
elm-fluent Documentation

Release 0.6.0

Luke Plant

Apr 09, 2020

Contents:

1	Overview	1
2	Installation	3
2.1	Stable release	3
2.2	From sources	3
2.3	Elm dependencies	4
3	Tutorial	7
3.1	Intro	7
3.2	One-time project changes	8
3.3	Extract localized text	10
3.4	Substitutions	12
3.5	Number substitutions and plural rules	13
3.6	Message attributes	14
3.7	HTML output	15
3.8	Dynamic HTML output	16
3.9	Final steps	19
4	Number handling functions	21
4.1	Fluent.number	21
4.2	Fluent.formattedNumber	21
5	Date handling functions	23
5.1	Fluent.date	23
5.2	Fluent.formattedDate	23
6	Contributing	25
6.1	Types of Contributions	25
6.2	Get Started!	26
6.3	Pull Request Guidelines	27
6.4	Deploying	27
7	Credits	29
7.1	Development Lead	29
7.2	Contributors	29
8	History	31

8.1	0.6.0 (2020-04-09)	31
8.2	0.5.0 (2019-08-15)	31
8.3	0.4.0 (2019-03-18)	31
8.4	0.3.0 (2019-03-06)	31
8.5	0.2.1 (2018-12-19)	31
8.6	0.2.0 (2018-09-10)	32
8.7	0.1.0 (2018-07-27)	32
9	Indices and tables	33

CHAPTER 1

Overview

The big picture for using elm-fluent looks as follows (with some design justifications thrown in). If you think this fits your needs, please see the [Installation](#) docs and the [Tutorial](#) for the details you'll need to implement this in your app.

With elm-fluent, all localized text (i.e. text in a specific human language) is stored in `.ftl` files in a `locales` directory, and not in Elm source files.

Justification

Some tools such as gettext encourage you to put localized text in source code files, at least for 'source language', and then extract this text into other files from which translation is done for the other ('destination') languages.

That approach has some advantages, but major disadvantages when it comes to [Fluent](#). Fluent is a language in its own right, that puts (just enough) power into the hands of translators to produce good translations, while maintaining proper separation of concerns.

For Fluent, developers need to put localized text into `.ftl` files, so the full power of that language will be used. This includes:

- choosing good message IDs, and being aware of the issues surrounding changes to IDs or changes to text without changing IDs
- adding comments that will help translators - this is often vital to give context, because a single English word could be interpreted as a noun or adjective or verb.
- using Fluent constructs for things like variant selection, rather than using Elm flow control constructs for variant selection, which would cause the destination languages to suffer.

So, if your web app has a 'notifications' component with a title 'Notifications' and some intro text, you would have a `locales/en/notifications.ftl` file that looks something like this:

```
notifications-title = Notifications

notifications-intro = Hello { $username }, you have unread notifications
```

You compile this to Elm files using `ftl2elm`. Normally the generated `.elm` files, which appear under `Ftl` by default, should not be stored in your VCS, you store only the `.ftl` files. You can then use these generated functions from your Elm source code.

Your app first needs some way to determine the user's current locale. This is usually best done by allowing them to choose from a list, and then saving this in the model somewhere. Let's assume we have `model.locale` set up already. Then our Elm source code might look like this:

```
import Ftl.Translations.Notifications as T

viewNotifications model =
    Html.div []
        [ Html.h2 (T.notificationsTitle model.locale ())
          , Html.p (T.notificationsIntro model.locale { username = model.username_
↵ })
        ]
```

The generated functions (in this case `notificationsTitle` and `notificationsIntro`) all follow the same pattern in terms of signature - they take a locale value and a strongly typed record of substitution parameters, and return a string. (For advanced use cases, the functions return `Html` and take an additional parameter that allows attributes to be added to the HTML).

Justification

Using a strongly typed record type means that we can catch the vast majority of errors at compile time. If a translator includes a parameter in their translation that is not passed by the developer, the code will not compile. We can also ensure that numbers get passed as numbers etc.

Depending on the locale parameter, the generated functions dispatch to the function for the correct language (we just have one so far).

You now need to distribute your `.ftl` files and get translations for the other languages you support. These are saved into the correct sub folder in your locales directory and committed to VCS. (Mozilla has developed the [Pontoon](#) system which provides a GUI for editing `.ftl` files, but elm-fluent doesn't have good integration with it yet).

Finally, you can now compile the `.ftl` files for all the languages, compile your Elm app and deploy.

2.1 Stable release

elm-fluent works primarily as a command line application that compiles `.ftl` files to Elm code. This application is written in Python, and if you already have a Python installation (3.4+) you can easily install it as follows.

We recommend using [pipx](#) to allow you to install Python binaries in their own virtualenvs which are isolated from your system Python. With pipx installed, do:

```
pipx install elm-fluent
```

Alternatively, you can manually create the virtualenv:

```
$ virtualenv my_virtual_env
$ . my_virtual_env/bin/activate
```

Then install the latest version from PyPI using pip:

```
$ pip install elm-fluent
```

An executable `ftl2elm` should have been added to your `PATH`, in your system `bin` or the virtualenv `bin` directory. If you don't have Python, [pip](#) or `virtualenv` installed, this [Python installation guide](#) can guide you through the process.

2.2 From sources

The sources for elm-fluent can be downloaded from the [Github repo](#).

2.3 Elm dependencies

The `.elm` files that elm-fluent produces have dependencies. At this point in time, these dependencies are a bit problematic: the Fluent spec has [built-in date and number formatting functions](#) (`DATE` and `NUMBER`), as well as support for handling of plural forms, which are covered by built-in browser Javascript modules `Intl.DateTimeFormat`, `Intl.NumberFormat` and `Intl.PluralRules` respectively.

However, these Javascript APIs are not yet wrapped by any core Elm libraries, although [the core team are looking into it](#), and implementing them in pure Elm has many difficulties. Until there is an official solution, elm-fluent depends on the `elm-intl` wrapping.

Since this uses native/kernel modules, it can't be uploaded to `packages.elm-lang.org`, but it can be installed using `elm-github-install`.

2.3.1 Elm 0.18 instructions

First install `elm-github-install` if you don't have it. Quick version:

```
$ npm install elm-github-install -g
```

Add the following dependencies to your `elm-package.json`:

```
"dependencies": {
  "thetalecrafter/elm-intl": "2.0.0 <= v < 3.0.0",
  "elm-fluent/elm-fluent": "1.0.0 <= v < 2.0.0"
}
```

and dependency sources - this will be a new section if you haven't used `elm-github-install` before:

```
"dependency-sources": {
  "thetalecrafter/elm-intl": {
    "url": "https://github.com/vanwagonet/elm-intl",
    "ref": "master"
  },
  "elm-fluent/elm-fluent": {
    "url": "https://github.com/elm-fluent/elm-fluent",
    "ref": "master"
  }
},
```

This adds the `elm-fluent` package (a very small module provided by this project), and the `elm-intl` package. You should also check [elm-intl](#) installation notes for info regarding polyfills.

Finally, run:

```
$ elm-install
```

Warning: NOTE: by using `elm-github-install` and adding these dependencies, you are opening yourself up to the problems that Javascript code brings - you are essentially trusting these packages in the same way that you currently trust Javascript from core Elm packages, rather than relying on the Elm compiler to protect you from many issues that Javascript brings.

2.3.2 Elm 0.19

Elm 0.19 has added restrictions that make it harder to use third party packages with native code, so for the time being there is no easy way to install the required dependencies with Elm 0.19.

Hopefully for Elm 0.19 we will have official `Intl` wrappers of some kind soon. It will probably not be too hard to adapt the elm-fluent compiler to depend on those libraries instead. This will likely mean some changes to user code, but possibly just types/imports.

It is hoped that this project will provide feedback/prototyping that will help to shape a useful set of `Intl` wrappers for `package.elm-lang.org`.

Tutorial contents

- *Intro*
- *One-time project changes*
- *Extract localized text*
- *Substitutions*
- *Number substitutions and plural rules*
- *Message attributes*
- *HTML output*
- *Dynamic HTML output*
- *Final steps*

3.1 Intro

This tutorial will start from the assumption that you already have an app which hard codes a single language. This is the ‘source language’, and we’ll assume it’s English for simplicity. You will be introduced to most of the key features of the Fluent language (although it is recommended that you also read [the Fluent guide](#) fully), and some key principles of internationalization.

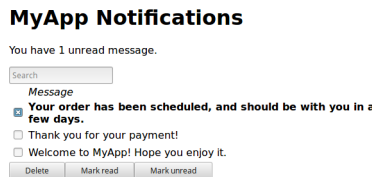
If you are already using another i18n/translation solution, hopefully it should be obvious how to adapt this tutorial for your situation.

Our example will be a ‘notifications’ page in a web app. The full sources of this demo app are available, and it may be useful to have them open as you read the tutorial:

- [Notifications app without Fluent](#)

- Notifications app with Fluent

It consists of a title, a search bar, a list of notifications and some buttons for deleting or ‘mark as read’ — here it is in all its unstyled glory:



It is assumed you have followed the [Installation](#) documentation for setting up your project with elm-fluent.

3.2 One-time project changes

As you start to internationalize your app, there are some one time changes you’ll need, usually across your whole app. For this demo, our whole app is one page, to keep it simple, but for your app you would need to put the initial plumbing in the right place so that it applies across everything.

Our first step is to keep track of the user’s choice of locale. This could come from a server side setting that is passed into your app via flags, or it could be handled entirely from Elm code. For our demo, we will default to English and have a simple select box for changing the language, and we won’t be saving the preference anywhere (but you would need to do that for a real app).

First we need an import:

```
import Intl.Locale as Locale
```

Then we need to add a field to our model:

```
type alias Model = {
  { locale : Locale.Locale
  -- (etc.)
}
```

and add a default value:

```
defaultLocale : Locale.Locale
defaultLocale =
  Locale.en

init : Flags -> ( Model, Cmd Msg )
init flags =
  ( { locale = defaultLocale
    -- (etc.)
  }
  , Cmd.none
  )
```

We’ll want a list of locales/languages we support, with some kind of caption:

```
availableLanguageChoices : List ( String, String )
availableLanguageChoices =
  [ ( "en", "English" )
```

(continues on next page)

(continued from previous page)

```
, ( "tr", "Türkçe" )
]
```

Here we've used the localized name of the language as the caption. To make it easier for people to find their own language, and get back if they accidentally change the language to Chinese etc., we won't translate these depending on the user's choice of language. So this is one of the few times that we'll hard code localized text into our Elm file (although there are other ways to do this).

For each language tag (the first value in the tuples above), we want a `Locale` object as well:

```
availableLocales : List ( Locale.Locale, String )
availableLocales =
    List.map
        (\( languageTag, caption ) ->
            ( Locale.fromLanguageTag
              languageTag
              |> Maybe.withDefault defaultLocale
            , caption
            )
        )
    availableLanguageChoices
```

Note: The `Locale.fromLanguageTag` function can fail, in which case it returns `Nothing`, hence the use of `withDefault` above. The function actually only fails if you pass invalid language tags — it does not fail if you pass a valid BCP47 language tag that the browser doesn't happen to support. We only passed valid language tags, but the compiler doesn't know that, so we still need the fallback here although it won't be used in practice.

We'll need a message to change locale:

```
type Msg
    = ChangeLocale Locale.Locale
    -- (etc.)
```

And we need to handle that message:

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    let
        newModel =
            case msg of
                ChangeLocale locale ->
                    { model
                      | locale = locale
                    }
    in
        -- etc.
```

We need some UI to select a different locale (you can skip this step for now if you just want to try the system out using the default English locale). The following is a simple `select` widget that will trigger `ChangeLocale` when it is changed:

```
viewLocaleSwitcher : Model -> List (H.Html Msg)
viewLocaleSwitcher model =
    [ H.text "Change language: "
    , H.select []
```

(continues on next page)

(continued from previous page)

```

    (List.map
      (\( locale, caption ) ->
        H.option [ E.onClick (ChangeLocale locale) ]
          [ H.text caption ]
      )
      availableLocales
    )
  ]

```

And we need to include this somewhere in the page by adding it to the main `view` function (left as an exercise for you!).

Also left as an exercise for you - if you change a page locale/language, you should always remember to [set the lang attribute](#) on the root `<html>` element, and you may want to update the document `<title>` as well. You'll probably need to do both of these using a port (because typically these elements are outside of the root element that Elm controls).

3.3 Extract localized text

Now that the basic setup of the project has been done, we can begin extracting localized text (i.e. all the English text that needs to be translated) into `.ftl` files.

You need to create a `locales` directory to store these files. This can go anywhere, as the `ftl2elm` program allows you to specify the location using `--locales-dir`. We will use the default, which is to place it in the same directory as your top level Elm source files.

Within it, create an `en` directory to store the English `.ftl` files. Within that you can create any number of sub folders, as your project structure dictates. We will create a file `notifications.ftl` directly within `en` that will contain the messages for our Notifications module.

Note: You have flexibility in how you structure your FTL files and the folders that contain them. A good pattern is to structure these files and folders in a way that mirrors the rest of your project structure.

Now we will create our first FTL message. Our view function currently looks like:

```

view : Model -> H.Html Msg
view model =
  H.div []
    [ H.h1 [] [ H.text "MyApp Notifications" ]
      -- etc
    ]

```

"MyApp Notifications" needs to be localized, so we'll pull it out into `notifications.ftl`. This is one of the simplest possible messages — an entirely static string with no substitutions, which will look like this in our `notifications.ftl` file:

```

# This title appears at the top of the notifications page
notifications-title = MyApp Notifications

```

Notice:

1. The comment which begins with `#` — comments can be really important in creating `.ftl` files that are understandable and well organized. The Fluent guide has [more information](#).

For brevity this tutorial will omit comments for the rest of the messages we add.

2. The `notification-` prefix.

This is used as an adhoc prefix to indicate the component/page this message belongs to. This is not strictly necessary with elm-fluent, but has several advantages:

- If you want to combine FTL files at some point, it will help to avoid name clashes. You may also want to use the same FTL files with other technologies (e.g. server side rendering), and these technologies tend to use bundles that combine multiple `.ftl` files.
- It will result in an Elm function that has a longer name, and so helps reduce the possibility of name clashes with other functions. To avoid problems, elm-fluent refuses to generate functions that would clash with [default imports](#).
- It is the convention used by Mozilla for their message IDs, and so is a standard.

3. The naming style:

The naming style used here is words-separated-with-hyphens. This is the normal convention in Fluent. These are not valid identifiers in Elm, and so are converted to camelCase by `ftl2elm`. You could also use camelCase directly as your message ID e.g. `notificationsTitle`

Now run the `ftl2elm`:

```
ftl2elm --verbose
```

(See also `ftl2elm --help` for other options, especially if you are using a different directory layout.)

You'll hopefully see output like the following:

```
Writing files:
Writing Ftl/EN/Notifications.elm
Writing Ftl/Translations/Notifications.elm
Success!
```

(This tutorial won't mention running `ftl2elm` again - every time you change `.ftl` files you'll need to run `ftl2elm` again - or, use the `--watch` option and leave it running).

Let's look at `Ftl/EN/Notifications.elm`. This is a compilation of our `notifications.ftl`, and it looks something like this (slightly abridged):

```
module Ftl.EN.Notifications exposing (notificationsTitle)

import Intl.Locale as Locale

notificationsTitle : Locale.Locale -> a -> String
notificationsTitle locale_ args_ =
    "MyApp Notifications"
```

The body is as simple as you would expect for this simple case — it just returns the string!

In addition, there is a similar function in `Ftl/Translations.elm`. The sole purpose of this function is to dispatch to the correct language, depending on the locale you pass in. It looks something like this (a bit redundant at the moment, as we only have one language so far):

```
module Ftl.Translations.Notifications exposing (notificationsTitle)

import Ftl.EN.Notifications as EN
import Intl.Locale as Locale
```

(continues on next page)

(continued from previous page)

```
notificationsTitle : Locale.Locale -> a -> String
notificationsTitle locale_ args_ =
    case String.toLower (Locale.toLanguageTag locale_) of
        "en" ->
            EN.notificationsTitle locale_ args_
        _ ->
            EN.notificationsTitle locale_ args_
```

This is the function we'll be using from our `Notifications.elm` module. First some imports are needed:

```
import Fluent
import Ftl.Translations.Notifications as T
```

Then we modify the view function to use the generated function:

```
view : Model -> H.Html Msg
view model =
    H.div []
        [ H.h1 [] [ H.text (T.notificationsTitle model.locale ()) ]
          -- etc.
        ]
```

Note we always pass:

- the locale value
- a value containing substitution parameters (if any).

In this case we have no substitution parameters, so we can pass any value, and we chose the empty value `()` for simplicity.

First message done! You can check your project compiles and works.

Note: Generated Ftl files

A bunch of files have been generated in a `Ftl` folder. You shouldn't edit these directly, or add them to VCS. Be sure to add this folder to `.gitignore` (etc.)

3.4 Substitutions

The next bit of our view consists of some Elm code that generates text like the following:

Hello, Mary. You have 2 unread messages.

We can split this into two messages.

Note: Splitting of localized text has to be done very carefully. One of the key principles of translating apps is that **you cannot split a sentence or label into parts and translate the parts separately**. This might work for English, but will fail for other languages. Usually if you have several unrelated sentences in a block of text, you can split them into a message for each sentence, but do not split up a sentence (or other text fragment like a title) further.

Our first message is a greeting, but it has a substitution. It will look like this in our FTL file:

```
notifications-greeting = Hello, { $username }.
```

And we use it like this in our Elm code:

```
[ H.text (T.notificationsGreeting model.locale { username = model.userName })
-- (etc.)
]
```

Notice how we pass substitutions in a record type. The signature for `notificationsGreeting` looks like this:

```
notificationsGreeting : Locale.Locale -> { a | username : String } -> String
```

This means if you fail to pass this argument in your Elm code, or try to pass something of the wrong type, you'll get a compilation error, which is exactly what we want to happen.

3.5 Number substitutions and plural rules

The next sentence we need to internationalize is "You have X unread messages."

The Elm code we are starting with looks like this:

```
H.text
  (let
    unreadNotificationsCount =
      unreadNotifications model |> List.length
  in
    if unreadNotificationsCount == 1 then
      "You have 1 unread message."
    else
      ("You have " ++ toString unreadNotificationsCount ++ " unread messages.")
  )
```

There are some big issues with this code:

- It hard-codes English language pluralization rules. In English, we use the singular form for 1, and plural form for everything else (including zero). This pattern does not apply to other languages. So we need to move this logic into the FTL file, where translators are able to apply the correct logic for their language.
- It hard-codes a number formatting function that is only appropriate for some languages. If we were using floating point numbers, we can easily get completely wrong number formatting. For example, `1.002` ("one and two one-thousandths") can be written in English as the string `"1.002"`. However, in most European locales, that string means "one thousand and two", while `"1,002"` means "one and two one-thousandths".

Fluent fixes these things for us. We pull out the entire message into FTL like this, using [Fluent selector syntax](#), to produce the following which is appropriate for English:

```
notifications-unread-count = { $count ->
  [one]      You have 1 unread message.
  *[other]   You have { NUMBER($count) } unread messages.
}
```

Our view code looks like this:

```
H.text
  (T.notificationsUnreadCount model.locale
```

(continues on next page)

(continued from previous page)

```

    { count = Fluent.number (unreadNotifications model |> List.length) }
  )

```

This time, instead of passing a string as an argument, we pass a number. We actually pass it as a `Fluent.FluentNumber` number value. The purpose of `FluentNumber` is that it allows us to optionally pass formatting options along with the number. In this case we didn't add formatting options, so we just used `Fluent.number`. We could have used `Fluent.formattedNumber` to specify formatting options (which includes things like currencies).

It is also possible for the translators to add parameters using the `NUMBER` builtin.

Similarly, not covered in this tutorial, there is the `Fluent.DATETIME` builtin, and elm-fluent functions `Fluent.date` and `Fluent.formattedDate` which are used to pass dates into messages.

3.6 Message attributes

The next piece of HTML we need to internationalize is the search box. The Elm code looks like this:

```

viewSearchBar : Model -> H.Html Msg
viewSearchBar model =
  H.div []
    [ H.input
      [ A.type_ "search"
      , A.name "q"
      , A.placeholder "Search"
      , A.attribute "aria-label" "Search through notifications"
      , A.value model.searchBoxText
      , E.onInput SearchBoxInput
      ]
      []
    ]

```

We can continue with this in exactly the same way as before — creating a message for each piece of English text that needs to be translated. One need that often crops up in web apps is that we have some closely related pieces of text, and we might want to group them more tightly. In the case above we have two pieces of text that appear as part of the search box - "Search" as the placeholder, and "Search through notifications" as the `ARIA` label, which will be used by screen readers.

There is a Fluent feature designed exactly for this case, called attributes. We can create an FTL message as follows:

```

notifications-search-box =
    .placeholder = Search
    .aria-label = Search through notifications

```

This will create two message functions for us. In this case, to produce valid Elm identifiers, the `.` is translated to an underscore, so we end up with `notificationsSearchBox_placeholder` and `notificationsSearchBox_ariaLabel`. These can be used in the same way as before, so our Elm code becomes:

```

viewSearchBar : Model -> H.Html Msg
viewSearchBar model =
  H.div []
    [ H.input
      [ A.type_ "search"
      , A.name "q"

```

(continues on next page)

(continued from previous page)

```

    , A.placeholder (T.notificationsSearchBox_placeholder model.locale ())
    , A.attribute "aria-label" (T.notificationsSearchBox_ariaLabel model.
↪ locale ())
    , A.value model.searchBoxText
    , E.onInput SearchBoxInput
  ]
  []
]

```

This example also illustrates the use of localized text in attributes — because our translation functions simply return strings, their output can be used in a wide variety of contexts.

3.7 HTML output

Although string output is the most flexible, there are times when you need HTML output.

For example, we may simply want to highlight some text in some way. Suppose in our greeting we want to highlight the username in bold:

```
Hello <b>Mary</b>, so nice to have you back!
```

Remember that a key rule of i18n is that we mustn't attempt to split this into parts, translate separately, and put back together. We need to keep these parts together as a single, translatable unit.

This is handled in elm-fluent simply by giving the message ID a special suffix -html or Html — and then putting the desired HTML markup directly in the .ftl file:

```
notifications-greeting-html = Hello <b>{ $username }</b>, so nice to have you back!
```

In this case, `ftl2elm` will generate a function which returns `List (Html msg)` instead of `String`. You can use it just as you expect, e.g.:

```

H.p []
  (T.notificationsGreetingHtml model.locale { username = model.userName } [])

```

Note, however, this takes one extra parameter, for which we passed an empty list. The purpose of this parameter is explained in the next section.

Before we go on, there are a few things to note:

- Compare the combined FTL + Elm you have above with what you would have had before if you had written it all in Elm:

```

H.p []
  [ H.text "Hello "
  , H.b [] [ H.text model.userName ]
  , H.text ", so nice to have you back!"
  ]

```

You may notice that your code, even when you add back in the FTL, is now shorter and much more readable! This is just a little bonus that comes from using a purpose-designed language like FTL, and a compiler (`ftl2elm`) that will generate all that `Html` code for you...

- elm-fluent can handle a lot more than this simple case — all the other Fluent features can be combined with HTML. This includes also being able to use attributes in the HTML, and substitutions within attributes etc.

There are a few limitations. You cannot have substitutions or other similar constructs in element names or attribute names, for example:

```
bad-message-1-html = Some <{ $arg }>text</{ $arg }>

bad-message-2-html = Some <b { $arg }="value">text</b>
```

There are also limitations with respect to Fluent’s mechanism for [message references](#). You can reference both HTML and plain text messages from HTML messages, and it will do the right thing, but you cannot reference an HTML message from a plain text one. (This might be obvious from the type system - we can embed `String` into `List (Html msg)` via `Html.text`, but we can’t embed `List (Html msg)` into `String`.)

- You should consider carefully how much HTML you should put into your FTL files, and keep it as simple as possible. Remember it will be read and translated by translators who may not be experts in HTML. Use the mechanisms described in the next section to keep as many HTML attributes in your Elm code as possible.

3.8 Dynamic HTML output

The above mechanism works fine for simple cases, but in Elm we often have more complex HTML, and in particular we have `Html` constructs that cannot be simply embedded into FTL files as a string.

For example, in our demo app, when you press the ‘Delete’ button, you get a confirmation panel that has hyperlinks like this:

These 2 messages will be permanently deleted - **cancel** or **confirm**

...except that ‘cancel’ and ‘confirm’ are hyperlinks with behavior attached. The original Elm code looks like this:

```
[ H.text "These "
, H.text <| toString <| Set.size model.selectedNotifications
, H.text " messages will be permanently deleted - "
, H.a
  [ A.href "#"
  , onClickSimply DeleteCancel
  ]
  [ H.text "cancel" ]
, H.text " or "
, H.a
  [ A.href "#"
  , onClickSimply DeleteConfirm
  ]
  [ H.text "confirm" ]
]
```

We need the `a` elements to be embedded in translatable text (as discussed above — we don’t want to split this text up). But we also need a way to attach those event handlers to the anchors, and we need to ensure we don’t mix up those event handlers.

(Also, we got lazy above and didn’t handle the case of a single message very well — we’ll fix that as we go).

This is probably one of the most complex examples you’ll come across, so the next section will be a little bit heavy. But if you can master this you have the tools to handle any similar situation.

Let’s start with a first attempt at our FTL message:

```
notifications-delete-confirm-panel-html =
  { $count ->
```

(continues on next page)

(continued from previous page)

```
[one]      This message
*[other]   These { $count } messages
} will be permanently deleted - <a>cancel</a> or <a>confirm</a>
```

and our Elm code to call it:

```
T.notificationsDeleteConfirmPanelHtml model.locale { count = Fluent.number <| Set.
  ↪size model.selectedNotifications } []
```

This will produce the right text, but the ‘cancel’ and ‘confirm’ parts don’t do anything, or even look like links yet.

If you look at the type signature of the generated `notificationsDeleteConfirmPanelHtml`, you’ll find this:

```
notificationsDeleteConfirmPanelHtml : Locale.Locale -> { a | count : Fluent.
  ↪FluentNumber number } -> List (String, List (Html.Attribute msg)) -> List (Html.
  ↪Html msg)
```

That final parameter looks a bit intimidating, but it is simply a way of passing the attributes that we would have attached directly before. To break it down, it is a list of tuples, where each tuple consists of:

1. A string element, whose value should be a CSS selector (a very limited subset of CSS selectors, to be precise)
2. A list of attributes that are attached to the elements that match those CSS selectors.

So, for a start, for our `a` elements to be actually rendered as hyperlinks, they need an `href` value. Both anchors match the CSS selector `a`, so let’s replace that final empty list with:

```
[ ("a", [ A.href "#" ] ) ]
```

Rebuild, and you’ll find the two `a` elements at least appear as links.

We now need to attach the event handlers. In most cases, in a single piece of translatable text we would only have a single link or ‘active’ element that needs attributes, so we would just add more attributes to the list above — like this:

```
[ ( "a", [ A.href "#"
           , onClickSimply DeleteConfirm
         ] )
]
```

But in this case we need to attach different handlers to the different elements, and they are both anchors. This means we will need to change the FTL message so that the two `a` elements can be distinguished somehow.

To enable this, elm-fluent supports a subset of CSS selectors. The full list is:

Type	Example
element	<code>a</code>
class	<code>.foo</code>
id	<code>#mything</code>
attribute present	<code>[data-bar]</code>
attribute value	<code>[data-bar="value"]</code>
element and class	<code>a.foo</code>
element and id	<code>a#mything</code>
element and attribute present	<code>a[data-bar]</code>
element and attribute value	<code>a[data-bar="value"]</code>

For our purposes, we’re going to add two adhoc `data-` attributes to our message — plus an explanatory note for the translators. We’ll actually start them both `data-ftl-` — these attributes will appear in the final rendered HTML,

and we probably want to avoid clashing with other data- attributes we might be using for other purposes. So our FTL looks like this:

```
# Confirmation message when deleting notifications.
# It includes two hyperlinks - 'cancel' to cancel the deletion,
# and 'confirm' to continue.
# You must wrap the 'cancel' text in:
#
#   <a data-ftl-cancel>...</a>
#
# and wrap the 'confirm' text in:
#
#   <a data-ftl-confirm>...</a>
#
notifications-delete-confirm-panel-html =
  { $count ->
    [one]   This message
    *[one]  These { $count } messages
  } will be permanently deleted - <a data-ftl-cancel>cancel</a> or <a data-ftl-
  ↪confirm>confirm</a>
```

Our Elm code becomes:

```
T.notificationsDeleteConfirmPanelHtml model.locale
  { count = Fluent.number <| Set.size model.selectedNotifications }
  [ ( "a", [ A.href "#" ] )
  , ( "[data-ftl-cancel]", [ onClickSimply DeleteCancel ] )
  , ( "[data-ftl-confirm]", [ onClickSimply DeleteConfirm ] )
  ]
```

That's it! The generated code in `notificationsDeleteConfirmPanelHtml` takes care of adding the passed in attributes to the right nodes, so you get the same functionality as before.

Let's just take stock of what we have to do for these cases:

1. Pull out the text and the essential HTML structure into FTL.
2. For all the attributes we were attaching before, create a CSS selector to match the HTML node they should be attached to, and put them together in a tuple, passing the list of tuples as the final parameter to the message function.
3. If necessary, add some attributes to the HTML in the FTL file to disambiguate the HTML nodes, and adjust the CSS selectors accordingly.

The result is that we have a clean separation of concerns. Our localized text is in one place, and it has the flexibility to structure the text and any embedded HTML in any way necessary for the language. The text is actually much more succinct and readable than before.

Our behavior is still all defined in Elm, though, where we want it. Admittedly it has become more dense, but we've only had to make simple, local changes.

Note: Performance

This use of CSS selectors might make some people worry about performance. Is there some heavy HTML parsing going on behind the scenes?

In reality, all the HTML parsing happens at compile time i.e. when `ftl2elm` runs. We know at compile time the exact list of supported CSS selectors that each node can match. For example, if we have `<a class="foo">` in the FTL, the complete list is `a`, `.foo` and `a.foo`. The generated code for that node does a simple `List.filter` on

the passed in attributes to find these 3 selectors. This is a little bit more expensive than the original code, but there isn't any heavy lifting going on at runtime.

3.9 Final steps

Having now completed our English `notifications.ftl` file, the next step is to distribute this to translators, so that they can produce similar ones, but in the target languages. When returned, these will be added to your project in new directories under `locales` using the same structure as before and committed to source control (like the original).

Finally, we can use `ftl2elm` to compile everything as part of our deployment process. `ftl2elm` can pick up quite a few errors in FTL files itself, and will complain loudly by default and fail. Other errors will be picked up by the Elm compiler (e.g. missing arguments to messages), to ensure that you can't deploy with broken translations.

Number handling functions

4.1 `Fluent.number`

TODO!

4.2 `Fluent.formattedNumber`

TODO!

Example of how it works is in the tests:

- https://github.com/elm-fluent/elm-fluent/blob/641a3bd889a95f165ff70348a321698455ce907d/tests/test_project/Main.elm#L154
- https://github.com/elm-fluent/elm-fluent/blob/641a3bd889a95f165ff70348a321698455ce907d/tests/test_project/Main.elm#L88

Date handling functions

5.1 `Fluent.date`

TODO!

5.2 `Fluent.formattedDate`

TODO!

Example of how it works is in the tests:

- https://github.com/elm-fluent/elm-fluent/blob/641a3bd889a95f165ff70348a321698455ce907d/tests/test_project/Main.elm#L160
- https://github.com/elm-fluent/elm-fluent/blob/641a3bd889a95f165ff70348a321698455ce907d/tests/test_project/Main.elm#L103

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at <https://github.com/elm-fluent/elm-fluent/issues>.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

6.1.4 Write Documentation

elm-fluent could always use more documentation, whether as part of the official elm-fluent docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/elm-fluent/elm-fluent/issues>.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *elm_fluent* for local development.

1. Fork the *elm-fluent* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/elm-fluent.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv elm_fluent
$ cd elm-fluent/
$ python setup.py develop
$ pip install -r requirements_dev.txt
```

We also need Elm and some other tools installed. If you don't already have it installed and available on your path, you can use nodeenv to first create a nodejs virtualenv:

```
$ nodeenv --python-virtualenv
```

Reload the virtualenv for changes to take affect:

```
$ deactivate
$ workon elm_fluent
```

And then install things into it:

```
$ npm install -g elm@0.18 elm-test@0.18 elm-github-install
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 elm_fluent tests
$ py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

A fast test run:

```
$ py.test -k-slow
```

See py.test docs for more collection options

Show the browser for the end to end tests:

```
$ TEST_SHOW_BROWSER=1 py.test
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/elm-fluent/elm-fluent/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 7

Credits

7.1 Development Lead

- Luke Plant <L.Plant.98@cantab.net>

7.2 Contributors

None yet. Why not be the first?

8.1 0.6.0 (2020-04-09)

- Big rewrite of the type inference/checking mechanism, with more accurate error messages now.

8.2 0.5.0 (2019-08-15)

- Updated to Syntax 1.0 (fluent.syntax 0.15)
- Various bug fixes and small improvements

8.3 0.4.0 (2019-03-18)

- `ftl2elm --include` option
- Dropped Python 2.7 support

8.4 0.3.0 (2019-03-06)

- Syntax 0.8 support, including parameterized terms.
- Better compile-time resolution of some expressions.

8.5 0.2.1 (2018-12-19)

- Fixed python-fluent dependency to an older version (< 0.9), because it doesn't work with newer versions. Thanks @stasm for the report.

8.6 0.2.0 (2018-09-10)

- Better handling for a large variety of error conditions
- Proper fallback mechanism implemented
- Added `--watch` option.
- Eliminate unused imports from generated code
- Various bug fixes:
 - Avoid outputting `.elm` files with no exports
 - Bugs with HTML attributes and non-string message args
 - Crasher with multi-line messages
 - Crasher when a message is missing from default locale

8.7 0.1.0 (2018-07-27)

- First release on PyPI.

CHAPTER 9

Indices and tables

- `genindex`
- `modindex`
- `search`