Bolt – version 1.0-beta

http://bolt.x9c.fr

May 31, 2009

Abstract: This document presents Bolt, its purpose and the way it works. This document is structured in three parts explaining how to build, how to use, and how to customize Bolt.

Introduction

Bolt is a logging tool for the Objective Caml language[1]. Its name stems from the following acronym: *Bolt is an Ocaml Logging Tool*. It is inspired by and modeled after the Apache log4j utlity[2].
Bolt provides both a comprehensive library for log handling, and a camlp4-based syntax extension that allows to remove log directives. The latter is useful to be able to distribute an executable that incurs no runtime penalty if logging is used only during development.

Bolt, in its 1.0-beta version, is designed to work with version 3.11.0 of Objective Caml.
Bolt is released under the LGPL version 3.
Bugs should be reported at `http://bugs.x9c.fr`.

Building Bolt

Bolt can be built from sources using `make`, and Objective Caml version 3.11.0. Under usual circumstances, there should be no need to edit the `Makefile`. Bolt is compiled by executing the command `make all` and installed by executing the command `make install` with root privileges. The following targets are available:

`all` compiles all versions, and generates html documentation;

`bytecode` compiles the bytecode version (ocamlc);

`native` compiles the bytecode version (ocamlopt);

`java` compiles the bytecode version (ocamljava);

`html-doc` generates html documentation;

`clean-all` deletes all produced files (including documentation);

[1] The official Caml website can be reached at `http://caml.inria.fr` and contains the full development suite (compilers, tools, virtual machine, *etc.*) as well as links to third-party contributions.
[2] `http://logging.apache.org/log4j`

`clean` deletes all produced files (excluding documentation);

`clean-doc` deletes documentation files;

`install` copies library files;

`tests` runs the tests;

`depend` generates dependency files.

The Java[3] version will be built only if the `ocamljava`[4] compiler is present and located by the makefile. The syntax extension will be compiled only to bytecode.

Using Bolt

Bolt is based on the following concepts:

Event: the event is the entity built each time the application executes a log statement.

Level: the level characterizes how critical an event is.
An event will be recorded iff its level is below the level of logger.
The levels are, in asending order: `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, and `TRACE`.

Filter: each logger has an associated filter, ensuring that only the events satistying the filter will be recorded.

Layout: each logger has an associated layout that defines how an event is rendered into a string.

Output: each logger has an associated output that defines where event are actually recorded (two loggers should not have the same destination).

Loggers have names that are strings composed of dot-separated components; they are thus akin to module names, and it is actually good practice to use the logger `M` to log events of the module `M`. It is possible to register several loggers with the same name; this feature is useful to record the events related to a given module to several different destinations (using possibly different filters, layout, and outputs).

Logger are also organized into a hierarchy (meaning that logger `P` is a parent of logger `P.S`). When a log statement is executed, it is associated with a logger name; the event will be presented to all logger with that name, and to all loggers with a parent name. Each logger will decide according to its level and filter if the event should actually be recorded. Finally, all events are presented to all loggers having the special empty name (corresponding to the string `""`). The hierarchy of the loggers is a key feature that allows to easily enable or disable logging for large parts of an application.

[3] The official website for the Java Technology can be reached at http://java.sun.com.
[4] OCaml compiler generating Java bytecode, by the same author – http://ocamljava.x9c.fr

Adding log statements

There are two ways to add a log statement: either by calling explicitly the `Bolt.Logger.log` function, or by using the `bolt_pp.cmo` camlp4 syntax extension. One is advised to use the latter method: first, using the syntax extension is lightweight (elements such as line and column are automatically computed); second, it allows to remove the log statements at compilation (it is useful to have a development version packed with a lot of debug log statements and a distributed version that suffer no runtime penalty related to logging).

To log using the `Bolt.Logger.log` function, one has to call it with the following parameters (*cf.* code sample 1):

- a `string` parameter giving the name of the logger to use;
- a `Bolt.Level.t` parameter giving the level of the event to log;
- an optional `string` parameter (named *file*) giving the file associated with the log event;
- an optional `int` parameter (named *line*) giving the line number associated with the log event;
- an optional `int` parameter (named *column*) giving the column number associated with the log event;
- an optional `(string * string) list` parameter (named *properties*) giving the property list associated with the log event;
- an optional `exn option` parameter (named *error*) giving the exception associated with the log event;
- a `string` parameter giving the message of the log event.

Code sample 1 Explicit logging.

```
let () =
  ...
  Bolt.logger.log "mylogger" Bolt.Level.DEBUG "some debug info";
  ...
```

To log using the syntax extension, one has to use the Bolt-introduced log expression. This is done by passing the `-pp 'camlp4o /path/to/bolt_pp.cmo'` option to the OCaml compiler. The new `LOG` expression can be used in an OCaml program wherever an expression is waited. The BNF definition of this expression is as follows:

log_expr ::= LOG *string attributes* LEVEL *level*

attributes ::= *attributes attribute* | ϵ

attribute ::= NAME *string* | PROPERTIES *expr* | EXCEPTION *expr*

level ::= FATAL | ERROR | WARN | INFO | DEBUG | TRACE

The string following the `LOG` keyword is the message of the log event. The attributes are optional, and have the following meaning:

`NAME` defines the name of the logger to be used;

`PROPERTIES` defines the properties associated with the log event (the expression should have the type `(string * string) list`);

`EXCEPTION` defines the exception associated with the log event (the expression should have type `exn`).

Code sample 2 shows how the expression can be used. Compared to explicit logging through the `Bolt.Logger.log`, when using the `LOG` expression file, line number and column number are determined automatically.
When no `NAME` attribute is provided, the logger name is computed from the source file name: the `.ml` suffix is removed and the result is capitalized. More, the `bolt_pp.cmo` accepts the following parameters:

- `-logger <n>` sets the logger name to n for all `LOG` expression of the compiled file;
- `-for-pack <P>` sets the prefix to the logger names used throughout the compiled file to "`P.`".

Finally, the `bolt_pp.cmo` syntax extension recognizes a third parameter `-level <l>` where l should be either `NONE` or a level. If l is `NONE`, all `LOG` expression will be removed from the source file; otherwise, only the `LOG` expression with a level inferior or equal to the passed value will be kept.

Code sample 2 Implicit logging.

```
let () =
  ...
  LOG "some debug info" LEVEL DEBUG;
  ...
```

Configuring log

There are two ways to configure log, that is to register loggers that will handle the log events produced by the application. The first way is to explicitly call `Bolt.Logger.register` while the second one is to use a configuration file that will be interpreted by Bolt at runtime.

To register (*i.e.* to create) a logger using the `Bolt.Logger.register` function, one as to call it with the following parameters:

- a `string` parameter giving the name of the logger;
- a `Bolt.Level.t` parameter giving the maximum level for events to be logged;
- a `string` parameter giving the filter of the logger;
- a `string` parameter giving the layout of the logger;
- a `string` parameter giving the output of the logger;
- a `string * float option` couple that gives the parameters used for output creation: the first component is the name of the output while the second one is the optional rotate value (the actual semantics of both component is dependent on the actual output used).

To register a logger using a configuration file, one should set the `BOLT_FILE` environment variable to the path of the configuration file. If the configuration file cannnot be loaded, an error message is written on the standard error unless the `BOLT_SILENT` environment variable is set to either "YES" or "ON" (defaulting to "OFF", case being ignored).

The format of the configuration file is as follows:

- the format is line-oriented;
- comments start with the '#' character and end at the end of the line;
- sections start with a line of the form `[a.b.c]`, "a.b.c" being the name of the section;
- a section ends when a new section starts;
- at the beginning of the file, the section named "" is currently opened;
- section properties are defined by lines of the form "key=value";
- others lines should be empty (only populated with whitespaces and comments).

Each section defines a logger whose name is the section name. The following properties are used to customize the logger:

- `level` defines the level of the logger;
- `filter` defines the filter of the logger;
- `layout` defines the layout of the logger;
- `output` defines the output of the logger;
- `name` is the first parameter passed to create the actual output;
- `rotate` is the second parameter passed to create the actual output.

The level can have one of the following values: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`. The possible values for the other properties are discussed in the following sections.
Code sample 3 examplifies a typical configuration file. It defines three loggers (with names "", "`"Pack.Main"`", and "`"Pack.Aux"`"). When executed, the application will produce three files "`bymodule.result`", "`bymodule1.result`", and "`bymodule2.result`": the first file will contain the log information for the whole application while the other ones will contain respectively the log information associated with the "`Pack.Main`" and "`Pack.Aux`" loggers.

Predefined filters

The following filters are predefined:

- `all` keeps all events;
- `none` keeps no event;
- `trace_or_below` keeps events with level inferior or equal to `TRACE`;
- `debug_or_below` keeps events with level inferior or equal to `DEBUG`;

Code sample 3 Example of configuration file.

```
level=trace
filter=all
layout=simple
output=file
name=bymodule.result

[Pack.Main]
level=trace
filter=all
layout=simple
output=file
name=bymodule1.result

[Pack.Aux]
level=trace
filter=all
layout=simple
output=file
name=bymodule2.result
```

- `info_or_below` keeps events with level inferior or equal to `INFO`;
- `warn_or_below` keeps events with level inferior or equal to `WARN`;
- `error_or_below` keeps events with level inferior or equal to `ERROR`;
- `fatal_or_below` keeps events with level inferior or equal to `FATAL`;
- `file_defined` keeps events with an actual filename;
- `file_undefined` keeps events with no filename;
- `line_defined` keeps events with a strictly positive line number;
- `line_undefined` keeps events with a negative or null line number;
- `column_defined` keeps events with a strictly positive column number;
- `column_undefined` keeps events with a negative or null column number;
- `message_defined` keeps events with a non-empty message;
- `message_undefined` keeps events with an empty message;
- `properties_empty` keeps events with an empty property list;
- `properties_not_empty` keeps events with an non-empty property list;
- `exception_some` keeps events with an exception;
- `exception_none` keeps events with no exception.

Predefined layouts

Bolt predefines the following non-configurable layouts:

- `simple` with format: `LEVEL - MESSAGE`;
- `default` with format: `TIME [FILE LINE] LEVEL MESSAGE`;
- `html` whose format is HTML, storing events into a table;
- `xml` whose format is XML.

Two other layouts are predefined:

- `pattern` whose actual format is specified by defining a property named `pattern`
  This property is a string that can contain `$(x)` elements where x is a key (defined below) or `$(x:n)` where x is a key and n is a padding instruction (the absolute value of n is the total width; the padding is left is n is negative, and right if n is positive)
- `csv` whose actual format is specified by properties named `csv-separator` and `csv-elements`
  `csv-separator` is the string to be used as the separator between values
  `csv-elements` is a whitespace-separated list of the keys of the values to render

The following keys are available for use by the `pattern` and `csv` layouts:

- `id` event identifier;
- `sec` seconds of event timestamp;
- `min` minutes of event timestamp;
- `hour` hour of event timestamp;
- `mday` day of month of event timestamp;
- `month` month of year of event timestamp;
- `year` year of event timestamp;
- `wday` day of week of event timestamp;
- `time` event timestamp;
- `relative` time elapsed between initilization and event creation;
- `level` event level;
- `logger` event logger;
- `file` event file;
- `filebase` event file (without directory information);
- `line` event line;
- `column` event column;

- `message` event message;
- `properties` property list of event (format: ["[k1: v1; ...; kn: vn]"]);
- `exception` event exception;
- `backtrace` event exception backtrace.

Predefined outputs

There are two predefined outputs, namely `void` and `file`. The `void` output discards all data. The `file` output writes data to a bare file, the `name` property (or the `string` value when using `Bolt.Logger.register`) defines the path of the file to be used, and the `rotate` property (or the `float option` value when using `Bolt.Logger.register`) gives the rates in seconds at which files will be rotated. When using rotation, the name should contain a `%` character that is expanded to a timestamp, ensuring that a different file is created at each rotatation. If no `%` character is employed, the same file will be written over and over again.

Reviewing log

Once the log information has been produced by the application, the developper and/or the user will have to review it. Although this can easily be done using Unix commands (such as `grep`, `cut`, `sed`; *etc*), a GUI can be helpful. For this reason, the XML layout of Bolt produces log4j-compatible XML files allowing the use of the Chainsaw application[5].
Code sample 4 shows a XML file that could be used to wrap the XML data produced by Bolt (in `bolt.xml` file) in such a way that Chainsaw can load it. This code sample is a reproduction of the one provided in the Javadoc of the log4j `org.apache.log4j.xml.XMLLayout` class[6].

Code sample 4 Wrapping produced XML data into a Chainsaw-compatible XML.

```
<?xml version="1.0"?>

<!DOCTYPE log4j:eventSet SYSTEM "log4j.dtd" [<!ENTITY data SYSTEM "bolt.xml">]>

<log4j:eventSet version="1.2" xmlns:log4j="http://jakarta.apache.org/log4j/">
        &data;
</log4j:eventSet>
```

Customizing Bolt

It is possible to customize Bolt by defining new filters, layouts, and outputs. This is easily done by using respectively the `Bolt.Filter.register`, `Bolt.Layout.register`, and `Bolt.Output.register` functions. More information about the actual types of these functions can be found in the `ocamldoc`-generated documentation (available in the `ocamldoc` directory, generation being triggered by the `make html-doc` command).

[5] http://logging.apache.org/chainsaw/
[6] http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/xml/XMLLayout.html

When custom elements have been registered using the previously mentioned functions, they can be used from the configuration files or from the `Bolt.Logger.register` function. However, it is necessary for the custom elements to be registered before *any* log event concerned with theses custom elements is built. Otherwise, elements won't be found and Bolt will resort to default values.

Code sample 5 shows how to register a new filter that keeps only event with an even line number, and a new layout programmed using the `Printf.sprintf` machinery.

Code sample 5 Customizing Bolt with new filter and layout.

```
let () =
  Bolt.Filter.register
    "myfilter"
    (fun e -> (e.Bolt.Event.line mod 2) = 0)

let () =
  Bolt.Layout.register
    "mylayout"
    ([],
     [],
     (fun e ->
       Printf.sprintf "file \"%s\" says \"%s\" with level \"%s\" (line: %d)"
         e.Bolt.Event.file
         e.Bolt.Event.message
         (Bolt.Level.to_string e.Bolt.Event.level)
         e.Bolt.Event.line))
```