



# Drupal 8 Multilingual APIs

## Building Multilingual Websites for the Entire World

### Table of Contents

Drupal 8 Multilingual APIs .....	2
Improving Upon Drupal 7.....	3
Drupal 8 APIs.....	4
The Four Drupal 8 Core Pillars for Multilingual .....	5
Language Setup .....	5
Interface.....	9
Content .....	11
Configuration Translation.....	14
Conclusion .....	17



## Drupal 8 Multilingual APIs

Localization strategy is gaining momentum because creating site content in a customer's language personalizes experience and improves engagement. That's leading more organizations to prioritize making their Drupal websites multilingual.

In his State of Drupal presentation at DrupalCon New Orleans, Dries Buytaert listed all the new features that Drupal 8 offers that are contributing to its success:

- Mobile optimization
- Modern PHP standards
- Configuration management
- Better caching
- Web services APIs
- 100 languages
- and 200+ more features

### **Drupal 8 offers better support for multilingual websites**

One big reason for its popularity is that Drupal 8 provides better support for multilingual sites. The Drupal 8 Multilingual Initiative made that a priority for this release. The power of Drupal 8 lies in its ability to assign languages to any content or configuration, along with the improvements in the APIs and user interfaces for localizing and translating your site.

# Improving Upon Drupal 7

**Drupal 8 simplifies multilingual management. It's more deeply integrated, which leads to more automation, more seamless workflows, & better publication management.**

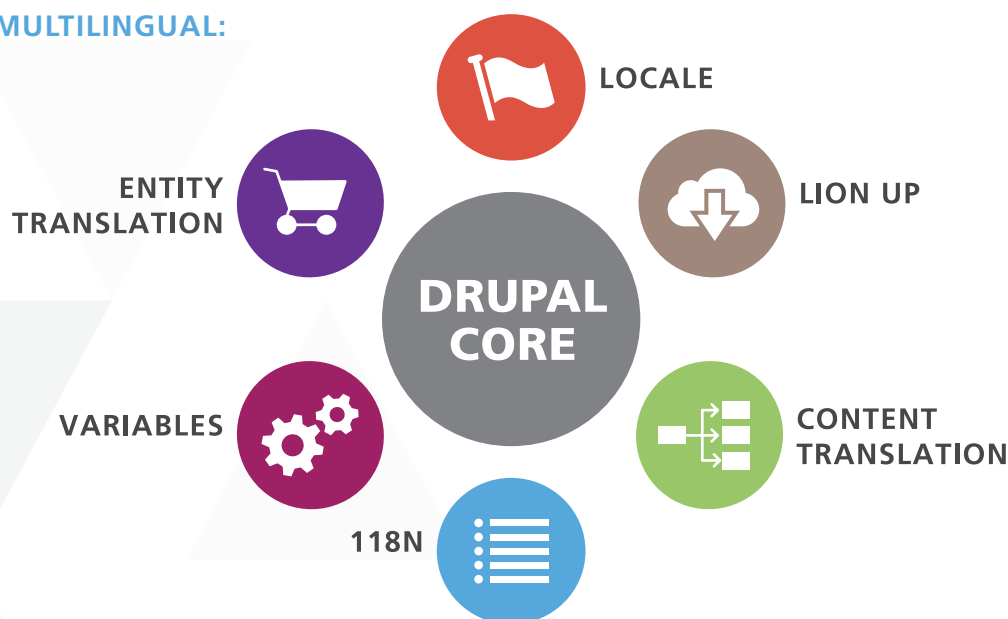
Drupal 7 is a very stable and well-used platform and it supports a vast array of use cases, but it wasn't built with multilingual in mind, so it earned a reputation for being complex for multilingual sites. The core has limited support for localization, so you need a lot of contributed modules to make a site multilingual ready. Locale is in core and you can translate your interface with it, but soon you need other contributed modules like localization update (l10n\_update), the internationalization suite (i18n), the variable module, or entity translation, to name a few.

Creating a truly multilingual site in Drupal 7 has been a real challenge for Drupal developers. It requires navigating several options and building a strategy for rolling out a multilingual site that has proven to be an extremely time-intensive process for developers.

For language support in Drupal 7, if you have several modules and several languages, you need the localization update module for avoiding downloading and importing several PO files for providing interface translations. And if you want to translate a full site, you soon will end up requiring a large number of extra modules, and the potential combination of different settings and installed modules can be hard to build and maintain.

The Drupal community went to work to rebuild language support in Drupal, so that everything in Drupal 8 understood language from the beginning. The result is that most custom or contributed modules or themes don't really have to think about language support – it's already built in or really easy to support.

## DRUPAL 7 MULTILINGUAL:



# Drupal 8 is a great platform to work with, not only because it is multilingual capable out-of-the-box, but also because you can easily expand while maintaining the translatability of your data.

Drupal 8 is a modern platform, and thanks to semantic versioning, it is quickly iterating with the latest technology trends. You won't have to wait several years for a new feature release. Content on Drupal 8 is centered around entities, which simplifies multilingual management. Multilingual is more deeply integrated with core, which paves the way for more automation, more seamless workflows, and better publication management.

Four pillars support building multilingual sites—language setup, interface translation, content translation, and configuration translation—making multilingual sites in Drupal easier than ever. Drupal 8 is a great platform to work with, not only because it is multilingual capable out-of-the-box, but also because you can easily expand while maintaining the translatability of your data.

---

## Drupal 8 APIs

Every Drupal developer who works with contributed or custom modules must be aware of the requirements for providing a solution that is multilingual ready. Drupal themers also need to be aware of what is needed for making their templates translation-ready.

In this guide, we'll look at the most important APIs and best practices to ensure that your project is multilingual-ready.



# The Four Drupal 8 Core Pillars for Multilingual

## Language Setup, Interface Translation, Content Translation & Configuration Translation

### FOUR PILLARS IN DRUPAL 8:



#### LANGUAGE

Base services for all modules dealing with data. Not just multilingual.



#### INTERFACE

Interface translation has built-in update feature, improved usability.



#### CONTENT

Field translation in built-in API for all entities. Content translation module provides user interface.



#### CONFIG

Common configuration system handles blocks, views, field settings. Unified translation.

## Language Setup

In Drupal 7, using the locale module allows you to add languages to your site and show a translated interface. This functionality was split in Drupal 8, so the language module only provides the ability of managing languages, and Drupal core already takes care of dealing with data and which language your data is in. It's useful not only for multilingual sites, but also for monolingual sites that can install in another language, track which language your site data is on, and you can add more languages later.

In Drupal 8, there is already a `LanguageManager` service in core without the need for installing the language module. Calling the `\Drupal::languageManager()` method returns a

### DEALING WITH LANGUAGE

```
\Drupal::languageManager()
```



WITH LANGUAGE  
MODULE ENABLED

```
ConfigurableLanguageManager
```

LanguageManager object—a service implementing LanguageManagerInterface that you can use for tracking and managing your language. If you install the language module, this same call will return a ConfigurableLanguageManager service. They share the same interface, so you can call the same methods on them. You don't need to know if you have a multilingual site or a monolingual site when you are dealing with APIs from your modules.

So for knowing which languages are available on a site, our LanguageManager has the `getLanguages()` method. If you don't have the language module enabled, this call will return three languages: the default which in Drupal 8 is English (EN: English (Default)), and two special languages: UND: Not Specified, and ZXX: Not Applicable. In Drupal 7, we had a special language—UND—which was confusing for users and it was not clear where it should be used. In Drupal 8, to make things clearer, there is an additional one—Not Applicable.

If you don't know the language of the data you are dealing with, you can assign Not Specified. For example, this can be useful if you are importing data from external sites that don't track languages. If the language doesn't really apply, you can use Not Applicable. For example, you may be uploading an image and the image has no text, so it doesn't make sense to assign a language in that case.

In Drupal 8, if you install the language module, you can use more languages.

If you install the language module, languages on your site will be configurable, so you can edit them, add them, or edit their attributes. Configurable languages are config objects, so you can export them and import them between different sites or environments like you would do with any other configuration element in Drupal 8. They are saved like any other config entities in the system and you can export them to a YAML file.

## DEALING WITH LANGUAGE

`->getLanguages()`

UND: NOT SPECIFIED

HU: HUNGARIAN

ZXX: NOT APPLICABLE

IT: ITALIAN



LOCKED



NOT LOCKED

**Language.entity.\$langcode.yml**

If you navigate to the Languages list in the Drupal 8 administration interface, you will find out that the languages Not Specified and Not Applicable are not listed, even when you can pick them in the language selection of content creation, for example. As special languages, they are locked, so you cannot edit them via the interface. When you export your configuration including your configurable languages, you will see that they are exported and have a "locked" property set to TRUE; for your regular languages, this property will be FALSE.



## HOW TO CREATE LANGUAGES

If you have the language module installed, you can use the `ConfigurableLanguage` class, which has a `create` method. If you want to use the 100 or so standard languages that come with Drupal you don't want to provide their details, you only have to provide the language code for creating them, by using the `ConfigurableLanguage::createFromLangcode` method, which will return a `ConfigurableLanguage` instance object with proper default names and writing direction for that language, and then you only need to call `save()` on that instance. But if you want to define your own languages or locales, you can use the constructor of this class providing your custom attribute values and save it the same way.

In Drupal 8, you can perform any CRUD (Create, Read, Update, Delete) operation over your languages using the APIs. As configurable languages are config entities, the API will look exactly the same as the API you would use when dealing with, e.g., node types.

If you need to know which language is currently being used, you can use the `LanguageManager` service's `getCurrentLanguage()` method, and it will return the language object being used. This allows the developer to ignore any details about how language negotiation is configured at this point, making it simpler to deal with multilingual sites.

To sum it up: in Drupal 8, languages can be configurable. You can manage them via the APIs, performing CRUD operations on them. You can also get the language that is being negotiated on the page without knowing the details of how it happened, and you can use the standard languages provided by Drupal core or create your own custom languages.

## DEALING WITH LANGUAGE

```
ConfigurableLanguage::  
createFromLangcode('fr')  
->save()
```

## TRANSLATING THE USER INTERFACE

In Drupal 8, interface translation has a built-in update feature and usability has been improved a lot when compared to Drupal 7. In terms of APIs, you are probably familiar with the `t()`-function used in Drupal 7. It is still there and in short—that's all you need to know about how to translate your user interface. There has been no change externally, but internally, it's very different. There are better ways for making your user interface texts localizable while having a proper testable and object-oriented code.

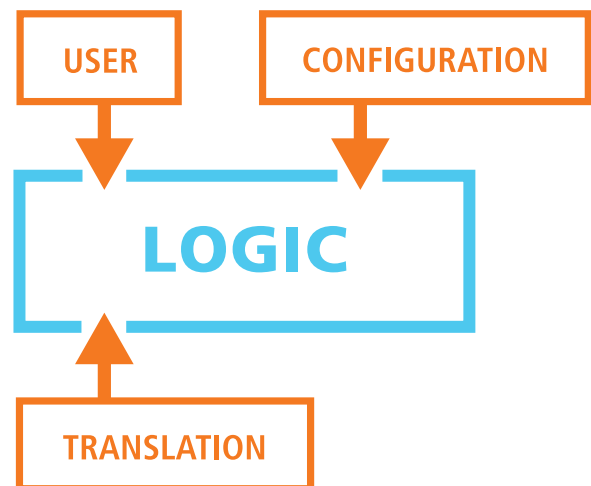
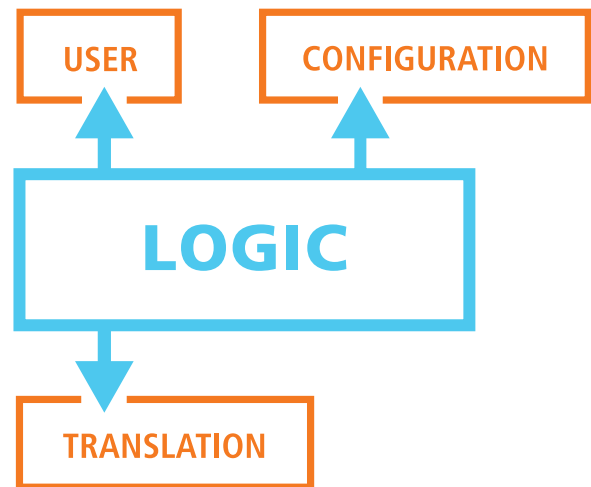
At this point, we need to introduce the concept of Dependency Injection, which is widely used in Drupal 8. While coding Drupal 7 modules, your code will contain logic that may need to know the language preference of a user visiting your site, accessing the site settings for knowing the languages available, and translate your user interface messages. For that, you could call methods for dealing with users, the configuration of the site, and translating texts into the right language.



## DEPENDENCY INJECTION

With the Dependency Injection pattern, instead of calling a globally available method, you can inject and pass through whatever dependencies are needed. You can keep in your services a reference to other service instances your code depends on.

The good thing about Dependency Injection is that you can switch them if you have an interface. When you need to know some data about the user, there is a method for that. You don't need to know the backend system being used. For example, when you are integrating with an external system and it provides the user, you don't need to know about the details. You know that there is a contract and the service will use that contract; you know which methods there will be, and you don't need to know what they are doing.





## Interface

You don't need to write different code based on whether there is a translation system in place or not. The LanguageManager can be different if you are on an English only site or if you have the language module installed.

There needed to be a way to provide this translation service; a way that it can be switched. For that, you will use this t()-method, and the switchability is hidden behind that. If you are creating a controller or a form, you are probably extending from base classes that are available in Drupal 8 core, like FormBuilder. It will already have a t()-method that you can use. It will work with the proper translation service in turn.

### INTERFACE LANGUAGE

```
format_plural(...)
$this->formatPlural(...)
```

If you are creating your own class that doesn't extend from a base class that already has translation, you can apply the use StringTranslationTrait statement and this trait will automatically provide you the t()-method and the setter method for the translation service.

### INTERFACE LANGUAGE

```
class Foo {
  use StringTranslationTrait;
  ...
}
```

### INTERFACE LANGUAGE

```
$this->t('English text')
->getOption('langcode');
```

This t()-function is internally quite different from Drupal 7. When you call t() in Drupal 7, you get a string translated just in time. In Drupal 8, the translation itself is deferred and you are actually getting an object. This object is a TranslatableMarkup object that wraps the original string and you can call methods on it. You can retrieve the langcode option (->getOption) used to set up the object and several further methods are available to gather data about this translation object. This way, text can be translated later, before the output is actually created.

In Drupal 7, if you had a t() call to return a translation, if a form alter removed it later, we are not really outputting that string. But Drupal already spent time translating it. Here in Drupal 8, if it's not going to be rendered, you are not really translating it, so at the end, you may even make the performance of your site better. The translation is only done when you are going to render the page.

In addition, format\_plural() from Drupal 7 is now gone in Drupal 8. There is \$this->formatPlural(...) for that and it will work a similar way. You get an object that you can actually alter to do whatever you need.

## JAVASCRIPT API

If you are doing JavaScript, it didn't change much from Drupal 7 for translation. You still have the Drupal t() method and the Drupal formatPlural() method.

```
Drupal.t(...)
Drupal.formatPlural()
```

## TWIG TEMPLATES

For templates, Drupal 8 uses Twig. In the PHP templates of Drupal 7, you would call t() on your translatable strings, but in Drupal 8, that is not available anymore. There are two different methods for translating your strings in your templates. The first one is the trans filter. If you have a string, you can pass it through the trans filter, and it will provide the proper translation.

Like in the t() function in Drupal 7, you can pass the context. If you want to request a translation into a specific language other than the interface language, you need to specify that language. It is easiest to do that with the trans block. All the text between the trans and endtrans tags will be used for calling the t() function internally. You can also have variables similar to those in your PHP code.

### INPUT – SEARCH.HTML.TWIG

```
<input {{ attributes }}
  placeholder="{{ 'Search'|trans }}" />
```

### NODE.HTML.TWIG

```
<div{{ author_attributes }}>
  {% trans %}
    Submitted by {{ author_name }} on {{ date }}
  {% endtrans %}
</div>
```

## \*LINKS.MENU.YML FILES

Another change is around menu items. In Drupal 7, you had your hook\_menu() where you had your route, your menu links, your tasks—everything was in there. Now in Drupal 8, it has been split. These are all located in YAML files. They are separated, so you have your menu links in one place, and your routing is in a different place. It's a more flexible system.

In Drupal 7, you couldn't call the t() function for your title and description in your hook\_menu(). This lets you cache your links and translate them later to other languages.

### LOCALE.LINKS.MENU.YML

```
local.translate_page:
  title: 'User interface translation'
  description: 'Configure the import ...'
  route_name: locale.translate_page
  parent: system.admin_config_regional
  weight: 15
```

```
locale.translate_status:
  title: 'Available translation updates'
  route_name: locale.translate_status
  description: 'Get a status report ...'
  parent: system.admin_reports
```

In Drupal 8, it's pretty much the same. They are YAML files, so you cannot call any function on them, but there are some special keys (in this case title and description) that Drupal 8 will take care of translating. The potx.module will take care of extracting them and making them available for translation on localize.drupal.org as well.

In summary, the interface translation module provides the ability to translate from English into any other language.



# Content



The content translation module allows you to translate all of your content, not only nodes and taxonomy terms, but also any kind of entity that you can have on your site. If you have Drupal commerce, it can translate your products and provides the user interface for actually translating them.

## DEFINING TRANSLATABLE CONTENT

The content translation module is not based on copies of your nodes or your entities. Drupal 8 has clever objects that know about them. If you are creating a `ContentEntityType`, Drupal 8 will make it easier to translate them.

### NODE.PHP (SNIPPET)

```
/**
 * Defines the node entity class.
 * @ContentEntityType(
 *   id = "noted",
 *   label = @Translation("File"),
 *   translatable = TRUE,
 *   entity_keys = {
 *     "id" = "nid",
 *     "label" = "title",
 *     "langcode" = "langcode",
 *     "label" = "title",
 *   }
 * )
```

For creating a `ContentEntityType`, you need to create a class, in a special namespace in your module and add an annotation, which describes the metadata. This defines your content entity.

In annotations, you can use `@Translation()` so your label will be extracted and you can translate your labels on your UI. For making the content possible to configure as translatable, you only need to specify `translatable=TRUE`. If you want to translate your entities, you need to know which language they are

in. You need to define the `langcode` in the entity keys. By doing this, you are telling Drupal that these entities may be configured as translatable. That doesn't mean that all of your nodes can be translated, it means that the user can go to the content translation configuration interface and set up if they want to translate them or not.

## DEFINE THE LANGCODE BASE FIELD

In Drupal 8 as in Drupal 7, we have two different kind of fields: the first one are the fields that are defined as part of the entity, and the second are the fields that you can configure (add and remove) on the interface of the entity.

If you are starting from `ContentEntityTypeBase`, when creating the content entity (which you should do), you have to define the base field definitions. It provides the base properties that every entity of this kind should have.

### CONTENTENTITYBASE.PHP (SNIPPET)

```
function baseFieldDefinitions($entity_type) {
  // ...
  if ($entity_type->hasKey('langcode')) {
    $fields[$entity_type->getKey('langcode')] =
      BaseFieldDefinition::create('language')
        ->setLabel(new TranslatableMarkup('Language'))
        ->setDisplayOptions('view', [
          'type' => 'hidden',
        ])
  }
}
```

## DEFINE TRANSLATABLE BASE FIELDS

It already has the langcode for creating the field for you if you defined the langcode key in the entity keys.

For your additional base fields, ensure that you call the `parent::baseFieldDefinitions()` method to get the common fields and then add your own fields. For your own fields, you also have a method for making them translatable, or rather as a field possible to configure for translation.

### NODE.PHP (SNIPPET)

```
function baseFieldDefinitions($entity_type) {
    $fields = parent::baseFieldDefinitions($entity_type);
    // ...
    $fields['title'] = BaseFieldDefinition::create('string')
        ->setLabel(t('Title'))
        ->setRequired(TRUE)
        ->setTranslatable(TRUE);
}
```

If you call `setTranslatable(TRUE)` method on a base field, Drupal will know about its translatability. The user can also configure it differently then. If you make an entity translatable, any field that has this flag will be translatable by default.

## FIELD TYPES THEMSELVES

If you are creating your own field types, how do we make them translatable? There is nothing you need to do. They will be translatable automatically. Drupal 8 knows how to make them translatable, but for some kind of fields you may want to make more granular options.

## MULTICOLUMN FIELDS

To define an image, you will have a reference to a file entity. You will need to know the width and the height of the image, then you can define the alternative text and the title of the image.

Like images, other fields may be composed of different properties. You may also want to provide different translatability options for them. In the case of an image, you may not want to translate the image itself, but you do want to translate the alternative text in the title. To make this possible, you can create groups of

### MULTICOLUMN FIELD SNIPPET

```
* @FieldType(
*   id = "image",
*   column_group = {
*     "file" = {
*       "label" = @Translation("File"),
*       "columns" = {
*         "target_id", "width", "height"
*       },
*     },
*     "alt" = {
*       "label" = @Translation("Alt"),
*       "translatable" = TRUE
*     },
*   }
*)
```

data columns. In this example, a file group is created and the columns of this group—"target\_id", "width", "height"—are part of this group. An alternative text group is also created where we are saying "translatable"=TRUE.

This provides sane defaults for when you are configuring your site, so when you check translating an image file and an image field in an entity, you will see by default that you can provide different configurations for the file itself, for alternative text and for the title (not represented here for brevity). And by default, the alternative text and the title will be translatable, but not the file itself. You can make it translatable if you want.

## ENTITY LANGUAGE API

The Entity Language API is quite straightforward. You have a node class for dealing with your nodes. You have the same kind for any entity you create. You have a `load()` method on it, and a `getTranslation()` method to retrieve specific translations.

### ENTITY LANGUAGE API

```

$node = Node::load(42);
↓
$node = $node
->getTranslation('hu');
↓
$node = $entityRepository
->getTranslationFromContext($node);

```

If you call the `getTranslation()` method, you can request which translation you want to work with and you will get a reference to the translation. It's the same node, so you can do any operations you want with it and then save it or edit the fields as you would with nodes in any other situation. Language here is quite easy to use and translations from these nodes are quite easy to access.

You may not know which translation you want. If all you know is you want a translation appropriate for this request, use the entity repository. Call the `getTranslationFromContext()` method and pass the node. What you get back is a node in the current negotiated language.

There are a lot useful methods that we can use in our entities:

- `$node->getUntranslated()`—gets us the source translation
- `$node->language()`—the language of the current loaded translation
- `$node->getTranslationLanguages()`—the list of the language objects for translations of the entity
- `$node->hasTranslation('hu')`—is there a translation for a specific langcode
- `$node->addTranslation('hu')`—for creating it
- `$node->removeTranslation('hu')`—for removing it

The best thing is this does not only apply to nodes but also other content entity types, such as users, comments, taxonomy terms, etc.

### ENTITY LANGUAGE API

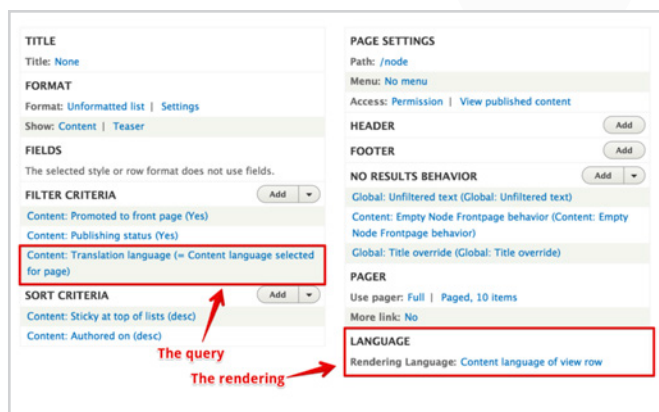
```

$node->getUntranslated()
$node->language()
$node->getTranslationLanguage()
$node->hasTranslation('hu')
$node->addTranslation('hu')
$node->removeTranslation('hu')

```

## VIEWS

Views has two key purposes: Creating queries on the system, and then render it. You can see this separation in the first two columns in the Views UI. Language is integrated in both.



You can filter by language. You can filter your views by a specific language, or you can filter by the language that was negotiated for the page. On the rendering side, you can pick which languages you want to render the found entities in. With Views, you can say give me all the nodes that are in English and then render the German translations. These are the two different ways to deal with language in Views. It's quite powerful; you may not need the PHP APIs at all.

Content translation provides the ability to translate from any language to any other language. Your entities are intelligent objects so you can actually call the methods on them—for getting the translations, for dealing with translations, editing translations, etc.



## Configuration Translation

We have two kinds of configuration: configuration objects for global settings and configuration entities for things that can have multiple instances, like configurable languages. Configuration is stored in YAML files with a langcode property.

For making our config translatable, you need to provide a schema. It was introduced in Drupal 8 for knowing how to translate or deal with languages in your configuration, but it is now used for a lot of other things. Drupal defines some base types such as `config_object` which has a `langcode`. Global configuration is usually typed as `config_object`.

There is another key data type for translatability, which is `text`. It is a string that is translatable. When these data types are properly used, a big advantage is Drupal knows to type-cast data to the right types, such as integers to numbers.

### CORE.DATA\_TYPES.SCHEMA.YML

```
config_object:
  type: mapping
  mapping
    langcode: ←
      type: string
      label: 'Language code'
    ...

text:
  type: string
  label: 'Text'
  translatable: true ←
```

To create your own schemas, you need to create a schema YAML file and then define the types of your configuration elements. Here we are defining the system maintenance configuration. We are saying that it is a config object. This means that it will have a langcode and you can define its further attributes. We know it's a message of type text, so we know it will also be a translatable string.

#### CONFIG/SCHEMA/SYSTEM.SCHEMA.YML

```
system.maintenance:
  type: config_object LANGCODE
  label: 'Maintenance mode'
  mapping:
    message:
      type: text TRANSLATABLE STRING
      label: 'Message to display...'
```

Here is the message and langcode. When it's translated in configuration storage, there will be a language folder for each language with a langcode as a name. There will also be a file with the same name. Only the translatable elements may show up in these files.

#### SYSTEM.MAINTENANCE.YML

```
message: '@site is currently under
maintenance. We should be back
shortly. Thank you for your patience.'
langcode: en
```

#### LANGUAGES/HU/SYSTEM....YML

```
message: '@site karbantartás alatt all...'
```

#### LANGUAGES/IT/SYSTEM....YML

```
message: '@site ...'
```

#### CONFIGURATION API

```
$manager = \Drupal::languageManager();
$hu = $manager->getLanguage('hu');

$original = $manager-
>getConfigOverrideLanguage();
$manager->setConfigOverrideLanguage($hu);

$config = \Drupal::config('system.maintenance');
// ...

$manager->setConfigOverrideLanguage($original);
```

To access the configuration of your site, use the Drupal config service. You pass the name of the configuration object that you want to retrieve and then you can access the properties with the `->get()` method so we can get the message from the system maintenance object with `->get('message')`.

If you have language overrides, they will apply as appropriate. If you have a settings PHP override that will also apply.

Configuration with overrides applied is the most common for settings used at runtime. Drupal uses the negotiated language for the page to initialize overrides. If you know what language overrides you want to apply instead, you need to use the Language Manager and you can call `>getConfigOverrideLanguage()` for storing the language Drupal is currently using. Then you can set a different one with `setConfigOverrideLanguage()`.



This can be quite useful if you are sending emails. An admin may be visiting your site—your admin interface in English—but you want the users they are sending emails to to get emails in the language they prefer. You can get the language from the user object, set the config override language, then use the configuration with the appropriate language overrides applied for sending the email. Then you can set it back to the original language Drupal was using before.

We have three different ways for dealing with languages with Configuration.

### CONFIGURATION API

```
\Drupal::config('system.maintenance');
```

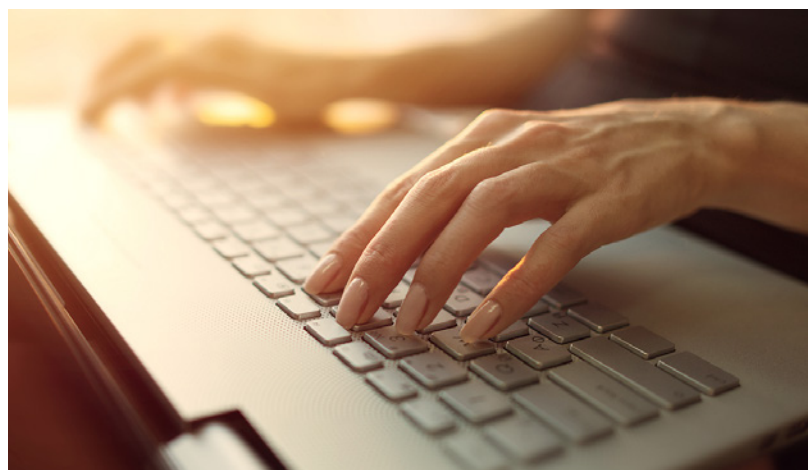
**OVERRIDES APPLY  
AS APPROPRIATE**

```
\Drupal::configFactory()-  
>getEditable('system.maintenance');
```

**NO OVERRIDES  
APPLY**

```
\Drupal::languageManager()  
->getLanguageConfigOverride  
('hu', 'system.maintenance')  
->set('message', 'Karbantartás...')  
->save();
```

**THE OVERRIDE  
ITSELF**



`\Drupal::config('system.maintenance')` provides your configuration with every override applied. If there are competing overrides for a value, Drupal's priority system is used to decide which override key prevails. `\Drupal::configFactory() >getEditable` is for getting the current configuration in the original raw format, so absolutely no overrides applied. This is useful for actually making configuration changes. Finally `\Drupal::languageManager()` has a method for getting a concrete config override itself: `->getLanguageConfigOverride()`. So you can work with the overrides themselves, the raw config without overrides, or we can ask the system to apply concrete overrides.

The config translation module allows you to translate your configurations from any language to any other language. The difference with the content translation module where you have intelligent objects, here you mostly have overrides. The APIs are not as beautiful as the content translation ones, but they are also a lot more flexible to allow for arbitrary variance in configuration not just by language.

## Conclusion



These APIs were the result of the work of over 1,600 Drupal contributors. It was an amazing collective effort to make multilingual sites work better in Drupal 8. If you are interested in contributing, please join one of the Contribution Sprints.

Even if your project is not immediately intended to be multilingual, having a multilingual-capable module, theme, or distribution makes your solution appealing to a much broader audience and is likely to provide value to global users.

### MAIN OFFICE

3400 North Ashton Blvd, #150  
Lehi, UT 84043  
+1 801 331 7777  
[sales@lingotek.com](mailto:sales@lingotek.com)

### WASHINGTON DC OFFICE

12020 Sunrise Valley Dr #100  
Reston, VA 20191  
[sales@lingotek.com](mailto:sales@lingotek.com)

### UNITED KINGDOM OFFICE

1 Bell Street,  
Maidenhead,  
Berkshire, SL6 1BU  
United Kingdom  
+44 (0)1628 421525  
[sales@lingotek.com](mailto:sales@lingotek.com)

[www.lingotek.com](http://www.lingotek.com)

# Contributors



**GÁBOR HOJTSY**

Gábor Hojtsy

Drupal 8 Multilingual Initiative lead

**Acquia**  
THINK AHEAD.



**PENYASKITO**

Christian López Espínola

Drupal 8 Multilingual Initiative contributor

 **Lingotek**

