Advanced TraceUtils

Hobo DRYML is perhaps the most advanced templating language available for Ruby (and Rails) and is one of the core features of Hobo. The DRYML internals are however not currently very well documented. I had a wish to integrate the ExtJS javascript frameworks, mainly the widgets for use in Hobo through DRYML. I also thought it would be nice to be able to create PDF and similar builder output using DRYML. However I soon discovered that DRYML was specifically created for outputting HTML (or simple concatenated text output) at its core.

In order extend it for other scenarios, I would have to understand its inner workings and then tweak DRYML and bend it to my will. I started off by inserting logging statements but it was soon obvious that something more advanced would be useful. From this requirement, `adv-trace-util` was born.

This chapter describes what `adv-trace-util` is, why you might find it useful and how it can be used for tracing DRYML. Tracing makes it easier to understand how DRYML works "under the hood", how to extend it (if you need to) and exactly what kind of .erb code it generates from the tag definitions, tag calls etc.

Installation and configuration

To use the gem, complete these steps:

1. Install `gemcutter`
2. Install the `adv-trace-util` gem
3. Create a new project using edge Hobo
4. Configure Hobo to use the `adv-trace-util` gem

Install gemcutter

Gemcutter is the latest and greatest gem hosting service on the web, a replacement for github and rubyforge when it comes to ruby gems. To install it:

Install the latest Rubygems (1.3.3 or greater required):

```
$ gem update --system
```

Install the Gemcutter gem from Rubyforge:

```
$ gem install gemcutter
```

Use the built-in command to make gemcutter.org your primary gem source:

```
$ gem tumble
```

Install the adv-trace-util gem

```
$ gem install adv-trace-util
```

This should download and install the gem from gemcutter.

Create a new project using edge hobo

There are several ways to do this. One option is to simply install Hobo as a plugin by cloning it from github into you `plugin` folder. Then remove the `config.gem 'hobo'` line from `environment.rb`.

Another approach is to install my bash script helpers from `ruby-rails-dev-bashing`, at http://github.com/kristianmandrup/ruby-rails-dev-bashing, which includes a function to set up a project with edge hobo with a simple one liner from the console.

```
$ hb_edge my_project_name
```

Configure Hobo to use the gem

Copy the `dryml_trace` folder into the lib folder of your Hobo project.
Inside the `dryml_trace` folder is a folder called `appenders`. This folder contains sample appenders specifically created to be used when tracing DRYML code.
In `environment.rb` add the following:

```
require 'dryml_trace/config'
```

The file `config.rb` is where you should configure DRYML tracing for your Hobo project.
The following sections will explain the general design of `adv-trace-util` and how to use it.

Architecture

The following figure illustrates the conceptual architectural design of TraceUtils.
TraceExt is a mixin of TraceCalls. The configuration of TraceUtils is done by registering different TraceUtil related objects on TraceExt, which become available to TraceCalls.
In a later version, the architecture might change to use the Singleton pattern instead, using the Ruby Singleton library.

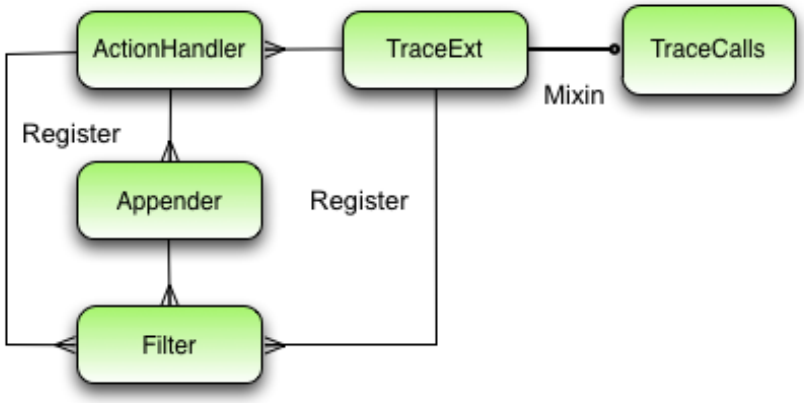

Figure 1 TraceUtils configuration

The filters registered directly with TraceExt are used to decide for which methods TraceUtil (and thus tracing) should be applied at all.

TraceUtil execution

When a method configured for tracing is called, a trace message is sent to TraceCalls which delegates the trace to each of its registered ActionHandlers. Each ActionHandler in turn delegates the trace to each of its registered Appenders. The Appenders calls its registered Tracer to generate the final output (trace) to send to the target of the appender.

The following figure illustrates the main flow of a traced method, and how the tracing aspects are delegated to TraceCalls to handle.

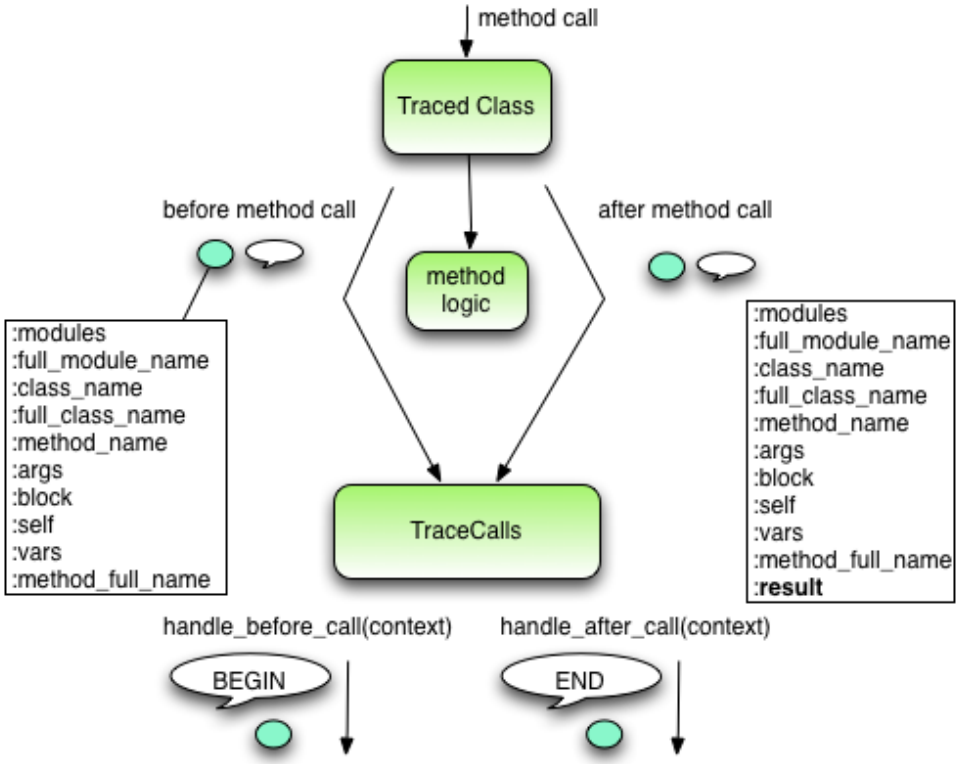

Figure 2 TraceCalls flow

The method bef

The figure below illustrates the program flow within TraceCalls when it receives a trace message.

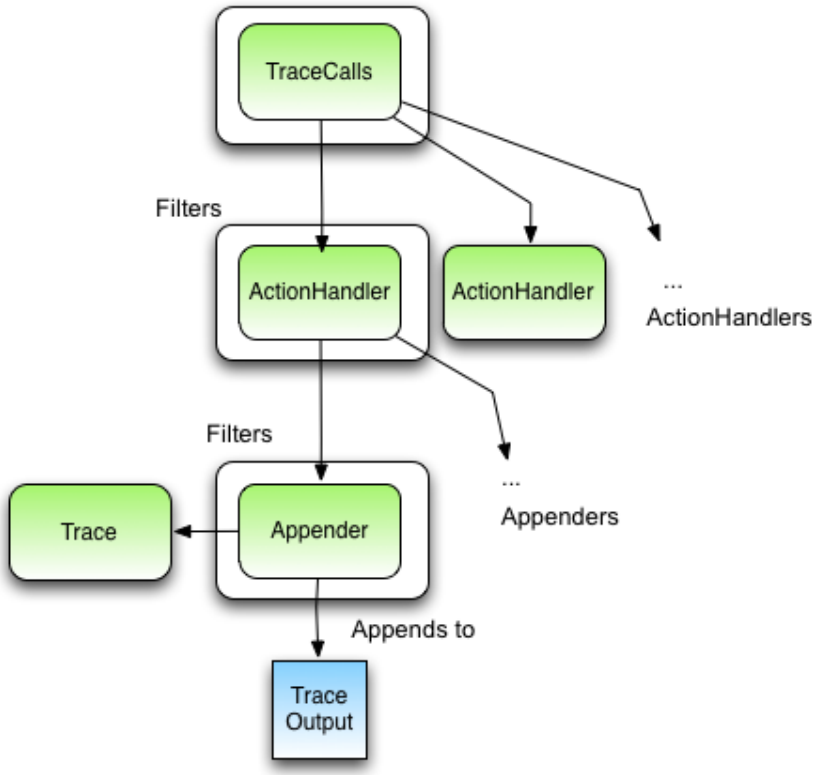

Figure 3 TraceUtil execution flow

Using TraceUtils with Hobo and DRYML

The recommended way to use TraceUtils for tracing Hobo or DRYML, is to apply tracing in an unobtrusive fshion. This can be done like this:

```
dryml_classes = ['DRYMLBuilder']
dryml_template_classes = ['Template', 'TemplateEnvironment']
dryml_classes_to_trace = dryml_classes + template_classes

dryml_classes_to_trace.each do |cls|
  eval "Hobo::Dryml::#{cls}.class_eval { include Tracing::TraceCalls }"
end
```

Here we first construct an array of the classes to trace are set up. Then we iterate and for each class construct the full class name (including the module Hobo::Dryml) and then evaluate and code to apply on each class instance. We include Tracing::TraceCalls as a mixin on each class.

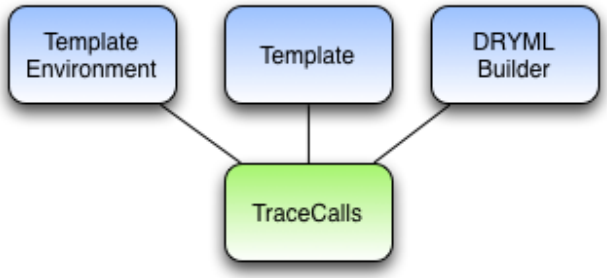

Figure 4 Configuring DRYML for tracing

When TraceCalls is mixed in, it will iterate all instance methods of the class it is mixed into, and see if it can wrap the methods with tracing aspects for when the method is called and when the method exits. TraceCalls uses the filters registered with TraceExt (a mixin to TraceCalls) to decide this for each method. The filters can be registered like this:

```
Tracing::TraceExt.configure(
  :appenders => basic_html_log_appender,
  :filters => method_filters
)
```

The :filters symbol can point to either an array of Tracing::BaseFilter or to a Hash or Array of Hashes. In order for a Hash to be parsed as a filter it must follow a specific syntax.
The following demonstrates such a Hash based filter for tracing DRYMLBuilder.

```
dryml_builder_method_filter = {
  :name => 'DRYMLBuilder method filter',
  :method_rules => [{
    :name => 'core builder methods',
    :include => dryml_builder_methods[:core] + dryml_builder_methods[:template],
    :exclude => ['add_build_instruction', 'template', 'template_path'],
    :default => :exclude
  }]
}
```

The filter contains a name and an array of rules. This filter is a list filter, since it can contain multiple rules. A simpler form is the following:

```
include_method_filter = {:imethod_filter => "method_a, method b"}
exclude_method_filter = {:xmethod_filter => "method_b, method c"}
```

Here we are stating to create a simple method filter, which includes or excludes the methods contained in the string. The string will be split using "," to get the list of methods. The prefix "i" says to create an include filter, "x" an exclude filter. These prefixes can be used in front of all the default types of filters included with TraceUtils:

```
  :method_filter, :class_filter, :module_filter, :vars_filter
```

In a coming version, the prefixes "iy" and "xy" will be added as convenient ways to add filters for `:include_and_yield` and `:exclude_and_yield` (see below)

Method filter is only one of the filters provided out-of-the-box by adv-trace-util.
The following figure illustrates some other filters that might be useful, especially when combined in a filter chain.

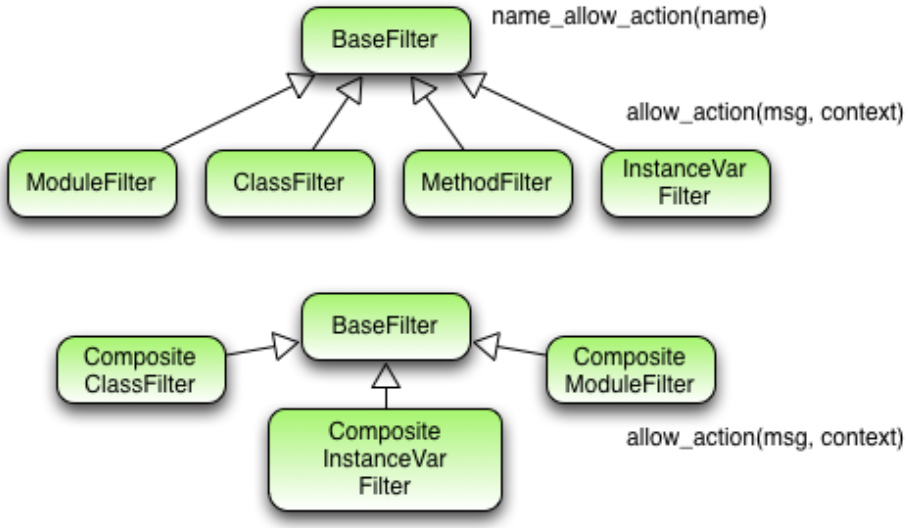

Figure 5 Filters

A class filter could be used in combination to ensure only methods within a certain class are allowed through. You could also create a CompositeClassFilter which can contains rules for methods of a specific set of classes that should be allowed. Same goes for Module- and InstanceVar filters.

Chaining filters

Filters can be chained to construct more complex filters or combined into composite filters. Examples of composite filters can be seen in `sample_filters.rb` in the source code for `TraceUtils`. Let us illustrate chaining filters using the previous example.

```
:filters => [include_method_filter, exclude_method_filter]
```

Notice that `method_b` occurs in both filters. The filter registration order is important.
The include filter will be applied first. If the class contains a `method_b` and it sees `method_b` is in the filter include list, the filter will break out of the filter chain and return `:include`.
Hence the method will be included for in that particular context.
The following are the types of rules you can setup within a filter.

```
:include              - include and end filter chain
:exclude              - exclude and end filter chain
:yield                - no decision, let following filters decide
:include_and_yield    - include for now, but let following filters overrule
:exclude_and_yield    - exclude for now, but let following filters overrule
```

The result of the filter chain can be either `:include`, `:exclude` or `:yield`. If the end result is `:yield` a global configuration `:final_yield_action` can be set on `TraceExt` to define if `:yield` should result in `:include` or `:exclude`. Simply set it as an extra parameter in the configure call or you can set it independently.

```
Tracing::TraceExt.configure(:final_yield_action  => :include)
Tracing::TraceExt.final_yield_action  => :include
```

The following figure illustrates the filter chain.

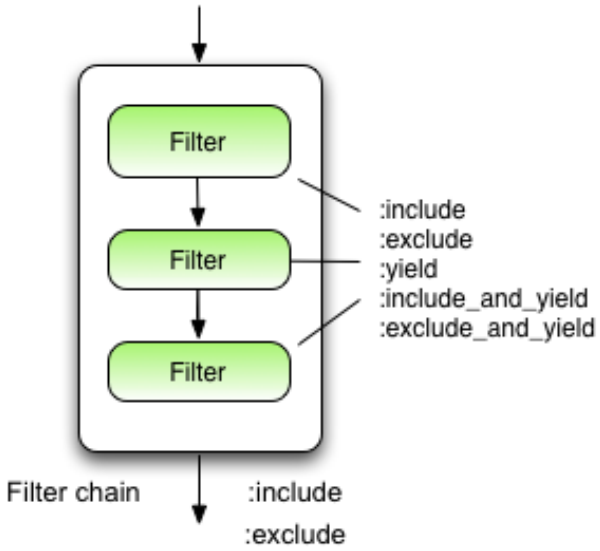

Figure 6 Filter chain

The figure demonstrates how the filters are called in turn and that the end result of calling the filter chain is always an `:exclude` or `:include` outcome which is translated to allow an action in the particular context or not, fx whether to allow an `Appender` to take effect in the particular context of a trace.

ActionHandlers

`ActionHandlers` are registered with `TraceExt` and are used to handle a trace.
Multiple ActionHandlers can be registered, and each `ActionHandler` is called in turn to handle the trace. An ActionHandler can be configured with their own Filters and Appenders.

The Filters registered on an ActionHandler are used to determine if the ActionHandler should take effect in the particular context. If this filter chain is passed, the registered Appenders of the ActionHandlers are called in turn with the trace message and context.

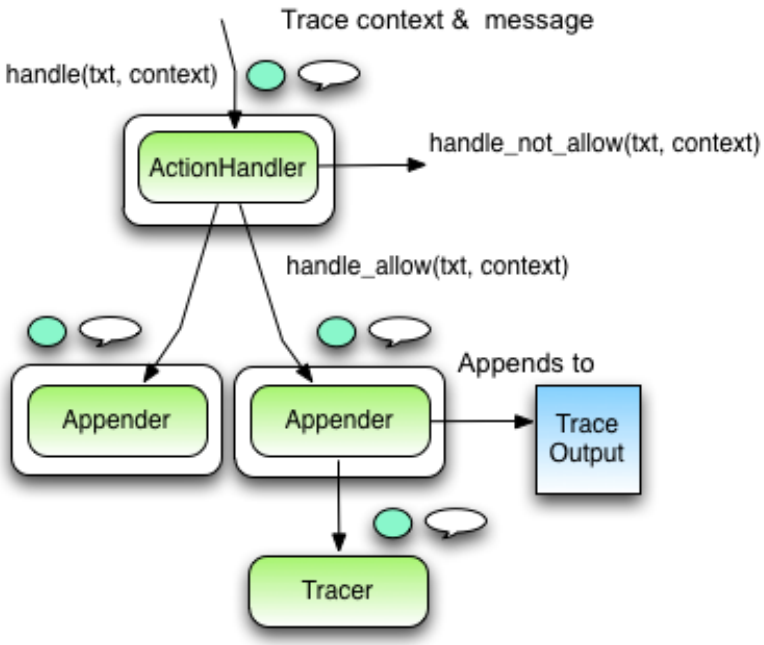

Figure 7 Actionhandler execution flow

The figure illustrates how the trace message and context is passed along at each step of the way and used in filtering and to generate the final trace result. The ActionHandler handle method iterates each appender it has registered, calls the appender filter chain and then delegates to handle_allow or handle_not_allow depending on whether the

Appenders

Appenders can be registered on an `ActionHandler` instance. Each `Appender` has a default Tracer, which is used to apply template logic on the context to generate the final text output to append to the target (consumer) of the appender. The target can be a file, a stream, a logger or some other object that makes sense. An appender can be created explicitly or constructed implicitly using a Hash of options.

```
basic_html_log_appender = {:type => :html, :to_file => 'dryml_trace.html'}
```

The :type can point to one of the default appender types:

```
:string, :xml, :html
```

The available appenders are illustrated in the following figure.

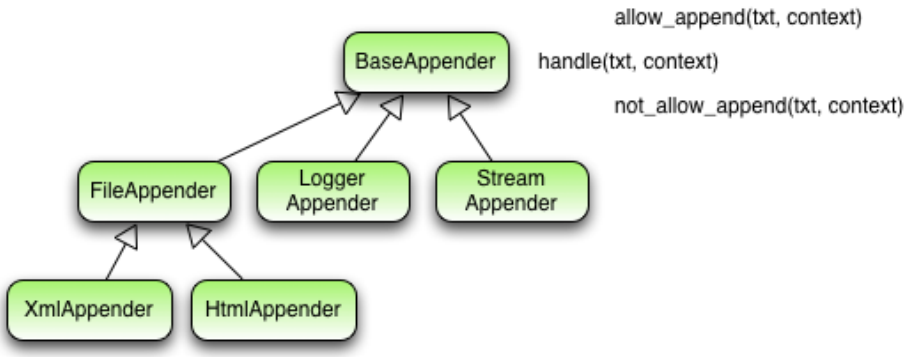

Figure 8 Appenders

The appender options can be more complex depending on the type of appender to create. Here we set up a logger `Appender`, which should not overwrite a previous target (fx a log file) unless it is more than 2 minutes old and should use the deault string tracer to output to the log.

The available tracers are shown in the following figure.

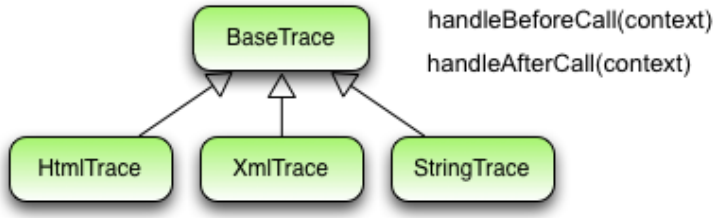

Figure 9 Tracers

Feel free to add and customize such settings on the appenders, perhaps by creating extension classes or through meta-programming.

```
appender_options = {
  :options => {:overwrite => false, :time_limit => 2.minutes},
  :type => :logger,
  :tracer => :string
}
```

To register the appenders you can do something like this:

```
Tracing::TraceExt.configure(
  :appenders => basic_html_log_appender,
  :filters => method_filters
)
```

This will in effect register a single `ActionHandler` with `TraceExt` and will construct an appender and some filters and then register these on the `ActionHandler` instance.

Using a vars filter for DRYML tracing

A `vars_filter` (instance variable filter) can be used to filter based on the current value of instance variables. This is very useful for DRYML tracing in particular. When the DRYML engine parses DRYML templates (.dryml files) it holds on to a `@template_path` instance variable for each template it processes. This variable is a reference to where the template was located. We can use this information to get an idea of what kind of template is currently being processed and accordingly decide how to trace it.

Using a `vars_filter` we can send the traces to different appenders depending on the value of this instance variable. Fx, if it contains '/taglib/', we are likely generating a template for one of the tag libraries that come with DRYML. If it contains 'application' it is the application template, if it contains '/views/' it is likely one of our own templates and so on.
This way we could send the trace to different log files or whatever we like.

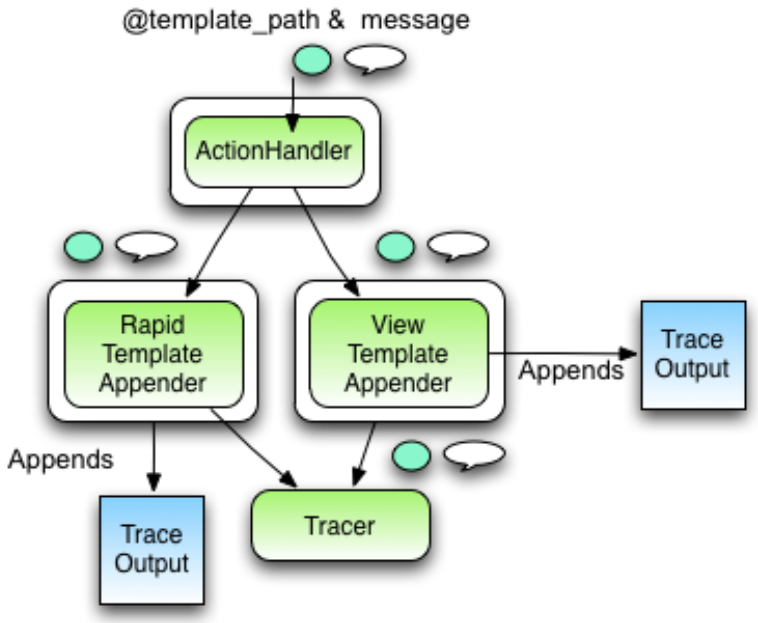

Figure 10 Dryml template tracing

Configuring appenders and filters for DRYML tracing

To trace DRYML execution we would like to define filters to include core (interesting) DRYML methods and exclude the various utility-like methods used by the core methods.
On github at http://github.com/kristianmandrup/trace_hobo I have created some utility functions to assist with this. The file dryml_trace_rules.rb contains a module `DrymlTraceRules` with a single method `dryml_methods` which returns a hash with an entry for each of the core DRYML classes. Each class key contains a new hash with symbols for various groupings of the methods for that class, fx :core, :element, :template.
We could use it like this:

```
dryml_methods = DrymlTraceRules.dryml_methods
```

```
template_methods = dryml_methods['Template']

Template_method_filter = {
  :name => 'Template method filter',
  :method_rules => [{
    # id of method rule set
    :name => 'my_methods',
    :include => template_methods[:core],
    :default => :yield
  }]
}

template_env_methods = dryml_methods['TemplateEnvironment']

TemplateEnvironment_method_filter = {
  :name => 'TemplateEnvironment method filter',
  :method_rules => [{
    # id of method rule set
    :name => 'my_methods',
    :include => template_env_methods[:core],
    :default => :yield
  }]
}
```

Here we define method filters for the DRYML classes Template and TemplateEnvironment. We want to include the :core methods for tracing and yield any other method to the next filter to decide. Then we can set the `:final_yield_action => :exclude` to ensure any method not specifically included in these filters are not traced.
We do something similar for the DRYMLBuilder class.

```
dryml_builder_methods = dryml_methods['DRYMLBuilder']

dryml_builder_method_filter = {
  :name => 'DRYMLBuilder method filter',
  :method_rules => [{
    # id of method rule set
    :name => 'core builder methods',
    :include => dryml_builder_methods[:core] + dryml_builder_methods[:template],
    :exclude => ['add_build_instruction', 'template', 'template_path'],
    :default => :yield
  }]
}
```

Here we explicitly exclude the methods `add_build_instruction`, `template` and `template_path`.

Then we set up a HTML appender, to append all traces to a single .html file. Setting the default_path here, means that all HTMLAppenders will prefix the file destination with this path. The html file will be output to `log/html/dryml_trace.html`.

```
Tracing::HtmlAppender.default_path = 'log/html/'
basic_html_log_appender = {:type => :html, :to_file => 'dryml_trace.html'}

dryml_builder_method_filters = [DRYMLBuilder_method_filter]

template_method_filters = [Template_method_filter, Template_method_filter]
```

```
method_filters = dryml_builder_method_filters + template_method_filters

Tracing::TraceExt.configure(
  :appenders => basic_html_log_appender,
  :filters => method_filters,
  :final_yield_action => :exclude
)
```

An example output is shown in the following figure.

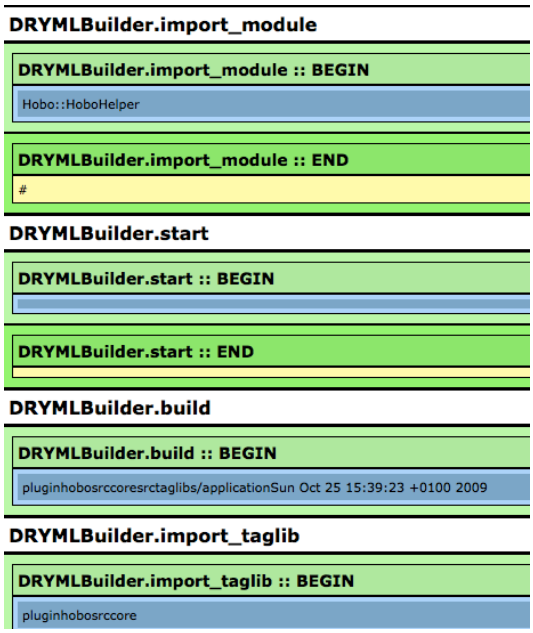

Figure 11 HTML appender - DRYML tracing

You can expand the nested methods by clicking on the white blocks. In a future version there are plans to allow some way to specify declaratively (or programmatically) which methods should be expanded in the HTML result.

Configuring channel rules for DRYML templates

To create a specific "trace channel" for @template_path, we could do something like the following. First set up some filters on @template_path to activate a different appender depending on which filter it matches. Then configure each appender, fx to output to a different log file for each appender.

```
template_path_filter1 = {
  :name => 'template_path1',
  :vars => [{
    :name => 'template_path',
    :type => :string,
    :var_rules => {
      :include => [/.*\/taglib\/.*/],
      :exclude => [/.*\/rapid_.*/]
    }
  }]
}
```

This filter allows passage when the context includes an instance variable @template_path (:name of entry in :vars array), and that variable has a value which matches the regexp (any string containing path = /taglib/). It will exclude if the value matches “rapid_”, the rapid taglibs. So in effect it will allow passage when templates within taglib are processed that are not part of the rapid library, in effect allowing passage for core taglibs only. We could be even more specific:

```
:include => [/.*\/taglib\/.*/, [/.*\/core_.*/]]
```

DRYMLBuilder has the responsibility to build the .erb templates from the .dryml templates. These .erb templates are then stored in an internal cache.

If we would like to see how this is done we have to intercept the build method of DRYMLBuilder.

```
def build(local_names, auto_taglibs, src_mtime)

  auto_taglibs.each { |t| import_taglib(t) }

  @build_instructions._?.each do |instruction|
    name = instruction[:name]
    case instruction[:type]
     …
    end
  end

  @last_build_mtime = src_mtime
end
```

The build method unfortunately returns the build time and not the actual output of the build process. To get the output in order to trace it, we would have to refactor the build method code directly or alias the method.

If we saved the output in an instance variable, we could then have access to it in our tracing and ultimately output it in a tracer and then in the appender output target.

The refactored build version would look something like this:

```
…
  result = case instruction[:type]
     …
    end
  end

  @last_build_mtime = src_mtime
  result
end
```

In the Appender, we could then use the @template_path to output the .erb result to a file in the same folder as the original .dryml source. If the @template_path was ‘/my/path/myfile.dryml’ we could output the .erb generate by build to ‘/my/path/myfile.dryml.erb’

To do this we would have to extend the FileAppender (part of adv-trace-util) and set the to_file accessor to @template_path + ".erb", overriding the to_file set when the HTMLAppender was initialized.

```
Class TemplateHtmlAppender < HtmlAppender
  def allow_append(txt, context)
    obj = context[:self]
    path_value = obj.instance_variable_get("template_path")
    to_file = path_value + ".erb"
    super(txt, context)
  end
end
```

Then simply register instances of this appender using TraceExt.configure as described previously. Multiple appenders can be registered simply by setting :appenders to an array containing a hash for each appender to register.

Examples of .erb outputs using this technique can be seen at:

http://github.com/kristianmandrup/hobo_dryml_uncovered

Here is an example for:
/app/views/front/index.dryml.erb

```
<% concat(page({:title => "Home"}, {
  :body => proc { [{:class => "front-page"}, {}] },
  :content => proc { [{}, { :default => proc { |_content__default_content|
new_context { %>
    <% concat(comma({}, { :default => proc { |_comma__default_content|
new_context { %>
      <% concat(x({}, { :default => proc { |_x__default_content| new_context {
%>hello<% } }, })) %>
      <% concat(x({}, { :default => proc { |_x__default_content| new_context {
%>hello you<% } }, })) %>
    <% } }, })) %>
  <% } }, }] },
})) %>
```

You can see similar .erb outputs for the rapid and core taglibs in:
/app/views/taglibs/rapid/
/app/views/taglibs/auto/rapid

Notice the multiple nested `concat` statements. DRYML basically works by outputting .erb which does a number of function call each resulting in something that has a .to_s and then concatenating these into a final string, which could then take part in a higher level concat and so on. To use DRYML for other scenarios we would have to short circuit this "hardcoded" DRYML approach and replace concat with a custom function, which does something else.

For the ExtJS case, we would like to join all non-empty elements with a "," to create nice javascript collections (valid JSON in effect). Thus we could find places in DRYML that outputs a string with "concat(…" and replace with something like "concat_comma(…".

I have some other projects in progress (or planned) that attempt to tweak DRYML for use with ExtJS, and similar javascript widget frameworks in general, PDF and other uses. If you get familiar with the inner workings of DRYML using the tracer as described here, I'm sure you can help in this effort or provide your own suggestions in a short while.

Have fun!