Bolt – version 1.2

http://bolt.x9c.fr

August 15, 2011

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 production, 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.

The importance of logging is frequently overlooked, but (quite ironically) in the same time, the most used debugging *method* is by far the `print` statement. Bolt aims at providing Objective Caml developpers with a framework that is comprehensive, yet easy to use. It also tries to leverage the benefits of both compile-time and run-time configuration to produce a flexible library with a manageable computational cost.

Bolt, in its 1.2 version, is designed to work with version 3.12.1 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` (in its `GNU Make 3.81` flavor), and Objective Caml version 3.12.1. No other dependency is needed. Following the classical Unix convention, the build and installation process consists in these three steps:

1. `sh configure`
2. `make all`
3. `make install`

During the first step, one can specify elements if they are not correctly inferred by the `./configure` script; the following switches are available:

[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`

- `-ocaml-prefix` to specify the prefix path to the Objective Caml installation (usually `/usr/local`);
- `-ocamlfind` to specify the path to the `ocamlfind` executable (notice that the presence of `ocamlfind`[3] is optional, and that the tool is used only at installation if present);
- `-no-native-dynlink` to disable dynamic linking.

During the third and last step, according to local settings, it may be necessary to acquire privileged accesses, running for example `sudo make install`.
The Java[4] version will be built only if the `ocamljava`[5] compiler is present and located by the makefile. The syntax extension will be compiled only to bytecode.

Using Bolt

Base concepts

The central concept of Bolt is loggers. 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. Figure 1 shows the hierarchy of loggers for an application using the loggers whose name appears in black. The loggers whose names appear in gray are implicitly added by Bolt in order to have a complete tree of loggers: those actually used in the program are the leaves, and the root is the special `""` logger. The arrows define the is-a-child-of relation.

Every log 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. Figure refdispatch shows how a message initially created for the `Library.PartB.Module` loggers is dispatched to all loggers with parent names, including loggers that are not explicitly used in the application (those whose name appears in gray). The dashed arrows show the order in which the event is presented to the different loggers.

Bolt is also 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`.

[3] Findlib, a library manager for Objective Caml, is available at http://projects.camlcity.org/projects/findlib.html

[4] The official website for the Java Technology can be reached at http://java.sun.com.

[5] Ojective Caml compiler generating Java bytecode, by the same author – http://ocamljava.x9c.fr

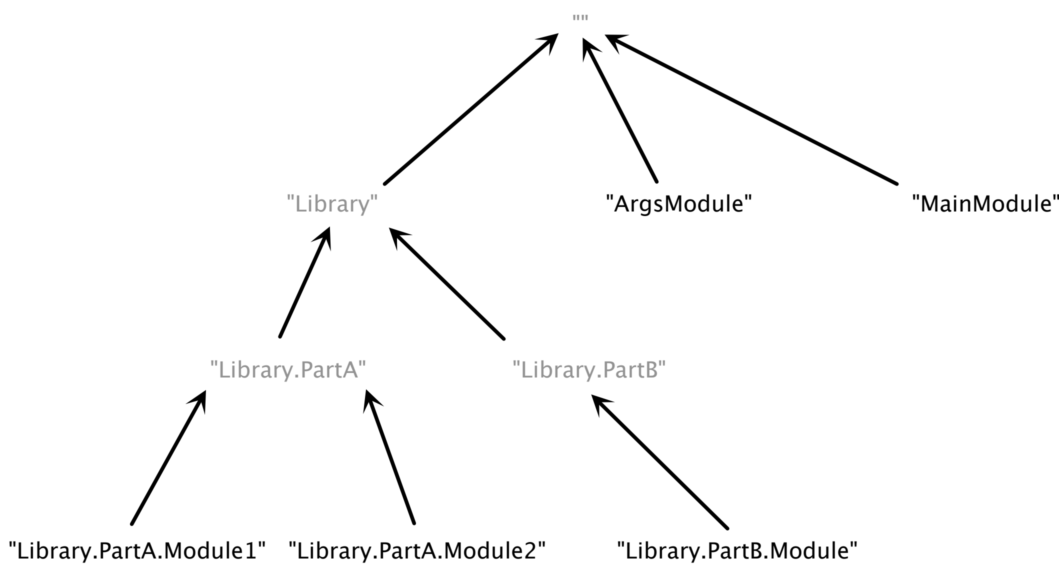

Figure 1: Example of logger hierarchy.

- **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).

Linking with the library

Linking with Bolt is usually done by adding one of the following library to the linking command-line:

- `-I +bolt bolt.cma` (for `ocamlc` compiler);
- `-I +bolt bolt.cmxa` (for `ocamlopt` compiler);
- `-I +bolt bolt.cmja` (for `ocamljava` compiler).

In order, to use Bolt in multithread applications, it is necessary to also link with the `BoltThread` module. This also implies to pass the `-linkall` option to the compiler.

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 may be useful to have a development version packed with a lot of debug log statements and a distributed version that suffers no runtime penalty related to logging). Moreover, only a given part of log statements may be removed, on a level basis.

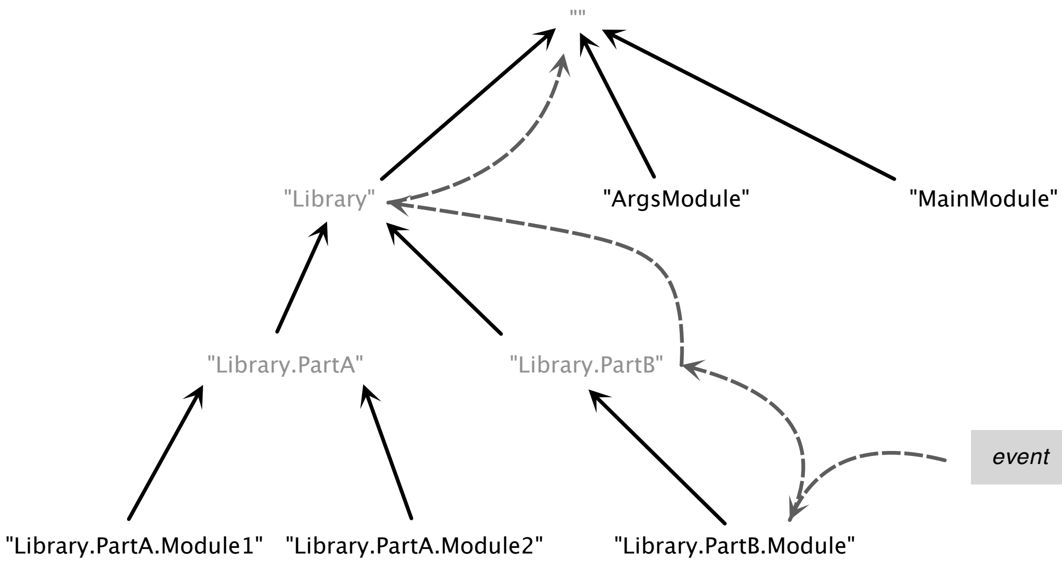

Figure 2: Dispatch of an event generated for the "`Library.PartB.Module`" logger.

Explicit 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";
  ...
```

Implicit logging

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 Objective Caml compiler. The new `LOG` expression can be used in an Objective Caml program wherever an expression of type `unit` is waited. The BNF definition of this expression is as follows:

log_expr ::= `LOG` (*string* | *ident*) *arguments* *attributes* `LEVEL` *level*
arguments ::= *list of expressions* | ϵ
attributes ::= *attributes* *attribute* | ϵ
attribute ::= `NAME` *string* | (`PROPERTIES` | `WITH`) *expr* | (`EXCEPTION` | `EXN`) *expr*
level ::= `FATAL` | `ERROR` | `WARN` | `INFO` | `DEBUG` | `TRACE`

The string following the `LOG` keyword is the message of the log event, it can be either a literal string or an identifier whose type is string. This string can be followed by expressions; in this case the string is interpreted as a `printf` format string, using the following expressions as values for the `%` placeholders of the format string.

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` syntax extension accepts the following parameters:

- `-logger <n>` sets the logger name to n for all `LOG` expressions 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` expressions 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;
  ...
```

When compiling in *unsafe* mode, the `-unsafe` switch should be passed to camlp4 instead of the compiler. Indeed, as camlp4 is building a syntax tree that is passed to the compiler, issuing the `-unsafe` switch to the compiler has no effect because it is too late: the code has been built by camlp4 in *safe* mode. In such a case, the compiler warns the user with the following message: `Warning: option -unsafe used with a preprocessor returning a syntax tree`. The correct command-line switch is hence `-pp 'camlp4o -unsafe /path/to/bolt_pp.cmo'`.

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 has 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.

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
```

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`;
- `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`;
- `paje`, and `paje_noheader` whose format is the Pajé trace format[6] (the two format only differ in that the latter one does not output definitions, which is useful when one wants to merge several files);
- `daikon_decls`, and `daikon_trace` that respectively follow Daikon[7] declaration (*i.e.* program points, and associated variable types) and trace format (*i.e.* actual variable values for the various program points visits);
- `html` whose format is HTML, storing events into a table;
- `xml`, or `log4j` whose format is XML (compatible with log4j).

[6] http://sourceforge.net/projects/paje/
[7] http://groups.csail.mit.edu/pag/daikon/

Pajé layouts

The Pajé layout support the file format as defined at https://gforge.inria.fr/projects/paje/; however, Bolt does not support the extensibility feature of the Pajé format. This means that only the kinds of events predefined by the standard are available. Nevertheless, it is still possible to add new fields to predefined events.
Code sample 4 shows how the functions from the `Paje` module could be used to record the fact that a container *cnt* change its state when receiving and handling a mail. The full list of supported event can be found in the ocamldoc of the `Paje` module.

Code sample 4 Pajé example.

```
LOG Paje.t
  PROPERTIES Paje.new_event ~typ:"mail" ~container:"cnt" ~value:msg []
  LEVEL TRACE;
LOG Paje.t
  PROPERTIES Paje.set_state ~typ:"state" ~container:"cnt" ~value:"computing" []
  LEVEL TRACE;
(...)
LOG Paje.t
  PROPERTIES Paje.set_state ~typ:"state" ~container:"cnt" ~value:"waiting" []
  LEVEL TRACE;
```

Daikon layouts

When using the Daikon tool, one is usually interested in having both the declaration and the traces for the program to analyze. As a result, the configuration file is similar to the one depicted in 5. The program to be analyzed should itself contain log statement to record information to be fed to the Daikon analyzer. Program 6 shows a simple program producing Daikon data.

The result of Daikon analysis with the aforementioned log configuration and program will be the following:

```
===========================================================================
f:::ENTER
===========================================================================
f:::EXIT1
"x" == orig("x")
"res" one of { 0, 1 }
"res" <= "x"
```

Each Daikon-related element should use `Daikon.t` as the log message, and one of the property-building functions from the `Daikon` module to build a list of element. As of version 1.2, these functions are:

- `enter` that is used to mark the start of a function, giving its name and parameters;
- `exit` that is used to mark the end of a function, giving its name, return value and parameters;
- `point` that can be used to mark any point in a program, associating it with a list of values.

Values, independently of their *kind* (parameters, return values, bare variables) are encoded using a variable-build function from the `Daikon` module. All these functions take as first parameter the name of the value, and as second parameter the value itself. As of version 1.2, they are:

- t for type t;
- t`_list` for type t `list`;
- t`_array` for type t `array`;

where t is one of `bool`, `int`, `float`, or `string`.

Code sample 5 Daikon configuration.

```
[]
level=trace
filter=all
layout=daikon_decls
output=file
name=daikon.decls

[]
level=trace
filter=all
layout=daikon_dtrace
output=file
name=daikon.dtrace
```

Code sample 6 Daikon-instrumented program.

```
 let f x =
  LOG Daikon.t
    WITH Daikon.enter "f" [Daikon.int "x" x] LEVEL TRACE;
  let res = (x * x) mod 2 in
  LOG Daikon.t
    WITH Daikon.exit "f" (Daikon.int "res" res) [Daikon.int "x" x] LEVEL TRACE;
  res

let () =
  let l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10] in
  let l = List.map f l in
  List.iter (Printf.printf "%d\n") l
```

Pattern and comma-separated layouts

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)
it is also possible to specify through the `pattern-header-file` (respectively `pattern-footer-file`) property the name of a file whose contents is used as the header (respectively footer) that is written at start/end as well as at each rotation

- `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;
- `hostname` host name of running program;
- `process` process identifier of running program (*i.e. pid*);
- `thread` thread 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;
- `origin` first logger that received the event;
- `file` event file;
- `filebase` event file (without directory information);
- `line` event line;
- `column` event column;
- `message` event message;
- `properties` property list of event (formatted as ["[k1: v1; ...; kn: vn]"]);
- `exception` event exception;
- `backtrace` event exception backtrace.

Predefined outputs

There are three predefined outputs, namely `void`, `growlnotify`[8], 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[9], 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. It is also possible to use the `signal` property (set to one one the following values: `SIGHUP`, `SIGUSR1`, `SIGUSR2`) in order to request rotation upon signal reception.

When using rotation or several program instances in parallel, it is convenient for the name to contain a piece of information ensuring that the file name will be unique; otherwise, the same file will be written over and over again. In version 1.0, Bolt supported the `%` special character that was substituted by a timestamp. Since version 1.1, Bolt additionally supports a more general `$(key)` substitution mechanism with the following keys:

- `time` as a bare alternative to `%`;
- `pid` that designates the process identifier;
- `hostname` that designates the process hostname (useful when using a shared file system);
- *var* that designates any environment variable available from the process.

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 classical Unix commands (such as `grep`, `cut`, `sed`; *etc*), a dedicated tool such as a GUI can be helpful. For this reason, the XML layout of Bolt produces log4j-compatible XML files allowing the use of the Apache Chainsaw application[10]. Code sample 7 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[11].

Code sample 7 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>
```

[8] Command-line utility associated with the Growl program available at http://growl.info/

[9] Two special filenames are recognized: <`stdout`> for standard output, and <`stderr`> for standard error.

[10] http://logging.apache.org/chainsaw/

[11] http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/xml/XMLLayout.html

Complete example

Code sample 8 shows a short program using the implicit logging feature of Bolt. The program can be compiled and executed by the Makefile shown by code sample 9. The `compile` target underlines that compilation should be done through the Bolr preprocessor, and that link entails references to the `str`, `unix`, and `dynlink` libraries (all of them being shipped with the standard Objective Caml distribution).

Code sample 8 Source example.

```
let funct n =
  LOG "funct(%d)" n LEVEL DEBUG;
  for i = 1 to n do
    print_endline "..."
  done

let () =
  LOG "application start" LEVEL TRACE;
  funct 3;
  funct 7;
  LOG "application end" LEVEL TRACE
```

Code sample 9 Makefile example.

```
DEPENDENCIES=str.cma unix.cma dynlink.cma

default: clean compile run view

clean:
	rm -f *.cm* log bytecode

compile:
	ocamlc -c -I +bolt bolt.cma \
	    -pp 'camlp4o path/to/bolt/bolt_pp.cmo' source.ml
	ocamlc -o bytecode -I +bolt $(DEPENDENCIES) bolt.cma source.cmo

run:
	BOLT_FILE=config ./bytecode

view:
	cat log
```

The target `run` of the Makefile shows that the environment variable `BOLT_FILE` should be set to the path of the configuration file defining the actual runtime-configuration of logging. The related configuration file is represented by code sample 10. As a result of execution, a plain text file named `log` will be produced, and can be viewed using the `view` target of the Makefile.

Code sample 10 Configuration example.

```
level=trace
filter=all
layout=default
output=file
name=log
```

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).

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.

A good practice is to define the new filters, layouts, and outputs in modules that are not part of the application. One should not forget to pass the `-linkall` switch to the compilers when linking such modules. Another option is to avoid linking these modules with the application, and to use the `BOLT_PLUGINS` environment variable to load them. The `BOLT_PLUGINS` environment variable contains a comma-separated list of files that will be loaded through `Dynlink`.

Code sample 11 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 11 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))
```