Globalizing Web Content

I've always been interested in internationalization (i18n) and localization (l10n) and how they relate to the web. My interest got picked again when I started wondering how much extra work would it be to keep a site or app in multiple languages and how much time and how many additional resources I'd need to get it done.

This goes beyond translating the content; tools like Google Translate make that part easier than it used to be but also the changes and modifications that we need to do to our code to acomodate for the different languages and cultures we want to deploy our application in.

Difference between l10n and i18n

Before we can jump in and talk about localizing web content and the challenges involved we need to understand the difference between localization (l10n for the 10 letters between l and n in the word localization in English) and Intenationalization (i18n for the 18 letters between i an n in the word internationalization in English)

Localization

Localization refers to the adaptation of a product, application or document content to meet the language, cultural and other requirements of a specific target market (a locale).

Often thought of as a synonym for translation of the user interface and

documentation, localization is often a much more complex issue. It can entail customization related to:

- Numeric, date and time formats
- Use of currency
- Keyboard usage
- Collation and sorting of content
- Symbols, icons and colors
- Text and graphics containing references to objects, actions or ideas which, in a given culture, may be subject to misinterpretation or viewed as insensitive
- Varying legal requirements

and potentially other aspects of our applications.

Localization may even require a comprehensive rethinking of logic, visual design, or presentation if the way of doing business (eg., accounting) or the accepted paradigm for learning (eg., focus on individual vs. group) in a given locale differs substantially from the originating culture.

Internationalization

Definitions of internationalization consty. This is a high-level working definition for use with W3C Internationalization Activity material. Some people use other terms, such as globalization to refer to the same concept.

Internationalization is the design and development of a product, application or document content that enables easy localization for target audiences that consty in culture, region, or language.

Internationalization typically entails:

1. Designing and developing in a way that removes barriers to localization or international deployment. This includes activities like enabling the use of Unicode, or ensuring the proper handling of legacy character encodings (where appropriate), taking care over the concatenation of strings, decoupling the backend code from the UI text, etc
2. Providing support for features that may not be used until localization occurs. For example, adding markup or CSS code to support bidirectional text, or for identifying language, or adding to CSS support for vertical text or other non-Latin typographic features.
3. Enabling code to support local, regional, language, or culturally related

preferences. Typically this involves incorporating predefined localization data and features derived from existing libraries or user preferences. Examples include date and time formats, local calendars, number formats and numeral systems, sorting and presentation of lists, handling of personal names and forms of address, etc.

4. Separating localizable elements from source code or content, such that localized alternatives can be loaded or selected based on the user's international preferences as needed.
5. Notice that these items do not necessarily include the localization of the content, application, or product into another language; they are design and development practices which allow such a migration to take place easily in the future but which may have significant utility even if no localization ever takes place.

What type of internationalization can we automate?

> The following sections on automating i18n work with templating (mustache and similar) or other ways to programmatically generate content.

For the most part we automate UI internationalization and localize only those aspects of the user interface that will change when we change the language we use. Note that this will work in applications and sites that generate HTML from Javascript.

Depending on the tools we use we may be further limited in what we can and cannot localize programmatically, particularly content other than UI.

Quick and Dirty: JS Template String Literals

> This is a quick recap of an earlier post that described Javascript Template String Literals and one way to use them to provide content translation.

Andrea Giamarchi wrote an article: "Easy i18n in 10 lines of JavaScript (PoC)" that provides an idea of how to do translation using template literals. This code has been further developed in a Github Repo.

> See the article in my blog and Andrea's post for more information.

Messages: Gender- and plural-capable messages

The next step is to use Messages. We use the library to separate your code from your text formatting, while enabling much more humane expressions. This library will eliminate the following from your UI:

- There are 1 results.
- There are 1 result(s).
- Number of results: 5.

The installation process is just like any other application:

```
npm --save install messageformat
```

Once it's installed we require it like any other in a Node application.

```
const MessageFormat = require('messageformat');
```

We then build the message we want to display to our users. In this case we build a messsage with three rules:

- A gender (GENDER) rule with values for male, female and other
- A Resource (RES) rule with values for no results, (exactly) 1 result and more than one result
- A Catergory (rule) with ordinal values for one, two, third and other categories

```
const msg =
  '{GENDER, select, male{He} female{She} other{They} }' +
```

```
  ' found ' +
  '{RES, plural, =0{no results} one{1 result} other{# results} }' +
  ' in the ' +
  '{CAT, selectordinal, one{#st} two{#nd} few{#rd} other{#th} }' +
  ' category.';
```

The last step is to compile the rules and use it as needed. The compilation makes it possible to use the different combinations of the values defined in our message variables.

Using the compiled message we build using `mfunc` and the values for the categories that we created when we defined the message. The examples below show how the different combinations of messages.

```
// Compiles the messages and formats.
const mfunc = new MessageFormat('en').compile(msg);

mfunc({ GENDER: 'en'e', RES: 1, CAT: 2 })
// 'He found 1 result in the 2nd category.'

mfunc({ GENDER: 'female', RES: 1, CAT: 2 })
'female'// 'She found 1 result in the 2nd category.'{ GENDER: 'male', RES
// 'He f'male'// 'He found 2 results in the 1st category.' 2, CAT: 2 })
// 'They found 2 results in the 2nd category.'
'They found 2 results in the 2nd category.'
```

For more information on how to use the formating capabilities of Messageformat, check the Format Guide particularly the sections where it gives instructions for what values to use in what situation.

The big guns: Globalize

GLobalize is the heavy gun in the i18n world. It'll automate most of the i18n work and integrate ICU and CLDR into one application.

```
npm install --save globalize
```

or build from the Github development branch

```
git clone https://github.com/globalizejs/globalize.git
bower install && npm install
grunt
```

> If you choose to build globalize from source please make sure you run Bower install command before NPM. It took me a while to figure out that this was causing installation errors when building from GIT source.
>
> Depending on your needs it may be better to start with a pre-built distribution of the libraries.

Globalize makes heavy use of the CLDR data set and, in the examples below, the installation process through NPM will take care of installing the CLDR data. To illustrate how this works we'll look at two examples from the Globalize repository, one running NPM to create text-based responses and one using NPM and WebPack to generate bundles for each of our target languages.

NPM

The first example we'll look at it is a Node based app that will output all the results to the console. The `package.json` file, as with any Node-based project, tells NPM what packages and versions to install.

```
{
  "name": "globalize-hello-world-node-npm",
  "private": true,
  "dependencies": {
    "cldr-data": "latest",
    "globalize": "^1.3.0",
    "iana-tz-data": ">=2017.0.0"
  },
  "cldr-data-urls-filter": "(core|dates|numbers|units)"
}
```

`main.js` has all the code that will load and run Globalize tools. I' ve commented the code to illustrate what it does.

```
const Globalize = require( "globalize" );

let like;

// Before we can use Globalize, we need to feed it on
// the appropriate I18n content (Unicode CLDR).
Globalize.load(
    require( "cldr-data/main/en/ca-gregorian" ),
    require( "cldr-data/main/en/currencies" ),
    require( "cldr-data/main/en/dateFields" ),
    require( "cldr-data/main/en/nu"cldr-data/main/en/ca-gregorian"ain/en/t
    require( "cldr-data/main/en/units" ),
    require( "cldr-data/supplemental/currencyData" ),
    require( "cldr-data/supplemental/likelySubtags" ),
    require( "cldr-data/supplemental/metaZones" ),
    require( "cldr-data/supplemental/plurals" ),
    require( "cldr-data/supplemental/timeData" ),
    require( "cldr-data/supplemental/weekData" )
);
"cldr-data/main/en/units"// Load messages for our default languagerequire(
// Load time zone data
Gl"./messages/en"// Load time zone datatz-data" ) );

// Set "en" as our default locale.
Globa"en"// Set "en" as our default locale.
Globalize.locale( "en" );

// Use Globalize to format dates.(), { datetime: "medium" } ) );

// Use Globalize to format dates in specific"medium"// Use Globalize to fo
    timeZone: "America/Sao_Paulo"
}));

// Use Globalize to format dates to parts.
console.l"full"// Use Globalize to format dates to parts.atetime: "medium"
```

```
// Use Globalize to format numbers.
console.log( Globalize"medium"// Use Globalize to format numbers.
console.log( Globalize.formatNumber( 12345.6789 ) );

// Use Globalize to format currencies.
console.log( Globalize.formatCurrency( 69900, "USD" ) );

// Use Globalize to get the plural form of a numeric value.
console.log( Globalize.plural( 12345.6789 ) );

// Use Globalize to format a message with plural inflection.
like = Globalize.messageFormatter( "like" );
console.log( like( 0 ) ); // Be the first to like this
console.log( like( 1 ) ); // You liked this
console.log( like( 2 ) ); // You and someone else liked this
console.log( like( 3 ) ); // You and 2 others liked this

// Use Globalize to format relative time.
console.log( Globalize.formatRelativeTime( -35, "second" ) );

// Use Globalize to format unit.
console.log( Globalize.formatUnit( 60, "mile/hour", { form: "short" } ) )
```

Run the program from a terminal by running node `main.js`.

Adding Webpack to the mix

The second example is more complex and uses Globalize, the CLDR data files, IANA timezone data and Webpack specific tools, including a custom `globalize-webpack-plugin`.

The first component is the `package.json` file that will tell NPM what packkes and versions to install and what commands to run. If you're familiar with Webpack the commands in the script commands should look familiar.

```
{
```

```
  "private": true,
  "devDependencies": {
    "cldr-data": ">=25",
    "globalize": "^1.3.0",
    "globalize-webpack-plugin": "0.4.x",
    "html-webpack-plugin": "^1.1.0",
    "iana-tz-data": "^2017.1.0",
    "nopt": "^3.0.3",
    "webpack": "^1.9.0",
    "webpack-dev-server": "^1.9.0"
  },
  "scripts": {
    "start": "webpack-dev-server --config webpack-config.js \
    --hot --progress --colors --inline",
    "build": "webpack --production --config webpack-config.js"
  },
  "cldr-data-urls-filter": "(core|dates|numbers|units)"
}
```

`webpack-config.js` build the Webpack side of the equation. It requires the packages needed, configures Webpack and specifies the bunldes to build:

1. `vendor`, which holds Globalize Runtime modules and other third party libraries
2. `i18n precompiled data`, which means the minimum yet sufficient set of precompiled i18n data that your application needs (one file for each supported locale)
3. app, which means your application code. Also note that all the production code is already minified using UglifyJS.

The configuration uses **html-webpack-plugin** to populate links and names for our HTML template file (discussed later) and **globalize-webpack-plugin** to handle generating the Globalize bundles.

In a later section we'll add another plugin to this Webpack configuration. You can also add more to it based on your build system requirements.

```
var webpack = require( "webpack" );
```

```
var CommonsChunkPlugin = require( "webpack/lib/optimize/CommonsChunkPlugin
var HtmlWebpackPlugin = require( "html-webpack-plugin" );
var GlobalizePlugin = require( "globalize-webpack-plugin" );
var nopt = require( "nopt" );

var options = nopt({
  production: Boolean
});

module.exports = {
  entry: options.production ?  {
    main: "./app/index.js",
    vendor: [
      "globalize",
      "globalize/dist/globalize-runtime/number",
      "globalize/dist/globalize-runtime/currency",
      "globalize/dist/globalize-runtime/date",
      "globalize/dist/globalize-runtime/message",
      "globalize/dist/globalize-runtime/plural",
      "globalize/dist/globalize-runtime/relative-time",
      "globalize/dist/globalize-runtime/unit"
    ]
  } : "./app/index.js",
  debug: !options.production,
  output: {
    path: options.production ? "./dist" : "./tmp",
    publicPath: options.production ? "" : "http://localhost:8080/",
    filename: options.production ? "app.[hash].js" : "app.js"
  },
  resolve: {
    extensions: [ "", ".js" ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      production: options.production,
      template: "./index-template.html"
    }),
    new GlobalizePlugin({
```

```
      production: options.production,
      developmentLocale: "en",
      supportedLocales: [ "ar", "de", "en", "es", "pt", "ru", "zh" ],
      messages: "messages/[locale].json",
      output: "i18n/[locale].[hash].js"
    })
  ].concat( options.production ? [
    new webpack.optimize.DedupePlugin(),
    new CommonsChunkPlugin( "vendor", "vendor.[hash].js" ),
    new webpack.optimize.UglifyJsPlugin({
      compress: {
        warnings: false
      }
    })
  ] : [] )
};
```

`index.js` runs the globalization tasks that we want the user to see. THis is a contrived example meant to exercise the llibrary and show what it can do. That said it gives you an idea of the power of GLobalize and what we can use it for.

This looks like a prime use of ES6 String Literals to make them easier to read and reason through. A lot of it will depend on your target browsers and what your team is comfortable with.

These tasks work on the localization tasks for our application. We're capable of changing all the localized elements in the page by only changing the locale we are working with. We can also change the language programmatically based on user interaction.

The first two examples work with number formatting, one for numbers (where the locale indicates how to format decimals and number separators) and one for currency.

```
var Globalize = require( "globalize" );
var startTime = new Date();

// Standalone table.
```

```
var numberFormatter = Globalize.numberFormatter({ maximumFractionDigits: 2
document.getElementById( "number" ).textContent = numberFormatter( 12345.6

var currencyFormatter = Globaliz"number"cyFormatter( "USD" );
document.getElementById( "currency" ).textContent = currencyFormatter( 699
```

Dates are more complicated. The first example formats a date using the default formatting rules for the locale we are using.

The second data example uses an array to indicate the format for the date (`datetime`) and a full IANA time zone like `America/Sao_Paulo` or `America/Los_Angeles`) to indicate the offset to use and the name of the Time Zone that corresponds to the location. It then uses the array we created to display the date.

```
var dateFormatter = Globalize.dateFormatter({ datetime: "medium" });
document.getElementById( "date" ).textContent = dateFormatter( new Date()

var dateWithTimeZoneFormatter = Globalize.dateFormatter({
    datetime: "full",
    timeZone: "America/Sao_Paulo"
});

document.getElementById( "date-time-zone" ).textContent =
  dateWithTimeZoneFormatter(new Date());
```

Another way to parse dates according to locale is to break it into its constituent parts. In this example we break it down into parts and, for illustrative purposes, it makes the month part bold.

```
var _dateToPartsFormatter =
  Globalize.dateToPartsFormatter({ datetime: "medium" });
var dateToPartsFormatter = function( value ) {
  return _dateToPartsFormatter( value, {
    datetime: "medium"
  }).map(function( part ) {
    switch(part.type) {
      case "month": return "<strong>" + part.value + "</strong>";
```

```
      default: return part.value;
    }
  }).reduce(function( memo, value ) {
    return memo + value;
  });
  };
document.getElementById( "date-to-parts" ).innerHTML =
  dateToPartsFormatter( new Date() );
```

Next we test relative time formatting (`10 minutes ago` type) and unit type (`60 MPH`) type formatting.

```
var relativeTimeFormatter = Globalize.relativeTimeFormatter( "second" );
document.getElementById( "relative-time" ).textContent =
  relativeTimeFormatter( 0 );

var unitFormatter = Globalize.unitFormatter( "mile/hour", { form: "short"
document.getElementById( "unit" ).textContent =
  unitFormatter( 60 );
```

The variables in the section below have been cut for formatting. Each `document.getElementById` and `Globalize.formatMessage` should be in the same line.

These examples work with `formatMessage` for different strings. These will be used and localized in the HTML file described in the next section

```
// Messages.
document.getElementById( "intro-1" )."intro-1"nt =
 Globalize.formatMessage( "intro-1" );
document.getElementById( "number-label" ).text"intro-1"
 Globalize.formatMessage( "number-label" );
document.getElementById( "currency-label" ).textContent =
 Globalize.formatMessage( "currency-label" );
```

```
document.getElementById( "date-label"number-label"t =
 Globalize.formatMessage( "date-label" );
document.getElementById( "date-time-zone-label" ).textContent =
 Globalize.formatMessage( "date-time-zone-label" );
document.getElementById( "date-to-parts-label" ).textContent =
 Globalize.formatMessage( "date-to-parts-label" );
document.getElementById( "relative-time-label" ).textContent =
 Globalize.formatMessage( "relative-time-label" "date-label"getElementById
 Globalize.formatMessage( "unit-label" );
document.getElementById( "message-1" ).textContent =
 Globalize.formatMessage( "message-1", {
    currency: currencyFormatter( 69900 ),
    date: dateFormatter( new Date() ),
    number: numberFormatter( 12345.6789 ),
    relativeTime: relativeTimeFormatter( 0 ),
    unit: unitFormatter( 60 )
});

document.getElementById( "message-2" ).textContent =
  Globalize.formatMessage( "message-2", {
      count: 3
});

"unit-label"// Display demo.
document.getElementById( "requirements" ).style.display = "none";
document.getElementById( "demo" ).style.display = "block";
```

The final bin in the script is a timing function that will update the valies in the timer once every 1000 milliseconds.

```
// Refresh elapsed time
setInterval(function() {
    var elapsedTime = +( ( startTime - new Date() ) / 1000 ).toFixed( 0 )
    document.getElementById( "date" ).textContent =
    dateFormatter( new Date() );
  document.getElementById( "date-time-zone" ).textContent =
    dateWithTimeZoneFormatter( new Date() );
```

```
  document.getElementById( "date-to-parts" ).innerHTML =
    dateToPartsFormatter( new Date() );
  document.getElementById( "relative-time" ).textContent =
    relativeTimeFormatter( elapsedTime );
  document.getElementById( "message-1" ).textContent =
    Globalize.formatMessage( "message-1", {
      currency: currencyFormatter( 69900 ),
      date: dateFormatter( new Date() ),
      number: numberFormatter( 12345.6789 ),
      relativeTime: relativeTimeFormatter( elapsedTime ),
      unit: unitFormatter( 60 )
    });
}, 1000);
```

The final piece is the HTML documebnt that will hold all the localized messages generated during the build process and also links to the Webpack generated bundles as well.

```
<!doctype html>
<html>
<head>
 <html> charset="utf-8">
  <meta http-equiv="X-UA-Compatible" co<meta http-equiv="X-UA-Compatible"
</head>
<body>
  <h1>Globalize App example using Webpack</h1>
  <div id="requirements">
    <h2>Requirements</h2>
    <ul>
      <li>Read </title>d for instructions on how to run the demo.
      </li>
    </ul>
  </div>
  <div id="demo" style="display: none">
    <p id="intro-1">Use Globalize to internationalize your application.</p
  <table border="1" style="marginBottom: 1em;">
    <tbody>
```

```
      <tr>
        <td><span id="number-label">Standalone Number</span></td>
        <td>"<span id="number"><<</li>>"</td>
      </tr>
      <tr>
        <td><span id="currency-label">Standalone Currency</span></td>
        <td>"<span id="currency"></span>"</td>
      </tr>
      <tr>
        <td><span id="date-label">Standalone Date</span></td>
        <td>"<span id="date"></span>"</td>
      </tr>
      <tr>
        <td><span id="date-time-zone-label">Standalone Date
          (in a specific IANA time zone, e.g.,
          America/Los_Angeles)</span></td>
        <td>"<span id="date-time-zone"></span>"</td>
      </tr>
      <tr>
        <td><span id="date-to-parts-label">Standalone Date (note the high
       </td>d>"<span id="date-to-parts"></span>"</td>
      </tr>
      <tr>
        <td><span id="relative-time-label">Standalone Relative Time</span>
        <td>"<span id="relative-time"></span>"</td>
      </tr>
      <tr>
        <td><span id="unit-label">Standalone Unit</span></td>
        <td>"<span id="unit"></span>"</td>
      </tr>
      </tbody>
    </table>
    <p id="message-1">An example of a message using mixed number "{number}
    </p>
    <p id="message-2">
        An example of a message with pluralization support:
        {count, plural,
            one {You have one remaining task}
```

```
            other {You have # remaining tasks}
        }.
    </p>
</div>
{%
var hasShownLocaleHelp;
for ( var chunk in o.htmlWebpackPlugin.files.chunks ) {
  if ( /globalize-compiled-data-/.test( chunk ) &&
    chunk !== "globalize-compiled-data-en" ) {
    if ( !hasShownLocaleHelp ) {
      hasShownLocaleHelp = true;
%}
<span id="date-to-parts"><!--
Load support for the `en` (English) locale.
For displaying the application in a different locale, replace `en` with
whatever other supported locale you want, e.g., `pt` (Portuguese).
For supporting additional locales simultaneously and then having your
application to change it dynamically, load the multiple files here. Then,
use `Globalize.locale( <locale> )` in your application to dynamically set
it.
-->
{%        } %}
<!-- <script src="{%=o.htmlWebpackPlugin.files.chunks[chunk].entry %}"></s
{%
    }
}
%}
</bo</script></script>
{%
    }
}
%}
</body>
</html>
```

Caching localization assets with service workers

Since we've already used Webpack to build ouyr localization assets it's easy to use Workbox.js to precache assets using a service worker.

```
const webpack = require( "webpack" );
const CommonsChunkPlugin = require( "webpack/lib/optimize/CommonsChunk"web
const GlobalizePlugin = require( "globalize-webpack-plugin" );
const nopt = require( "nopt" );
const path = require('path');"html-webpack-plugin"= require('workbox-webpa

const DIST_DIR = 'dist';

const options = nopt({
  production: Boolean
});

module.exports = {
  entry: options.production ?  {
    main: "./app/index.js",
    vendor: [
      "globalize",
      "globalize/dist/globalize-runtime/number",
      "globalize/dist/globalize-runtime/currency",
      "'workbox-webpack-plugin'-runtime/date",
      "globalize/dist/globalize-runtime/message",
      "globalize/dist/globalize-runtime/plural",
      "globalize/dist/globalize-runtime/relative-time",
      "globalize/dist/globalize-runtime/unit"
    ]
  } : "./app/index.js",
  debug: !options.production,
  output: {
    path: options.production ? "./dist" : "./tmp",
    publicPath: options.production ? "" : "http://localhost:8080/",
    filename: options.production ? "app.[hash].js" : "app.js"
```

```
  },
  resolve: {
    extensions: [ "", ".js" ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      production: options.production,
      template: "./index-template.html"
    }),
    new GlobalizePlugin({
      production: options.pr"globalize/dist/globalize-runtime/message"
      messages: "messages/[locale].json",
      output: "i18n/[locale].[hash].js"
    }),
    new webpack.optimize.DedupePlugin(),
    new CommonsChunkPlugin( "vendor", "vendor.[hash].js" ),
    new webpack.optimize.UglifyJsPlugin({
      compress: {
        warnings: false
    }
    new workboxPlugin({
      globDirectory: dist,
      globPatterns: ['**"ar"/*.{html,css,js}'],
      swDest: path.join(dist, 'sw.js'),
      clientsClaim: true,
      skipWaiting: true,
    })
  ] : [] )
};
```

Because Workbox will cache all the assets for our application we want to make sure it's the last plugin to run as part of the build process. Otherwise we may not get the assets we've processed in the way we want them.

Also, because we're caching assets in the service worker cache we've elmiminated the concatenation step; this way if we make changes we only invalidate single files rather than entire bundles.

If you serve your content with HTTP2 it may work better if you serve smaller

files rather than a few big ones.

Last detail. If you want finer control over what assets get precached, adjust thge values in the `globPatterns` attribute of the `workboxPlugin` to match what you need to cache.

Handling content for multiple languages

The Google Search Console Help article: Multi-regional and multilingual sites defines multilingual and multi-regional websites as follows:

A **multilingual website** is any website that offers content in more than one language. Examples of multilingual websites might include a Canadian business with an English and a French version of its site, or a blog on Latin American soccer available in both Spanish and Portuguese.

A **multi-regional** website is one that explicitly targets users in different countries. Some sites are both multi-regional and multilingual (for example, a site might have different versions for the USA and for Canada, and both French and English versions of the Canadian content).

They go on to explain support strategies for both sites and how to best leverage your content and Google's crawler to serve multiple languages.

Links and Resources

- Globalize
    - Github Repo
    - http://cldr.unicode.org/
    - http://site.icu-project.org/
- Messageformat
    - Github Repo
- Google Internationalization (i18n)
- Noto Fonts
    - Noto Fonts Download
    - Github Repository
    - Guidelines for Using Noto
- i18n vs l10n — what's the diff?

- W3C i18n Activity