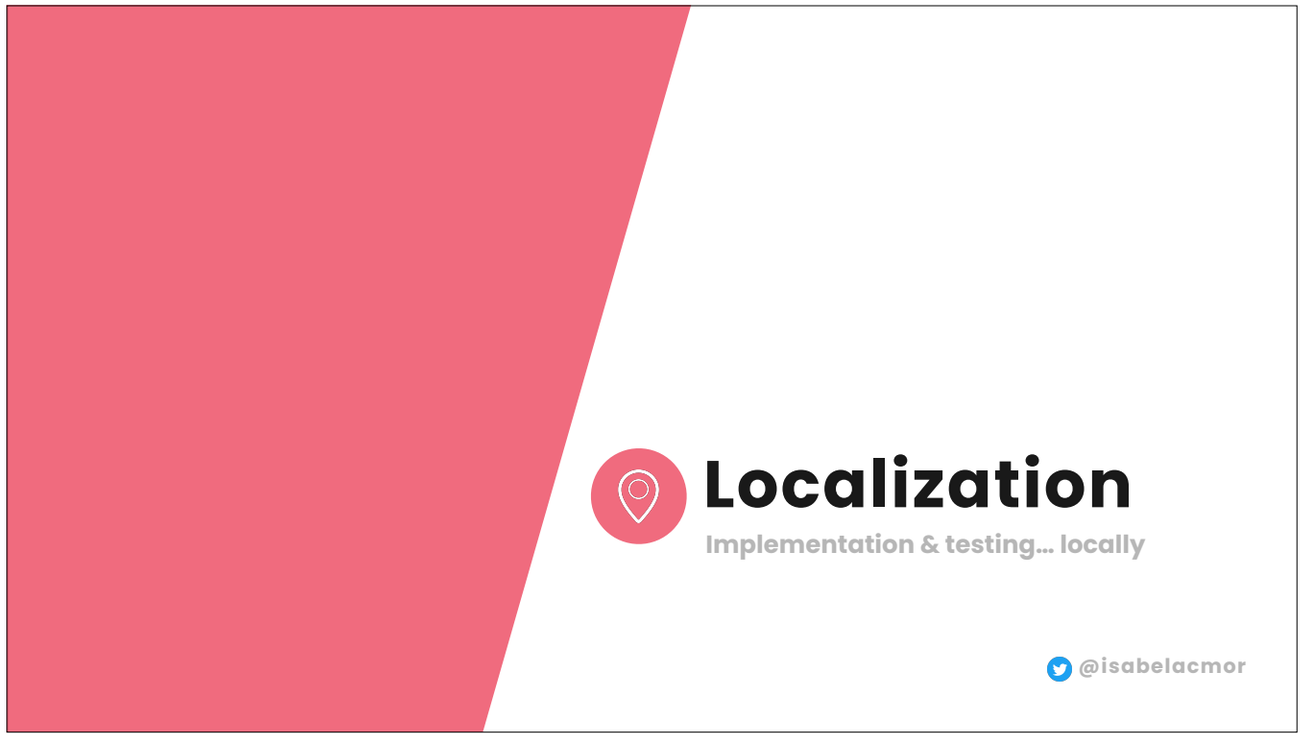
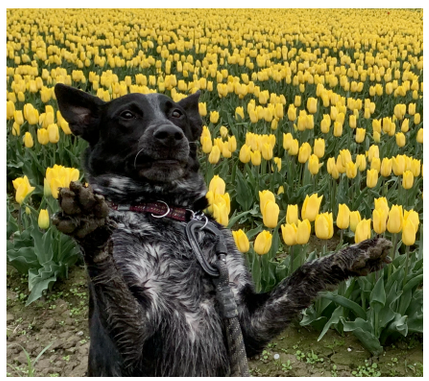

Localization
Implementation & testing... locally
@isabelacmor



Before we dive in, I want to tell you guys a bit more about me. The two most important things:

1. This is my dog, Atlas. He's very excited about this talk and I hope you guys are too.
2. I work at a tiny startup in Seattle, Washington called CedarAI. We work on railroad technology and the entire industry is still run on pen and paper. This is a huge challenge and it's even harder to tackle for a brand new product in a brand new space. In this talk, I'll be sharing how to get your app ready to be shipped to a global user base with a few different localization techniques and gotchas. I'm super excited to be giving this talk here in Sweden to a diverse audience, so this couldn't be a more fitting topic. Let's get started.

WHAT WE'LL COVER

- What's localization?
- Localization vs internationalization
- Why is localization important?
- Localizing a React & vanilla JS app
- Testing a localized app

@isabelacmor

WHAT'S LOCALIZATION?

Adapting a product to meet language and cultural requirements

- Numeric, date, time formats
- Currency, measurements
- Keyboard usage
- Symbols, icons, colors
- Legal requirements
- Text and graphics that may be insensitive

@isabelacmor

Basically, everything your user sees is fair game to be localized

LOCALIZATION VS INTERNATIONALIZATION

Removes barriers to localization

- Bidirectional text support
- Unicode support
- String concatenation

Support for dynamic UI

- No hardcoded strings!

@isabelacmor

They both share the same goal, but internalization is a more broad term that describes the overall process of creating a product that can be shipped on a global scale, whereas localization gets down into the details of what needs to be done in order for that to happen.

Internationalization significantly affects the ease of the product's localization. Retrofitting a linguistically- and culturally-centered deliverable for a global market is obviously much more difficult and time-consuming than designing a deliverable with the intent of presenting it globally. (Think back to the Y2K effort and trying to "undo" two-character year fields that were built on the assumption of “19xx").

The end result and overarching goal of both is to [bullets]

WHY IS LOCALIZATION IMPORTANT?

Expand user-base

- 25.2% of internet users worldwide speak English
- 76.3% of internet users across top 10 languages

Sensitive across all cultures

In-context localization

Localization itself can be expensive. It can take a lot of dev time to setup the infrastructure to handle this is if it wasn't built into the product's design from the beginning. It can also be costly to have professional translations done across all the strings in your product.

But localizing your product also expands your user base and, therefore, your income stream.

In-context review focuses more on quality and experience, rather than functionality.

WHY IS LOCALIZATION IMPORTANT?

Finger lickin' good = eat your fingers off in Chinese

WHY IS LOCALIZATION IMPORTANT?

VS Installer welcome image contains offensive element for Chinese

Closed - Fixed fixed in: visual studio 2019 version 16.0 rc.2 Fixed In: Visual Studio 2017 version 15.9.10
Fixed In: Visual Studio 2019 version 16.0 visual studio 2019 version 16.0 rc Setup Fixed In: Visual Studio 2019 version 16.0 Preview 4

EDI Edi Wang reported Mar 02 at 02:17 PM

After install VS2019 in the Visual Studio Installer. There is a welcome background image showes a guy wearing a GREEN HAT riding a bike.

However, GREEN HAT is offensive in Chinese culture. A guy with a green hat means his girlfriend or wife is cheating on him.

Please just Bing “green hat china”, this is serious. Please consider changing the color of the hat in that image.

WHY IS LOCALIZATION IMPORTANT?

promoted its toothpaste in a distinct area in Southeast Asia by highlighting that it “whitens your teeth.” This campaign entirely failed because the locals chew betel nuts to blacken their teeth as it is considered attractive.

LOCALIZATION GOALS

Single source of truth

Arbitrary number of languages

Decoupled

Automated

@isabelacmor

Single source of truth. There should be no confusion about which underlying English string is the correct one. If inconsistencies appear, it should be clear what to use as a reference.

Arbitrary number of languages. There should be no upper bound to how many languages we are able to support. Adding a new language should require minimal effort from developers.

Decoupled. Adding translations and developing features should be kept as independent as possible.

Automated. Automate as much as possible. This makes the process both more efficient and less prone to errors.

LOCALIZING REACT & VANILLA JS APP

react-intl

```
import { injectIntl } from 'react-intl';

class MyComponent extends Component {
  render() {
    const intl = this.props;
    const title = intl.formatMessage({ id: 'title' });
    return (<div>{title}</div>);
  }
};

export default injectIntl(MyComponent);
```

react-intl-universal

```
// JS code
import intl from 'react-intl-universal';

const greeting = intl.get('HELLO', { name: 'Isabela' })};
const location = intl.get('WELCOME', { place: 'NordicJS' });

<h1>{greeting}</h1>
<h2>{location}</h2>

// Outputs "Hello, Isabela. Welcome to NordicJS!"
```


One library you may run into for react apps is react-intl.
I've found this to be a fine solution for very simple react apps, but of course, it doesn't support vanilla JS. So even in your React app, if you have a generic JS or TS file you need to return some text, you wouldn't be able to do that. Any text has to be generated from inside a JSX or TSX file because the library uses an HOC to inject functionality. This breaks our decoupling goal because now when we're developing features, we need to keep in mind that they must be HOC wrapped and need to limit our tech stack to React.

So what's the alternative?

LOCALIZING REACT & VANILLA JS APP

react-intl

```
import { injectIntl } from 'react-intl';

class MyComponent extends Component {
  render() {
    const intl = this.props;
    const title = intl.formatMessage({ id: 'title' });
    return (<div>{title}</div>);
  }
};

export default injectIntl(MyComponent);
```

react-intl-universal

```
// JS code
import intl from 'react-intl-universal';

const greeting = intl.get('HELLO', { name: 'Isabela' })};
const location = intl.get('WELCOME', { place: 'NordicJS' });

<h1>{greeting}</h1>
<h2>{location}</h2>

// Outputs "Hello, Isabela. Welcome to NordicJS!"
```


With the react-intl-universal library, you can handle all your localization needs from any JS app - including vanilla JS.

You define a JSON with your locales and all the string resources you'll use in your app.

Then in the JS file, you do an intl.get call with the key for the string resource.

USING REACT-INTL-UNIVERSAL

Simple strings

```
// JS code
import intl from 'react-intl-universal';

const greeting = intl.get('HELLO', { name: 'Isabela' })};
const location = intl.get('WELCOME', { place: 'NordicJS' });

<h1>{greeting}</h1>
<h2>{location}</h2>

// Outputs "Hello, Isabela. Welcome to NordicJS!"
```

```
// Locale data JSON
en: {
  HELLO: 'Hello, {name}.',
  WELCOME: 'Welcome to {place}!'
}
```


This setup is nice because your strings are separated from your app and you don't have hardcoded values anymore, but we still don't quite have internationalization done since we're only pulling English values. We're going to dig in a little more into implementation details for that, but for now, I want to cover some more areas of internationalize that we'll need to handle and that this library can help us with.

For this to work in many languages, a message has to be written and translated as a single unit. Typically, this is a string with placeholder syntax for the variable elements.

You should never concatenate your own strings from various formatted elements and your own injected variables. This makes it extremely hard for translators to understand the full context of your message.

Not only might translators get context wrong, entire translated strings could end up with the wrong grammar. This is most commonly seen with languages that differentiate between masculine and feminine words. If you rely on passing in a data type as a variable to your translation, you might end up with incorrect pronouns.

It’s helpful to be explicit when translating strings. Usually in programming, you’ll hear that you need to write DRY code - Don’t Repeat Yourself. But this is pretty much the opposite of what we want to to do when handling localization. We want to follow WET - want everything translated.

I came up with that myself.

So that’s where this ICU MessageFormat comes into play. It provides a programming language agnostic format for translating strings.

USING REACT-INTL-UNIVERSAL

ICU MessageFormat

- Message has to be written and translated as a single unit

Do:

```
// Locale data JSON
en: {
  NO_MESSAGES: 'No new messages'
},
br: {
  NO_MESSAGES: 'Sem novas mensagens'
}

// JS code
intl.get('NO_MESSAGES');

// English output: "No new messages"
// Portuguese output: "Sem novas mensagens"
```

Don't:

```
// Locale data JSON
{
  NO_MESSAGES: 'No new {data_type}'
}

// JS code
intl.get('NO_MESSAGES', {data_type: 'messages'});

// English output: "No new messages"
// Portuguese output: "Sem novos mensagens"
```


Drawing from these guidelines, here's what a very basic translation would, using only strings.

USING REACT-INTL-UNIVERSAL

ICU MessageFormat

- Message has to be written and translated as a single unit

```
{ name, type, format }
```


Following those guidelines, the ICU MessageFormat looks like this.

The first parameter is the variable name. This is the key in our JSON. [click]

The second parameter is the type of value that will be passed in. This can be number, date, time, or plural.

The third parameter is the format. It's optional and provides additional context for how the value should be displayed.

USING REACT-INTL-UNIVERSAL

Pluralization

```
// Locale data JSON
{
  PHOTO: 'You have {num, plural, =0 {no photos.} =1 {one photo.} other {# photos.}}'
}

// JS code
intl.get('PHOTO', { num: 0 });
intl.get('PHOTO', { num: 1 });
intl.get('PHOTO', { num: 100 });

// Output
// "You have no photos."
// "You have one photo."
// "You have 100 photos"
```


Let’s dig into some examples for each of the type and format options.

First, you can pass in variable numbers to the API call and corresponding numerical values to determine what pluralization you want.

The keyword “plural” comes from the ICU User Guide. This is used to select sub-messages based on numerical value, which we see here being done with =0 and =1. The “other” keyboard acts like the “default” keyword in a switch statement.

The ICU User Guide also suggests using full sentences when possible for sub-messages. This gives both translators the most context possible when translating the strings and allows developers to know exactly what strings they should be referencing.

USING REACT-INTL-UNIVERSAL

Dates

```
// Locale data JSON
{
  SALE_START: 'Sale begins {start, date}',
  SALE_END: 'Sale ends {end, date, long}'
}

// JS code
intl.get('SALE_START', { start: new Date() });
intl.get('SALE_END', { end: new Date() });

// Output
// "Sale begins 4/19/2017"
// "Sale ends April 19, 2017"
```

- `short` shows date as shortest as possible
- `medium` shows short textual representation of the month
- `long` shows long textual representation of the month
- `full` shows dates with the most detail

Localization doesn't mean just translating strings. As probably everyone in this room knows, everyone hates the American date format (except Americans of course). A global product launch from an American-based company would probably be a complete disaster if they didn't localize their date formats.

Thankfully, this library allows us to do just that.

ICU MessageFormat allows us to specify the type of value we're localizing. In this example, we use date. [click]

The third parameter is how we want to format our value. [click] For date, we can pass along short, medium, long, or full. If we don't pass along a third parameter, it'll default to short.

USING REACT-INTL-UNIVERSAL

Times

```
// Locale data JSON
{
  COUPON: 'Coupon expires at {expires, time, medium}'
}

// JS code
intl.get('COUPON', {expires:new Date()});

// Output
// Coupon expires at 6:45:44 PM
```

`short` shows times with hours and minutes

`medium` shows times with hours, minutes, and seconds

`long` shows times with hours, minutes, seconds, and timezone

Just like with dates, we can localize timestamps by passing in time as the second parameter and a format as the third parameter.

USING REACT-INTL-UNIVERSAL

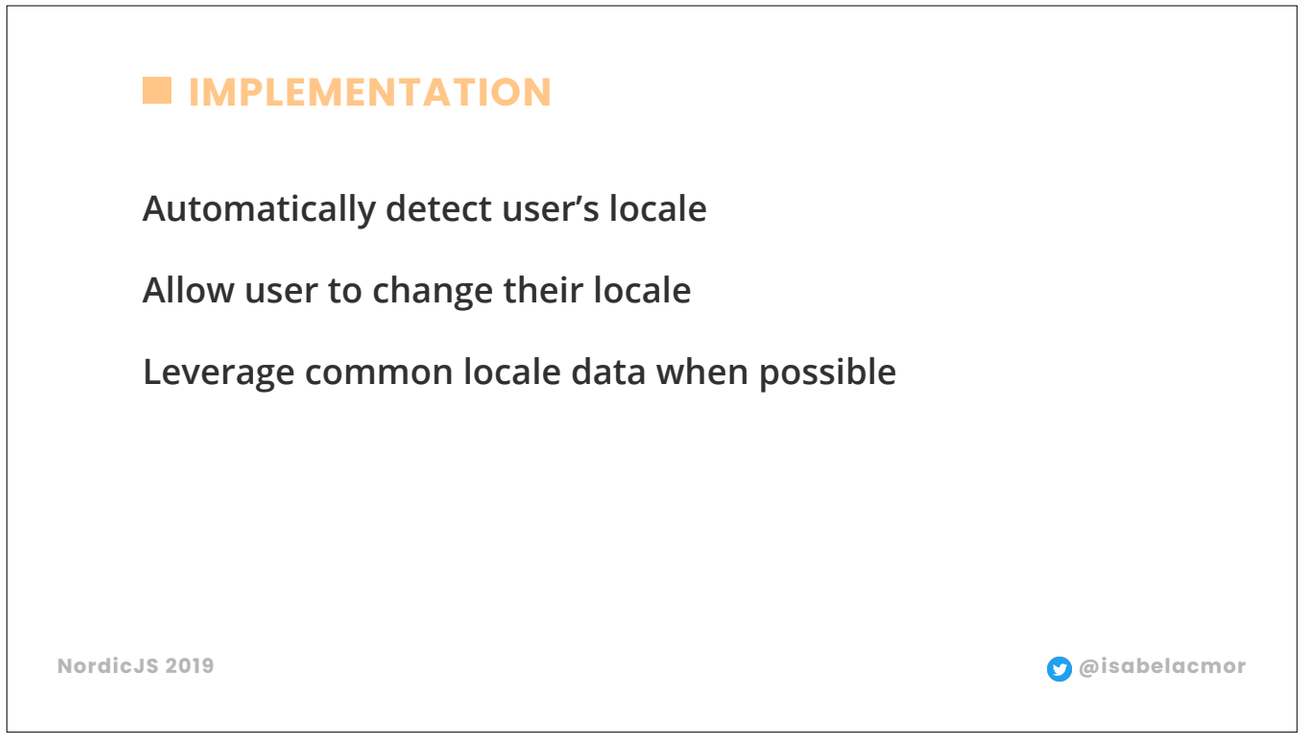
Currency

```
// Locale data JSON
{
  SALE_PRICE: 'The price is {price, number, USD}'
}

// JS code
intl.get('SALE_PRICE', { price: 123456.781 });

// Output
// "The price is $123,456.78"
```

https://www.currency-iso.org/dam/downloads/lists/list_one.xml

We also need to handle formatting numerical values, which also includes currency.

We have a similar structure to handle this as we did for dates. [click]. This time we set our second parameter to number. The format parameter then becomes the country code for the currency. You can find the currency codes from the ISO standards website [click]



All this is great, but as you can see, we still have hardcoded formatting options, like USD in the last example. So let's shift gears and look into actually implementing this library into our existing React or JS app.

In order to make this low-friction for end users of our app, we should automatically recognize the user's default locale. Of course, from a complete user experience standpoint, we should also allow the user to change this from a settings page.

For browser rendering, the common locale data such as date, currency, and number formats are automatically loaded from CDN on demand, based on the locale we have the library configured to use.

IMPLEMENTATION

```
componentDidMount() {
  this.loadLocale();
}
```


The first thing we can do is start the setup of determining what the user's current locale is. Most of this example will be in React for simplicity, but keep in mind that like library can be used in VanillaJS as well, you might just need to tweak some of the code (for example, instead of componentWillMount, we might us a window load function in vanillaJS).

IMPLEMENTATION

```
loadLocales() {
  let currentLocale = intl.determineLocale({
    urlLocaleKey: "lang",
    cookieLocaleKey: "lang"
  });

  // Default to English if no language found
  if (!_.find(SUPPORTED_LOCALES, { value: currentLocale })) {
    currentLocale = "en-US";
  }

  intl.init({
    // Loads common locale for currency, date, etc
    currentLocale,
    locales: {
      [currentLocale]: require(`json!../localeData/${currentLocale}.json`)
    }
  });

  this.setState({ localeLoaded: true });
}
```


Here's what it looks like to load the user's current locale. First, I use the library to try to determine the user's locale. This is done with either a URL parameter or a cookie. If a locale isn't set, we can default to English.

IMPLEMENTATION

```
loadLocales() {
  let currentLocale = intl.determineLocale({
    urlLocaleKey: "lang",
    cookieLocaleKey: "lang"
  });

  // Default to English if no language found
  if (!_.find(SUPPORTED_LOCALES, { value: currentLocale })) {
    currentLocale = "en-US";
  }

  intl.init({
    // Loads common locale for currency, date, etc
    currentLocale,
    locales: {
      [currentLocale]: require(`json!../localeData/${currentLocale}.json`)
    }
  });

  this.setState({ localeLoaded: true });
}
```


@isabelacmor

Then, I initialize the library with the data required for that locale.

By passing in the currentLocale string, I'll get access to all the common locale data that I'll use to format currency, dates, times, etc. Then I pass in an object called locales with the locale data. I use a library called json-loader to dynamically require the json file so that I don't have to import a ton of locale files that I won't be using.

IMPLEMENTATION

```
const SUPPORTED_LOCALES = [
  { name: "English", value: "en-US" },
  { name: "简体中文", value: "zh-CN" },
  { name: "繁體中文", value: "zh-TW" },
  { name: "français", value: "fr-FR" },
  { name: "日本の", value: "ja-JP" }
];
```


Now we can set up a way for users to change their locale.We'll make this a dropdown of all the supported languages. We can create an array of objects where the name shown in the dropdown maps to the locale string for that language, since that's the value we'll be sending the intl library to initialize the locale.

IMPLEMENTATION

```
location.search = `?lang=${e.target.value}`;
```

```
loadLocales() {
  let currentLocale = intl.determineLocale({
    urlLocaleKey: "lang",
    cookieLocaleKey: "lang"
  }) || this.state.currentLocale;

  // Default to English if no language found
  if (!_.find(SUPPORTED_LOCALES, { value: currentLocale })) {
    currentLocale = "en-US";
  }

  intl.init({
    // Loads common locale for currency, date, etc
    currentLocale,
    locales: {
      [currentLocale]: require(`json!../localeData/${currentLocale}.json`)
    }
  });

  this.setState({ localeLoaded: true });
}
```

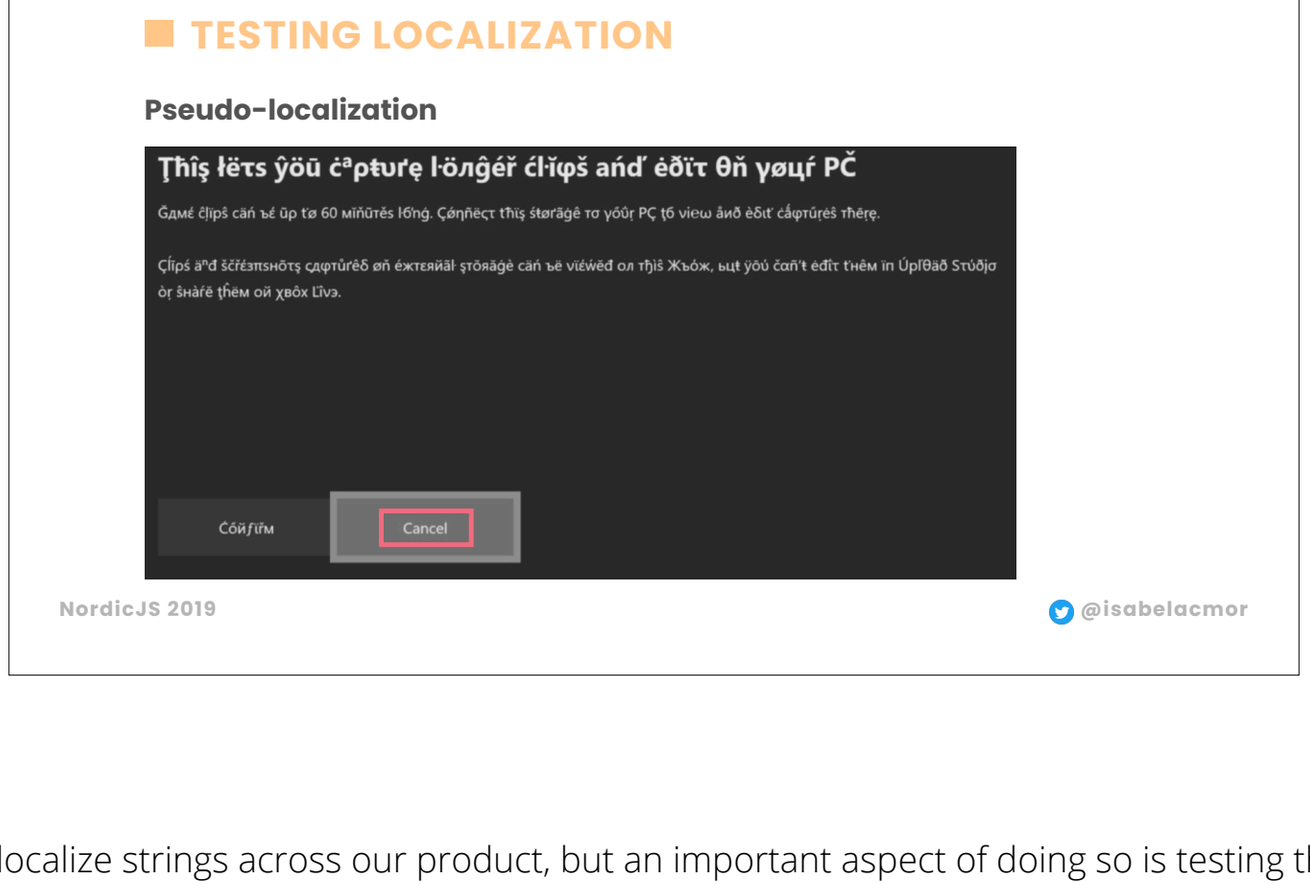

Now, whenever the selected locale has been changed from the dropdown, we can either update the URL that the determineLocale function looks up to determine the current locale by doing location.search = `?lang=${lang}`;

Or we can set the state and update our loadLocales function to check the state for the locale first.

IMPLEMENTATION

```
if (intl.getInitOptions().currentLocale === 'fr-FR') {
    // Render something specific for French locale
} else {
    // Render the standard element
}
```


Going back to localization including symbols and images, we can use the currentLocale from the library to determine what elements to render as well.



We've covered a lot about how we'd localize strings across our product, but an important aspect of doing so is testing that we haven't missed any strings.

Now you may be thinking... maybe you only know one language or don't want to fiddle with manually testing if strings are translated across your product by relying on faking a locale (or toggling some user setting).

To solve this, we can use a strategy called pseudo-localization. By leveraging this, we can force our dev environment to use a specific locale file that returns unicode text that closely matches English, but looks obviously different with various accent marks and ligatures. We used this technique a lot when I was at Microsoft. This is a screenshot of what a dev build of Windows looks like with pseudo-localization enabled.

As you can see from this screenshot, the text is still fairly readable, and it makes it very obvious what strings are still hardcoded and aren't being sent through our localization service. Here we can see that cancel doesn't show up in our unicode font, so we know it hasn't been translated.

TESTING LOCALIZATION

Pseudo-localization

```
npm install pseudo-localization
```

```
npx pseudo-localization ./path/to/locales/en.json -o ./path/to/locales/pseudo.json
```


I really like CLI tools that will generate pseudo-localized JSON files that we can use in place of the locale JSON files in local dev builds.

I use the package pseudo-localization. Once you install it, you can use the CLI version to generate the pseudo-localized JSON file

TESTING LOCALIZATION

Pseudo-localization

```
loadLocales() {
  // ... intl.determineLocale

  intl.init({
    currentLocale,
    locales: {
       [currentLocale]: process.env.NODE_ENV === 'development'
                        ? require(`json!../localeData/pseudoLocale.json`)
                        : require(`json!../localeData/${currentLocale}.json`)
    }
  });

  // ...
}
```


@isabelacmor

From there, we can do something like this in our app code.

If the process environment is dev, we can immediately require the pseudoLocale.json file. Otherwise, we'll require whatever locale was detected.

The only thing you need to remember to do (or automate), is to re-run the pseudo-localization CLI tool whenever you create or edit a string in your base locale JSON file. Otherwise, an empty string will be rendered if it doesn't exist in the pseudo-localized JSON file.

TESTING LOCALIZATION

Pseudo-localization

```
expect(wrapper.find('my-element').text()).toEqual(intl.get('MY_STRING'));
```


So that solves the problem from a manual testing standpoint, but we should really update our tests to make sure they catch any non-localized strings and make them independent of what locale we run them under.

This is actually super simple to do. We can run our tests like normal and instead of expecting text to equal some hard coded string, we can expect it to equal a value returned by the intl library

WRAP UP

No hardcoded strings

Translate messages as a single unit (minimize string variables)

No manual string concatenation

Use react-intl-universal and pseudo-localization

Don’t DIY localization

@isabelacmor

So to wrap things up:

- No hardcoded strings (even if you’re not quite ready to jump to localization, moving strings to their own JSON file will be a huge stepping stone and make your codebase cleaner in general)

THANKS

NordicJS 2019

@isabelacmor

That’s all it takes to localize your app! I hope I’ve been able to show you how easy it is to do and encourage you to set up localization in your products.