  let m' = encodeToLazyText $ trimTime $ K.itemJson (verbosity r) (logitem r)
  in (fromIntegral $ TL.length m', m')
-- keep only two digits for the fraction of seconds
trimTime :: Value → Value
trimTime (Object o) = Object $ HM.adjust
  keep2Decimals
  "at"
  o
where
  keep2Decimals :: Value → Value
  keep2Decimals v = case fromJSON v of
    Success (utct :: UTCTime) →
      String $ pack $ formatTime defaultTimeLocale jformat utct
    _ → v
  jformat :: String
  jformat = "%FT%T%QZ"
trimTime v = v
mkTextFileScribe :: Maybe RotationParameters → FileDescription → Bool → IO K.Scribe

```

```

mkTextFileScribe rotParams fdesc colorize = do
  mkFileScribe rotParams fdesc formatter colorize
where
  formatter :: (K.LogItem a) ⇒ Handle → Rendering a → IO Int
  formatter hdl r =
    case KC._itemMessage (logitem r) of
      K.LogStr "" →
        -- if message is empty do not output it
        return 0
      _ → do
        let (mlen, tmsg) = renderTextMsg r
        TIO.hPutStrLn hdl tmsg
        return mlen

mkJsonFileScribe :: Maybe RotationParameters → FileDescription → Bool → IO K.Scribe
mkJsonFileScribe rotParams fdesc colorize = do
  mkFileScribe rotParams fdesc formatter colorize
where
  formatter :: (K.LogItem a) ⇒ Handle → Rendering a → IO Int
  formatter h r = do
    let (mlen, tmsg) = renderJsonMsg r
    TIO.hPutStrLn h tmsg
    return mlen

mkFileScribe
  :: Maybe RotationParameters
  → FileDescription
  → (forall a ◦ K.LogItem a ⇒ Handle → Rendering a → IO Int)
  → Bool
  → IO K.Scribe

mkFileScribe (Just rotParams) fdesc formatter colorize = do
  let prefixDir = prefixPath fdesc
  (createDirectoryIfMissing True prefixDir)
  'catchIO' (prtoutException ("cannot log prefix directory: " ++ prefixDir))
  let fpath = filePath fdesc
  trp ← initializeRotator rotParams fpath
  scribestate ← newMVar trp -- triple of (handle), (bytes remaining), (rotate time)
  -- sporadically remove old log files - every 10 seconds
  cleanup ← mkAutoUpdate defaultUpdateSettings {
    updateAction = cleanupRotator rotParams fpath
    , updateFreq = 10000000
  }
  let finalizer :: IO ()
  finalizer = withMVar scribestate $
    λ(h, _, _) → hClose h
  let logger :: forall a ◦ K.LogItem a ⇒ K.Item a → IO ()
  logger item =
    modifyMVar_ scribestate $ λ(h, bytes, rotime) → do
      byteswritten ← formatter h (Rendering colorize K.V0 item)
      -- remove old files
      cleanup
      -- detect log file rotation
      let bytes' = bytes - (toInteger $ byteswritten)

```

```

let tdiff' = round $ diffUTCTime rottime (K._itemTime item)
if bytes' < 0 ∨ tdiff' < (0 :: Integer)
  then do -- log file rotation
    hClose h
    (h2, bytes2, rottime2) ← evalRotator rotParams fpath
    return (h2, bytes2, rottime2)
  else
    return (h, bytes', rottime)
return $ K.Scribe logger finalizer
-- log rotation disabled.
mkFileScribe Nothing fdesc formatter colorize = do
  let prefixDir = prefixPath fdesc
  (createDirectoryIfMissing True prefixDir)
  'catchIO' (prtoutException ("cannot log prefix directory: " ++ prefixDir))
  let fpath = filePath fdesc
  h ← catchIO (openFile fpath WriteMode) $
    λe → do
      prtoutException ("error while opening log: " ++ fpath) e
      -- fallback to standard output in case of exception
      return stdout
  hSetBuffering h LineBuffering
  scribestate ← newMVar h
  let finalizer :: IO ()
    finalizer = withMVar scribestate hClose
  let logger :: forall a ◦ K.LogItem a ⇒ K.Item a → IO ()
    logger item =
      withMVar scribestate $ λhandler →
        void $ formatter handler (Rendering colorize K.V0 item)
  return $ K.Scribe logger finalizer

formatItem :: Bool → K.Verbosity → K.Item a → Builder
formatItem withColor _verb K.Item {..} =
  fromText header <>
  fromText " " <>
  brackets (fromText timestamp) <>
  fromText " " <>
  KC.unLogStr _itemMessage
where
  header = colorBySeverity _itemSeverity $
    "[" <> mconcat [namedcontext <> ":", severity <> ":", threadid <> "] "
  namedcontext = KC.intercalateNs _itemNamespace
  severity = KC.renderSeverity _itemSeverity
  threadid = KC.getThreadIdText _itemThread
  timestamp = pack $ formatTime defaultTimeLocale tsformat _itemTime
  tsformat :: String
  tsformat = "%F %T%2Q %Z"
  colorBySeverity s m = case s of
    K.EmergencyS → red m
    K.AlertS     → red m
    K.CriticalS  → red m
    K.ErrorS     → red m

```

```

    K.NoticeS    → magenta m
    K.WarningS  → yellow m
    K.InfoS     → blue m
    _           → m
    red = colorize "31"
    yellow = colorize "33"
    magenta = colorize "35"
    blue = colorize "34"
    colorize c m
      | withColor = "\ESC[" <> c <> "m" <> m <> "\ESC[0m"
      | otherwise = m

-- translate Severity to Log.Severity
sev2klog :: Severity → K.Severity
sev2klog = λcase
  Debug    → K.DebugS
  Info     → K.InfoS
  Notice   → K.NoticeS
  Warning  → K.WarningS
  Error    → K.ErrorS
  Critical → K.CriticalS
  Alert    → K.AlertS
  Emergency → K.EmergencyS

data FileDescription = FileDescription {
  filePath :: !FilePath
  deriving (Show)
  prefixPath :: FileDescription → FilePath
  prefixPath = takeDirectory ∘ filePath

# ifdef ENABLE_SYSTEMD
mkJournalScribe :: IO K.Scribe
mkJournalScribe = return $ journalScribe Nothing (sev2klog Debug) K.V3
-- taken from https://github.com/haskell-service/katip-libsystemd-journal
journalScribe :: Maybe Facility
  → K.Severity
  → K.Verbosity
  → K.Scribe
journalScribe facility severity verbosity = K.Scribe liPush scribeFinalizer
where
  liPush :: K.LogItem a ⇒ K.Item a → IO ()
  liPush i = when (K.permitItem severity i) $
    sendJournalFields $ itemToJournalFields facility verbosity i
  scribeFinalizer :: IO ()
  scribeFinalizer = pure ()

```

Converts a *Katip* *Item* into a *libsystemd-journal* *JournalFields* map.

```

itemToJournalFields :: K.LogItem a
  ⇒ Maybe Facility
  → K.Verbosity

```

```

→ K.Item a
→ JournalFields
itemToJournalFields facility verbosity item = mconcat [ defaultFields item
, maybe HM.empty facilityFields facility
, maybe HM.empty locFields (K._itemLoc item)
]
where
  defaultFields kItem =
    mconcat [ message (TL.toStrict $ toLazyText $ KC.unLogStr (KC._itemMessage kItem))
, priority (mapSeverity (KC._itemSeverity kItem))
, syslogIdentifier (unNS (KC._itemApp kItem))
, HM.fromList [ (environment, T.encodeUtf8 $ KC.getEnvironment (KC._itemEnv kItem))
, (namespace, T.encodeUtf8 $ unNS (KC._itemNamespace kItem))
, (payload, BL.toStrict $ encode $ KC.payloadObject verbosity (KC._itemPayload kItem))
, (thread, T.encodeUtf8 $ KC.getThreadIdText (KC._itemThread kItem))
, (time, T.encodeUtf8 $ formatAsIso8601 (KC._itemTime kItem))
]
]
  facilityFields = syslogFacility
  locFields Loc {..} = mconcat [ codeFile loc_filename
, codeLine (fst loc_start)
]
  environment = mkJournalField "environment"
  namespace = mkJournalField "namespace"
  payload = mkJournalField "payload"
  thread = mkJournalField "thread"
  time = mkJournalField "time"
  unNS ns = case K.unNamespace ns of
    [] → T.empty
    [p] → p
    parts → T.intercalate "." parts
  mapSeverity s = case s of
    K.DebugS → J.Debug
    K.InfoS → J.Info
    K.NoticeS → J.Notice
    K.WarningS → J.Warning
    K.ErrorS → J.Error
    K.CriticalS → J.Critical
    K.AlertS → J.Alert
    K.EmergencyS → J.Emergency
# endif

```

1.7.28 Cardano.BM.Backend.LogBuffer

Structure of LogBuffer

```

newtype LogBuffer a = LogBuffer
  { getLogBuf :: LogBufferMVar a }
type LogBufferMVar a = MVar (LogBufferInternal a)
data LogBufferInternal a = LogBufferInternal

```



```
{logBuffer :: LogBufferMap a
}
```

Relation from log context name to log item

We keep the latest **LogObject** from a log context in a *HashMap*.

```
type LogBufferMap a = HM.HashMap LoggerName (LogObject a)
```

Read out the latest **LogObjects**

```
readBuffer :: LogBuffer a → IO [(LoggerName, LogObject a)]
readBuffer buffer =
  withMVar (getLogBuf buffer) $ λcurrentBuffer →
    return $ HM.toList $ logBuffer currentBuffer
```

LogBuffer is an effectuator

Function *effectuate* is called to pass in a **LogObject** for log buffering.

```
instance IsEffectuator LogBuffer a where
  effectuate buffer lo@(LogObject logname _lmeta (LogValue lvname _lvalue)) = do
    modifyMVar_ (getLogBuf buffer) $ λcurrentBuffer →
      return $ LogBufferInternal $ HM.insert ("#buffered." <> logname <> "." <> lvname) lo $ logBuffer c
  effectuate buffer lo@(LogObject logname _lmeta _logitem) = do
    modifyMVar_ (getLogBuf buffer) $ λcurrentBuffer →
      return $ LogBufferInternal $ HM.insert ("#buffered." <> logname) lo $ logBuffer currentBuffer
  handleOverflow _ = TIO.hPutStrLn stderr "Notice: overflow in LogBuffer, dropping log items"
```

LogBuffer implements Backend functions

LogBuffer is an **IsBackend**

```
instance FromJSON a ⇒ IsBackend LogBuffer a where
  typeOf _ = LogBufferBK
  realize _ = do
    let emptyBuffer = LogBufferInternal HM.empty
    LogBuffer <$> newMVar emptyBuffer
  unrealize _ = return ()
```

1.7.29 Cardano.BM.Backend.EKGView

Structure of EKGView

```
type EKGViewMVar a = MVar (EKGViewInternal a)
newtype EKGView a = EKGView
  {getEV :: EKGViewMVar a}
data EKGViewInternal a = EKGViewInternal
```

```

{evQueue   :: TBQ.TBQueue (Maybe (LogObject a))
, evLabels :: EKGViewMap Label.Label
, evGauges :: EKGViewMap Gauge.Gauge
, evServer  :: Server
, evDispatch :: Async.Async ()
, evPrometheusDispatch :: Maybe (Async.Async ())
}

```

Relation from variable name to label handler

We keep the label handlers for later update in a *HashMap*.

```
type EKGViewMap a = HM.HashMap Text a
```

Internal Trace

This is an internal **Trace**, named "#ekgview", which can be used to control the messages that are being displayed by EKG.

```

ekgTrace :: ToObject a => EKGView a -> Configuration -> Trace IO a
ekgTrace ekg _c =
  Trace.appendName "#ekgview" $ ekgTrace' ekg
where
  ekgTrace' :: ToObject a => EKGView a -> Tracer IO (LogObject a)
  ekgTrace' ekgview = Tracer $ \lo@(LogObject lname _ _) -> do
    let setLabel :: Text -> Text -> EKGViewInternal a -> IO (Maybe (EKGViewInternal a))
        setLabel name label ekg_i@(EKGViewInternal _ labels _ server _ _) =
          case HM.lookup name labels of
            Nothing -> do
              ekghdl <- getLabel name server
              Label.set ekghdl label
              return $ Just $ ekg_i {evLabels = HM.insert name ekghdl labels}
            Just ekghdl -> do
              Label.set ekghdl label
              return Nothing
    let setGauge :: Text -> Int64 -> EKGViewInternal a -> IO (Maybe (EKGViewInternal a))
        setGauge name value ekg_i@(EKGViewInternal _ _ gauges server _ _) =
          case HM.lookup name gauges of
            Nothing -> do
              ekghdl <- getGauge name server
              Gauge.set ekghdl value
              return $ Just $ ekg_i {evGauges = HM.insert name ekghdl gauges}
            Just ekghdl -> do
              Gauge.set ekghdl value
              return Nothing
    update :: ToObject a => LogObject a -> EKGViewInternal a -> IO (Maybe (EKGViewInternal a))
    update (LogObject lname _ (LogMessage logitem)) ekg_i =
      setLabel lname (pack $ show $ toObject logitem) ekg_i
    update (LogObject lname _ (LogValue iname value)) ekg_i =
      let lname' = lname <> " ." <> iname
      in

```

```

    case value of
      (Microseconds x) → setGauge ("us:" <> logname') (fromIntegral x) ekg_i
      (Nanoseconds x) → setGauge ("ns:" <> logname') (fromIntegral x) ekg_i
      (Seconds      x) → setGauge ("s:" <> logname') (fromIntegral x) ekg_i
      (Bytes        x) → setGauge ("B:" <> logname') (fromIntegral x) ekg_i
      (PureI        x) → setGauge ("int:" <> logname') (fromIntegral x) ekg_i
      (PureD        _) → setLabel ("real:" <> logname') (pack $ show value) ekg_i
      (Severity      _) → setLabel ("sev:" <> logname') (pack $ show value) ekg_i
    update _ _ = return Nothing
  modifyMVar_ (getEV ekgview) $ \ekgview → do
    let -- strip off some prefixes not necessary for display
        lognam1 = case stripPrefix "#ekgview.#aggregation." loname of
          Nothing → loname
          Just ln' → ln'
        logname = case stripPrefix "#ekgview." lognam1 of
          Nothing → lognam1
          Just ln' → ln'
    upd ← update lo {loName = logname} ekgview
    case upd of
      Nothing → return ekgview
      Just ekgview' → return ekgview'

```

EKG view is an effectuator

Function *effectuate* is called to pass in a **LogObject** for display in EKG. If the log item is an *AggregatedStats* message, then all its constituents are put into the queue. In case the queue is full, all new items are dropped.

```

instance IsEffectuator EKGView a where
  effectuate ekgview item = do
    ekg ← readMVar (getEV ekgview)
    let enqueue a = do
      nocapacity ← atomically $ TBQ.isFullTBQueue (evQueue ekg)
      if nocapacity
      then handleOverflow ekgview
      else atomically $ TBQ.writeTBQueue (evQueue ekg) (Just a)
    case item of
      (LogObject logname lometa (AggregatedMessage ags)) → liftIO $ do
        let traceAgg :: [(Text, Aggregated)] → IO ()
            traceAgg [] = return ()
            traceAgg ((n, AggregatedEWMA ewma) : r) = do
              enqueue $ LogObject (logname <> "." <> n) lometa (LogValue "avg" $ avg ewma)
              traceAgg r
            traceAgg ((n, AggregatedStats stats) : r) = do
              let statsname = logname <> "." <> n
              qbasestats s' nm = do
                enqueue $ LogObject nm lometa (LogValue "mean" (PureD $ meanOfStats s'))
                enqueue $ LogObject nm lometa (LogValue "min" $ fmin s')
                enqueue $ LogObject nm lometa (LogValue "max" $ fmax s')
                enqueue $ LogObject nm lometa (LogValue "count" $ PureI $ fromIntegral $ fcount s')
                enqueue $ LogObject nm lometa (LogValue "stdev" (PureD $ stdevOfStats s'))

```

```

        enqueue $ LogObject statsname lometata (LogValue "last" $ flast stats)
        qbasestats (fbasic stats) $ statsname <> ".basic"
        qbasestats (fdelta stats) $ statsname <> ".delta"
        qbasestats (ftimed stats) $ statsname <> ".timed"
        traceAgg r
        traceAgg ags
        (LogObject _ _ (LogMessage _)) → enqueue item
        (LogObject _ _ (LogValue _ _)) → enqueue item
        _ → return ()
    handleOverflow _ = TIO.hPutStrLn stderr "Notice: EKGViews's queue full, dropping log items"

```

EKGView implements Backend functions

EKGView is an IsBackend

```

instance (ToObject a, FromJSON a) => IsBackend EKGView a where
    typeOf _ = EKGViewBK
    realize _ = fail "EKGView cannot be instantiated by 'realize'"
    realizeFrom config sbtrace _ = do
        evref ← newEmptyMVar
        let ekview = EKGView evref
        evport ← getEKGport config
        ehdl ← forkServer "127.0.0.1" evport
        ekghdl ← getLabel "iohk-monitoring version" ehdl
        Label.set ekghdl $ pack (showVersion version)
        let ekgtrace = ekgTrace ekview config
    # ifdef PERFORMANCE_TEST_QUEUE
        let qSize = 1000000
    # else
        let qSize = 5120
    # endif
        queue ← atomically $ TBQ.newTBQueue qSize
        dispatcher ← spawnDispatcher config queue sbtrace ekgtrace
        -- link the given Async to the current thread, such that if the Async
        -- raises an exception, that exception will be re-thrown in the current
        -- thread, wrapped in ExceptionInLinkedThread.
        Async.link dispatcher
    # ifdef ENABLE_PROMETHEUS
        prometheusPort ← getPrometheusPort config
        prometheusDispatcher ← spawnPrometheus ehdl prometheusPort
        Async.link prometheusDispatcher
    # endif
        putMVar evref $ EKGViewInternal
            { evLabels = HM.empty
            , evGauges = HM.empty
            , evServer = ehdl
            , evQueue = queue
            , evDispatch = dispatcher
            }
    # ifdef ENABLE_PROMETHEUS
        , evPrometheusDispatch = Just prometheusDispatcher
    # else

```

```

    ,evPrometheusDispatch = Nothing
# endif
}
return ekgview
unrealize ekgview = do
  let clearMVar :: MVar b → IO ()
      clearMVar = void ∘ tryTakeMVar
  (dispatcher, queue, prometheusDispatcher) ←
    withMVar (getEV ekgview) (λev →
      return (evDispatch ev, evQueue ev, evPrometheusDispatch ev))
  -- send terminating item to the queue
  atomically $ TBQ.writeTBQueue queue Nothing
  -- wait for the dispatcher to exit
  res ← Async.waitCatch dispatcher
  either throwM return res
  case prometheusDispatcher of
    Just d → Async.cancel d
    Nothing → return ()
  withMVar (getEV ekgview) $ λekg →
    killThread $ serverThreadId $ evServer ekg
  clearMVar $ getEV ekgview

```

Asynchronously reading log items from the queue and their processing

```

spawnDispatcher :: Configuration
    → TBQ.TBQueue (Maybe (LogObject a))
    → Trace.Trace IO a
    → Trace.Trace IO a
    → IO (Async.Async ())
spawnDispatcher config evqueue sbtrace ekgtrace = do
  now ← getCurrentTime
  let messageCounters = resetCounters now
  countersMVar ← newMVar messageCounters
  timer ← Async.async $ sendAndResetAfter
    sbtrace
    "#messagecounters.ekgview"
    countersMVar
  60000 -- 60000 ms = 1 min
  Debug
  Async.async $ qProc countersMVar
where
  {- lazy qProc -}
  qProc :: MVar MessageCounter → IO ()
  qProc counters = do
    processQueue
      evqueue
      processEKGView
      counters
    (\_ → pure ())
  processEKGView obj@(LogObject logname _ _) counters = do

```

```

obj' ← testSubTrace config ("#ekgview." <> logname) obj
case obj' of
  Just lo@(LogObject logname' meta content) → do
    let trace = Trace.appendName logname' ekgttrace
    Trace.traceNamedObject trace (meta,content)
    -- increase the counter for the type of message
    modifyMVar_ counters $ \cnt → return $ updateMessageCounters cnt lo
  Nothing → pure ()
return counters

```

1.7.30 Cardano.BM.Backend.Editor

This simple configuration editor is accessible through a browser on <http://127.0.0.1:13789>, or whatever port has been set in the configuration.

A number of maps that relate logging context name to behaviour can be changed. And, most importantly, the global minimum severity that defines the filtering of log messages.

links

The GUI is built on top of *Threepenny-GUI* (<http://hackage.haskell.org/package/threepenny-gui>). The appearance is due to *w3-css* (<https://www.w3schools.com/w3css>).

Structure of Editor

```

type EditorMVar a = MVar (EditorInternal a)
newtype Editor a = Editor
  {getEd :: EditorMVar a}
data {-ToObj a = λ -} EditorInternal a = EditorInternal
  {edSBtrace :: Trace IO a
  ,edThread :: Async.Async ()
  ,edBuffer :: LogBuffer a
  }

```

Editor implements Backend functions

Editor is an **IsBackend**

```

instance (ToObj a, FromJSON a) ⇒ IsBackend Editor a where
  typeOf _ = EditorBK
  realize _ = fail "Editor cannot be instantiated by 'realize'"
  realizefrom config sbtrace _ = do
    gref ← newEmptyMVar
    let gui = Editor gref
    port ← getGUIport config
    when (port ≤ 0) $ fail "cannot create GUI"
    -- local LogBuffer
    logbuf :: Cardano.BM.Backend o LogBuffer.LogBuffer a ← Cardano.BM.Backend o LogBuffer.realize c
    thd ← Async.async $
      startGUI defaultConfig {jsPort = Just port

```

```

        ,jsAddr                = Just "127.0.0.1"
        ,jsStatic              = Just "iohk-monitoring/static"
        ,jsCustomHTML = Just "configuration-editor.html"
    } $ prepare gui config
  Async.link thd
  putMVar gref $ EditorInternal
    { edSBtrace = sbtrace
    , edThread = thd
    , edBuffer = logbuf
    }
  return gui
unrealize editor =
  withMVar (getEd editor) $ \ed →
    Async.cancel $ edThread ed

```

Editor is an effectuator

Function *effectuate* is called to pass in a **LogObject** for display in the GUI.

```

instance ToObject a ⇒ IsEffectuator Editor a where
  effectuate editor item =
    withMVar (getEd editor) $ \ed →
      effectuate (edBuffer ed) item
  handleOverflow _ = TIO.hPutStrLn stderr "Notice: overflow in Editor!"

```

Prepare the view

```

data Cmd = Backends | Scribes | Severities | SubTrace | Aggregation | Buffer | ExportConfiguration
  deriving (Enum, Eq, Show, Read)

prepare :: ToObject a ⇒ Editor a → Configuration → Window → UI ()
prepare editor config window = void $ do
  let commands = [Backends..]
  inputKey ← UI.input #. "w3-input w3-border" # set UI.size "34"
  inputValue ← UI.input #. "w3-input w3-border" # set UI.size "60"
  outputMsg ← UI.input #. "w3-input w3-border"
  currentCmd ← UI.p #. "current-cmd"
  let performActionOnId anId action =
    getElementById window anId >> λcase
      Nothing → return ()
      Just anElement → action anElement
  let turn anElement toState = void $ element anElement # set UI.enabled toState
  let setValueOf anElement aValue = void $ element anElement # set UI.value aValue
  let setClasses classes anElement = void $ element anElement # set UI.class_ classes
  let setError m = setValueOf outputMsg ("ERROR: " ++ m)
  let setMessage m = setValueOf outputMsg m
  let enable anElement = turn anElement True
  let disable anElement = turn anElement False
  let clean anElement = setValueOf anElement ""

```



```

let cleanAndDisable anElement = clean anElement >> disable anElement
let rememberCurrent cmd = setValueOf currentCmd $ show cmd
let removeItem Backends k = CM.setBackends config k Nothing
    removeItem Severities k = CM.setSeverity config k Nothing
    removeItem Scribes k = CM.setScribes config k Nothing
    removeItem SubTrace k = CM.setSubTrace config k Nothing
    removeItem Aggregation k = CM.setAggregatedKind config k Nothing
    removeItem _ = pure ()
let updateItem Backends k v = case (readMay v :: Maybe [BackendKind]) of
    Nothing → setError "parse error on backend list"
    Just v' → liftIO $ CM.setBackends config k $ Just v'
    updateItem Severities k v = case (readMay v :: Maybe Severity) of
    Nothing → setError "parse error on severity"
    Just v' → liftIO $ CM.setSeverity config k $ Just v'
    updateItem Scribes k v = case (readMay v :: Maybe ScribeId) of
    Nothing → setError "parse error on scribe list"
    Just v' → liftIO $ CM.setScribes config k $ Just v'
    updateItem SubTrace k v = case (readMay v :: Maybe SubTrace) of
    Nothing → setError "parse error on subtrace"
    Just v' → liftIO $ CM.setSubTrace config k $ Just v'
    updateItem Aggregation k v = case (readMay v :: Maybe AggregatedKind) of
    Nothing → setError "parse error on aggregated kind"
    Just v' → liftIO $ CM.setAggregatedKind config k $ Just v'
    updateItem _ = pure ()
disable inputKey
disable inputValue
disable outputMsg
let saveItemId = "save-item-button"
let cancelSaveItemId = "cancel-save-item-button"
let addItemButtonId = "add-item-button"
let outputTableId = "output-table"
let addItemButton = performActionOnId addItemButtonId
let saveItemButton = performActionOnId saveItemId
let cancelSaveItemButton = performActionOnId cancelSaveItemId
let cleanOutputTable = performActionOnId outputTableId $ λt → void $ element t # set children []
let mkLinkToFile :: String → FilePath → UI Element
    mkLinkToFile str file = UI.anchor # set (attr "href") file
    # set (attr "target") "_blank"
    #+ [string str]
let mkSimpleRow :: ToObject a ⇒ LoggerName → LogObject a → UI Element
    mkSimpleRow n lo@(LogObject _lonm _lmeta _lov) = UI.tr #. "itemrow" #+
    [UI.td #+ [string (unpack n)]
    , UI.td #+ [string $ BS8.unpack $ {-TL.toStrict -} encode (toObject lo)]
    ]
let mkTableRow :: Show t ⇒ Cmd → LoggerName → t → UI Element
    mkTableRow cmd n v = UI.tr #. "itemrow" #+
    [UI.td #+ [string (unpack n)]
    , UI.td #+ [string (show v)]
    , UI.td #+
    [do

```



```

    b ← UI.button #. "w3-small w3-btn w3-ripple w3-orange edit-item-button"
      #+ [ UI.bold #+ [string "Edit"] ]
on UI.click b $ const $ do
  saveItemButton enable
  cancelSaveItemButton enable
  clean outputMsg
  enable inputKey
  enable inputValue
  setValueOf inputKey (unpack n)
  setValueOf inputValue (show v)
  rememberCurrent cmd
return b
,UI.span # set html "&nbsp;&nbsp;&nbsp;"
,do
  b ← UI.button #. "w3-small w3-btn w3-ripple w3-red"
    #+ [ UI.bold #+ [string "Delete"] ]
on UI.click b $ const $ do
  liftIO $ removeItem cmd n
  cleanAndDisable inputKey
  cleanAndDisable inputValue
  -- Initiate a click to current menu to update the items list after deleting
  performActionOnId (show cmd) $ runFunction ∘ ffi "$(%1).click()"
return b
]
]
let showCurrentTab cmd = do
  let baseClasses = "w3-bar-item w3-button"
    classesForCurrentTab = baseClasses <> " " <> "w3-light-grey"
performActionOnId (show cmd) $ setClasses classesForCurrentTab
let otherTabs = delete cmd commands
forM_ otherTabs $ λtabName →
  performActionOnId (show tabName) $ setClasses baseClasses
let displayItems cmd sel = do
  showCurrentTab cmd
  rememberCurrent cmd
  saveItemButton disable
  cancelSaveItemButton disable
  addItemButton enable
  cleanOutputTable
performActionOnId outputTableId $
  λt → void $ element t #+
    [ UI.tr #+
      [ UI.th #+ [string "LoggerName"]
        , UI.th #+ [string $ show cmd <> " value" ]
        , UI.th #+ [string "" ]
      ]
    ]
cg ← liftIO $ readMVar (CM.getCG config)
forM_ (HM.toList $ sel cg) $
  λ(n,v) → performActionOnId outputTableId $
    λt → void $ element t #+ [ mkTableRow cmd n v ]

```

```

let displayBuffer :: ToObject a ⇒ Cmd → [(LoggerName, LogObject a)] → UI ()
displayBuffer cmd sel = do
  showCurrentTab cmd
  rememberCurrent cmd
  saveItemButton disable
  cancelSaveItemButton disable
  addItemButton disable
  cleanOutputTable
  performActionOnId outputTableId $
    λt → void $ element t #+
      [ UI.tr #+
        [ UI.th #+ [string "LoggerName"]
          , UI.th #+ [string $ show cmd <> " value"]
          , UI.th #+ [string ""]
        ]
      ]
  forM_ (sel) $
    λ(n,v) → performActionOnId outputTableId $
      λt → void $ element t #+ [mkSimpleRow n v]

let accessBufferMap = do
  ed ← liftIO $ readMVar (getEd editor)
  liftIO $ readBuffer $ edBuffer ed

let exportConfiguration = do
  currentDir ← liftIO getCurrentDirectory
  let dir = currentDir </> "iohk-monitoring/static/conf"
  liftIO $ createDirectoryIfMissing True dir
  tsnow ← formatTime defaultTimeLocale tsformat <$> liftIO getCurrentTime
  let filename = "config.yaml" ++ "-" ++ tsnow
  filepath = dir </> filename
  res ← liftIO $ catch
    (CM.exportConfiguration config filepath >>
     return ("Configuration was exported to the file: " ++ filepath))
    (λ(e :: SomeException) → return $ show e)
  setMessage res
  performActionOnId outputTableId $
    λt → void $ element t #+ [mkLinkToFile
      "Link to configuration file"
      ("/static/conf" </> filename)
    ]

let displayExport cmd = do
  showCurrentTab cmd
  rememberCurrent cmd
  saveItemButton disable
  cancelSaveItemButton disable
  addItemButton disable
  cleanOutputTable
  exportConfiguration

let switchToTab c@Backends = displayItems c $ CM.cgMapBackend
  switchToTab c@Severities = displayItems c $ CM.cgMapSeverity
  switchToTab c@Scribes = displayItems c $ CM.cgMapScribe

```

```

switchToTab c@SubTrace    = displayItems c $ CM.cgMapSubtrace
switchToTab c@Aggregation = displayItems c $ CM.cgMapAggregatedKind
switchToTab c@Buffer      = accessBufferMap >>= displayBuffer c
switchToTab c@ExportConfiguration = displayExport c

let mkEditInputs =
  row [element inputKey
    , UI.span #. "key-value-separator" #+ [string ":"]
    , element inputValue
    , UI.span #. "key-value-separator" #+ [string ""]
    , do
      b ← UI.button #. "w3-btn w3-ripple w3-green save-item-button"
      #set (UI.attr "id") addItemButtonId
      #set UI.enabled False
      #+ [UI.bold #+ [string "New"]]
      on UI.click b $ const $ do
        enable inputKey
        enable inputValue
        saveItemButton enable
        cancelSaveItemButton enable
      return b
    , UI.span #. "key-value-separator" #+ [string ""]
    , do
      b ← UI.button #. "w3-btn w3-ripple w3-lime save-item-button"
      #set (UI.attr "id") saveItemButtonId
      #set UI.enabled False
      #+ [UI.bold #+ [string "Save"]]
      on UI.click b $ const $ do
        k ← inputKey # get UI.value
        v ← inputValue # get UI.value
        m ← currentCmd # get UI.value
        case (readMay m :: Maybe Cmd) of
          Nothing → setError "parse error on cmd"
          Just c → do
            cleanAndDisable inputKey
            cleanAndDisable inputValue
            saveItemButton disable
            cancelSaveItemButton disable
            setMessage $ "Setting '" + k + "' to '" + v + "' in " + m
            updateItem c (pack k) v
            switchToTab c
      return b
    , UI.span #. "key-value-separator" #+ [string ""]
    , do
      b ← UI.button #. "w3-btn w3-ripple w3-white"
      #set (UI.attr "id") cancelSaveItemButtonId
      #set UI.enabled False
      #+ [UI.bold #+ [string "Cancel"]]
      on UI.click b $ const $ do
        cleanAndDisable inputKey
        cleanAndDisable inputValue
        saveItemButton disable

```

```

        cancelSaveItemButton disable
    return b
]

let minimumSeveritySelection = do
    confMinSev ← liftIO $ minSeverity config
    let setMinSev _el Nothing = pure ()
        setMinSev _el (Just sev) = liftIO $
            setMinSeverity config (toEnum sev :: Severity)
    mkSevOption sev = UI.option # set UI.text (show sev)
        # set UI.value (show sev)
        # if (confMinSev == sev) then set UI.selected True else id
    minsev ← UI.select #. "minsevfield" #+
        map mkSevOption (enumFrom Debug)
    on UI.selectionChange minsev $ setMinSev minsev
    row [string "Set minimum severity to:"
        , UI.span # set html "&nbsp;"
        , UI.span #. "severity-dropdown big" #+ [element minsev]
    ]

let commandTabs =
    row $ flip map commands $ \cmd → do
        b ← UI.button #. "w3-bar-item w3-button w3-grey"
            # set (UI.attr "id") (show cmd)
            #+ [ UI.bold #+ [string (show cmd)] ]
        on UI.click b $ const $ do
            cleanAndDisable inputKey
            cleanAndDisable inputValue
            clean outputMsg
            switchToTab cmd
        return b

getElementById window "main-section" >>= \case
    Nothing → pure ()
    Just mainSection → void $ element mainSection #+
        [ UI.div #. "w3-panel" #+
            [ UI.div #. "w3-border w3-border-dark-grey" #+
                [ UI.div #. "w3-panel" #+ [minimumSeveritySelection]
                ]
            , UI.div #. "w3-panel" #+ [
                , UI.div #. "w3-border w3-border-dark-grey" #+
                    [ UI.div #. "w3-bar w3-grey" #+ [commandTabs]
                    , UI.div #. "w3-panel" #+ [mkEditInputs]
                    , UI.div #. "w3-panel" #+ [element outputMsg]
                ]
            ]
        ]
]

```

1.7.31 Cardano.BM.Backend.Graylog

Structure of Graylog

```

type GraylogMVar a = MVar (GraylogInternal a)
newtype Graylog a = Graylog
    {getGL :: GraylogMVar a}
data GraylogInternal a = GraylogInternal
    {glQueue :: TBQ.TBQueue (Maybe (LogObject a))
    ,glDispatch :: Async.Async ()
    }

```

Graylog is an effectuator

Function *effectuate* is called to pass in a **LogObject** to forward to Graylog. In case the queue is full, all new items are dropped.

```

instance IsEffectuator Graylog a where
    effectuate graylog item = do
        gelf ← readMVar (getGL graylog)
        let enqueue a = do
            nocapacity ← atomically $ TBQ.isFullTBQueue (glQueue gelf)
            if nocapacity
            then handleOverflow graylog
            else atomically $ TBQ.writeTBQueue (glQueue gelf) (Just a)
        case item of
            (LogObject logname lometa (AggregatedMessage ags)) → liftIO $ do
                let traceAgg :: [(Text, Aggregated)] → IO ()
                traceAgg [] = return ()
                traceAgg ((n, AggregatedEWMA ewma) : r) = do
                    enqueue $ LogObject (logname <> "." <> n) lometa (LogValue "avg" $ avg ewma)
                    traceAgg r
                traceAgg ((n, AggregatedStats stats) : r) = do
                    let statsname = logname <> "." <> n
                    qbasestats s' nm = do
                        enqueue $ LogObject nm lometa (LogValue "mean" (PureD $ meanOfStats s'))
                        enqueue $ LogObject nm lometa (LogValue "min" $ fmin s')
                        enqueue $ LogObject nm lometa (LogValue "max" $ fmax s')
                        enqueue $ LogObject nm lometa (LogValue "count" $ PureI $ fromIntegral $ fcount s')
                        enqueue $ LogObject nm lometa (LogValue "stdev" (PureD $ stdevOfStats s'))
                    enqueue $ LogObject statsname lometa (LogValue "last" $ flast stats)
                    qbasestats (fbasic stats) $ statsname <> ".basic"
                    qbasestats (fdelta stats) $ statsname <> ".delta"
                    qbasestats (ftimed stats) $ statsname <> ".timed"
                    traceAgg r
                traceAgg ags
            (LogObject _ _ (LogMessage _)) → enqueue item
            (LogObject _ _ (LogValue _)) → enqueue item
            _ → return ()
    handleOverflow _ = TIO.hPutStrLn stderr "Notice: Graylogs's queue full, dropping log items

```

Graylog implements Backend functions

Graylog is an IsBackend

```

instance (ToObject a, FromJSON a) ⇒ IsBackend Graylog a where
  typeof _ = GraylogBK
  realize _ = fail "Graylog cannot be instantiated by 'realize'"
  realizefrom config sbtrace _ = do
    glref ← newEmptyMVar
    let graylog = Graylog glref
  # ifdef PERFORMANCE_TEST_QUEUE
    let qSize = 1000000
  # else
    let qSize = 1024
  # endif
  queue ← atomically $ TBQ.newTBQueue qSize
  dispatcher ← spawnDispatcher config queue sbtrace
  -- link the given Async to the current thread, such that if the Async
  -- raises an exception, that exception will be re-thrown in the current
  -- thread, wrapped in ExceptionInLinkedThread.
  Async.link dispatcher
  putMVar glref $ GraylogInternal
    { glQueue = queue
    , glDispatch = dispatcher
    }
  return graylog
unrealize graylog = do
  let clearMVar :: MVar b → IO ()
    clearMVar = void ∘ tryTakeMVar
  (dispatcher, queue) ← withMVar (getGL graylog) (λgelf →
    return (glDispatch gelf, glQueue gelf))
  -- send terminating item to the queue
  atomically $ TBQ.writeTBQueue queue Nothing
  -- wait for the dispatcher to exit
  res ← Async.waitCatch dispatcher
  either throwM return res
  clearMVar $ getGL graylog

```

Asynchronously reading log items from the queue and their processing

```

spawnDispatcher :: forall a ∘ ToObject a
  ⇒ Configuration
  → TBQ.TBQueue (Maybe (LogObject a))
  → Trace.Trace IO a
  → IO (Async.Async ())
spawnDispatcher config evqueue sbtrace = do
  now ← getCurrentTime
  let messageCounters = resetCounters now
  countersMVar ← newMVar messageCounters
  _timer ← Async.async $ sendAndResetAfter

```

```

sbtrace
"#messagecounters.graylog"
countersMVar
60000 -- 60000 ms = 1 min
Debug
let gltrace = Trace.appendName "#graylog" sbtrace
Async.async $ Net.withSocketsDo $ qProc gltrace countersMVar Nothing
where
{- lazy qProc -}
qProc :: Trace.Trace IO a → MVar MessageCounter → Maybe Net.Socket → IO ()
qProc gltrace counters conn =
  processQueue
    evqueue
    processGraylog
      (gltrace, counters, conn)
    (λ(→, →, c) → closeConn c)
processGraylog :: LogObject a → (Trace.Trace IO a, MVar MessageCounter, Maybe Net.Socket) → IO (Trace.Trace IO a)
processGraylog item (gltrace, counters, mConn) = do
  case mConn of
    (Just conn) → do
      sendLO conn item
      'catch' λ(e :: SomeException) → do
        let trace' = Trace.appendName "sending" gltrace
        mle ← mkLOMeta Error Public
        Trace.traceNamedObject trace' (mle, LogError (pack $ show e))
        threadDelay 50000
        void $ processGraylog item (gltrace, counters, mConn)
        modifyMVar_ counters $ λcnt → return $ updateMessageCounters cnt item
        return (gltrace, counters, mConn)
    Nothing → do
      mConn' ← tryConnect gltrace
      processGraylog item (gltrace, counters, mConn')
sendLO :: ToObject a ⇒ Net.Socket → LogObject a → IO ()
sendLO conn obj =
  let msg = BS8.toStrict $ encodeMessage obj
  in sendAll conn msg
closeConn :: Maybe Net.Socket → IO ()
closeConn Nothing = return ()
closeConn (Just conn) = Net.close conn
tryConnect :: Trace.Trace IO a → IO (Maybe Net.Socket)
tryConnect gltrace = do
  port ← getGraylogPort config
  let hints = Net.defaultHints {Net.addrSocketType = Net.Datagram}
  (addr: _) ← Net.getAddrInfo (Just hints) (Just "127.0.0.1") (Just $ show port)
  sock ← Net.socket (Net.addrFamily addr) (Net.addrSocketType addr) (Net.addrProtocol addr)
  res ← Net.connect sock (Net.addrAddress addr) >> return (Just sock)
  'catch' λ(e :: SomeException) → do
    let trace' = Trace.appendName "connecting" gltrace
    mle ← mkLOMeta Error Public
    Trace.traceNamedObject trace' (mle, LogError (pack $ show e))
    return Nothing

```

```

    return res
encodeMessage :: ToObject a => LogObject a -> BS8.ByteString
encodeMessage lo = encode $ mkGelfItem lo

```

Gelf data structure

GELF defines a data format of the message payload: <https://docs.graylog.org/en/3.0/pages/gelf.html>

```

data GelfItem = GelfItem {
    version :: Text,
    host :: Text,
    short_message :: Text,
    full_message :: Value,
    timestamp :: Double,
    level :: Int,
    _tid :: Text,
    _privacy :: Text
}

mkGelfItem :: ToJSON a => LogObject a -> GelfItem
mkGelfItem (LogObject loname lometa locontent) = GelfItem {
    version = "1.1",
    host = "hostname",
    short_message = loname,
    full_message = toJSON locontent,
    timestamp = (fromInteger o toInteger $ (utc2ns $ tstamp lometa) :: Double) / 1000000000,
    level = (fromEnum $ maxBound@Severity) - (fromEnum $ severity lometa),
    _tid = tid lometa,
    _privacy = pack $ show $ privacy lometa
}

instance ToJSON GelfItem where
    toJSON gli = object [
        "version" .= version gli,
        "host" .= host gli,
        "short_message" .= short_message gli,
        "full_message" .= full_message gli,
        "timestamp" .= (printf "%0.3f" $ timestamp gli :: String),
        "level" .= level gli,
        "_tid" .= _tid gli,
        "_privacy" .= _privacy gli
    ]

```

1.7.32 Cardano.BM.Backend.Aggregation

Internal representation

```

type AggregationMVar a = MVar (AggregationInternal a)
newtype Aggregation a = Aggregation
    { getAg :: AggregationMVar a }
data AggregationInternal a = AggregationInternal
    { agQueue :: TBQ.TBQueue (Maybe (LogObject a))

```



```
,agDispatch :: Async.Async ()
}
```

Relation from context name to aggregated statistics

We keep the aggregated values (**Aggregated**) for a named context in a *HashMap*.

```
type AggregationMap = HM.HashMap Text AggregatedExpanded
```

Info for Aggregated operations

Apart from the **Aggregated** we keep some valuable info regarding to them; such as when was the last time it was sent.

```
type Timestamp = Word64
data AggregatedExpanded = AggregatedExpanded
  { aeAggregated :: !Aggregated
  , aeResetAfter :: !(Maybe Word64)
  , aeLastSent :: {-# UNPACK #-} !Timestamp
  }
```

Aggregation implements *effectuate*

Aggregation is an **IsEffectuator** Enter the log item into the **Aggregation** queue.

```
instance IsEffectuator Aggregation a where
  effectuate agg item = do
    ag ← readMVar (getAg agg)
    nocapacity ← atomically $ TBQ.isFullTBQueue (agQueue ag)
    if nocapacity
    then handleOverflow agg
    else atomically $ TBQ.writeTBQueue (agQueue ag) $! Just item
  handleOverflow _ = TIO.hPutStrLn stderr "Notice: Aggregation's queue full, dropping log it"
```

Aggregation implements **Backend** functions

Aggregation is an **IsBackend**

```
instance FromJSON a => IsBackend Aggregation a where
  typeOf _ = AggregationBK
  realize _ = fail "Aggregation cannot be instantiated by 'realize'"
  realizefrom config trace _ = do
    aggref ← newEmptyMVar
  # ifdef PERFORMANCE_TEST_QUEUE
    let qSize = 1000000
  # else
    let qSize = 2048
  # endif
  aggregationQueue ← atomically $ TBQ.newTBQueue qSize
  dispatcher ← spawnDispatcher config HM.empty aggregationQueue trace
```

```

-- link the given Async to the current thread, such that if the Async
-- raises an exception, that exception will be re-thrown in the current
-- thread, wrapped in ExceptionInLinkedException.
Async.link dispatcher
putMVar aggref $ AggregationInternal aggregationQueue dispatcher
return $ Aggregation aggref
unrealize aggregation = do
  let clearMVar :: MVar a → IO ()
      clearMVar = void ∘ tryTakeMVar
  (dispatcher, queue) ← withMVar (getAg aggregation) (λag →
    return (agDispatch ag, agQueue ag))
  -- send terminating item to the queue
  atomically $ TBQ.writeTBQueue queue Nothing
  -- wait for the dispatcher to exit
  -- TODO add a timeout to waitCatch in order
  -- to be sure that it will finish
  res ← Async.waitCatch dispatcher
  either throwM return res
  (clearMVar ∘ getAg) aggregation

```

Asynchronously reading log items from the queue and their processing

```

spawnDispatcher :: Configuration
                → AggregationMap
                → TBQ.TBQueue (Maybe (LogObject a))
                → Trace.Trace IO a
                → IO (Async.Async ())
spawnDispatcher conf aggMap aggregationQueue basetrace = do
  now ← getCurrentTime
  let trace = Trace.appendName "#aggregation" basetrace
  let messageCounters = resetCounters now
  countersMVar ← newMVar messageCounters
  timer ← Async.async $ sendAndResetAfter
    basetrace
    "#messagecounters.aggregation"
    countersMVar
    60000 -- 60000 ms = 1 min
    Debug
  Async.async $ qProc trace countersMVar aggMap
where
  {- lazy qProc -}
  qProc trace counters aggregatedMap = do
    processQueue
      aggregationQueue
      processAggregated
      (trace, counters, aggregatedMap)
    (\_ → pure ())
  processAggregated lo@(LogObject logname lm _) (trace, counters, aggregatedMap) = do
    (updatedMap, aggregations) ← update lo aggregatedMap trace

```

```

unless (null aggregations) $
  sendAggregated trace (LogObject logname lme (AggregatedMessage aggregations))
-- increase the counter for the specific severity and message type
modifyMVar_counters $ \cnt → return $ updateMessageCounters cnt lo
return (trace, counters, updatedMap)

createNupdate :: Text → Measurable → LOMeta → AggregationMap → IO (Either Text Aggregated)
createNupdate name value lme agmap = do
  case HM.lookup name agmap of
    Nothing → do
      -- if Aggregated does not exist; initialize it.
      aggregatedKind ← getAggregatedKind conf name
      case aggregatedKind of
        StatsAK → return $ Right $ singletonStats value
        EwmaAK aEWMA → do
          let initEWMA = EmptyEWMA aEWMA
          return $ AggregatedEWMA < $ > ewma initEWMA value
      Just a → return $ updateAggregation value (aeAggregated a) lme (aeResetAfter a)

update :: LogObject a
  → AggregationMap
  → Trace.Trace IO a
  → IO (AggregationMap, [(Text, Aggregated)])
update (LogObject logname lme (LogValue iname value)) agmap trace = do
  let fullname = logname <> "." <> iname
  eitherAggregated ← createNupdate fullname value lme agmap
  case eitherAggregated of
    Right aggregated → do
      now ← getMonotonicTimeNSec
      let aggregatedX = AggregatedExpanded {
          aeAggregated = aggregated
          , aeResetAfter = Nothing
          , aeLastSent = now
        }
      namedAggregated = [(iname, aeAggregated aggregatedX)]
      updatedMap = HM.alter (const $ Just $ aggregatedX) fullname agmap
      return (updatedMap, namedAggregated)
    Left w → do
      let trace' = Trace.appendName "update" trace
      Trace.traceNamedObject trace' ≪
        (,) < $ > liftIO (mkLOMeta Warning Public)
        < * > pure (LogError w)
      return (agmap, [])

update (LogObject logname lme (ObserveDiff counterState)) agmap trace =
  updateCounters (csCounters counterState) lme (logname, "diff") agmap [ ] trace
update (LogObject logname lme (ObserveOpen counterState)) agmap trace =
  updateCounters (csCounters counterState) lme (logname, "open") agmap [ ] trace
update (LogObject logname lme (ObserveClose counterState)) agmap trace =
  updateCounters (csCounters counterState) lme (logname, "close") agmap [ ] trace
update (LogObject logname lme (LogMessage _)) agmap trace = do
  let iname = pack $ show (severity lme)
  let fullname = logname <> "." <> iname

```

```

eitherAggregated ← createNupdate fullname (PureI 0) lme agmap
case eitherAggregated of
  Right aggregated → do
    now ← getMonotonicTimeNSec
    let aggregatedX = AggregatedExpanded {
      aeAggregated = aggregated
      ,aeResetAfter = Nothing
      ,aeLastSent = now
    }
    namedAggregated = [(iname, aeAggregated aggregatedX)]
    updatedMap = HM.alter (const $ Just $ aggregatedX) fullname agmap
    return (updatedMap, namedAggregated)
  Left w → do
    let trace' = Trace.appendName "update" trace
    Trace.traceNamedObject trace' ≪
      (,) <$> liftIO (mkLOMeta Warning Public)
      <*> pure (LogError w)
    return (agmap, [])
-- everything else
update _ agmap _ = return (agmap, [])
updateCounters :: [Counter]
  → LOMeta
  → (LoggerName, LoggerName)
  → AggregationMap
  → [(Text, Aggregated)]
  → Trace.Trace IO a
  → IO (AggregationMap, [(Text, Aggregated)])
updateCounters [] _ _ aggrMap aggs _ = return $ (aggrMap, aggs)
updateCounters (counter : cs) lme (logname, msgname) aggrMap aggs trace = do
  let name = cName counter
      subname = msgname <> "." <> (nameCounter counter) <> "." <> name
      fullname = logname <> "." <> subname
      value = cValue counter
  eitherAggregated ← createNupdate fullname value lme aggrMap
  case eitherAggregated of
    Right aggregated → do
      now ← getMonotonicTimeNSec
      let aggregatedX = AggregatedExpanded {
        aeAggregated = aggregated
        ,aeResetAfter = Nothing
        ,aeLastSent = now
      }
      namedAggregated = (subname, aggregated)
      updatedMap = HM.alter (const $ Just $ aggregatedX) fullname aggrMap
      updateCounters cs lme (logname, msgname) updatedMap (namedAggregated : aggs) trace
    Left w → do
      let trace' = Trace.appendName "updateCounters" trace
      Trace.traceNamedObject trace' ≪
        (,) <$> liftIO (mkLOMeta Warning Public)
        <*> pure (LogError w)
      updateCounters cs lme (logname, msgname) aggrMap aggs trace

```

```

sendAggregated :: Trace.Trace IO a → LogObject a → IO ()
sendAggregated trace (LogObject logname meta v@(AggregatedMessage _)) = do
  -- enter the aggregated message into the Trace
  let trace' = Trace.appendName logname trace
  liftIO $ Trace.traceNamedObject trace' (meta, v)
  -- ignore every other message
sendAggregated _ _ = return ()

```

Update aggregation

We distinguish an uninitialized from an already initialized aggregation. The latter is properly initialized.

We use Welford's online algorithm to update the estimation of mean and variance of the sample statistics. (see [https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's](https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_algorithm)

```

updateAggregation :: Measurable → Aggregated → LOMeta → Maybe Word64 → Either Text Aggregated
updateAggregation v (AggregatedStats s) lme resetAfter =
  let count = fcount (fbasic s)
      reset = maybe False (count ≥) resetAfter
  in
  if reset
  then
    Right $ singletonStats v
  else
    Right $ AggregatedStats $! Stats {flast = v
      ,fold = mkTimestamp
      ,fbasic = updateBaseStats 1 v (fbasic s)
      ,fdelta = updateBaseStats 2 deltav (fdelta s)
      ,ftimed = updateBaseStats 2 timediff (ftimed s)
      }
  where
    deltav = subtractMeasurable v (flast s)
    mkTimestamp = Nanoseconds $ utc2ns (tstamp lme)
    timediff = Nanoseconds $ fromInteger $ (getInteger mkTimestamp) - (getInteger $ fold s)
updateAggregation v (AggregatedEWMA e) _ _ =
  let !eitherAvg = ewma e v
  in
    AggregatedEWMA < $ > eitherAvg
updateBaseStats :: Word64 → Measurable → BaseStats → BaseStats
updateBaseStats startAt v s =
  let newcount = fcount s + 1 in
  if (startAt > newcount)
  then s {fcount = fcount s + 1}
  else
    let newcountRel = newcount - startAt + 1
        newvalue = getDouble v
        delta = newvalue - fsum_A s
        dincr = (delta / fromIntegral newcountRel)
        delta2 = newvalue - fsum_A s - dincr
        (minim, maxim) =
          if startAt ≡ newcount

```

```

    then (v, v)
    else (min v (fmin s), max v (fmax s))
in
  BaseStats {fmin = minim
            ,fmax      = maxim
            ,fcount = newcount
            ,fsum_A = fsum_A s + dincr
            ,fsum_B = fsum_B s + (delta * delta2)
            }

```

Calculation of EWMA

Following https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average we calculate the exponential moving average for a series of values Y_t according to:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

The pattern matching below ensures that the **EWMA** will start with the first value passed in, and will not change type, once determined.

```

ewma :: EWMA → Measurable → Either Text EWMA
ewma (EmptyEWMA a) v = Right $ EWMA a v
ewma (EWMA a s@(Microseconds _)) y@(Microseconds _) =
  Right $ EWMA a $ Microseconds $ round $ a * (getDouble y) + (1 - a) * (getDouble s)
ewma (EWMA a s@(Seconds _)) y@(Seconds _) =
  Right $ EWMA a $ Seconds $ round $ a * (getDouble y) + (1 - a) * (getDouble s)
ewma (EWMA a s@(Bytes _)) y@(Bytes _) =
  Right $ EWMA a $ Bytes $ round $ a * (getDouble y) + (1 - a) * (getDouble s)
ewma (EWMA a (PureI s)) (PureI y) =
  Right $ EWMA a $ PureI $ round $ a * (fromInteger y) + (1 - a) * (fromInteger s)
ewma (EWMA a (PureD s)) (PureD y) =
  Right $ EWMA a $ PureD $ a * y + (1 - a) * s
ewma _ _ = Left "EWMA: Cannot compute average on values of different types"

```

1.7.33 Cardano.BM.Backend.Monitoring

Structure of Monitoring

```

type MonitorMVar a = MVar (MonitorInternal a)
newtype Monitor a = Monitor
  { getMon :: MonitorMVar a }
data MonitorInternal a = MonitorInternal
  { monQueue :: TBQ.TBQueue (Maybe (LogObject a))
  , monDispatch :: Async.Async ()
  , monBuffer :: LogBuffer a
  }

```

Relation from context name to monitoring state

We remember the state of each monitored context name.

```
data MonitorState = MonitorState {
  _preCondition :: MEvPreCond
  , _expression  :: MEvExpr
  , _actions     :: [MEvAction]
  , _environment :: Environment
} deriving Show
type MonitorMap = HM.HashMap LoggerName MonitorState
```

Monitor view is an effectuator

Function *effectuate* is called to pass in a **LogObject** for monitoring.

```
instance IsEffectuator Monitor a where
  effectuate monitor item = do
    mon ← readMVar (getMon monitor)
    effectuate (monBuffer mon) item
    nocapacity ← atomically $ TBQ.isFullTBQueue (monQueue mon)
    if nocapacity
    then handleOverflow monitor
    else atomically $ TBQ.writeTBQueue (monQueue mon) $ Just item
  handleOverflow _ = TIO.hPutStrLn stderr "Notice: Monitor's queue full, dropping log items!"
```

Monitor implements Backend functions

Monitor is an **IsBackend**

```
instance FromJSON a => IsBackend Monitor a where
  typeOf _ = MonitoringBK
  realize _ = fail "Monitoring cannot be instantiated by 'realize'"
  realizefrom config sbtrace _ = do
    monref ← newEmptyMVar
    let monitor = Monitor monref
  # ifdef PERFORMANCE_TEST_QUEUE
    let qSize = 1000000
  # else
    let qSize = 512
  # endif
  queue ← atomically $ TBQ.newTBQueue qSize
  dispatcher ← spawnDispatcher queue config sbtrace monitor
  monbuf :: Cardano.BM.Backend ◦ LogBuffer.LogBuffer a ← Cardano.BM.Backend ◦ LogBuffer.realize
  -- link the given Async to the current thread, such that if the Async
  -- raises an exception, that exception will be re-thrown in the current
  -- thread, wrapped in ExceptionInLinkedThread.
  Async.link dispatcher
  putMVar monref $ MonitorInternal
    { monQueue = queue
    , monDispatch = dispatcher
```

```

    ,monBuffer = monbuf
  }
  return monitor
unrealize monitoring = do
  let clearMVar :: MVar b → IO ()
      clearMVar = void ∘ tryTakeMVar
  (dispatcher, queue) ← withMVar (getMon monitoring) (λmon →
    return (monDispatch mon, monQueue mon))
  -- send terminating item to the queue
  atomically $ TBQ.writeTBQueue queue Nothing
  -- wait for the dispatcher to exit
  res ← Async.waitCatch dispatcher
  either throwM return res
  clearMVar $ getMon monitoring

```

Asynchronously reading log items from the queue and their processing

```

spawnDispatcher :: TBQ.TBQueue (Maybe (LogObject a))
    → Configuration
    → Trace.Trace IO a
    → Monitor a
    → IO (Async.Async ())
spawnDispatcher mqueue config sbtrace monitor = do
  now ← getcurrentTime
  let messageCounters = resetCounters now
  countersMVar ← newMVar messageCounters
  _timer ← Async.async $ sendAndResetAfter
    sbtrace
    "#messagecounters.monitoring"
    countersMVar
  60000 -- 60000 ms = 1 min
  Debug
  Async.async (initMap ≫ qProc countersMVar)
where
  {- lazy qProc -}
  qProc counters state = do
    processQueue
      mqueue
      processMonitoring
        (counters, state)
    (\_ → pure ())
  processMonitoring lo@(LogObject _ _ _) (counters, state) = do
    let accessBufferMap = do
      mon ← tryReadMVar (getMon monitor)
      case mon of
        Nothing → return []
        Just actualMon → readBuffer $ monBuffer actualMon
    mbuf ← accessBufferMap
    let sbtraceWithMonitoring = Trace.appendName "#monitoring" sbtrace
    valuesForMonitoring ← getVarValuesForMonitoring config mbuf

```



```

state' ← evalMonitoringAction sbtraceWithMonitoring
      state
      lo
      valuesForMonitoring
-- increase the counter for the type of message
modifyMVar counters $ \cnt → return $ updateMessageCounters cnt lo
return (counters, state')

initMap = do
  ls ← getMonitors config
  return $ HM.fromList $ map (\(n, (precond, e, as)) → (n, MonitorState precond e as HM.empty))
    $ HM.toList ls

getVarValuesForMonitoring :: Configuration
→ [(LoggerName, LogObject a)]
→ IO [(VarName, Measurable)]
getVarValuesForMonitoring config mbuf = do
  -- Here we take all var names for all monitors, just in case.
  monitorsInfo ← HM.elems < $ > getMonitors config
  let varNames = concat [extractVarNames mEvExpr | (_, mEvExpr, _) ← monitorsInfo]
  return ∘ catMaybes ∘ concat $ map (getVNnVal varNames) mbuf
where
  extractVarNames expr = case expr of
    Compare vn _ → [vn]
    AND e1 e2 → extractVarNames e1 ++ extractVarNames e2
    OR e1 e2 → extractVarNames e1 ++ extractVarNames e2
    NOT e → extractVarNames e
  getVNnVal varNames logObj = case logObj of
    (_, LogObject _ (LogValue vn val)) → if vn ∈ varNames
      then [Just (vn, val)]
      else []
    (_, LogObject _ (AggregatedMessage agg)) → concat $ map getMeasurable agg
    (_, _) → []
  where
    getMeasurable :: (Text, Aggregated) → [Maybe (VarName, Measurable)]
    getMeasurable agg = case agg of
      (vn, AggregatedEWMA (EWMA _ val)) → if vn ∈ varNames
        then [Just (vn <> ".ewma.avg", val)]
        else []
      (vn, AggregatedStats st) → if vn ∈ varNames
        then stValues vn st
        else []
      _ → []
  where
    stValues vn st =
      [Just (vn <> ".flast", flast st)
      , Just (vn <> ".fold", fold st)
      , Just (vn <> ".fbasic.fmin", fmin ∘ fbasic $ st)
      , Just (vn <> ".fbasic.fmax", fmax ∘ fbasic $ st)
      , Just (vn <> ".fbasic.mean", PureD ∘ meanOfStats ∘ fbasic $ st)
      , Just (vn <> ".fbasic.stdev", PureD ∘ stdevOfStats ∘ fbasic $ st)
      , Just (vn <> ".fbasic.fcount", PureI ∘ fromIntegral ∘ fcount ∘ fbasic $ st)

```

```

,Just (vn <> ".fdelta.fmin", fmin ∘ fdelta $ st)
,Just (vn <> ".fdelta.fmax", fmax ∘ fdelta $ st)
,Just (vn <> ".fdelta.mean", PureD ∘ meanOfStats ∘ fdelta $ st)
,Just (vn <> ".fdelta.stdev", PureD ∘ stdevOfStats ∘ fdelta $ st)
,Just (vn <> ".fdelta.fcount", PureI ∘ fromIntegral ∘ fcount ∘ fdelta $ st)
,Just (vn <> ".ftimed.fmin", fmin ∘ ftimed $ st)
,Just (vn <> ".ftimed.fmax", fmax ∘ ftimed $ st)
,Just (vn <> ".ftimed.mean", PureD ∘ meanOfStats ∘ ftimed $ st)
,Just (vn <> ".ftimed.stdev", PureD ∘ stdevOfStats ∘ ftimed $ st)
,Just (vn <> ".ftimed.fcount", PureI ∘ fromIntegral ∘ fcount ∘ ftimed $ st)
]

```

Evaluation of monitoring action

Inspect the log message and match it against configured thresholds. If positive, then run the action on the current state and return the updated state.

```

evalMonitoringAction :: Trace.Trace IO a
  → MonitorMap
  → LogObject a
  → [(VarName, Measurable)]
  → IO MonitorMap
evalMonitoringAction sbtrace mmap logObj@(LogObject logname0 _ content) variables = do
  let logname = case content of
    ObserveOpen _ → logname0 <> ".open"
    ObserveDiff _ → logname0 <> ".diff"
    ObserveClose _ → logname0 <> ".close"
    _ → logname0
  let sbtrace' = Trace.appendName logname sbtrace
  case HM.lookup logname mmap of
    Nothing → return mmap
    Just mon@(MonitorState precondition expr acts env0) → do
      let env1 = updateEnv env0 logObj
      let env' = HM.union env1 $ HM.fromList variables
      let doMonitor = case precondition of
        -- There's no precondition, do monitor as usual.
        Nothing → True
        -- Precondition is defined, do monitor only if it is True.
        Just preCondExpr → evaluate env' preCondExpr
      -- In this place env' already must contain opvn..
      let thresholdIsReached = evaluate env' expr
      if doMonitor ∧ thresholdIsReached then do
        now ← getMonotonicTimeNSec
        let env'' = HM.insert "lastalert" (Nanoseconds now) env'
        mapM_ (evaluateAction sbtrace' env' expr) acts
        return $ HM.insert logname mon { _environment = env'' } mmap
      else return mmap
  where
    updateEnv env (LogObject lname lometata (ObserveOpen (CounterState counters))) =
      let addenv = HM.fromList $ ("timestamp", Nanoseconds $ utc2ns (tstamp lometata))
        : countersEnvPairs (lname <> ".open") counters

```

```

    in
      HM.union addenv env
    updateEnv env (LogObject lname lmeta (ObserveDiff (CounterState counters))) =
      let addenv = HM.fromList $ ("timestamp", Nanoseconds $ utc2ns (tstamp lmeta))
        : countersEnvPairs (lname <> ".diff") counters
    in
      HM.union addenv env
    updateEnv env (LogObject lname lmeta (ObserveClose (CounterState counters))) =
      let addenv = HM.fromList $ ("timestamp", Nanoseconds $ utc2ns (tstamp lmeta))
        : countersEnvPairs (lname <> ".close") counters
    in
      HM.union addenv env
    updateEnv env (LogObject _ lmeta (LogValue vn val)) =
      let addenv = HM.fromList [ (vn, val)
        , ("timestamp", Nanoseconds $ utc2ns (tstamp lmeta))
        ]
    in
      HM.union addenv env
    updateEnv env (LogObject _ lmeta (LogMessage _ logitem)) =
      let addenv = HM.fromList [ ("severity", (Severity (severity lmeta)))
        -- , ("selection", (liSelection logitem))
        -- , ("message", (liPayload logitem))
        , ("timestamp", Nanoseconds $ utc2ns (tstamp lmeta))
        ]
    in
      HM.union addenv env
    updateEnv env (LogObject _ lmeta (AggregatedMessage vals)) =
      let addenv = ("timestamp", Nanoseconds $ utc2ns (tstamp lmeta)) : aggs2measurables vals []
    in
      HM.union (HM.fromList addenv) env
  where
    aggs2measurables [] acc = acc
    aggs2measurables ((n, AggregatedEWMA ewma) : r) acc = aggs2measurables r $ (n <> ".avg", avg ewma)
    aggs2measurables ((n, AggregatedStats s) : r) acc = aggs2measurables r $
      (n <> ".mean", PureD ◦ meanOfStats $ fbasic s)
      : (n <> ".flast", flast s)
      : (n <> ".fcount", PureI ◦ fromIntegral ◦ fcount $ fbasic s)
      : acc
  -- catch all
  updateEnv env _ = env
  countersEnvPairs loggerName = map $ \counter →
    let name = loggerName <> "." <> (nameCounter counter) <> "." <> cName counter
      value = cValue counter
    in
      (name, value)
  evaluateAction sbtrace' env expr (CreateMessage sev alertMessage) = do
    lmeta ← mkLOMeta sev Public
    let fullMessage = alertMessage
      <> "; environment is: " <> pack (show env)
      <> "; threshold expression is: " <> pack (show expr)
    Trace.traceNamedObject sbtrace' (lmeta, MonitoringEffect (MonitorAlert fullMessage))

```

```

evaluateAction sbtrace' _ _ (SetGlobalMinimalSeverity sev) = do
  lometa ← mkLOMeta sev Public
  Trace.traceNamedObject sbtrace' (lometa, MonitoringEffect (MonitorAlterGlobalSeverity sev))
evaluateAction sbtrace' _ _ (AlterSeverity loggerName sev) = do
  lometa ← mkLOMeta sev Public
  Trace.traceNamedObject sbtrace' (lometa, MonitoringEffect (MonitorAlterSeverity loggerName sev))

```

1.7.34 Cardano.BM.Backend.Prometheus

Spawn Prometheus client from existing EKG server

```

spawnPrometheus :: EKG.Server → Port → IO (Async.Async ())
spawnPrometheus s p = Async.async $ passToPrometheus s p
passToPrometheus :: EKG.Server → Port → IO ()
passToPrometheus server port =
  let store = EKG.serverMetricStore server
      reg = execRegistryT $ registerEKGStore store $ AdapterOptions mempty Nothing 1
      in serveMetrics (reg >> sample)
where
  serveMetrics :: MonadIO m ⇒ IO RegistrySample → m ()
  serveMetrics = liftIO ∘ runSettings settings ∘ prometheusApp [ "metrics" ]
  settings = setPort port ∘ setHost "127.0.0.1" $ defaultSettings

```

1.7.35 Cardano.BM.Backend.ExternalAbstraction

Abstraction for the communication between *ExternalLogBK* and *LogToPipeBK* backends.

```

class Pipe p where
  data family PipeHandler p
  create :: FilePath → IO (PipeHandler p)
  open   :: FilePath → IO (PipeHandler p)
  close  :: PipeHandler p → IO ()
  write  :: PipeHandler p → BS.ByteString → IO ()
  getLine :: PipeHandler p → IO BS.ByteString

data NoPipe
data UnixNamedPipe

instance Pipe NoPipe where
  data PipeHandler NoPipe = NP ()
  create = \_ → pure $ NP ()
  open   = \_ → pure $ NP ()
  close  = \_ → pure ()
  write  = \_ → pure ()
  getLine = \_ → pure ""

instance Pipe UnixNamedPipe where
  data PipeHandler UnixNamedPipe = P Handle
  # ifndef mingw32_HOST_OS
  create pipePath =
    (createNamedPipe pipePath stdFileMode >> (P < $ > openFile pipePath ReadWriteMode))

```

```

-- use of ReadWriteMode instead of ReadMode in order
-- EOF not to be written at the end of file
'catch' (λ(e :: SomeException) →
  case fromException e of
    Just (IOError _ AlreadyExists _ _ _) →
      P < $ > openFile pipePath ReadWriteMode
    _ → do
      hPutStrLn stderr $ "Creating pipe threw: " ++ show e
      throw e)
# else
  create _ = error "UnixNamedPipe not supported on Windows"
# endif
open pipePath = do
  h ← openFile pipePath WriteMode
  'catch' (λ(e :: SomeException) → do
    hPutStrLn stderr $ "Opening pipe threw: " ++ show e
    ++ "\nForwarding its objects to stderr"
    hDuplicate stderr)
  hSetBuffering h NoBuffering
  return $ P h
close (P h) = hClose h 'catch' (λ(_ :: SomeException) → pure ())
getLine (P h) = BS.hGetLine h
write (P h) bs = BSC.hPutStrLn h $! bs

```

1.7.36 Cardano.BM.Backend.TraceAcceptor

TraceAcceptor is a backend responsible for processing **LogObjects** of an external process captured by a pipe or socket. At the time being it redirects the **LogObjects** to the *SwitchBoard*.

Structure of TraceAcceptor

```

newtype TraceAcceptor p a = TraceAcceptor
  { getTA :: TraceAcceptorMVar p a }
type TraceAcceptorMVar p a = MVar (TraceAcceptorInternal p a)
data TraceAcceptorInternal p a = TraceAcceptorInternal
  { accPipe :: PipeHandler p
    , accDispatch :: Async.Async ()
  }
effectuate :: LogObject a → IO ()
effectuate = \_ → return ()
realizefrom :: (Pipe p, FromJSON a)
  ⇒ Trace.Trace IO a → FilePath → IO (TraceAcceptor p a)
realizefrom sbtrace pipePath = do
  elref ← newEmptyMVar
  let externalLog = TraceAcceptor elref
  h ← create pipePath
  dispatcher ← spawnDispatcher h sbtrace
  -- link the given Async to the current thread, such that if the Async
  -- raises an exception, that exception will be re-thrown in the current
  -- thread, wrapped in ExceptionInLinkedThread.

```

```

Async.link dispatcher
putMVar elref $ TraceAcceptorInternal
  {accPipe = h
   ,accDispatch = dispatcher
  }
return externalLog
unrealize :: Pipe p => TraceAcceptor p a -> IO ()
unrealize accView = withMVar (getTA accView) (\acc -> do
  Async.cancel $ accDispatch acc
  let hPipe = accPipe acc
  -- close the pipe
  close hPipe)

```

Reading log items from the pipe

```

spawnDispatcher :: (Pipe p, FromJSON a)
  => PipeHandler p
  -> Trace.Trace IO a
  -> IO (Async.Async ())
spawnDispatcher hPipe sbtrace = do
  Async.async $ pProc hPipe
where
  {- lazy pProc -}
  pProc h = do
    bs <- CH.getLine h
    if not (BS.null bs)
    then do
      case decodeStrict bs of
        Just lo ->
          traceWith sbtrace lo
        Nothing -> do
          let trace = Trace.appendName "#external" sbtrace
              Trace.traceNamedObject trace <-<
                (,) <$> (mkLOMeta Warning Public)
              <*> pure (LogError "Could not parse external log objects.")
          pProc h
    else return () -- stop here

```

1.7.37 Cardano.BM.Backend.TraceForwarder

TraceForwarder is a new backend responsible for redirecting the logs into a pipe or a socket to be used from another application. It puts **LogObjects** as *ByteStrings* in the provided handler.

Structure of TraceForwarder

Contains the handler to the pipe or to the socket.

```

newtype TraceForwarder p a = TraceForwarder
  {getTF :: TraceForwarderMVar p a}
type TraceForwarderMVar p a = MVar (TraceForwarderInternal p a)

```

```
data Pipe p  $\Rightarrow$  TraceForwarderInternal p a =
  TraceForwarderInternal
    { tfPipeHandler :: PipeHandler p
    }
```

TraceForwarder is an effectuator

Every **LogObject** before being written to the given handler is converted to *ByteString* through its *JSON* representation.

```
instance (Pipe p, ToJSON a)  $\Rightarrow$  IsEffectuator (TraceForwarder p) a where
  effectuate tf lo =
    withMVar (getTF tf) $  $\lambda$ (TraceForwarderInternal h)  $\rightarrow$ 
      let (_, bs) = jsonToBS lo
      in
        write h bs
        handleOverflow _ = return ()
  jsonToBS :: ToJSON a  $\Rightarrow$  a  $\rightarrow$  (Int, BS.ByteString)
  jsonToBS a =
    let bs = BL.toStrict $ encode a
    in (BS.length bs, bs)
```

TraceForwarder implements Backend functions

TraceForwarder is an IsBackend

```
instance (Pipe p, FromJSON a, ToJSON a)  $\Rightarrow$  IsBackend (TraceForwarder p) a where
  typeOf _ = TraceForwarderBK
  realize cfg = do
    ltpref  $\leftarrow$  newEmptyMVar
    let logToPipe = TraceForwarder ltpref
    pipePath  $\leftarrow$  fromMaybe "log-pipe" <$> getLogOutput cfg
    h  $\leftarrow$  open pipePath
    putMVar ltpref $ TraceForwarderInternal
      { tfPipeHandler = h
      }
    return logToPipe
  unrealize tf = withMVar (getTF tf) ( $\lambda$ (TraceForwarderInternal h)  $\rightarrow$ 
    -- close the pipe
    close h
    'catch' ( $\lambda$ (_ :: SomeException)  $\rightarrow$  pure ()))
```

Chapter 2

Testing

2.1 Test main entry point

```
{-# LANGUAGE CPP #-}
module Main
  (
    main
  ) where
import Test.Tasty
# ifdef ENABLE_AGGREGATION
import qualified Cardano.BM.Test.Aggregated (tests)
# endif
import qualified Cardano.BM.Test.STM (tests)
import qualified Cardano.BM.Test.Trace (tests)
import qualified Cardano.BM.Test.Configuration (tests)
import qualified Cardano.BM.Test.LogItem (tests)
import qualified Cardano.BM.Test.Rotator (tests)
import qualified Cardano.BM.Test.Routing (tests)
import qualified Cardano.BM.Test.Structured (tests)
import qualified Cardano.BM.Test.Tracer (tests)
# ifdef ENABLE_MONITORING
import qualified Cardano.BM.Test.Monitoring (tests)
# endif
main :: IO ()
main = defaultMain tests
tests :: TestTree
tests =
  testGroup "iohk-monitoring"
  [
    # ifdef ENABLE_AGGREGATION
      Cardano.BM.Test ◦ Aggregated.tests
    ,
    # endif
      Cardano.BM.Test ◦ STM.tests
    , Cardano.BM.Test ◦ Trace.tests
    , Cardano.BM.Test ◦ Configuration.tests
    , Cardano.BM.Test ◦ LogItem.tests
```



```

    ,Cardano.BM.Test ◦ Rotator.tests
    ,Cardano.BM.Test ◦ Routing.tests
    ,Cardano.BM.Test ◦ Structured.tests
    ,Cardano.BM.Test ◦ Tracer.tests
# ifdef ENABLE_MONITORING
    ,Cardano.BM.Test ◦ Monitoring.tests
# endif
]

```

2.2 Test case generation

2.2.1 instance Arbitrary Aggregated

We define an instance of *Arbitrary* for an **Aggregated** which lets *QuickCheck* generate arbitrary instances of **Aggregated**. For this an arbitrary list of *Integer* is generated and this list is aggregated into a structure of **Aggregated**.

```

instance Arbitrary Aggregated where
  arbitrary = do
    vs' ← arbitrary :: Gen [Integer]
    let vs = 42 : 17 : vs'
        ds = map (\(a,b) → a - b) $ zip vs (tail vs)
        (m1,s1) = updateMeanVar $ map fromInteger vs
        (m2,s2) = updateMeanVar $ map fromInteger ds
        mkBasicStats = BaseStats
            (PureI (minimum vs))
            (PureI (maximum vs))
            (fromIntegral $ length vs)
            (m1)
            (s1)
        mkDeltaStats = BaseStats
            (PureI (minimum ds))
            (PureI (maximum ds))
            (fromIntegral $ length ds)
            (m2)
            (s2)
        mkTimedStats = BaseStats
            (Nanoseconds 0)
            (Nanoseconds 0)
            (0)
            (0)
            (0)
    return $ AggregatedStats (Stats
        (PureI (last vs))
        (Nanoseconds 0)
        mkBasicStats
        mkDeltaStats
        mkTimedStats)

```

Estimators for mean and variance must be updated the same way as in the code.

```

updateMeanVar :: [Double] → (Double, Double)
updateMeanVar [] = (0, 0)

```

```

updateMeanVar (val : vals) = updateMeanVar' (val, 0) 1 vals
  where
    updateMeanVar' (m, s) _ [] = (m, s)
    updateMeanVar' (m, s) cnt (a : r) =
      let delta = a - m
          newcount = cnt + 1
          m' = m + (delta / newcount)
          s' = s + (delta * (a - m'))
      in
        updateMeanVar' (m', s') newcount r

```

2.3 Tests

2.3.1 Testing aggregation

```

tests :: TestTree
tests = testGroup "Aggregation measurements" [
  propertyTests
  , unitTests1
  , unitTests2
]

propertyTests :: TestTree
propertyTests = testGroup "Properties" [
  testProperty "minimal" prop_Aggregation_minimal
  , testProperty "commutative" prop_Aggregation_comm
]

unitTests1 :: TestTree
unitTests1 = testGroup "Unit tests for Aggregated" [
  testCase "compare equal >" unitAggregatedEqualGT
  , testCase "compare equal <" unitAggregatedEqualLT
  , testCase "compare different >" unitAggregatedDiffGT
  , testCase "compare different <" unitAggregatedDiffLT
]

unitTests2 :: TestTree
unitTests2 = testGroup "Unit tests for Aggregation" [
  testCase "initial -1" unitAggregationInitialMinus1
  , testCase "initial +1" unitAggregationInitialPlus1
  , testCase "initial +0" unitAggregationInitialZero
  , testCase "initial +1, -1" unitAggregationInitialPlus1Minus1
  , testCase "stepwise" unitAggregationStepwise
]

```

Property tests

```

prop_Aggregation_minimal :: Bool
prop_Aggregation_minimal = True

lometa :: LOMeta
lometa = unsafePerformIO $ mkLOMeta Debug Public

```

```

prop_Aggregation_comm :: Integer → Integer → Aggregated → Property
prop_Aggregation_comm v1 v2 ag =
  let Right agg2 = updateAggregation (PureI v2) ag lometa Nothing
      Right agg1 = updateAggregation (PureI v1) ag lometa Nothing
      Right (AggregatedStats stats21) = updateAggregation (PureI v1) agg2 lometa Nothing
      Right (AggregatedStats stats12) = updateAggregation (PureI v2) agg1 lometa Nothing
  in
    fbasic stats21 === fbasic stats12.&&.
    (v1 == v2) 'implies' (flast stats21 === flast stats12)
-- implication: if p1 is true, then return p2; otherwise true
implies :: Bool → Property → Property
implies p1 p2 = property (¬ p1) .|. p2

```

Unit tests for Aggregation

```

unitAggregationInitialMinus1 :: Assertion
unitAggregationInitialMinus1 = do
  let Right (AggregatedStats stats1) = updateAggregation (-1) firstStateAggregatedStats lometa Nothing
      flast stats1 @? = (-1)
      (fbasic stats1) @? = BaseStats (-1) 0 2 (-0.5) 0.5
      (fdelta stats1) @? = BaseStats (-1) (-1) 2 (-1) 0
      -- AggregatedStats (Stats (-1) x (BaseStats (-1) 0 2 (-0.5) 0.5) (BaseStats (-1)
unitAggregationInitialPlus1 :: Assertion
unitAggregationInitialPlus1 = do
  let Right (AggregatedStats stats1) = updateAggregation 1 firstStateAggregatedStats lometa Nothing
      flast stats1 @? = 1
      (fbasic stats1) @? = BaseStats 0 1 2 0.5 0.5
      (fdelta stats1) @? = BaseStats 1 1 2 1 0
      -- AggregatedStats (Stats 1 x (BaseStats 0 1 2 0.5 0.5) (BaseStats 1 1 2 1 0) (B
unitAggregationInitialZero :: Assertion
unitAggregationInitialZero = do
  let Right (AggregatedStats stats1) = updateAggregation 0 firstStateAggregatedStats lometa Nothing
      flast stats1 @? = 0
      (fbasic stats1) @? = BaseStats 0 0 2 0 0
      (fdelta stats1) @? = BaseStats 0 0 2 0 0
      -- AggregatedStats (Stats 0 x (BaseStats 0 0 2 0 0) (BaseStats 0 0 2 0 0) (BaseS
unitAggregationInitialPlus1Minus1 :: Assertion
unitAggregationInitialPlus1Minus1 = do
  let Right agg1 = updateAggregation (PureI 1) firstStateAggregatedStats lometa Nothing
      Right (AggregatedStats stats1) = updateAggregation (PureI (-1)) agg1 lometa Nothing
      (fbasic stats1) @? = BaseStats (PureI (-1)) (PureI 1) 3 0.0 2.0
      (fdelta stats1) @? = BaseStats (PureI (-2)) (PureI 1) 3 (-0.5) 4.5
unitAggregationStepwise :: Assertion
unitAggregationStepwise = do
  stats0 ← pure $ singletonStats (Bytes 3000)
  -- putStrLn (show stats0)
  threadDelay 50000 -- 0.05 s
  t1 ← mkLOMeta Debug Public
  Right stats1 ← pure $ updateAggregation (Bytes 5000) stats0 t1 Nothing
  -- putStrLn (show stats1)

```

```

-- showTimedMean stats1
threadDelay 50000-- 0.05 s
t2 ← mkLOMeta Debug Public
Right stats2 ← pure $ updateAggregation (Bytes 1000) stats1 t2 Nothing
-- putStrLn (show stats2)
-- showTimedMean stats2
checkTimedMean stats2
threadDelay 50000-- 0.05 s
t3 ← mkLOMeta Debug Public
Right stats3 ← pure $ updateAggregation (Bytes 3000) stats2 t3 Nothing
-- putStrLn (show stats3)
-- showTimedMean stats3
checkTimedMean stats3
threadDelay 50000-- 0.05 s
t4 ← mkLOMeta Debug Public
Right stats4 ← pure $ updateAggregation (Bytes 1000) stats3 t4 Nothing
-- putStrLn (show stats4)
-- showTimedMean stats4
checkTimedMean stats4
where
checkTimedMean (AggregatedEWMA _) = return ()
checkTimedMean (AggregatedStats s) = do
  let mean = meanOfStats (ftimed s)
  assertBool "the mean should be >= the minimum" (mean ≥ getDouble (fmin (ftimed s)))
  assertBool "the mean should be <= the maximum" (mean ≤ getDouble (fmax (ftimed s)))

```

commented out:

```

showTimedMean (AggregatedEWMA _) = return ()
showTimedMean (AggregatedStats s) = putStrLn $ "mean = " ++ show (meanOfStats (ftimed s))
  ++ showUnits (fmin (ftimed s))

```

```

firstStateAggregatedStats :: Aggregated
Stats
  z
  z'
  (BaseStats z z 1 0 0)
  (BaseStats z z 1 0 0)
  (BaseStats z' z' 1 0 0)
where
  z = PureI 0
  z' = Nanoseconds 0

```

Unit tests for Aggregated

```

unitAggregatedEqualGT :: Assertion
unitAggregatedEqualGT = do
  assertBool "comparing seconds"
    ((Seconds 3) > (Seconds 2))

```

```

assertBool "comparing microseconds"
  ((Microseconds 3000) > (Microseconds 2000))
assertBool "comparing nanoseconds"
  ((Nanoseconds 3000000) > (Nanoseconds 2000000))
assertBool "comparing bytes"
  ((Bytes 2048) > (Bytes 1024))
assertBool "comparing doubles"
  ((PureD 2.34) > (PureD 1.42))
assertBool "comparing integers"
  ((PureI 2) > (PureI 1))
assertBool "comparing severities"
  ((Severity Error) > (Severity Warning))
unitAggregatedEqualLT :: Assertion
unitAggregatedEqualLT = do
  assertBool "comparing seconds"
    ((Seconds 2) < (Seconds 3))
  assertBool "comparing microseconds"
    ((Microseconds 2000) < (Microseconds 3000))
  assertBool "comparing nanoseconds"
    ((Nanoseconds 2000000) < (Nanoseconds 3000000))
  assertBool "comparing bytes"
    ((Bytes 1024) < (Bytes 2048))
  assertBool "comparing doubles"
    ((PureD 1.34) < (PureD 2.42))
  assertBool "comparing integers"
    ((PureI 1) < (PureI 2))
  assertBool "comparing severities"
    ((Severity Info) < (Severity Notice))
unitAggregatedDiffGT :: Assertion
unitAggregatedDiffGT = do
  assertBool "comparing time (s vs. s)"
    ((Microseconds 3000000) > (Seconds 2))
  assertBool "comparing time (s vs. ns)"
    ((Microseconds 30) > (Nanoseconds 29999))
  assertBool "comparing nanoseconds"
    ((Nanoseconds 3000000) > (Microseconds 2900))
  assertBool "comparing bytes"
    ((Bytes 2048) > (PureI 1024))
  assertBool "comparing doubles"
    ((PureD 2.34) > (PureI 1))
  assertBool "comparing integers"
    ((PureI 2) > (PureD 1.42))
unitAggregatedDiffLT :: Assertion
unitAggregatedDiffLT = do
  assertBool "comparing time (s vs. s)"
    ((Microseconds 2999999) < (Seconds 3))
  assertBool "comparing time (s vs. ns)"
    ((Microseconds 30) < (Nanoseconds 30001))
  assertBool "comparing nanoseconds"
    ((Nanoseconds 3000000) < (Microseconds 3001))
  assertBool "comparing bytes"

```

```

    ((PureI 1024) < (Bytes 2048))
  assertBool "comparing doubles"
    ((PureD 2.34) < (PureI 3))
  assertBool "comparing integers"
    ((PureI 2) < (PureD 3.42))

```

2.3.2 Cardano.BM.Test.STM

```

module Cardano.BM.Test.STM (
  tests
) where

import Test.Tasty
import Test.Tasty.QuickCheck

tests :: TestTree
tests = testGroup "Observing STM actions" [
  testProperty "minimal" prop_STM_observer
]

prop_STM_observer :: Bool
prop_STM_observer = True

```

2.3.3 Cardano.BM.Test.Trace

```

tests :: TestTree
tests = testGroup "Testing Trace" [
  unit_tests
  , testCase "forked traces stress testing" stressTraceInFork
# ifdef ENABLE_OBSERVABLES
  , testCase "stress testing: ObservableTraceSelf vs. NoTrace" timingObservableVsUntimed
# endif
  , testCaseInfo "demonstrating logging" simpleDemo
  , testCaseInfo "demonstrating nested named context logging" exampleWithNamedContexts
]

unit_tests :: TestTree
unit_tests = testGroup "Unit tests" [
  testCase "opening messages should not be traced" unitNoOpeningTrace
-- , testCase "hierarchy of traces" unitHierarchy
  , testCase "forked traces" unitTraceInFork
  , testCase "hierarchy of traces with NoTrace" $
    unitHierarchy' [Neutral, NoTrace, (ObservableTraceSelf observablesSet)]
    onlyLevelOneMessage
  , testCase "hierarchy of traces with DropOpening" $
    unitHierarchy' [Neutral, DropOpening, (ObservableTraceSelf observablesSet)]
    notObserveOpen
  , testCase "hierarchy of traces with UntimedTrace" $
    unitHierarchy' [Neutral, UntimedTrace, UntimedTrace]
    observeNoMeasures
  , testCase "changing the minimum severity of a trace at runtime"
    unitTraceMinSeverity

```

```

,testCase "changing the minimum severity of a named context at runtime"
  unitNamedMinSeverity
,testCase "appending names" unitAppendName
,testCase "create subtrace which duplicates messages" unitTraceDuplicate
,testCase "testing name filtering" unitNameFiltering
,testCase "testing throwing of exceptions" unitExceptionThrowing
,testCase "NoTrace: check lazy evaluation" unitTestLazyEvaluation
,testCase "private messages should not be logged into private files" unitLoggingPrivate
]
where
  observablesSet = [MonotonicClock, MemoryStats]
  notObserveOpen :: [LogObject a] → Bool
  notObserveOpen = all (λcase {LogObject _ _ (ObserveOpen _) → False; _ → True})
  notObserveClose :: [LogObject a] → Bool
  notObserveClose = all (λcase {LogObject _ _ (ObserveClose _) → False; _ → True})
  notObserveDiff :: [LogObject a] → Bool
  notObserveDiff = all (λcase {LogObject _ _ (ObserveDiff _) → False; _ → True})
  onlyLevelOneMessage :: [LogObject Text] → Bool
  onlyLevelOneMessage = λcase
    [LogObject _ _ (LogMessage "Message from level 1.")] → True
    _ → False
  observeNoMeasures :: [LogObject a] → Bool
  observeNoMeasures obs = notObserveOpen obs ∧ notObserveClose obs ∧ notObserveDiff obs

```

Helper routines

```

data TraceConfiguration = TraceConfiguration
  { tcConfig      :: Configuration
  , tcOutputKind :: MockSwitchboard Text
  , tcName        :: LoggerName
  , tcSubTrace    :: SubTrace
  }

setupTrace :: TraceConfiguration → IO (Trace IO Text)
setupTrace (TraceConfiguration cfg mockSB name subTr) = do
  let logTrace = traceMock mockSB cfg
  setSubTrace cfg name (Just subTr)
  return $ appendName name logTrace

```

Simple demo of logging.

```

simpleDemo :: IO String
simpleDemo = do
  cfg ← defaultConfigTesting
  logTrace :: Trace IO String ← Setup.setupTrace (Right cfg) "test"
  putStrLn "\n"
  logDebug logTrace "This is how a Debug message looks like."
  logInfo logTrace "This is how an Info message looks like."
  logNotice logTrace "This is how a Notice message looks like."

```

```

logWarning logTrace "This is how a Warning message looks like."
logError   logTrace "This is how an Error message looks like."
logCritical logTrace "This is how a Critical message looks like."
logAlert   logTrace "This is how an Alert message looks like."
logEmergency logTrace "This is how an Emergency message looks like."
return ""

```

Example of using named contexts with **Trace**

```

exampleWithNamedContexts :: IO String
exampleWithNamedContexts = do
  cfg ← defaultConfigTesting
  Setup.withTrace cfg "test" $ λ(logTrace :: Trace IO Text) → do
    putStrLn "\n"
    logInfo logTrace "entering"
    let logTrace0 = appendName "simple-work-0" logTrace
    work0 ← complexWork0 cfg logTrace0 "0"
    let logTrace1 = appendName "complex-work-1" logTrace
    work1 ← complexWork1 cfg logTrace1 "42"
    Async.wait work0
    Async.wait work1
    -- the named context will include "complex" in the logged message
    logInfo logTrace "done."
    threadDelay 100000
    -- force garbage collection to allow exceptions to be thrown
    performMajorGC
    threadDelay 100000
  return ""
where
  complexWork0 _ tr msg = Async.async $ logInfo tr ("let's see (0): " 'append' msg)
  complexWork1 cfg tr msg = Async.async $ do
    logInfo tr ("let's see (1): " 'append' msg)
    let trInner = appendName "inner-work-1" tr
    observablesSet = [ MonotonicClock ]
    setSubTrace cfg "test.complex-work-1.inner-work-1.STM-action" $
      Just $ ObservableTraceSelf observablesSet
  # ifdef ENABLE_OBSERVABLES
    _ ← STMObserver.bracketObserveIO cfg trInner Debug "STM-action" setVar_
  # endif
  logInfo trInner "let's see: done."

```

Show effect of turning off observables

```

# ifdef ENABLE_OBSERVABLES
runTimedAction :: Configuration → Trace IO Text → LoggerName → Int → IO Measurable
runTimedAction cfg logTrace name reps = do
  t0 ← getMonoClock
  forM_ [(1 :: Int)..reps] $ const $ observeAction logTrace

```



```

t1 ← getMonoClock
return $ diffTimeObserved (CounterState t0) (CounterState t1)
where
  observeAction trace = do
    _ ← MonadicObserver.bracketObserveIO cfg trace Debug name action
    return ()
  action = return $ forM [1 :: Int..100] $ \x → [x] ++ (init $ reverse [1 :: Int..10000])
  diffTimeObserved :: CounterState → CounterState → Measurable
  diffTimeObserved (CounterState startCounters) (CounterState endCounters) =
    let
      startTime = getMonotonicTime startCounters
      endTime = getMonotonicTime endCounters
    in
      endTime - startTime
  getMonotonicTime counters = case (filter isMonotonicClockCounter counters) of
    [(Counter MonotonicClockTime _ mus)] → mus
    _ → error "A time measurement is missing!"
  isMonotonicClockCounter :: Counter → Bool
  isMonotonicClockCounter = (MonotonicClockTime ≡) ∘ cType
  timingObservableVsUntimed :: Assertion
  timingObservableVsUntimed = do
    cfg1 ← defaultConfigTesting
    msgs1 ← STM.newTVarIO []
    traceObservable ← setupTrace $ TraceConfiguration cfg1
      (MockSB msgs1)
      "observables"
      (ObservableTraceSelf observablesSet)
    cfg2 ← defaultConfigTesting
    msgs2 ← STM.newTVarIO []
    traceUntimed ← setupTrace $ TraceConfiguration cfg2
      (MockSB msgs2)
      "no timing"
      UntimedTrace
    cfg3 ← defaultConfigTesting
    msgs3 ← STM.newTVarIO []
    traceNoTrace ← setupTrace $ TraceConfiguration cfg3
      (MockSB msgs3)
      "no trace"
      NoTrace
    t_observable ← runTimedAction cfg1 traceObservable "observables" 100
    t_untimed ← runTimedAction cfg2 traceUntimed "no timing" 100
    t_notrace ← runTimedAction cfg3 traceNoTrace "no trace" 100
    ms ← STM.readTVarIO msgs1
    assertBool
      ("Untimed consumed more time than ObservableTraceSelf " ++ (show [t_untimed, t_observable]
        (t_observable > t_untimed ∧ ¬ (null ms)))
    assertBool
      ("NoTrace consumed more time than ObservableTraceSelf" ++ (show [t_notrace, t_observable]
        (t_observable > t_notrace))
    assertBool

```

```

    ("NoTrace consumed more time than Untimed" ++ (show [t_notrace, t_untimed]))
  True
where
  observablesSet = [MonotonicClock, GhcRtsStats, MemoryStats, IOStats, ProcessStats]
# endif

```

Control tracing in a hierarchy of **Traces**

We can lay out traces in a hierarchical manner, that the children forward traced items to the parent **Trace**. A **NoTrace** introduced in this hierarchy will cut off a branch from messaging to the root.

```

_unitHierarchy :: Assertion
_unitHierarchy = do
  cfg ← defaultConfigTesting
  msgs ← STM.newTVarIO []
  basetrace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test" Neutral
  logInfo basetrace "This should have been displayed!"
  -- subtrace of trace which traces nothing
  setSubTrace cfg "test.inner" (Just NoTrace)
  let trace1 = appendName "inner" basetrace
  logInfo trace1 "This should NOT have been displayed!"
  setSubTrace cfg "test.inner.innermost" (Just Neutral)
  let trace2 = appendName "innermost" trace1
  logInfo trace2 "This should NOT have been displayed also due to the trace one level above"
  -- acquire the traced objects
  res ← STM.readTVarIO msgs
  -- only the first message should have been traced
  assertBool
    ("Found more or less messages than expected: " ++ show res)
    (length res == 1)

```

Change a trace's minimum severity

A trace is configured with a minimum severity and filters out messages that are labelled with a lower severity. This minimum severity of the current trace can be changed.

```

_unitTraceMinSeverity :: Assertion
_unitTraceMinSeverity = do
  cfg ← defaultConfigTesting
  msgs ← STM.newTVarIO []
  trace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test min severity" Neutral
  logInfo trace "Message #1"
  -- raise the minimum severity to Warning
  setMinSeverity cfg Warning
  msev ← Cardano.BM.Configuration.minSeverity cfg
  assertBool ("min severity should be Warning, but is " ++ (show msev))
    (msev == Warning)
  -- this message will not be traced

```

```

logInfo trace "Message #2"
-- lower the minimum severity to Info
setMinSeverity cfg Info
-- this message is traced
logInfo trace "Message #3"
-- acquire the traced objects
res ← STM.readTVarIO msgs
-- only the first and last messages should have been traced
assertBool
  ("Found more or less messages than expected: " ++ show res)
  (length res ≡ 2)
assertBool
  ("Found Info message when Warning was minimum severity: " ++ show res)
  (all
    (λcase
      LogObject _ (LOMeta _ _ Info _) (LogMessage "Message #2") → False
      _ → True)
    res)

```

Define a subtrace's behaviour to duplicate all messages

The **SubTrace** will duplicate all messages that pass through it. Each message will be in its own named context.

```

unitTraceDuplicate :: Assertion
unitTraceDuplicate = do
  cfg ← defaultConfigTesting
  msgs ← STM.newTVarIO []
  basetrace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test-duplicate" Neutral
  logInfo basetrace "Message #1"
  -- create a subtrace which duplicates all messages
  setSubTrace cfg "test-duplicate.orig" $ Just (TeeTrace "test-duplicate.dup")
  let trace = appendName "orig" basetrace
  -- this message will be duplicated
  logInfo trace "You will see me twice!"
  -- acquire the traced objects
  res ← STM.readTVarIO msgs
  -- only the first and last messages should have been traced
  assertBool
    ("Found more or less messages than expected: " ++ show res)
    (length res ≡ 3)

```

Change the minimum severity of a named context

A trace of a named context can be configured with a minimum severity, such that the trace will filter out messages that are labelled with a lower severity.

```

unitNamedMinSeverity :: Assertion
unitNamedMinSeverity = do

```

```

cfg ← defaultConfigTesting
msgs ← STM.newTVarIO []
basetrace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test-named-severity" Neutral
let trace = appendName "sev-change" basetrace
logInfo trace "Message #1"

-- raise the minimum severity to Warning
setSeverity cfg "test-named-severity.sev-change" (Just Warning)
msev ← Cardano.BM.Configuration.inspectSeverity cfg "test-named-severity.sev-change"
assertBool ("min severity should be Warning, but is " ++ (show msev))
  (msev ≡ Just Warning)
-- this message will not be traced
logInfo trace "Message #2"

-- lower the minimum severity to Info
setSeverity cfg "test-named-severity.sev-change" (Just Info)
-- this message is traced
logInfo trace "Message #3"

-- acquire the traced objects
res ← STM.readTVarIO msgs

-- only the first and last messages should have been traced
assertBool
  ("Found more or less messages than expected: " ++ show res)
  (length res ≡ 2)
assertBool
  ("Found Info message when Warning was minimum severity: " ++ show res)
  (all
    (λcase
      LogObject _ (LOMeta _ _ Info _) (LogMessage "Message #2") → False
      _ → True)
    res)

unitHierarchy' :: [SubTrace] → ([LogObject Text] → Bool) → Assertion
unitHierarchy' subtraces f = do
  cfg ← liftIO Cardano.BM.Configuration ◦ Model.empty
  let (t1 : t2 : t3 : _) = cycle subtraces
  msgs ← STM.newTVarIO []
  -- create trace of type 1
  trace1 ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test" t1
  logInfo trace1 "Message from level 1."
  -- subtrace of type 2
  setSubTrace cfg "test.inner" (Just t2)
  let trace2 = appendName "inner" trace1
  logInfo trace2 "Message from level 2."
  -- subsubtrace of type 3
  setSubTrace cfg "test.inner.innermost" (Just t3)
# ifdef ENABLE_OBSERVABLES
  _ ← STMObserver.bracketObserveIO cfg trace2 Debug "innermost" setVar_
# endif
  logInfo trace2 "Message from level 3."
  -- acquire the traced objects
  res ← STM.readTVarIO msgs

```

```
-- only the first message should have been traced
assertBool
  ("Found more or less messages than expected: " ++ show res)
(f res)
```

Logging in parallel

```
unitTraceInFork :: Assertion
unitTraceInFork = do
  cfg ← defaultConfigTesting
  msgs ← STM.newTVarIO []
  trace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test" Neutral
  let trace0 = appendName "work0" trace
      trace1 = appendName "work1" trace
  work0 ← work trace0
  threadDelay 5000
  work1 ← work trace1
  Async.wait $ work0
  Async.wait $ work1
  res ← STM.readTVarIO msgs
  let names@(_:namesTail) = map loName res
  -- each trace should have its own name and log right after the other
  assertBool
    ("Consecutive loggernames are not different: " ++ show names)
    (and $ zipWith (≠) names namesTail)
where
  work :: Trace IO Text → IO (Async.Async ())
  work trace = Async.async $ do
    logInfoDelay trace "1"
    logInfoDelay trace "2"
    logInfoDelay trace "3"
  logInfoDelay :: Trace IO Text → Text → IO ()
  logInfoDelay trace msg =
    logInfo trace msg >>
    threadDelay 10000
```

Stress testing parallel logging

```
stressTraceInFork :: Assertion
stressTraceInFork = do
  cfg ← defaultConfigTesting
  msgs ← STM.newTVarIO []
  trace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test" Neutral
  let names = map (λa → ("work-" <> pack (show a))) [1..(10 :: Int)]
  ts ← forM names $ λname → do
    let trace' = appendName name trace
    work trace'
  forM_ ts Async.wait
  res ← STM.readTVarIO msgs
```

```

let resNames = map loName res
let frequencyMap = fromListWith (+) [(x,1) | x ← resNames]
-- each trace should have traced totalMessages' messages
assertBool
  ("Frequencies of logged messages according to loggername: " ++ show frequencyMap)
  (all (λname → (lookup ("test." <> name) frequencyMap) ≡ Just totalMessages) names)
where
  work :: Trace IO Text → IO (Async.Async ())
  work trace = Async.async $ forM_ [1..totalMessages] $ (logInfo trace) o pack o show
  totalMessages :: Int
  totalMessages = 10

```

Dropping **ObserveOpen** messages in a subtrace

```

unitNoOpeningTrace :: Assertion
unitNoOpeningTrace = do
  cfg ← defaultConfigTesting
  msgs ← STM.newTVarIO []
  # ifdef ENABLE_OBSERVABLES
    logTrace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test" DropOpening
    _ ← STMObserver.bracketObserveIO cfg logTrace Debug "setTVar" setVar_
  # endif
  res ← STM.readTVarIO msgs
  assertBool
    ("Found non-expected ObserveOpen message: " ++ show res)
    (all (λcase { LogObject _ → (ObserveOpen _) → False; _ → True }) res)

```

Assert maximum length of log context name

The name of the log context cannot grow beyond a maximum number of characters, currently the limit is set to 80.

```

unitAppendName :: Assertion
unitAppendName = do
  cfg ← defaultConfigTesting
  msgs ← STM.newTVarIO []
  basetrace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test" Neutral
  let trace1 = appendName bigName basetrace
      trace2 = appendName bigName trace1
  forM_ [basetrace, trace1, trace2] $ (flip logInfo msg)
  res ← reverse < $ > STM.readTVarIO msgs
  let loggerNames = map loName res
  assertBool
    ("AppendName did not work properly. The loggerNames for the messages are: " ++
     show loggerNames)
    (loggerNames ≡ [ "test"
                    , "test." <> bigName
                    , "test." <> bigName <> "." <> bigName
                    ])
  where

```

```

    bigName = T.replicate 30 "abcdefghi jklmnopqrstuvwxyz"
    msg = "Hello!"

# ifdef ENABLE_OBSERVABLES
setVar_ :: STM.STM Integer
setVar_ = do
    t ← STM.newTVar 0
    STM.writeTVar t 42
    res ← STM.readTVar t
    return res
# endif

```

Testing log context name filters

```

unitNameFiltering :: Assertion
unitNameFiltering = do
    let contextName = "test.sub.1"
    let lname = "sum" -- would be part of a "LogValue lname 42"
    let filter1 = [(Drop (Exact "test.sub.1"), Unhide [ ])]
    assertBool ("Dropping a specific name should filter it out and thus return False")
        (False == evalFilters filter1 contextName)
    let filter2 = [(Drop (EndsWith ".1"), Unhide [ ])]
    assertBool ("Dropping a name ending with a specific text should filter out the context")
        (False == evalFilters filter2 contextName)
    let filter3 = [(Drop (StartsWith "test."), Unhide [ ])]
    assertBool ("Dropping a name starting with a specific text should filter out the context")
        (False == evalFilters filter3 contextName)
    let filter4 = [(Drop (Contains ".sub."), Unhide [ ])]
    assertBool ("Dropping a name starting containing a specific text should filter out the context")
        (False == evalFilters filter4 contextName)
    let filter5 = [(Drop (StartsWith "test."),
        Unhide [(Exact "test.sub.1")])]
    assertBool ("Dropping all and unhiding a specific name should the context name allow pass")
        (True == evalFilters filter5 contextName)
    let filter6a = [(Drop (StartsWith "test."),
        Unhide [(EndsWith ".sum"),
        (EndsWith ".other")])]
    assertBool ("Dropping all and unhiding some names, the LogObject should pass the filter")
        (True == evalFilters filter6a (contextName <> "." <> lname))
    assertBool ("Dropping all and unhiding some names, another LogObject should not pass the filter")
        (False == evalFilters filter6a (contextName <> ".value"))
    let filter6b = [(Drop (Contains "test."),
        Unhide [(Contains ".sum"),
        (Contains ".other")])]
    assertBool ("Dropping all and unhiding some names, the LogObject should pass the filter")
        (True == evalFilters filter6b (contextName <> "." <> lname))
    assertBool ("Dropping all and unhiding some names, another LogObject should not pass the filter")
        (False == evalFilters filter6b (contextName <> ".value"))
    assertBool ("Dropping others and unhiding some names, something different should still pass")
        (True == evalFilters filter6b "some.other.value")

```

```

let filter7 = [(Drop (StartsWith "test."),
  Unhide [(EndsWith ".product")])]
assertBool ("Dropping all and un hiding an inexistant named value, the LogObject should
  (False == evalFilters filter7 (contextName <> "." <> loname))
let filter8 = [(Drop (StartsWith "test."),
  Unhide [(Exact "test.sub.1")]),
  (Drop (StartsWith "something.else."),
  Unhide [(EndsWith ".this")])]
assertBool ("Disjunction of filters that should pass")
  (True == evalFilters filter8 contextName)
let filter9 = [(Drop (StartsWith "test."),
  Unhide [(Exact ".that")]),
  (Drop (StartsWith "something.else."),
  Unhide [(EndsWith ".this")])]
assertBool ("Disjunction of filters that should not pass")
  (False == evalFilters filter9 contextName)

```

Exception throwing

Exceptions encountered should be thrown. Lazy evaluation is really happening! This test fails if run with a configuration `defaultConfigTesting`, because this one will ignore all traced messages.

```

unitExceptionThrowing :: Assertion
unitExceptionThrowing = do
  action ← work msg
  res ← Async.waitCatch action
  assertBool
    ("Exception should have been rethrown")
    (isLeft res)
where
  msg :: Text
  msg = error "faulty message"
  work :: Text → IO (Async.Async ())
  work message = Async.async $ do
    cfg ← defaultConfigStdout
    trace ← Setup.setupTrace (Right cfg) "test"
    logInfo trace message
    threadDelay 10000

```

Check lazy evaluation of trace

Exception should not be thrown when type of `Trace` is `NoTrace`.

```

unitTestLazyEvaluation :: Assertion
unitTestLazyEvaluation = do
  action ← work msg
  res ← Async.waitCatch action
  assertBool

```



```

    ("Exception should not have been rethrown when type of Trace is NoTrace")
    (isRight res)
  where
    msg :: Text
    msg = error "faulty message"
    work :: Text → IO (Async.Async ())
    work message = Async.async $ do
      cfg ← defaultConfigTesting
      basetrace ← Setup.setupTrace (Right cfg) "test"
      setSubTrace cfg "test.work" (Just NoTrace)
      let trace = appendName "work" basetrace
      logInfo trace message

```

Check that private messages do not end up in public log files.

```

unitLoggingPrivate :: Assertion
unitLoggingPrivate = do
  tmpDir ← getTemporaryDirectory
  let privateFile = tmpDir </> "private.log"
      publicFile = tmpDir </> "public.log"
  conf ← empty
  setDefaultBackends conf [KatipBK]
  setSetupBackends conf [KatipBK]
  setDefaultScribes conf [ "FileSK::" <> pack privateFile
                          , "FileSK::" <> pack publicFile
                          ]
  setSetupScribes conf [ ScribeDefinition
    { scKind    = FileSK
    , scFormat  = ScText
    , scName    = pack privateFile
    , scPrivacy = ScPrivate
    , scRotation = Nothing
    }
  , ScribeDefinition
    { scKind    = FileSK
    , scFormat  = ScText
    , scName    = pack publicFile
    , scPrivacy = ScPublic
    , scRotation = Nothing
    }
  ]
  Setup.withTrace conf "test" $ λtrace → do
    -- should log in both files
    logInfo trace message
    -- should only log in private file
    logInfoS trace message
  countPublic ← length ∘ lines <$> readFile publicFile
  countPrivate ← length ∘ lines <$> readFile privateFile
  -- delete files

```

```

forM_ [privateFile, publicFile] removeFile
assertBool
  ("Confidential file should contain 2 lines and it contains " ++ show countPrivate ++ "
   Public file should contain 1 line and it contains " ++ show countPublic ++ ".\n"
  )
(countPublic  $\equiv$  1  $\wedge$  countPrivate  $\equiv$  2)
where
  message :: Text
  message = "Just a message"

```

2.3.4 Testing configuration

Test declarations

```

tests :: TestTree
tests = testGroup "config tests" [
  propertyTests
, unitTests
]
propertyTests :: TestTree
propertyTests = testGroup "Properties" [
  testProperty "minimal" prop_Configuration_minimal
]
unitTests :: TestTree
unitTests = testGroup "Unit tests" [
  testCase "static representation" unitConfigurationStaticRepresentation
, testCase "parsed representation" unitConfigurationParsedRepresentation
, testCase "parsed configuration" unitConfigurationParsed
, testCase "export configuration" unitConfigurationExport
, testCase "include EKG if defined" unitConfigurationCheckEKGpositive
, testCase "not include EKG if not def" unitConfigurationCheckEKGnegative
, testCase "check scribe caching" unitConfigurationCheckScribeCache
, testCase "test ops on Configuration" unitConfigurationOps
]

```

Property tests

```

prop_Configuration_minimal :: Bool
prop_Configuration_minimal = True

```

Unit tests

The configuration file only indicates that EKG is listening on port nnnnn. Infer that *EKGViewBK* needs to be started as a backend.

```

unitConfigurationCheckEKGpositive :: Assertion
unitConfigurationCheckEKGpositive = do
  # ifdef ENABLE_EKG
  return ()

```

```

# else
tmp ← getTemporaryDirectory
let c = [ "rotation:"
, "  rpLogLimitBytes: 5000000"
, "  rpKeepFilesNum: 10"
, "  rpMaxAgeHours: 24"
, "minSeverity: Info"
, "defaultBackends:"
, "  - KatipBK"
, "setupBackends:"
, "  - KatipBK"
, "defaultScribes:"
, "- - StdoutSK"
, "  - stdout"
, "setupScribes:"
, "- scName: stdout"
, "  scRotation: null"
, "  scKind: StdoutSK"
, "hasEKG: 18321"
, "options:"
, "  test:"
, "    value: nothing"
]
fp = tmp < / > "test_ekgv_config.yaml"
writeFile fp $ unlines c
repr ← parseRepresentation fp
assertBool "expecting EKGViewBK to be setup" $
  EKGViewBK ∈ (setupBackends repr)
# endif

```

If there is no port defined for EKG, then do not start it even if present in the config.

```

unitConfigurationCheckEKGnegative :: Assertion
unitConfigurationCheckEKGnegative = do
  # ifndef ENABLE_EKG
    return ()
  # else
    tmp ← getTemporaryDirectory
    let c = [ "rotation:"
, "  rpLogLimitBytes: 5000000"
, "  rpKeepFilesNum: 10"
, "  rpMaxAgeHours: 24"
, "minSeverity: Info"
, "defaultBackends:"
, "  - KatipBK"
, "  - EKGViewBK"
, "setupBackends:"
, "  - KatipBK"
, "  - EKGViewBK"
, "defaultScribes:"
, "- - StdoutSK"
, "  - stdout"

```

```

    , "setupScribes:"
    , "- scName: stdout"
    , "  scRotation: null"
    , "  scKind: StdoutSK"
    , "###hasEKG: 18321"
    , "options:"
    , "  test:"
    , "    value: nothing"
  ]
  fp = tmp < / > "test_ekgv_config.yaml"
  writeFile fp $ unlines c
  repr ← parseRepresentation fp
  assertBool "EKGVViewBK shall not be setup" $
    ¬ $ EKGViewBK ∈ (setupBackends repr)
  assertBool "EKGVViewBK shall not receive messages" $
    ¬ $ EKGViewBK ∈ (defaultBackends repr)
# endif

```

```

unitConfigurationStaticRepresentation :: Assertion
unitConfigurationStaticRepresentation =
  let r = Representation
    { minSeverity = Info
    , rotation = Just $ RotationParameters
      { rpLogLimitBytes = 5000000
      , rpMaxAgeHours = 24
      , rpKeepFilesNum = 10
      }
    , setupScribes =
      [ ScribeDefinition { scName = "stdout"
      , scKind = StdoutSK
      , scFormat = ScText
      , scPrivacy = ScPublic
      , scRotation = Nothing }
      ]
    , defaultScribes = [ (StdoutSK, "stdout") ]
    , setupBackends = [ EKGViewBK, KatipBK ]
    , defaultBackends = [ KatipBK ]
    , hasGUI = Just 12789
    , hasGraylog = Just 12788
    , hasEKG = Just 18321
    , hasPrometheus = Just 12799
    , logOutput = Nothing
    , options =
      HM.fromList [ ("test1", (HM.singleton "value" "object1"))
      , ("test2", (HM.singleton "value" "object2")) ]
    }
  in
  encode r @? =
    (intercalate "\n"
    [ "rotation:"
    , "  rpLogLimitBytes: 5000000"

```

```

    , " rpKeepFilesNum: 10"
    , " rpMaxAgeHours: 24"
    , "defaultBackends:"
    , "- KatipBK"
    , "setupBackends:"
    , "- EKGViewBK"
    , "- KatipBK"
    , "hasPrometheus: 12799"
    , "hasGraylog: 12788"
    , "hasGUI: 12789"
    , "defaultScribes:"
    , "- - StdoutSK"
    , " - stdout"
    , "options:"
    , " test2:"
    , "   value: object2"
    , " test1:"
    , "   value: object1"
    , "setupScribes:"
    , "- scName: stdout"
    , " scRotation: null"
    , " scKind: StdoutSK"
    , " scFormat: ScText"
    , " scPrivacy: ScPublic"
    , "logOutput: null"
    , "hasEKG: 18321"
    , "minSeverity: Info"
    , "-- to force a line feed at the end of the file
  ]
)

```

unitConfigurationParsedRepresentation :: Assertion

unitConfigurationParsedRepresentation = do

repr ← parseRepresentation "test/config.yaml"

encode repr @? =

```

  (intercalate "\n"
    [ "rotation:"
    , " rpLogLimitBytes: 5000000"
    , " rpKeepFilesNum: 10"
    , " rpMaxAgeHours: 24"
    , "defaultBackends:"
    , "- KatipBK"
    , "setupBackends:"
    , "- AggregationBK"
    , "- EKGViewBK"
    , "- KatipBK"
    , "- path: log-pipe"
    , " kind: TraceAcceptorBK"
    , "hasPrometheus: null"
    , "hasGraylog: 12788"
    , "hasGUI: null"
    , "defaultScribes:"

```

```

, "- - StdoutSK"
, " - stdout"
, "options:"
, "  mapSubtrace:"
, "    iohk.benchmarking:"
, "      contents:"
, "        - GhcRtsStats"
, "        - MonotonicClock"
, "      subtrace: ObservableTraceSelf"
, "    iohk.deadend:"
, "      subtrace: NoTrace"
, "  mapSeverity:"
, "    iohk.startup: Debug"
, "    iohk.background.process: Error"
, "    iohk.testing.uncritical: Warning"
, "  mapAggregatedkinds:"
, "    iohk.interesting.value: EwmaAK {alpha = 0.75}"
, "    iohk.background.process: StatsAK"
, "  cfokey:"
, "    value: Release-1.0.0"
, "  mapMonitors:"
, "    chain.creation.block:"
, "      actions:"
, "        - CreateMessage Warning \"chain.creation\""
, "        - AlterSeverity \"chain.creation\" Debug"
, "      monitor: ((time > (23 s)) Or (time < (17 s)))"
, "      '#aggregation.critproc.observable':"
, "      actions:"
, "        - CreateMessage Warning \"the observable has been too long too high!\""
, "        - SetGlobalMinimalSeverity Info"
, "      monitor: (mean >= (42))"
, "  mapScribes:"
, "    iohk.interesting.value:"
, "      - StdoutSK::stdout"
, "      - FileSK::testlog"
, "    iohk.background.process: FileSK::testlog"
, "  mapBackends:"
, "    iohk.user.defined:"
, "      - kind: UserDefinedBK"
, "      name: MyBackend"
, "      - KatipBK"
, "    iohk.interesting.value:"
, "      - EKGViewBK"
, "      - AggregationBK"
, "setupScribes:"
, "- scName: testlog"
, "  scRotation:"
, "    rpLogLimitBytes: 25000000"
, "    rpKeepFilesNum: 3"
, "    rpMaxAgeHours: 24"
, "  scKind: FileSK"

```

```

    , "  scFormat: ScText"
    , "  scPrivacy: ScPrivate"
    , "- scName: stdout"
    , "  scRotation: null"
    , "  scKind: StdoutSK"
    , "  scFormat: ScText"
    , "  scPrivacy: ScPublic"
    , "logOutput: null"
    , "hasEKG: 12789"
    , "minSeverity: Info"
    , "-- to force a line feed at the end of the file
  ]
)

unitConfigurationParsed :: Assertion
unitConfigurationParsed = do
  cfg ← setup "test/config.yaml"
  cfgInternal ← readMVar $ getCG cfg
  cfgInternal @? = ConfigurationInternal
    { cgMinSeverity      = Info
    , cgDefRotation      = Just $ RotationParameters
        { rpLogLimitBytes = 5000000
        , rpMaxAgeHours   = 24
        , rpKeepFilesNum  = 10
        }
    , cgMapSeverity      = HM.fromList [ ("iohk.startup", Debug)
        , ("iohk.background.process", Error)
        , ("iohk.testing.uncritical", Warning)
        ]
    , cgMapSubtrace      = HM.fromList [ ("iohk.benchmarking",
        ObservableTraceSelf [GhcRtsStats, MonotonicClock])
        , ("iohk.deadend", NoTrace)
        ]
    , cgOptions          = HM.fromList
        [ ("mapSubtrace",
            HM.fromList [ ("iohk.benchmarking",
                Object (HM.fromList [ ("subtrace", String "ObservableTraceSelf")
                    , ("contents", Array $ V.fromList
                        [ String "GhcRtsStats"
                        , String "MonotonicClock" ])))
                , ("iohk.deadend",
                    Object (HM.fromList [ ("subtrace", String "NoTrace") ])))
            ]
        ]
    , ("mapMonitors", HM.fromList [ ("chain.creation.block", Object (HM.fromList
        [ ("monitor", String "((time > (23 s)) Or (time < (17 s)))")
        , ("actions", Array $ V.fromList
            [ String "CreateMessage Warning \"chain.creation\" \"
            , String "AlterSeverity \"chain.creation\" Debug"
            ])))
        ]
    , ("#aggregation.critproc.observable", Object (HM.fromList
        [ ("monitor", String "(mean >= (42))")
        , ("actions", Array $ V.fromList
            [ String "CreateMessage Warning \"the observable has been too lo

```

```

        ,String "SetGlobalMinimalSeverity Info"
        ])))]))
,("mapSeverity",HM.fromList [("iohk.startup",String "Debug")
    ,("iohk.background.process",String "Error")
    ,("iohk.testing.uncritical",String "Warning")])
,("mapAggregatedkinds",HM.fromList [("iohk.interesting.value",
    String "EwmaAK {alpha = 0.75}")
    ,("iohk.background.process",
    String "StatsAK")])
,("cfokey",HM.fromList [("value",String "Release-1.0.0")])
,("mapScribes",HM.fromList [("iohk.interesting.value",
    Array$ V.fromList [String "StdoutSK::stdout"
    ,String "FileSK::testlog" ])
    ,("iohk.background.process",String "FileSK::testlog")])
,("mapBackends",HM.fromList [("iohk.user.defined",
    Array$ V.fromList [Object (HM.fromList [("kind",String "UserDefinedBK"
    ,("name",String "MyBackend")])
    ,String "KatipBK"
    ])
    ,("iohk.interesting.value",
    Array$ V.fromList [String "EKGViewBK"
    ,String "AggregationBK"
    ]))
]
,cgMapBackend = HM.fromList [("iohk.user.defined"
    ,[UserDefinedBK "MyBackend"
    ,KatipBK
    ]
    )
    ,("iohk.interesting.value"
    ,[EKGViewBK
    ,AggregationBK
    ]
    )
]
,cgDefBackendKs = [KatipBK]
,cgSetupBackends = [AggregationBK
    ,EKGViewBK
    ,KatipBK
    ,TraceAcceptorBK "log-pipe"
    ]
,cgMapScribe = HM.fromList [("iohk.interesting.value",
    [ "StdoutSK::stdout", "FileSK::testlog" ])
    ,("iohk.background.process",[ "FileSK::testlog" ])
    ]
,cgMapScribeCache = HM.fromList [("iohk.interesting.value",
    [ "StdoutSK::stdout", "FileSK::testlog" ])
    ,("iohk.background.process",[ "FileSK::testlog" ])
    ]
,cgDefScribes = [ "StdoutSK::stdout" ]
,cgSetupScribes = [ScribeDefinition

```



```

    {scKind = FileSK
    ,scFormat = ScText
    ,scName = "testlog"
    ,scPrivacy = ScPrivate
    ,scRotation = Just $ RotationParameters
      {rpLogLimitBytes = 25000000
      ,rpMaxAgeHours = 24
      ,rpKeepFilesNum = 3
      }
    }
  ,ScribeDefinition
    {scKind = StdoutSK
    ,scFormat = ScText
    ,scName = "stdout"
    ,scPrivacy = ScPublic
    ,scRotation = Nothing
    }
  ]
,cgMapAggregatedKind = HM.fromList [("iohk.interesting.value", EwmaAK {alpha = 0.75})
,("iohk.background.process", StatsAK)
]
,cgDefAggregatedKind = StatsAK
,cgMonitors = HM.fromList [("chain.creation.block"
, (Nothing
, (OR (Compare "time" (GT, (OpMeasurable (Agg.Seconds 23)))) (Compare "t
, [CreateMessage Warning "chain.creation"
, AlterSeverity "chain.creation" Debug
]
)
)
)
, ("#aggregation.critproc.observable"
, (Nothing
, Compare "mean" (GE, (OpMeasurable (Agg.PureI 42)))
, [CreateMessage Warning "the observable has been too long too hi
, SetGlobalMinimalSeverity Info
]
)
)
]
, cgPortEKG = 12789
, cgPortGraylog = 12788
, cgPortPrometheus = 12799 -- the default value
, cgPortGUI = 0
, cgLogOutput = Nothing
}

unitConfigurationExport :: Assertion
unitConfigurationExport = do
  cfg ← setup "test/config.yaml"
  cfg' ← withSystemTempFile "config.yaml-1213" $ \file _ → do
    exportConfiguration cfg file
  setup file

```

```

cfgInternal ← readMVar $ getCG cfg
cfgInternal' ← readMVar $ getCG cfg'
cfgInternal' @? = cfgInternal

```

Test caching and inheritance of Scribes.

```

unitConfigurationCheckScribeCache :: Assertion
unitConfigurationCheckScribeCache = do
  configuration ← empty
  let defScribes = ["FileSK::node.log"]
  setDefaultScribes configuration defScribes
  let scribes12 = ["StdoutSK::stdout", "FileSK::out.txt"]
  setScribes configuration "name1.name2" $ Just scribes12
  scribes1234 ← getScribes configuration "name1.name2.name3.name4"
  scribes1 ← getScribes configuration "name1"
  scribes1234cached ← getCacheScribes configuration "name1.name2.name3.name4"
  scribesXcached ← getCacheScribes configuration "nameX"
  assertBool "Scribes for name1.name2.name3.name4 must be the same as name1.name2" $
    scribes1234 == scribes12
  assertBool "Scribes for name1 must be the default ones" $
    scribes1 == defScribes
  assertBool "Scribes for name1.name2.name3.name4 must have been cached" $
    scribes1234cached == Just scribes1234
  assertBool "Scribes for nameX must not have been cached since getScribes was not called" $
    scribesXcached == Nothing

```

Test operations on Configuration.

```

unitConfigurationOps :: Assertion
unitConfigurationOps = do
  configuration ← defaultConfigStdout
  defBackends ← getDefaultBackends configuration
  setDefaultAggregatedKind configuration $ EwmaAK 0.01
  -- since loggename does not exist the default must be inherited
  defAggregatedKind ← getAggregatedKind configuration "non-existent loggename"
  setAggregatedKind configuration "name1" $ Just StatsAK
  name1AggregatedKind ← getAggregatedKind configuration "name1"
  setEKGport configuration 11223
  ekgPort ← getEKGport configuration
  setGUIport configuration 1080
  guiPort ← getGUIport configuration
  assertBool "Default backends" $
    defBackends == [KatipBK]
  assertBool "Default aggregated kind" $
    defAggregatedKind == EwmaAK 0.01
  assertBool "Specific name aggregated kind" $
    name1AggregatedKind == StatsAK
  assertBool "Set EKG port" $
    ekgPort == 11223

```

```
assertBool "Set GUI port" $
  guiPort == 1080
```

2.3.5 Rotator

```
tests :: TestTree
tests = testGroup "testing Trace" [
  property_tests
]
property_tests :: TestTree
property_tests = testGroup "Property tests" [
  testProperty "rotator: file naming" propNaming
# ifdef POSIX
  , testProperty "rotator: cleanup" $ propCleanup $ rot n
# endif
]
# ifdef POSIX
where
  n = 5
  rot num = RotationParameters
    { rpLogLimitBytes = 100000000 -- 10 MB
    , rpMaxAgeHours = 24
    , rpKeepFilesNum = num
    }
# endif
```

Check that the generated file name has only 15 digits added to the base name.

```
propNaming :: FilePath → Property
propNaming name = ioProperty $ do
  filename ← nameLogFile name
  return $ length filename == length name + 15
```

Test cleanup of rotator.

This test creates a random number of files with the same name but with different dates and afterwards it calls the `cleanupRotator` function which removes old log files keeping only `rpKeepFilesNum` files and deleting the others.

```
# ifdef POSIX
data LocalFilePath = Dir FilePath
  deriving (Show)
instance Arbitrary LocalFilePath where
  arbitrary = do
    start ← QC.sized $ λn → replicateM (n + 1) (QC.elements $ [ 'a' .. 'z' ])
    x ← QC.sized $ λn → replicateM n (QC.elements $ [ 'a' .. 'd' ] ++ [ "/" ])
    pure $ Dir $ start ++ removeAdjacentAndLastSlashes x
  shrink (Dir path) = map (Dir ∘ removeAdjacentAndLastSlashes ∘ (intercalate "/")) $
```

```

    product' $ map (filter (≠ "")) $ map QC.shrink (splitOn "/" path)
  where
    product' :: [[a]] → [[a]]
    product' = mapM (λx → x ≫ return)
removeAdjacentAndLastSlashes :: FilePath → FilePath
removeAdjacentAndLastSlashes = concat ∘ filter (≠ "/") ∘ groupBy (\_b → b ≠ '/')
data SmallAndLargeInt = SL Int
  deriving (Show)
instance Arbitrary SmallAndLargeInt where
  arbitrary = do
    QC.oneof [smallGen
              ,largeGen
              ]
  where
    smallGen :: QC.Gen SmallAndLargeInt
    smallGen = do
      QC.Small x ← (QC.arbitrary :: QC.Gen (QC.Small Int))
      pure $ SL $ abs x
    largeGen :: QC.Gen SmallAndLargeInt
    largeGen = do
      let maxBoundary = 0010000000000000 -- 10 years for the format which is used
          minBoundary = 000000000010000 -- 1 hour for the format which is used
          x ← QC.choose (minBoundary, maxBoundary)
      pure $ SL x
    shrink _ = []
data NumFiles = NF Int deriving (Show)
instance Arbitrary NumFiles where
  arbitrary = QC.oneof [return (NF 0), return (NF 1), return (NF 5), return (NF 7)]
propCleanup :: RotationParameters → LocalFilePath → NumFiles → SmallAndLargeInt → Property
propCleanup rotationParams (Dir filename) (NF nFiles) (SL maxDev) = QC.withMaxSuccess 20 $ ioProperty
  tmpDir0 ← getTemporaryDirectory
  let tmpDir = tmpDir0 </> "rotatorTest.base"
  let path = tmpDir </> filename
  -- generate nFiles different dates
  now ← getcurrentTime
  let tsnow = formatTime defaultTimeLocale tsformat now
  deviations ← replicateM nFiles $ QC.generate $ QC.choose (1, maxDev + 1)
  -- TODO if generated within the same sec we have a problem
  let dates = map show $ scanl (+) (read tsnow) deviations
      files = map (λa → path ++ ('-' : a)) dates
      sortedFiles = reverse $ sort files
      keepFilesNum = fromIntegral $ rpKeepFilesNum rotationParams
      toBeKept = reverse $ take keepFilesNum sortedFiles
  createDirectoryIfMissing True $ takeDirectory path
  forM_ (files) $ λf → openFile f WriteMode
  cleanupRotator rotationParams path
  filesRemained ← listLogFiles path
  let kept = case filesRemained of
    Nothing → []

```

```

    Just l → NE.toList l
  removeDirectoryRecursive tmpDir
  return $ kept == toBeKept
# endif

```

2.3.6 Cardano.BM.Test.Structured

```

tests :: TestTree
tests = testGroup "Testing Structured Logging" [
  testCase "logging simple text" logText
  -- , testCase "logging data structures" logStructures
]

```

Simple logging of text.

```

logText :: Assertion
logText = do
  cfg ← defaultConfigTesting
  baseTrace :: Trace IO Text ← Setup.setupTrace (Right cfg) "logText"
  let logTrace = toLogObject $ baseTrace
  traceWith logTrace "This is a simple message."
  traceWith logTrace ".. and another!"
  assertBool "OK" True

```

2.3.7 Cardano.BM.Test.Tracer

```

tests :: TestTree
tests = testGroup "Testing Extensions to Tracer" [
  testCase "simple tracing of messages in a named context" tracingInNamedContext,
  testCase "tracing with privacy and severity annotation" tracingWithPrivacyAndSeverityAnnotation,
  testCase "tracing with a predicate filter" tracingWithPredicateFilter,
  testCase "tracing with a filter that is evaluated in a monad" tracingWithMonadicFilter,
  testCase "tracing with filtering for both severity and privacy" tracingWithComplexFiltering
]

```

Utilities

```

data LogNamed item = LogNamed
  { lnName :: LoggerName
  , lnItem :: item
  } deriving (Show)

named :: Tracer m (LogNamed a) → Tracer m a
named = contramap (LogNamed mempty)

appendNamed :: LoggerName → Tracer m (LogNamed a) → Tracer m (LogNamed a)
appendNamed name = contramap $ (λ(LogNamed oldName item) →

```

```

LogNamed (name <> "." <> oldName) item)
renderNamedItemTracing :: Show a => Tracer m String -> Tracer m (LogNamed a)
renderNamedItemTracing = contramap $ \item ->
  unpack (lnName item) ++ ": " ++ show (lnItem item)

appendNamed' :: LoggerName -> Tracer m (LogObject a) -> Tracer m (LogObject a)
appendNamed' name = contramap $ (\(LogObject oldName meta item) ->
  if oldName == ""
  then LogObject name meta item
  else LogObject (name <> "." <> oldName) meta item)

renderNamedItemTracing' :: Show a => Tracer m String -> Tracer m (LogObject a)
renderNamedItemTracing' = contramap $ \item ->
  unpack (loName item) ++ ": " ++ show (loContent item) ++ ", (meta): " ++ show (loMeta item)

```

Tracing messages in a named context

```

tracingInNamedContext :: Assertion
tracingInNamedContext = do
  let logTrace = appendNamed' "named" $ renderNamedItemTracing' $ stdoutTracer
  void $ callFun2 logTrace
  assertBool "OK" True

callFun2 :: Tracer IO (LogObject Text) -> IO Int
callFun2 logTrace = do
  let logTrace' = appendNamed' "fun2" logTrace
  traceWith (toLogObject logTrace') "in function 2"
  callFun3 logTrace'

callFun3 :: Tracer IO (LogObject Text) -> IO Int
callFun3 logTrace = do
  traceWith (toLogObject $ appendNamed' "fun3" $ logTrace) "in function 3"
  return 42

```

Tracing messages with privacy and severity annotation

A **Tracer** transformer creating a **LogObject** from *PrivacyAndSeverityAnnotated*.

```

logObjectFromAnnotated :: Show a
=> Tracer IO (LogObject a)
-> Tracer IO (PrivacyAndSeverityAnnotated a)
logObjectFromAnnotated tr = Tracer $ \ (PSA sev priv a) -> do
  lometa <- mkLOMeta sev priv
  traceWith tr $ LogObject "" lometa (LogMessage a)

tracingWithPrivacyAndSeverityAnnotation :: Assertion
tracingWithPrivacyAndSeverityAnnotation = do
  let logTrace =
    logObjectFromAnnotated $ appendNamed' "example3" $ renderNamedItemTracing' $ stdoutTracer
  traceWith logTrace $ PSA Info Confidential ("Hello" :: String)
  traceWith logTrace $ PSA Warning Public "World"
  assertBool "OK" True

```

Filter Tracer

```

filterAppendNameTracing :: Monad m
    => m (LogObject a → Bool)
    → LoggerName
    → Tracer m (LogObject a)
    → Tracer m (LogObject a)
filterAppendNameTracing test name = (appendNamed' name) ◦ (condTracingM test)
tracingWithPredicateFilter :: Assertion
tracingWithPredicateFilter = do
    let appendF = filterAppendNameTracing oracle
        logTrace = appendF "example4" (renderNamedItemTracing' stdoutTracer)
    traceWith (toLogObject logTrace) ("Hello" :: String)
    let logTrace' = appendF "inner" logTrace
    traceWith (toLogObject logTrace') "World"
    let logTrace'' = appendF "innest" logTrace'
    traceWith (toLogObject logTrace'') "!!"
    assertBool "OK" True
where
    oracle :: Monad m => m (LogObject a → Bool)
    oracle = return $ ((≠) "example4.inner.") ◦ loName

-- severity anotated
tracingWithMonadicFilter :: Assertion
tracingWithMonadicFilter = do
    let logTrace =
        condTracingM oracle $
            logObjectFromAnnotated $
                appendNamed' "test5" $ renderNamedItemTracing' stdoutTracer
    traceWith logTrace $ PSA Debug Confidential ("Hello" :: String)
    traceWith logTrace $ PSA Warning Public "World"
    assertBool "OK" True
where
    oracle :: Monad m => m (PrivacyAndSeverityAnnotated a → Bool)
    oracle = return $ λ(PSA sev _priv _) → (sev > Debug)

tracing with combined filtering for name and severity

tracingWithComplexFiltering :: Assertion
tracingWithComplexFiltering = do
    let logTrace0 = -- the basis, will output using the local renderer to stdout
        appendNamed' "test6" $ renderNamedItemTracing' stdoutTracer
        logTrace1 = -- the trace from Privacy...Annotated to LogObject
            condTracingM oracleSev $ logObjectFromAnnotated $ logTrace0
        logTrace2 =
            appendNamed' "row" $ condTracingM oracleName $ logTrace0
        logTrace3 = -- oracle should eliminate messages from this trace
            appendNamed' "raw" $ condTracingM oracleName $ logTrace0
    traceWith logTrace1 $ PSA Debug Confidential ("Hello" :: String)

```

```

traceWith logTrace1 $ PSA Warning Public "World"
lometa ← mkLOMeta Info Public
traceWith logTrace2 $ LogObject "" lometa (LogMessage " , Row!")
traceWith logTrace3 $ LogObject "" lometa (LogMessage " , Row!")
assertBool "OK" True
where
  oracleSev :: Monad m ⇒ m (PrivacyAndSeverityAnnotated a → Bool)
  oracleSev = return $ λ(PSA sev _priv _) → (sev > Debug)
  oracleName :: Monad m ⇒ m (LogObject a → Bool)
  oracleName = return $ λ(LogObject name _ _) → (name ≡ "row") -- we only see the names from

```


Index

- addBackend, 81
- Aggregated, 49
 - instance of Semigroup, 49
 - instance of Show, 50
- AggregatedExpanded, 111
- AggregatedKind, 50
 - EwmaAK, 50
 - StatsAK, 50
- AggregatedMessage, 56
- Aggregation, 110
 - instance of IsBackend, 111
 - instance of IsEffectuator, 111
- appendName, 32
- Backend, 50
- BackendKind, 51
 - AggregationBK, 51
 - EditorBK, 51
 - EKGViewBK, 51
 - GraylogBK, 51
 - KatipBK, 51
 - LogBufferBK, 51
 - MonitoringBK, 51
 - SwitchboardBK, 51
 - TraceAcceptorBK, 51
 - TraceForwarderBK, 51
 - UserDefinedBK, 51
- Command, 56
- CommandValue, 58
- condTracing, 26
- condTracingM, 26
- Counter, 53
- Counters
 - Dummy
 - readCounters, 36
 - Linux
 - readCounters, 37
- CounterState, 54
- CounterType, 53
- debugTracer, 26
- diffCounters, 54
- Editor, 100
 - instance of IsBackend, 100
 - instance of IsEffectuator, 101
- EKGView, 95
 - instance of IsBackend, 98
 - instance of IsEffectuator, 97
- evalFilters, 75
- EWMA, 49
- ewma, 116
- exportConfiguration, 75
- getMonoClock, 36
- getOptionOrDefault, 65
- Graylog, 107
 - instance of IsBackend, 108
 - instance of IsEffectuator, 107
- IsBackend, 50
- IsEffectuator, 50
- KillPill, 56
- LOContent, 55
- Log, 84
 - instance of IsBackend, 86
 - instance of IsEffectuator, 85
- logAlert, 33
- logAlertS, 33
- LogBuffer, 94
 - instance of IsBackend, 95
 - instance of IsEffectuator, 95
- logCritical, 33
- logCriticalS, 33
- logDebug, 33
- logDebugS, 33
- logEmergency, 33
- logEmergencyS, 33
- LogError, 56
- logError, 33
- logErrorS, 33
- LoggerName, 54
- logInfo, 33
- logInfoS, 33
- LogMessage, 56
- logNotice, 33
- logNoticeS, 33

- LogObject, 55
- LogStructured, 56
- LogValue, 56
- logWarning, 33
- logWarningS, 33
- LOMeta, 55
- mainTraceConditionally, 78
- mapLOContent, 59
- mapLogObject, 59
- Measurable, 44
 - instance of Num, 46
 - instance of Show, 46
- mkLOMeta, 55
- MockSwitchboard, 84
- modifyName, 32
- Monitor, 116, 122
 - instance of IsBackend, 117
 - instance of IsEffectuator, 117
- MonitoringEffect, 56
- nameCounter, 54
- natTrace, 26
- nominalTimeToMicroseconds, 35
- nullTracer, 25
- ObservableInstance, 59
 - GhcRtsStats, 59
 - IOStats, 59
 - MemoryStats, 59
 - MonotonicClock, 59
 - NetStats, 59
 - ProcessStats, 59
- ObserveClose, 56
- ObserveDiff, 56
- ObserveOpen, 56
- parseRepresentation, 52
- Port, 52
- PrivacyAnnotation, 58
 - Confidential, 58
 - Public, 58
- readLogBuffer, 81
- readRTSStats, 36
- Representation, 52
- RotationParameters, 61
 - rpKeepFilesNum, 61
 - rpLogLimitBytes, 61
 - rpMaxAgeHours, 61
- ScribeDefinition, 60
 - scKind, 60
 - scName, 60
 - scPrivacy, 60
 - scRotation, 60
- ScribeFormat
 - ScJson, 60
 - ScText, 60
- ScribeId, 60
- ScribeKind
 - DevNullSK, 59
 - FileSK, 59
 - JournalSK, 59
 - StderrSK, 59
 - StdoutSK, 59
- ScribePrivacy, 60
 - ScPrivate, 60
 - ScPublic, 60
- setupTrace, 34
- Severity, 61
 - Alert, 61
 - Critical, 61
 - Debug, 61
 - Emergency, 61
 - Error, 61
 - Info, 61
 - instance of FromJSON, 61
 - Notice, 61
 - Warning, 61
- shutdown, 35
- singletonStats, 49
- Stats, 47
 - instance of Semigroup, 48
- stats2Text, 48
- stdoutTrace, 32
- stdoutTracer, 26
- SubTrace, 62
 - DropOpening, 62
 - FilterTrace, 62
 - NameOperator, 62
 - NameSelector, 62
 - Neutral, 62
 - NoTrace, 62
 - ObservableTrace, 62
 - ObservableTraceSelf, 62
 - SetSeverity, 62
 - TeeTrace, 62
 - UntimedTrace, 62
- Switchboard, 77
 - instance of IsBackend, 79
 - instance of IsEffectuator, 78
 - setupBackends, 82
- testSeverity, 65

- testSubTrace, 75
- ToLogObject, 64
 - toLogObject, 64
- ToObject, 64
 - toObject, 64
- toRepresentation, 74
- Trace, 63
- TraceAcceptor, 123
- TraceForwarder, 124
 - instance of IsBackend, 125
 - instance of IsEffectuator, 125
- traceInTVar, 33
- traceInTVarIO, 33
- traceMock, 84
- traceNamedItem, 33
- traceNamedObject, 32
- Tracer, 25
 - instance of Contravariant, 25
 - instance of Monoid, 25
- traceWith, 25

- updateAggregation, 115
- utc2ns, 55

- waitForTermination, 81
- withTrace, 35