Leveraging Mason to Build a Flexible, Localized Framework

Jason Gessner
Performics/DoubleClick
YAPC::NA 2006 – June 27, 2006

What are I18N and L10N?

- Internationalization (I18N) is the process of changing an application to be usable in different parts of the world.
- Localization (L10N) is adding the features, translations and formatting to adapt to the norms of the locations where your application is used.

Why Internationalize?

- That’s easy. I am being paid to.
- Customers in Germany don’t want to read English (most of the time).
- It is the right thing to do.
- It forces you to view your copy as code, and treat it with the same care as you would a class or a database call.

What Parts of an app change?

- All of the user facing copy should be translated.
- Numbers should use the appropriate formatting.
- Currency symbols should be accurate.
- Dates should appear in the proper formats.

What does Mason offer?

- Mason offers an application platform that lets the developer hook in at many different levels:
  - At any point in the Apache process lifecycle (mod_perl)
  - At any point in the Mason process lifecycle (request, resolver, compiler, etc).
  - At the template level.

What does Perl Offer?

- Core perl has the Locale::Maketext framework.
- Treats local text lookup as hash lookups.
- Offers variable substitution.
- Offers formatting to do number agreement (0 results vs. 1 result, vs. 9,000,000 results).
- Offers various backends.

What does GNU gettext offer?

- We chose the gettext backend (Locale::Maketext::gettext) for 4 simple reasons:
  - gettext is a standard.
  - gettext is one further step removed from our code.
  - gettext can work with any language, not just perl.
  - Simple:
    - msgid “yapcNa2006”
    - msgstr “Yet Another Perl Conference North America 2006”

General Usage For Text

```
my $lh = LanguageHandle->get_handle(
  $conf->{default_locale}
);
print $lh->maketext(“some phrase”), “\n”;
print $lh->maketext(
  “phrase with args”, 0, “BLEARGH!!!”), “\n”;
```

Snippets from our handler.pl

```
my $ah = new HTML::Mason::ApacheHandler (
  resolver_class    => ‘OurMason::Resolver',
  dhandler_name     => '',
  decline_dirs      => 1,
  preprocess        => \&OurMason::preprocess,
  comp_root => [
    ['common' => '/home/jgessner/common/web/interface'],
  ],
  data_dir          => '/home/jgessner/web/mason',
  error_mode        => $cfg->{mason_error_mode}
);
```

Handler.pl continued

```
# create our language handle
    $HTML::Mason::Commands::lh =
      Language->get_handle($cfg->{locale});

 # set the locale for the language handle
    if ( $HTML::Mason::Commands::session &&
         $HTML::Mason::Commands::session->locale()
  &&
         $HTML::Mason::Commands::session->locale()
  ne $cfg->{locale} ) {

         $HTML::Mason::Commands::lh =
          Language->get_handle(
            $HTML::Mason::Commands::session-
  >locale()
           );
    }
```

Using This in Mason

- Normal Print Block
  - **<% $lh->maketext(“boo”) %>**
- Actually, we use
  - **<% $lh->maketext(“boo”) | n,localized %>**
- That is no fun. Too much typing!
- Our “extension”
  - **<# “boo” #>**
  - **<# “results”, 12, “awesome” #>**

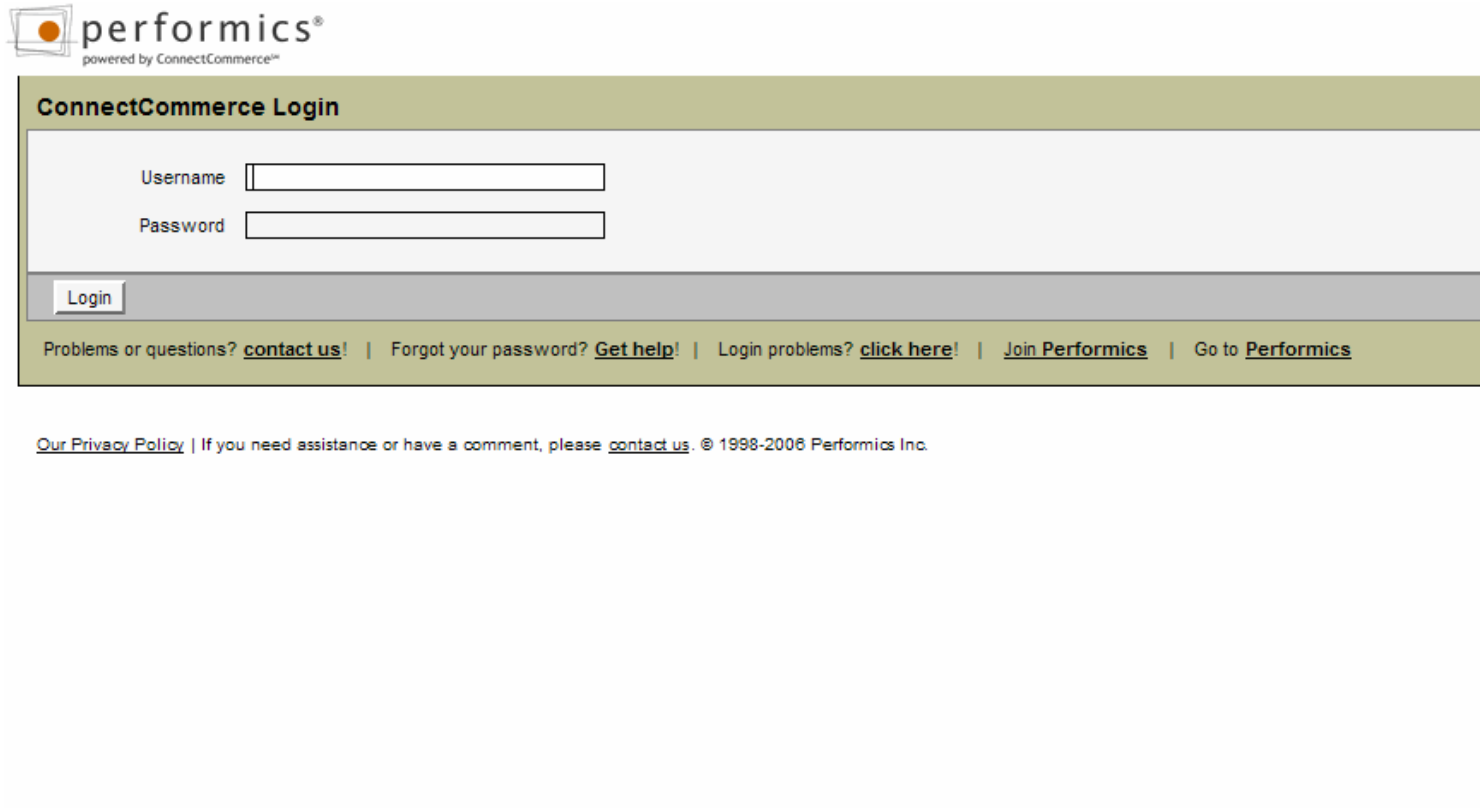
Prepocess()

- Receives a reference to the source of a component before it is handed off to the compiler.
- We cheated and just used a regex. ☺

Our Awesome Sub & Regex

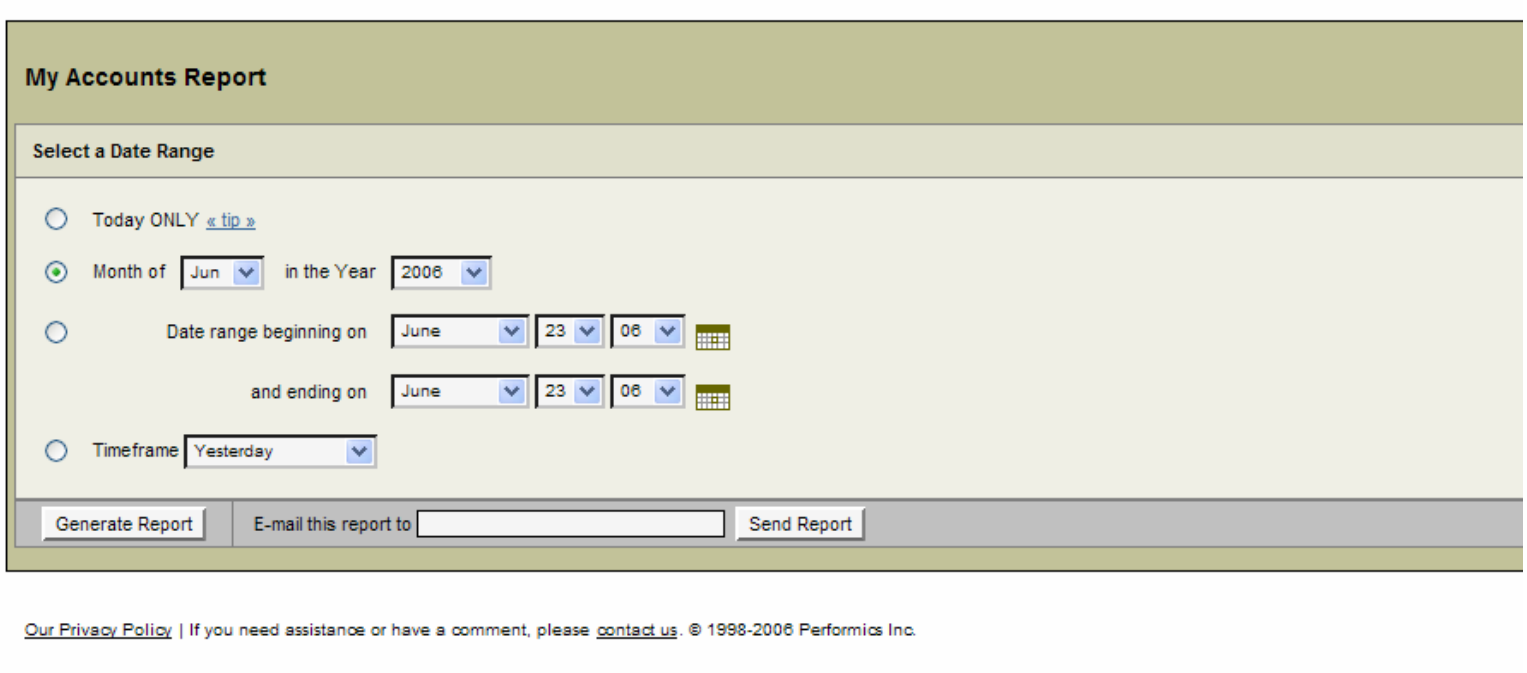
```
sub preprocess {
  $comp_src_ref = shift;

  $$comp_src_ref =~
    s/<\#                 # open tag
        \s*?              # optional whitespace
        (["'].*?["'])     # our id
        (.*?)             # everything else.
      \#>                 # our closing tag
    /<% \$lh->maketext($1$2) | n, localized %>/gix;

  return $comp_src_ref;
}
```

English Login Screen

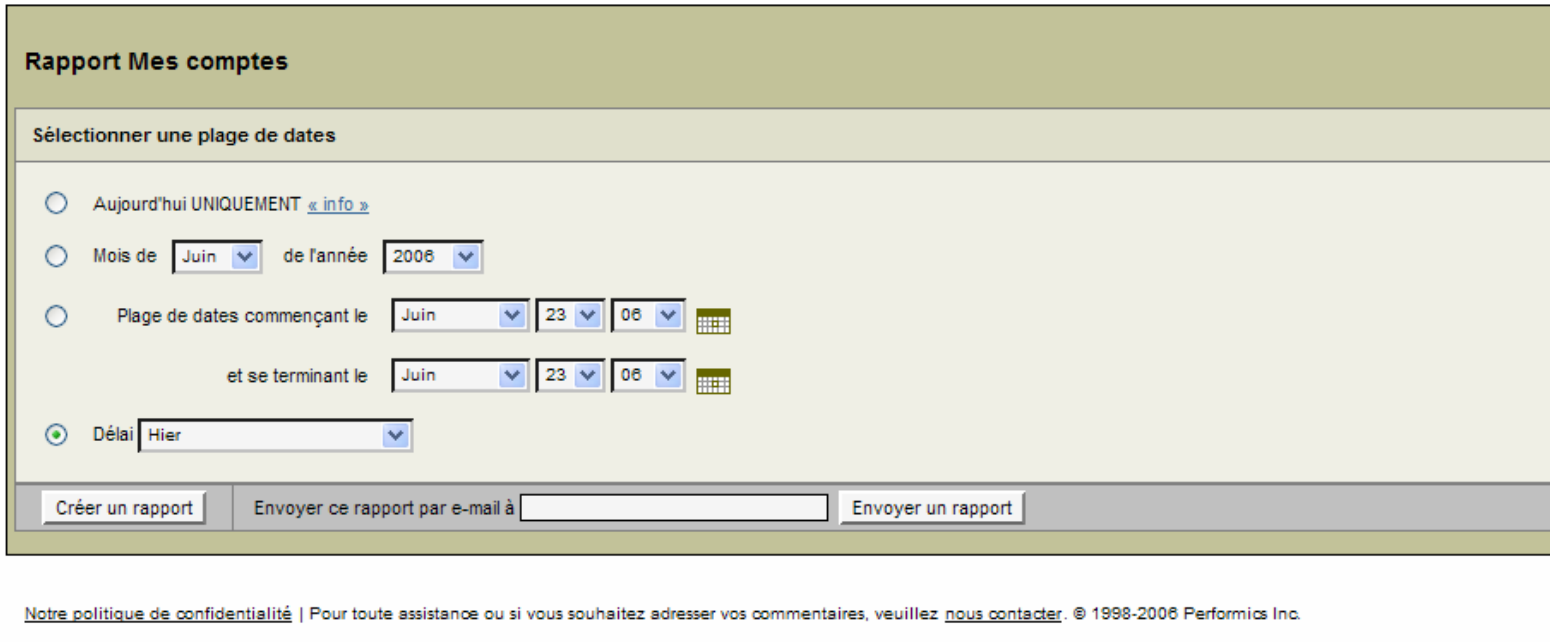


French Login Screen

Another Page in English



Another Page in French



Number formatting

- We added an initialized Number::Format object to our Language objects:
  - <% $lh->numf($some_dollar_amount) %>
  - Or
  - <% $curr_char . $lh->numf($value,2,2) %>
- This is fairly nice, but we need to wrap more succinct helper methods around it.
- numf_decimal, numf_money, numf_percent, etc.

English Number Formatting

504	$33.14	$0.07	6042	8.34%	1.20	39	7.74%	$3.53	$4,183.59
170	$38.86	$0.23	3909	4.35%	2.52	7	4.12%	$12.10	$1,834.00
63	$0.82	$0.01	258	24.42%	1.20	6	9.52%	$7.52	$1,475.80
25	$9.23	$0.37	1495	1.67%	9.66	2	8.00%	$23.08	$1,476.84
835	$1,702.10	$2.04	27155	3.07%	2.62	21	2.51%	$82.68	$1,142.06
6	$1.64	$0.27	484	1.24%	7.33	1	16.67%	$34.89	$1,329.99

European Number Formatting

399	0,25%	$83.821,01	30.230	$0,02	8.810	29,14%	
462	17,67%	$4.120,26	69.417	$0,25	2.517	3,63%	
219	15,41%	$3.249,28	50.804	$0,26	1.593	3,14%	
188	11,52%	$2.742,58	51.895	$0,25	998	1,92%	
130	10,87%	$1.439,00	18.845	$0,24	493	2,62%	
97	13,10%	$1.337,18	27.788	$0,31	463	1,67%	
123	8,17%	$1.551,04	10.981	$0,26	334	3,04%	

Tools we built around this

- Tests!
  - Do the .po files compile?
  - Every message in every language is run through maketext() to verify that they can be used.
- App that crawls our web app source files looking for phrasebook calls, and verifies that at least English has an entry for that phrase.

Other Options

- Letting users choose what number and date formatting they want to see.
- Auto-detecting the initial default from the browser.
  - Accept-lang header
- Falling back to a given language if you don’t have everything translated yet.
- Wrapped up date formatting.

Problems

- Character sets suck.
- Perl trying to be helpful about them sucks more.