Internationalization with react-intl

react-intl / format.js

Formatjs is a set of javascript libraries that focus on internationalization and formatting numbers, dates and strings.

It works at this point very well with react, ember and vue.

IntlProvider

In order to tell the application what language, region and timezone should be used, an IntlProvider will need to be defined in your application.

```
import { IntlProvider } from "react-intl";
<IntlProvider
  locale="en"
  defaultLocale="en"
  messages={messages}
  timeZone="Europe/Vienna"
>
  {children}
</IntlProvider>;
```

Formatting numbers

To format numbers within your application, you can use the `<FormattedNumber>` component. It will take into account the locale defined in the `IntlProvider`.

```
<IntlProvider locale="de" >
  {/* 1.000.000 */}
  <FormattedNumber value={1000000} />
</IntlProvider>

<IntlProvider locale="en-US" >
  {/* 1,000,000 */}
  <FormattedNumber value={1000000} />
</IntlProvider>
```

Formatted numbers options

The `FormattedNumber` component internally only wraps the native Int.NumberFormat functions, which expose a lot of options.

```
// 30,00 €
<FormattedNumber value={30} style="currency" currency="EUR" />

// 30,000
<FormattedNumber
  value={30}
  minimumFractionDigits={3}
/>
// 30,12
<FormattedNumber
  value={30.1233454}
  maximumFractionDigits={2}
/>
```

Formatting dates

To format dates within the application, you can use the `<FormattedDate>` component. It will take into account the `timeZone` and `locale` formats defined in the `IntlProvider` for display purposes.

```
<IntlProvider locale="de" >
  {/* 5.12.2023 */}
  <FormattedDate value={new Date()} />
</IntlProvider>

<IntlProvider locale="en-US" >
  {/* 12/5/2023 */}
  <FormattedDate value={new Date()} />
</IntlProvider>
```

Formatting date options

Same as the number formatter, the `FormattedDate` internally wraps the `Intl.FormatDateTime` functionality which will expose additional formatting options.

```
<IntlProvider locale="de" >
  {/* Dienstag, 5. Dezember 2023 */}
  <FormattedDate value={new Date()} dateStyle="full"/>
  {/* 05.12.23 */}
  <FormattedDate value={new Date()} dateStyle="short"/>
</IntlProvider>

<IntlProvider locale="en-US" >
  {/* Tuesday, December 5, 2023 */}
  <FormattedDate value={new Date()} dateStyle="full" />
  {/* 12/5/23 */}
  <FormattedDate value={new Date()} dateStyle="shot" />
</IntlProvider>
```

Formatting date options - time and timezone

```
<IntlProvider locale="de" timeZone="Europe/Vienna">
  {/* 05.12.23, 23:10:00 MEZ */}
  <FormattedDate value={new Date()} dateStyle="short" timeStyle="long"/>
  {/* 05.12.23, 23:10 */}
  <FormattedDate value={new Date()} dateStyle="short" timeStyle="short"/>
</IntlProvider>

<IntlProvider locale="en-US" timeZone="America/Detroit">
  {/* 12/5/23, 5:10:00 PM EST */}
  <FormattedDate value={new Date()} dateStyle="short" timeStyle="long" />
  {/* 12/5/23, 5:10 PM */}
  <FormattedDate value={new Date()} dateStyle="shot" timeStyle="short" />
</IntlProvider>
```

There is also an option for only formatting time with `<FormattedTime />`.

Relative time

Relative times can also be defined, by using the `<FormattedRelativeTime />` component.

```
<IntlProvider locale="de-DE">
  {/* vor 50 Sekunden */}
  <FormattedRelativeTime value={-50} />
  {/* vor 50 Tagen */}
  <FormattedRelativeTime value={-50} unit="days" />
</IntlProvider>

<IntlProvider locale="en-US">
  {/* 50 seconds ago*/}
  <FormattedRelativeTime value={-50} />
</IntlProvider>
```

List formats

```
<IntlProvider locale="de-DE">
  {/* Me, myself und I */}
  <FormattedList value={["Me", "myself", "I"]} />
  {/* Me, myself oder I */}
  <FormattedList value={["Me", "myself", "I"]} type="disjunction"/>
</IntlProvider>

<IntlProvider locale="en-US">
  {/* Me, myself, and I */}
  <FormattedList value={["Me", "myself", "I"]} />
</IntlProvider>
```

Plural rules

A simple version of adding plurals to your application is with the `FormattedPlural` component. It will take a `value` and a plurality message.

```
{ /* message */ }
<FormattedPlural value={10} one="message" other="messages" />;

{ /* messages */ }
<FormattedPlural value={10} one="message" other="messages" />;

{ /* messages */ }
<FormattedPlural value={0} zero="messages" one="message" other="messages" />;
```

MDN plurality formats

Messages

You can define custom messages, that will need to be translated manually and provided to the `IntlProvider`.

To define a translateable message, you will need to define a `FormattedMessage`.

```
const messages = {
  "custom-message": "Translated message",
};

<IntlProvider locale="en-US" messages={}>
  <FormattedMessage
    id="custom-message"
    defaultMessage="This is my default message"
  />
</IntlProvider>;
```

Dynamic values in messages

```
// You can use dynamic variables in the messages to make them reusable

// Hello, John
<FormattedMessage
  id="custom-message"
  defaultMessage="Hello, {user}"
  values={{
    user: 'John'
  }}
>
```

Dynamic values with plurals

```
// And also use plurals in the message definition

// You have 2 items
<FormattedMessage
  id="custom-message"
  defaultMessage="You have {count, plural,
    =0 {no items}
    one {# item}
    other {# items}
  }"
  values={{
    count: 2
  }}
>
```

Workflow for translations

Format.js provides a command line interface that provides some helpers in order to get all the messages that have been defined (and are in need for translation) in an application.

- extract
- compile
- compileFolder

FormatJS command line

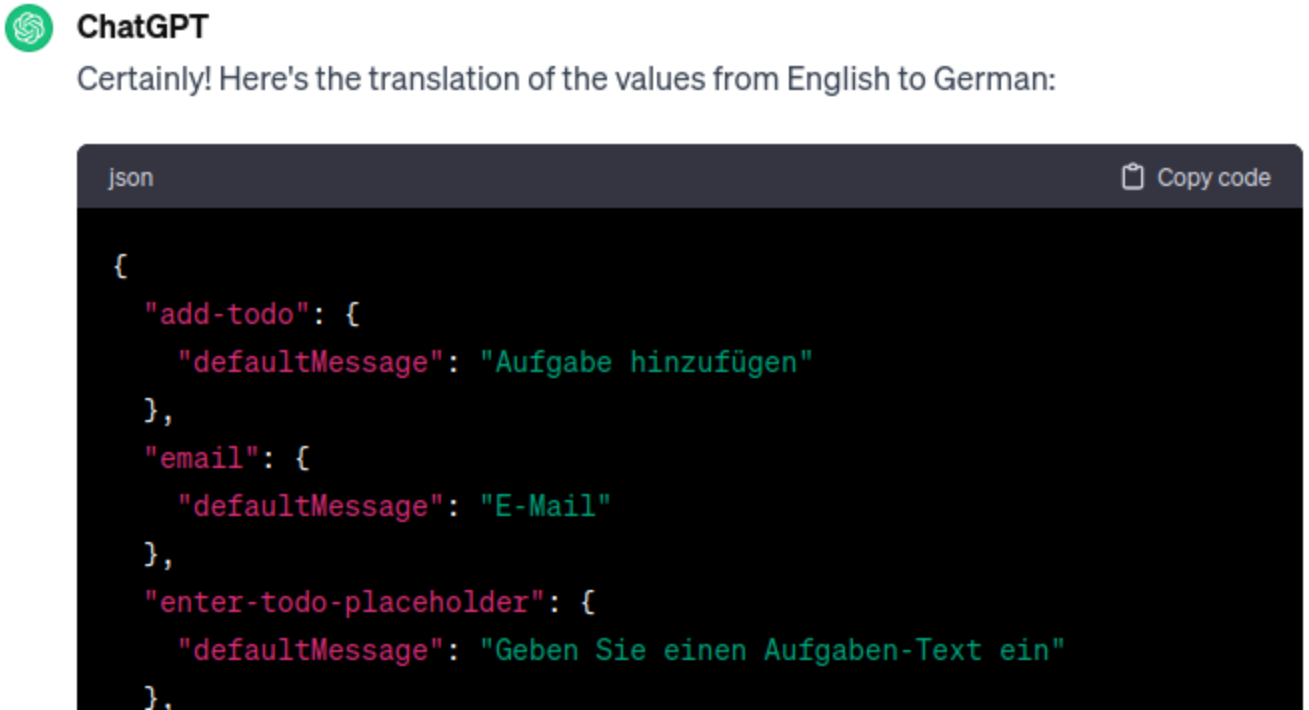
Extract

The extraction will look for all `FormattedMessage` or `intl.formatMessage`, `definedMessages` function calls and will extract the message declarations into a single json file with the defaultMessages.

```
{
  "add-todo": {
    "defaultMessage": "Add todo"
  },
  "email": {
    "defaultMessage": "Email"
  },
  "enter-todo-placeholder": {
    "defaultMessage": "Enter a todo text"
  }
}
```

Translation

You then can either copy then extracted .json file and create translations there. You could also upload these to any translation service / translation agency and get translated messages back. But openAI is also quite good with translating these:



ChatGPT

Certainly! Here's the translation of the values from English to German:

json — Copy code

```
{
  "add-todo": {
    "defaultMessage": "Aufgabe hinzufügen"
  },
  "email": {
    "defaultMessage": "E-Mail"
  },
  "enter-todo-placeholder": {
    "defaultMessage": "Geben Sie einen Aufgaben-Text ein"
  },
```

Use the messages

You can then either import the messages and pass them to the `IntlProvider` or add some fetching logic to get them async.

```
import germanMessages from './de.json';

<IntlProvider locale="de-DE" messages={germanMessages}>
 {/* ... */}
</IntlProvider>
```