Localization, i18n

Guides you through the steps to internationalize your application to provide localized versions with respect to both Localized Models as well as Localized Data.

'Localization' is a means to adapting your app to the languages of specific target markets.

This guide focuses on static texts such as labels. See CDS and Localized Data for information about how to manage and serve actual payload data in different translations.

Table of Contents

Externalizing Texts Bundles

All you have to do to internationalize your models is to externalize all of your literal texts to text bundles and refer to the respective keys from your models as annotation values. Here is a sample of a model and the corresponding bundle.

srv/my-service.cds

```cds
service Bookshop {
  entity Books @(
    UI.HeaderInfo: {
      Title.Label: '{i18n>Book}',
      TypeName: '{i18n>Book}',
      TypeNamePlural: '{i18n>Books}',
    },
  ){/*...*/}
}
```

_i18n/i18n.properties

```properties
Book = Book
Books = Books
foo = Foo
```

> You can define the keys of your properties entries.

↳ *Learn more about annotations in CSN.*

Then you can translate the texts in localized bundles, each with a language/locale code appended to its name, for example:

```sh
_i18n/
  i18n.properties           # dev main → 'default fallback'
  i18n_en.properties        # English  → 'default language'
  i18n_de.properties        # German
  i18n_zh_TW.properties     # Traditional Chinese
  ...
```

Where to Place Text Bundles?

Recommendation is to put your properties files in a folder named `_i18n` in the root of your project, as in this example:

```zsh
bookshop/
├─ _i18n/
│  ├─ i18n_en.properties
│  ├─ i18n_de.properties
│  ├─ i18n_fr.properties
│  └─ i18n.properties
│  ...
```

By default, text bundles are fetched from folders named *_i18n* or *i18n* in the neighborhood of models, i.e. all folders that contain `.cds` sources or parent folders thereof. For example, given the following project layout and sources:

```zsh
bookshop/
├─ app/
│  ├─ browse/
│  │  └─ fiori.cds
│  ├─ common.cds
│  └─ index.cds
├─ srv/
│  ├─ admin-service.cds
│  └─ cat-service.cds
├─ db/
│  └─ schema.cds
└─ readme.md
```

We will be loading i18n bundles from all of these locations, if existing:

```zsh
bookshop/app/browse/_i18n
bookshop/app/_i18n
bookshop/srv/_i18n
bookshop/db/_i18n
bookshop/_i18n
```

↳ *Learn more about the underlying machinery in the reference docs for* `cds.i18n`

CSV-Based Text Bundles

For smaller projects you can use CSV files instead of *.properties* files, which you can easily edit in *Excel*, *Numbers*, etc.

The format is as follows:

key	en	de	zh_CN	...
Book	Book	Buch	...	
Books	Books	Bücher	...	
...				

With this CSV source:

```csv
key,en,de,zh_CN,...
Book,Book,Buch,...
Books,Books,Bücher,...
...
```

Merging Algorithm

Each localized model is constructed by applying:

1. The *default fallback* bundle (that is, *i18n.properties*), then ...
2. The *default language* bundle (usually *i18n_en.properties*), then ...
3. The requested bundle (for example, *i18n_de.properties*)

In that order.

So, the complete stack of overlaid models for the given example would look like this (higher ones override lower ones):

Source	Content
_i18n/i18n_de.properties	specific language bundle
_i18n/i18n_en.properties	default language bundle
_i18n/i18n.properties	default fallback bundle
srv/my-service.cds	service definition
db/schema.cds	underlying data model

Set default language

> The *default language* is usually `en` but can be overridden by configuring `cds.i18n.default_language` ☼ in your project's *package.json*.

Merging Reuse Bundles

If your application is importing models from a reuse package, that package comes with its own language bundles for localization. These are applied upon import, so they can be overridden in your models as well as in your language bundles and their translations.

For example, assuming that your data model imports from a *foundation* package, then the overall stack of overlays would look like this:

Source
./_i18n/i18n_de.properties
./_i18n/i18n_en.properties
./_i18n/i18n.properties
./srv/my-service.cds
./db/schema.cds
foundation/_i18n/i18n_de.properties
foundation/_i18n/i18n_en.properties
foundation/_i18n/i18n.properties
foundation/index.cds
foundation/<private model a>.cds
foundation/<private model b>.cds
...

Determining User Locales

Upon incoming requests at runtime, the user's preferred language is determined as follows:

1. Read the preferred language from the first of:
    1. The value of the `sap-locale` URL parameter, if present.
    2. The value of the `sap-language` URL parameter, but only if it's `1Q`, `2Q` or `3Q` as described below.
    3. The first entry from the request's `Accept-Language` header.
2. Narrow to normalized locales as described below.

> **Differences between Node.js and Java runtimes**
>
> CAP Node.js accepts formats following the available standards of POSIX and RFC 1766, and transforms them into normalized locales. CAP Java only accepts language codes following the standard of RFC 1766 (or IETF's BCP 47).

Normalized Locales

To reduce the number of required translations, most determined locales are normalized by narrowing them to their main language codes only, for example, `en_US`, `en_CA`, `en_AU` → `en`, except for these preserved language codes:

Locale	Language
zh_CN	Chinese - China
zh_HK	Chinese - Hong Kong, China
zh_TW	Chinese traditional - Taiwan, China
en_GB	English - English
fr_CA	French - Canada
pt_PT	Portuguese - Portugal
es_CO	Spanish - Colombia
es_MX	Spanish - Mexico

Locale	Language
en_US_x_saptrc	SAP tracing translations w/ `sap-language=1Q`
en_US_x_sappsd	SAP pseudo translations w/ `sap-language=2Q`
en_US_x_saprigi	Rigi language w/ `sap-language=3Q`

Configuring Normalized Locales

For CAP Node.js, the list of preserved locales is configurable, for example in the *package.json* file, using the configuration option `cds.i18n.preserved_locales` ☼ as follows:

```jsonc
{"cds":{
  "i18n": {
    "preserved_locales": [
      "en_GB",
      "fr_CA",
      "pt_PT",
      "pt_BR",
      "zh_CN",
      "zh_HK",
      "zh_TW"
    ]
  }
}}
```

In this example we removed `es_CO` and `es_MX` from the list, and added `pt_BR` .

In CAP Java the preserved locales can be configured via the `cds.locales.normalization.includeList` ☼ property.

> ***Note:***
>
> However this list is configured, ensure to have translations for the listed locales, as the fallback language will otherwise be `en` .

Use Underscores in File Names

Due to the ambiguity regarding standards, for example, the usage of hyphens (`-`) in contrast to underscores (`_`), CAP follows the approach of the SAP Translation Hub . Using that approach, CAP normalizes locales to **underscores** as our de facto standard.

In effect, this means:

- We support incoming locales as language tags using hyphens to separate sub tags [1], for example `en-GB`.
- We always normalize these to underscores, which is `en_GB`.
- Always use underscores in filenames, for example, `i18n_en_GB.properties`
- Always use underscores when filling `LOCALE` columns of localized text tables (e.g. in CSV files).

[1] CAP Node.js also supports underscore separated tags, for example `en_GB`.

Previous page
SAP HANA Native

Next page
Localized Data