Localized Data

This guide extends the localization/i18n of static content, such as labels or messages, to serve localized versions of actual application data.

Localized data refers to the maintenance of different translations of textual data and automatically fetching the translations matching the users' preferred language, with per-row fallback to default languages, if the required translations aren't available. Language codes are in ISO 639-1 format.

> Find a **working sample** at https://github.com/capire/bookshop .

Table of Contents

- Declaring Localized Data
- Behind the Scenes
  - Resolving localized texts via views
  - Resolving search over localized texts at runtime
  - Base Entities Stay Intact
  - Extending .texts Entities
- Pseudo var $user.locale
  - Determining $user.locale from Inbound Requests
  - Programmatic Access to $user.locale
  - Propagating $user.locale to Databases
- Reading Localized Data
  - In Agnostic Code
  - For End Users
  - For Translation UIs
- Serving Localized Data

    - localized. Helper Views
    - Read Operations
    - Write Operations
    - Update Operations
    - Delete Operations
- Nested Localized Data
- Adding Initial Data

Declaring Localized Data

Use the `localized` modifier to mark entity elements that require translated texts.

```cds
entity Books {
  key ID       : UUID;
      title    : localized String;
      descr    : localized String;
      price    : Decimal;
      currency : Currency;
}
```

↳ *Find this source also in **capire/bookshop**.*

> ***Restriction***
>
> If you want to use the `localized` modifier, the entity's keys must not be associations.

> `localized` in entity sub elements isn't currently supported and is ignored. This includes `localized` in structured elements and structured types.

Behind the Scenes

The `cds` compiler automatically unfolds the previous definition as follows, applying the basic mechanisms of Managed Compositions, and Scoped Names:

First, a separate *Books.texts* entity is added to hold translated texts:

```cds
entity Books.texts {
  key locale : sap.common.Locale;
  key ID : UUID; //= source's primary key
  title : String;
  descr : String;
}
```

↳ *See the definition of* `sap.common.Locale` .

Second, the source entity is extended with associations to *Books.texts*:

```cds
extend entity Books with {
  texts : Composition of many Books.texts on texts.ID=ID;
  localized : Association to Books.texts on localized.ID=ID
    and localized.locale = $user.locale;
}
```

The composition `texts` points to all translated texts for the given entity, whereas the `localized` association points to the translated texts and is narrowed to the request's locale.

Third, views are generated in SQL DDL to easily read localized texts with an equivalent fallback:

```cds
entity localized.Books as select from Books {*,
  coalesce (localized.title, title) as title,
  coalesce (localized.descr, descr) as descr
};
```

Resolving localized texts via views

As we already mentioned, the CDS compiler is already creating views that resolve the translated texts internally. Once a CDS runtime detects a request with a user locale, it uses those views instead of the table of the involved entity.

Note that SQLite doesn't support locales like *SAP HANA* does. For *SQLite*, additional views are generated for different languages. Currently those views are generated for the

locales 'de' and 'fr' and the default locale is handled as 'en'.

```json
"i18n": { "for_sqlite": ["en", ...] }
```

> In *package.json* put this snippet in the `cds` block, but don't do so for *.cdsrc.json*.

> For testing with SQLite: Make sure that the *Books* table contains the English texts and that the other languages go into the *Books.texts* table.

For *H2*, you need to use the property as follows.

```json
"i18n": { "for_sql": ["en", ...] }
```

Resolving search over localized texts at runtime

Although the approach with the generated localized views is very convenient, it's limited on SQLite and shows suboptimal performance with large data sets on *SAP HANA*. Especially for search operations the performance penalty is very critical. Therefore, both CAP runtimes have implemented a solution targeted for search operations. If the `localized` association of your entity is present and accessible by the given CQL statement, the runtimes generate SQL statements that resolve the localized texts. This is optimized for the underlying database.

When your CQL queries select entities directly there is no issue as the `localized` association is automatically accessible in an entity with localized elements. If your CQL query selects from a view, it is important that your views' projection preserves the `localized` association.

The following view definitions preserve the `localized` association in the view, allowing you to optimize query execution, or for broader language support on SQLite, H2, and PostgreSQL.

Preferred - Exclude elements that mustn't be exposed:

```cds
entity OpenBookView as select from Books {*}
  excluding { price, currency };
```

Include the `localized` association:

```cds
entity ClosedBookView as select from Books {
  ID, title, descr, localized
```

```
};
```

Base Entities Stay Intact

In contrast to similar strategies, all texts aren't externalized but the original texts are kept in the source entity. This saves one join when reading localized texts with fallback to the original ones.

Extending *.texts* Entities

It's possible to collectively extend all generated *.texts* entities by extending the aspect `sap.common.TextsAspect`, which is defined in *common.cds*.

For example, the aspect can be used to add an association to the `Languages` code list entity, or to add flags that help you to control the translation process.

Example:

```cds
extend sap.common.TextsAspect with {
  language : Association to sap.common.Languages on language.code = locale;
}
```

The earlier description is simplified, *.texts* entities are generated with an include on `sap.common.TextsAspect`, if the aspect exists. For the *Books* entity, the generated *.texts* entity looks like:

```cds
entity Books.texts : sap.common.TextsAspect {
  key ID : UUID;
  title : String;
  descr : String;
}
```

When the include is expanded, the key element `locale` is inserted into *.texts* entities, alongside all the other elements that have been added to `sap.common.TextsAspect` via extensions.

```cds
entity Books.texts {
  // from sap.common.TextsAspect
```

```
    key locale: sap.common.Locale;
    language : Association to sap.common.Languages on language.code = locale;
    // from Books
    key ID : UUID;
    title : String;
    descr : String;
}
```

It isn't allowed to extend `sap.common.TextsAspect` with

- Managed Compositions of Aspects
- localized elements
- key elements

For entities that have an annotation `@fiori.draft.enabled` , the corresponding *.texts* entities also include the aspect, but the element `locale` isn't marked as a key and an element `key ID_texts : UUID` is added.

Pseudo var `$user.locale`

As shown in the second step, the pseudo variable `$user.locale` is used to refer to the user's preferred locale and join matching translations from `.texts` tables. This pseudo variable allows expressing such queries in a database-independent way, which is realized in the service runtimes as follows:

Determining `$user.locale` from Inbound Requests

The user's preferred locale is determined from request parameters, user settings, or the *accept-language* header of inbound requests as explained in the Localization guide.

Programmatic Access to `$user.locale`

The resulting normalized locale is available programmatically, in your event handlers.

- Node.js: `req.locale`
- Java: `eventContext.getParameterInfo().getLocale()`

Propagating `$user.locale` to Databases

Finally, the normalized locale is **propagated** to underlying databases using session variables, that is, `$user.locale` translates to `session_context('locale')` in native SQL of SAP HANA and most databases.

Not all databases support session variables. For example, for *SQLite* we currently would just create stand-in views for selected languages. With that, the APIs are kept stable but have restricted feature support.

Reading Localized Data

Given the asserted unfolding and user locales propagated to the database, you can read localized data as follows:

In Agnostic Code

Read *original* texts, that is, the ones in the originally created data entry:

```sql
SELECT ID, title, descr from Books
```

For End Users

Reading texts for end users uses the `localized` association, which requires prior propagation of `$user.locale` to the underlying database.

Read *localized* texts in the user's preferred language:

```sql
SELECT ID, localized.title, localized.descr from Books
```

For Translation UIs

Translation UIs read and write texts in all languages, independent from the current user's preferred one. They use the to-many `texts` association, which is independent from `$user.locale`.

Read texts in **different** translations:

```sql
SELECT ID, texts[locale='fr'].title, texts[locale='fr'].descr from Books
```

Read texts in **all** translations:

```sql
SELECT ID, texts.locale, texts.title, texts.descr from Books
```

Serving Localized Data

The generic handlers of the service runtimes automatically serve read requests from `localized` views. Users see all texts in their preferred language or the fallback language.

↳ *See also **Enabling Draft for Localized Data**.*

For example, given this service definition:

```cds
using { Books } from './books';
service CatalogService {
  entity BooksList as projection on Books { ID, title, price };
  entity BooksDetails as projection on Books;
  entity BooksShort as projection on Books {
    ID, price,
    substr(title, 0, 10) as title : localized String(10),
  };
}
```

`localized.` Helper Views

For each exposed entity in a service definition, and all intermediate views, a corresponding `localized.` entity is created. It has the same query clauses and all annotations, except for the `from` clause being redirected to the underlying entity's `localized.` counterpart. A helper view is only created if the corresponding entity contains at least one element with a `localized` property, or it exposes an association to an entity that is localized. You may need to cast an element if that property is not propagated, for example for expressions such as in `CatalogService.BooksShort` .

```cds
using { localized.Books } from './books_localized';

entity localized.CatalogService.BooksList as
  SELECT from localized.Books { ID, title, price };

entity localized.CatalogService.BooksDetails as
  SELECT from localized.Books;

entity localized.CatalogService.BooksShort as
    SELECT from localized.Books { ID, price,
        substr(title, 0, 10) as title : localized String(10),
    };
```

> **`localized` entities are only generated for SQL**
>
> They are not part of the CSN or exposed via OData.

Read Operations

The generic handlers in the service framework will automatically redirect all incoming read requests to the `localized_` helper views in the SQL database, unless in SAP Fiori draft mode.

The `@cds.localized: false` annotation can be used to explicitly switch off the automatic redirection to the localized views. All incoming requests to an entity annotated with `@cds.localized: false` will directly access the base entity.

```cds
using { Books } from './books';
service CatalogService {
  @cds.localized: false //> direct access to base entity; all fields are non-
```

```
  entity BooksDetails as projection on Books;
}
```

In Node.js applications, for requests with an `$expand` query option on entities annotated with `@cds.localized: false`, the expanded properties are not translated.

```http
// all fields from authors are non-localized defaults if BooksDetails
// is annotated with `@cds.localized: false`
GET /BooksDetails?$expand=authors
```

Write Operations

Since the corresponding text table is linked through composition, you can use deep inserts or upserts to fill in language-specific texts.

```http
POST /Entity HTTP/1.1
Content-Type: application/json

{
  "name": "Some name",
  "description": "Some description",
  "texts": [ {"name": "Ein Name", "description": "Eine Beschreibung", "locale
}
```

If you want to add a language-specific text to an existing entity, perform a `POST` request to the text table of the entity through navigation.

```http
POST /Entity(<entity_key>)/texts HTTP/1.1
Content-Type: application/json

{
  {"name": "Ein Name", "description": "Eine Beschreibung", "locale": "de"}
}
```

Update Operations

To update the language-specific texts of an entity along with the default fallback text, you can perform a deep update as a *PUT* or *PATCH* request to the entity through navigation.

```http
PUT/PATCH /Entity(<entity_key>) HTTP/1.1
Content-Type: application/json

{
  "name": "Some new name",
  "description": "Some new description",
  "texts": [ {"name": "Ein neuer Name", "description": "Eine neue Beschreibun
}
```

To update a single language-specific text field, perform a *PUT* or a *PATCH* request to the entity's text field via navigation.

```http
PUT/PATCH /Entity(<entity_key>)/texts(ID=<entity_key>,locale='<locale>')/<file
Content-Type: application/json

{
  "name": "Ein neuer Name"
}
```

> **Language codes need to follow BCP 47**
>
> Accepted language codes in the *locale* property need to follow the BCP 47 standard but use **underscore** (_) instead of **hyphen** (-), for example *en_GB* .

Delete Operations

To delete a locale's language-specific texts of an entity, perform a *DELETE* request to the entity's texts table through navigation. Specify the entity's key and the locale that you want to delete.

```http
DELETE /Entity(<entity_key>)/texts(ID=<entity_key>,locale='<locale>') HTTP/1.
```

Nested Localized Data

The definition of books has a `currency` element that is effectively an association to the `sap.common.Currencies` code list entity, which in turn has localized texts. Find the respective definitions in the reference docs for `@sap/cds/common`, in the section on Common Code Lists.

Upon unfolding, all associations to other entities with localized texts are automatically redirected as follows:

```cds
entity localized.Currencies as select from Currencies AS c {* /*...*/};
entity localized.Books as select from Books AS p mixin {
  // association is redirected to localized.Currencies
  country : Association to localized.Currencies on country = p.country;
} into {* /*...*/};
```

Given that, nested localized data can be easily read with independent fallback logic:

```sql
SELECT from localized.Books {
  ID, title, descr,
  currency.name as currency
} where title like '%pen%' or currency.name like '%land%'
```

In the result sets for this query, values for `title`, `descr`, as well as the `currency` name are localized.

Adding Initial Data

To add initial data, two *.csv* files are required. The first *.csv* file, for example *Books.csv*, should contain all the data in the default language. The second file, for example *Books_texts.csv* (please note **_texts** in the file name) should contain the translated data in all other languages your application is using.

For example, *Books.csv* can look as follows:

Books.csv

```csv
ID,title,descr,author_ID,stock,price,currency_code,genre_ID
201,Wuthering Heights,"Wuthering Heights, Emily Brontë's only novel ...",101,
207,Jane Eyre,Jane Eyre is a novel by English writer ...,107,11,12.34,GBP,11
251,The Raven,The Raven is a narrative poem by ...,150,333,13.13,USD,16
252,Eleonora,Eleonora is a short story by ...,150,555,14,USD,16
271,Catweazle,Catweazle is a British fantasy ...,170,22,150,JPY,13
...
```

This is the corresponding *Books_texts.csv*:

Books_texts.csv

```csv
ID,locale,title,descr
201,de,Sturmhöhe,Sturmhöhe (Originaltitel: Wuthering Heights) ist der einzige
201,fr,Les Hauts de Hurlevent,Les Hauts de Hurlevent (titre original : Wuther
207,de,Jane Eyre,Jane Eyre. Eine Autobiographie (Originaltitel: Jane Eyre. An
252,de,Eleonora,Eleonora ist eine Erzählung von Edgar Allan Poe. Sie wurde 18
...
```