ndtime js f

Pros:

* offers fine grained control on what gets logged where, it is very obvious how reporting works at an instance level

Cons:

* could be confusing to have two ways of adding differently scoped loggers.
* configuration scattered wherever there is an instance of logger

Option 2. Filtering by Name

We could filter by name and only support global level reporting.

Log.reporter(new ConsoleReporter()); // Adds reporter to all log instances

let log = new Log(‘http’);

Log.reporter(‘http’, new MyPersonalizedReporter()); // Adds reporter to just ‘http’ instance

Pros:

* offers fine grained control on what gets logged where.
* one way to add reporters

Cons:

* instead of instance level reporting the logger would have a global reporter
* does not filter by log level

Option 3. A Configuration Object for Filtering

We could filter by a lot of characteristics using a configuration object.

// Filters by log level `info` and logger named `http`

Log.reporter(new ConsoleReporter(), {

level: ‘info’,

logger: ‘http’

});

Log.reporter(new MyPersonalizedReporter()); // gets the firehose

Pros:

* Easy to offer more configuration options in the future (filter by regex, multiple levels using arrays etc), more keys etc
* one way to add reporters
* The configuration side can easily be replaced with a function that returns an object so log configuration could easily support environments and persistence (the localstorage conf story). E.G. Below

Cons:

* a bit more verbose

E.G.

// LoggerConfiguration could persist in local storage as well as check an environment

Log.reporter(new ConsoleReporter(), LoggerConfiguration(‘development’));

// Later in the app

Log.reporter(new CustomReporter(), {info: ['http'], warn: '\*'});

Log.config([{‘CustomReporter’, { info: [‘http’] }, …]);

Log.config(localStorage.diaryConfig['development'])

Option 4. Single config object

What if we had a config object that would define all of the config options and all of the logger instances would receive this object. The object could look like this:

{

default: {

warn: ['console'],

info: ['websocket']

},

http: {

debug: ['timeline', 'console']

},

otherComponent: {

info: ['console']

}

}

This object could then be persisted in local storage and conveniently edited via a browser extension.

A logger could then be created as:

let log = new Logger('http', configObject);

But what if we need to change the configuration at runtime?

We could use O.o() to observe the config object for changes, but that requires ability to ensure that only one instance of the config object exists and everyone who needs to mutate the object can get hold of it. This is not ideal.

One nice benefit (and also drawback) to this approach is that we could create component loggers completely independently from each other. Each logger would instantiate its reporters and we don't need to worry about global state.

This however also means that if a reporter tries to do some kind of batching or grouping, this grouping will be only for messages from the given logger/component and we won't be able to process messages in cross-component fashion. This could be a big limitation.

Option 5. Single config object + root logger

Variation of #4 where the config object is passed into the constructor of root logger and then root logger is passed into the constructor of module loggers:

let rootLog = new Logger(configObject);

let httpLog = new Logger('http', rootLog);

With this approach module loggers become just decorators that attach information about the module to each message and then relay the message to the root logger.