Internacionalización y puesta en producción

Las aplicaciones web, son la expresión perfecta del mundo globalizado en el que vivimos. Preparar tu aplicación para ese mundo se conoce como *internationalization*, o recortadamente **i18n**. Pero globalizar no significa "café con leche para todos". Atender a las necesidades específicas de tus usuarios es el proceso de **localization**.

Para empezar esto afecta las traducciones de los contenidos: sean datos o textos fijos. Pero también a las adaptaciones culturales para la presentación de fechas, números, iconos... En Angular tenemos herramientas y soluciones para poner en marcha proyectos globalizados.

Partiendo del código tal como quedó en Velocidad y SEO con el SSR de Angular Universal. Al finalizar tendrás una aplicación que adapta a la cultura del usuario.

> Código asociado a este tutorial en *GitHub*: AcademiaBinaria/angular-boss

1 Traducciones y contenido

Traducir una aplicación es el primer paso para que un usuario la acepte y comprenda mejor. Afecta a los datos que venga dese un API, y afecta a los textos fijos que acompañan a los datos.

Como estamos viendo Angular en frontend, me centraré en el problema de los literales en las *templates*. Existen varias estrategias para tratarlo, aquí veremos la oficial. Se apuesta por dos principios:

- El programador vea el texto en la plantilla en el idioma nativo escogido para el desarrollo. Así puede hacerse una idea del tamaño y apariencia inicial y facilita mucho la legibilidad del HTML.
- El usuario descargue la aplicación específica para su idioma, de forma que no necesite llamadas extra ni resolución dinámica para los textos y menajes fijos.

1.1 xi18n

Para cumplir con esos criterios se necesita extraer del código los literales que se van traducir. Para ello se usa una herramienta y un convenio.

La herramienta es `xi18n` que viene con el CLI. El convenio es la directiva `i18n` que viene con Angular. El uso es muy sencillo, como puedes ver este ejemplo.

`apps\warehouse\src\app\app.component.html`

```
<header>
  <h1 i18n>Welcome to the Angular Builders Warehouse</h1>
</header>
<img src="../assets/Warehouse-Building.jpg"
     alt="Warehouse building"
     i18n-alt>
<router-outlet></router-outlet>
<footer>
```

```
    <a href="https://angular.builders"
       target="blank">Angular.Builders: </a>
    <span i18n>a store of resources for developers and software architects.</span>
  </footer>
```

Para la extracción se recomienda, como siempre, crear un script en el `package.json`

```
{
  "i18n:warehouse": "ng xi18n warehouse --output-path src/locale",
}
```

> Atención IVY: por el momento debemos desactivar el renderizador *Ivy* para poder realizar la extracción.

`apps\warehouse\tsconfig.app.json`

```
{
"angularCompilerOptions": {
    "enableIvy": false
  }
}
```

Aparecerá un fichero *xml* un tanto feo: `/src/locale/messages.xlf`. Aunque es procesable automáticamente, por ahora vamos a usarlo de forma manual. Para empezar crea una copia y nómbrala incluyendo el idioma destino. Por ejemplo `messages.es.xlf`

Sólo tendrás que duplicar cada etiqueta `source` en su equivalente traducido `target`... y traducirlo, claro. Te pongo aquí un ejemplo de como quedaría.

```
<trans-unit id="5c08a98fac06c803712ab27dbb81d889af5ef5fb" datatype="html">
  <source>Welcome to the Angular Builders Warehouse</source>
  <target>Bienvenido al almacén de Angular Builders</target>
  <context-group purpose="location">
    <context context-type="sourcefile">src/app/app.component.html</context>
    <context context-type="linenumber">2</context>
  </context-group>
</trans-unit>
```

Necesitarás crear tantos ficheros como idiomas tengas que mantener. Si son muchos te recomiendo usar alguna herramienta. Tampoco debes olvidarte de realizar este proceso ante cada despliegue para incorporar los posibles cambios.

Y hablando de despliegue..

1.2 Build configurations

Una vez que tenemos las traducciones en los ficheros de idioma, ahora hay que compilarlos para generar una versión distribuible específica para cada idioma. De nuevo esto es un proceso un tanto tedioso la primera vez, y que desde luego merece ser automatizado.

Se trata de crear una nueva entrada en la rama `architect/build/configurations` de tu aplicación en el fichero `angular.json`.

```
"production-es": {
  "fileReplacements": [
    {
      "replace": "apps/warehouse/src/environments/environment.ts",
      "with": "apps/warehouse/src/environments/environment.prod.es.ts"
    },
  ],
  "outputPath": "dist/apps/warehouse/es/",
  "i18nFile": "apps/warehouse/src/locale/messages.es.xlf",
  "i18nFormat": "xlf",
  "i18nLocale": "es",
  "baseHref": "es",
}
```

Esencialmente se le dicen las rutas y los ficheros que debe utilizar durante el proceso de construcción. Para usarla definimos un par de scripts. Uno para la construcción en sí mediante `ng build` y el otro para poder ver el resultado con un servidor de ficheros estáticos.

```
{
  "build:warehouse-es": "ng build warehouse --configuration=production-es",
  "start:warehouse-es": "npm run build:warehouse-es && angular-http-server --path
./dist/apps/warehouse/es",
}
```

2 Adaptaciones culturales de tiempo y moneda

Pero no todo van a ser textos. También hay fechas, número, imágenes... Se necesitan hacer pequeñas adaptaciones en un montón de lugares. En Angular, algunas son casi gratis.

2.1 Registro manual en app.module o Auto registro en angular.json

Habrás usado *pipes* como `date` o `number` desde el día que empezaste con Angular. Si no haces nada especial seguro que te chocó ver que las fechas y los números en formato norte americano. Es el que viene de fábrica, pero e fácil cambiarlo.

Manual

Puedes asignar la cultura en tu aplicación de forma implícita registrándola al inicio del módulo raíz. La clave está en importar el fichero con las definiciones particulares de tu cultura.

Todo está descargado en node-modules. Pero sólo se empaqueta y envía al navegador la cultura seleccionada.

```
import { registerLocaleData } from '@angular/common';
import localeEs from '@angular/common/locales/es';

registerLocaleData(localeEs);
```

Automático

Otra forma es usar el fichero `package.json` para configurarlo. Puedes establece la cultura deseada en la propiedad `i18nLocale`.

```
"i18nLocale": "es"
```

2.2 Tiempo, moneda y contenido

En cualquier caso, todos los pipes deAngular empezarán a comportarse educadamente conforme a la cultura establecida.

`apps\warehouse\src\app\app.component.html`

```
<article class="card">
  <p>{{ building.date | date:'long' }}</p>
  <p>${{ building.value | number }}<i> {{ building.status }}</i></p>
</article>
```

`apps\warehouse\src\app\app.component.ts`

```
public building = {
  date: Date.now(),
  value: 2345.897,
  status: 'buy'
};
constructor() {
  if (this.building.status === 'buy') {
    this.building.status = environment.buy;
  } else {
    this.building.status = environment.sell;
  }
}
```

Otra cosa pueden ser pequeños textos, lógica , iconos o clases css. A veces todo esto varía de un país a otro. Esas variaciones puedes almacenarla en ficheros externos o dentro del código. Pero si optas por esto ultimo

entonces es mejor que uses los *environments* para almacenar configuraciones específicas.

```
# Original environment.prod.ts
{
  buy: 'for buy',
  sell: 'for sell'
}
# En español environment.prod.es.ts
{
  buy: 'para comprar',
  sell: 'para vender'
}
```

Ahora ya tienes una aplicación que se puede desplegar adaptada a las preferencias culturales de tus usuarios. Continúa tu formación avanzada para crear aplicaciones Angular.

> Aprender, programar, disfrutar, repetir. -- *Saludos, Alberto Basalo*