Если вы видите что-то необычное, просто сообщите мне.

I18n in Go: Managing Translations

Recently I've been building a fully internationalized (i18n) and localized (l10n) web application for the first time with Go's golang.org/x/text packages. I've found that the packages and tools that live under golang.org/x/text are really effective and well designed, although it's been a bit of a challenge to figure out how to put it all together in a real application.

In this tutorial I want to explain how you can use golang.org/x/text packages to manage translations in your application. Specifically:

How to use the golang.org/x/text/language and golang.org/x/text/message packages to print translated messages from your Go code. How to use the gotext tool to automatically extract messages for translation from your code into JSON files. How to use gotext to parse translated JSON files and create a catalog containing translated messages. How to manage variables in messages and provided pluralized versions of translations. Note: Just in case you're not already aware, the packages that live under golang.org/x are part of the official Go Project but outside the main Go standard library tree. They are held to looser standards that the standard library packages, which means they aren't subject to the Go compatibility promise (i.e. their APIs might change), and documentation may not always be complete. What we'll be building To help put this into context, we're going to create a simple pre-launch website for an imaginary online bookstore. We'll start off slowly and build up the code step-by-step.

Our application will have just a single home page, and we'll localize the page content based on a locale identifier at the start of the URL path. We'll set up our application to support three different locales: the United Kingdom, Germany, and the French-speaking part of Switzerland.

URL Localized for localhost:4018/en-gb United Kingdom localhost:4018/de-de Germany localhost:4018/fr-ch Switzerland (French-speaking) We're going to follow a common convention and use BCP 47 language tags as the locale identifier in our URLs. Simplifying things hugely for the

sake of this tutorial, BCP 47 language tags typically take the format {language}-{region}. The language part is a ISO 639-1 code and the region is a two-letter country code from ISO_3166-1. It's conventional to uppercase the region (like en-GB), but BCP 47 tags are technically case-insensitive and it's OK for us to use all-lowercase versions in our URLs.

Scaffolding a web application If you'd like to follow along with the application build, go ahead and run the following commands to setup a new project directory.

$ mkdir bookstore $ cd bookstore $ go mod init bookstore.example.com go: creating new go.mod: module bookstore.example.com At this point, you should have a go.mod file in the root of the project directory with the module path bookstore.example.com.

Next create a new cmd/www directory to hold the code for the bookstore web application, and add main.go and handlers.go files like so:

$ mkdir -p cmd/www $ touch cmd/www/main.go cmd/www/handlers.go Your project directory should now look like this:

. ├── cmd │ └── www │ ├── handlers.go │ └── main.go └── go.mod Let's begin in the cmd/www/main.go file and add the code to declare our application routes and start a HTTP server.

Because our application URL paths will always use a (dynamic) locale as a prefix — like /en-gb/bestsellers or /fr-ch/bestsellers — it's simplest if our application uses a third-party router which supports dynamic values in URL path segments. I'm going to use pat, but feel free to use an alternative like chi or gorilla/mux if you prefer.

Note: If you're not sure which router to use in your project, you might like to take a look at my comparison of Go routers blog post. OK, open up the main.go file and add the following code:

File: cmd/www/main.go package main

import ("log" "net/http"

```
 "github.com/bmizerany/pat"
```

)

func main() { // Initialize a router and add the path and handler for the homepage. mux := pat.New() mux.Get("/:locale", http.HandlerFunc(handleHome))

```
// Start the HTTP server using the router.
log.Print("starting server on :4018...")
err := http.ListenAndServe(":4018", mux)
log.Fatal(err)
```

} Then in the cmd/www/handlers.go file, add a handleHome() function which extracts the locale identifer from the URL path and echoes it in the HTTP response.

File: cmd/www/handlers.go package main

import ("fmt" "net/http")

func handleHome(w http.ResponseWriter, r *http.Request) { // Extract the locale from the URL path. This line of code is likely to
// be different for you if you are using an alternative router. locale := r.URL.Query().Get(":locale")

```
// If the locale matches one of our supported values, echo the locale
// in the response. Otherwise send a 404 Not Found response.
switch locale {
case "en-gb", "de-de", "fr-ch":
    fmt.Fprintf(w, "The locale is %s\n", locale)
default:
    http.NotFound(w, r)
}
```

} Once that's done, run go mod tidy to tidy your go.mod file and download any necessary dependencies, and then run the web application.

$ go mod tidy go: finding module for package github.com/bmizerany/pat go: found github.com/bmizerany/pat in github.com/bmizerany/pat v0.0.0-20210406213842-e4b6760bdd6f

$ go run ./cmd/www/ 2021/08/21 21:22:57 starting server on :4018... If you make some requests to the application using curl, you should find that the appropriate locale is echoed back to you like so:

$ curl localhost:4018/en-gb The locale is en-gb

$ curl localhost:4018/de-de The locale is de-de

$ curl localhost:4018/fr-ch The locale is fr-ch

$ curl localhost:4018/da-DK 404 page not found Extracting and translating text content Now that we've laid the groundwork for our web application, let's get into the core of this tutorial and update the handleHome() function so that it renders a "Welcome!" message translated for the specific locale.

In this project we'll use British English (en-GB) as the default 'source' or 'base' language in our application, but we'll want to render a translated version of the welcome message in German and French for the other locales.

To do this, we'll need to import the golang.org/x/text/language and golang.org/x/text/message packages and update our handleHome() function to do the following two things:

Construct a language.Tag which identifies the target language that we want to translate the message in to. The language package contains some pre-defined tags for common language variants, but I find that it's easier to use the language.MustParse() function to create a tag. This let's you create a language.Tag for any valid BCP 47 value, like language.MustParse("fr-CH"). Once you have a language tag, you can use the message.NewPrinter() function to create a message.Printer instance that prints out messages in that specific language. If you're following along, please go ahead and update your cmd/www/handlers.go file to contain the following code:

File: cmd/www/handlers.go package main

import ("net/http"

```
"golang.org/x/text/language"
"golang.org/x/text/message"
```

)

func handleHome(w http.ResponseWriter, r *http.Request) { locale := r.URL.Query().Get(":locale")

```
// Declare variable to hold the target language tag.
var lang language.Tag
```

```
// Use language.MustParse() to assign the appropriate language tag
// for the locale.
switch locale {
case "en-gb":
    lang = language.MustParse("en-GB")
case "de-de":
    lang = language.MustParse("de-DE")
case "fr-ch":
    lang = language.MustParse("fr-CH")
default:
    http.NotFound(w, r)
    return
}

// Initialize a message.Printer which uses the target language.
p := message.NewPrinter(lang)
// Print the welcome message translated into the target language.
p.Fprintf(w, "Welcome!\n")
```

} Again, run go mod tidy to download the necessary dependencies...

$ go mod tidy go: finding module for package golang.org/x/text/message go: finding module for package golang.org/x/text/language go: downloading golang.org/x/text v0.3.7 go: found golang.org/x/text/language in golang.org/x/text v0.3.7 go: found golang.org/x/text/message in golang.org/x/text v0.3.7 And then run the application:

$ go run ./cmd/www/ 2021/08/21 21:33:52 starting server on :4018... When you make a request to any of the supported URLs, you should now see the (untranslated) welcome message like this:

$ curl localhost:4018/en-gb Welcome!

$ curl localhost:4018/de-de Welcome!

$ curl localhost:4018/fr-ch Welcome! So in all cases we're seeing the "Welcome!" message in our en-GB source language. That's because we still need to provide Go's message package with the actual translations that we want to use. Without the actual translations, it falls back to displaying the message in the source language.

There are a number of ways to provide Go's message package with translations, but for most non-trivial applications it's probably sensible to use some automated tooling to help you manage the task. Fortunately, Go provides the gotext tool to assist with this.

Note: The gotext tool we're using is the one from golang.org/x/text/cmd/gotext. It shouldn't be confused with the github.com/leonelquinteros/gotext package (which is designed to work with GNU gettext utilities and PO/MO files). If you're following along, please use go install to install the gotext executable on your machine:

$ go install golang.org/x/text/cmd/gotext@latest All being well, the tool should be installed to your $GOBIN directory on your system path and you can run it like so:

$ which gotext /home/alex/go/bin/gotext

$ gotext gotext is a tool for managing text in Go source code.

Usage:

```
    gotext command [arguments]
```

The commands are:

```
    update      merge translations and generate catalog
    extract     extracts strings to be translated from code
    rewrite     rewrites fmt functions to use a message Printer
    generate    generates code to insert translated messages
```

Use "gotext help [command]" for more information about a command.

Additional help topics:

Use "gotext help [topic]" for more information about that topic. I really like the gotext tool — it's functionality is excellent — but there are a couple of important things to point out before we carry on.

The first thing is that go text is designed to work in conjunction with go generate, not as a standalone command-line tool. You can run it as a standalone tool, but weird things happen and it's a lot smoother if you use it in the way it's intended.

The other thing is that documentation and help functionality is basically non-existent. The best guidance on how to use it are the examples in the repository and, probably, this article that you're reading right now. There is an open issue about the lack of help functionality, and hopefully this is something that will improve in the future.

In this tutorial, we're going to store the all the code relating to translations in a new internal/translations package. We could keep all the translation code for our web application under cmd/www instead, but in my (limited) experience I've found that using a separate internal/translations package is better. It helps separate concerns and also makes it possible to reuse the same translations across different applications in the same project. YMMV.

If you're following along, go ahead and create that new directory and a translations.go file like so:

$ mkdir -p internal/translations $ touch internal/translations/translations.go At this point, your project structure should look like this:

. ├── cmd │ └── www │ ├── handlers.go │ └── main.go ├── go.mod ├── go.sum └── internal └── translations └── translations.go Next, let's open up the internal/translations/translations.go file and add a go generate command which uses gotext to extract the messages for translation from our application.

File: internal/translations/translations.go package translations

//go:generate gotext -srclang=en-GB update -out=catalog.go -lang=en-GB,de-DE,fr-CH bookstore.example.com/cmd/www There's a lot going on in this command, so let's quickly break it down.

The -srclang flag contains the BCP 47 tag for the source (or 'base') language that we are using in the application. In our case, the source language is en-GB. update is thegotext function that we want to execute. As well as update there are extract, rewrite and generate functions, but in the translation workflow for a web application the only one you actually need is update. The -out flag contains the path that you want the message catalog to be output to. This path should be relative to the file containing the go generate command. In our case, we've set the value to catalog.go, which means that the message catalog will be output to a new internal/translations/catalog.go file. We'll talk more about message catalogs and explain what they are shortly. The -lang flag contains a comma-separated list of the BCP 47 tags that you want to create translations for. You don't need

to include the source language here, but (as we'll demonstrate later in this article) it can be helpful for dealing with pluralization of text content. Lastly, we have the fully-qualified module path for the package(s) that you want to create translations for (in this case bookstore.example.com/cmd/www). You can list multiple packages if necessary, separated by a whitespace character. When we execute this go generate command, gotext will walk the code for the cmd/www application and look for all calls to a message.Printer†. It then extracts the relevant message strings and outputs them to some JSON files for translation.

† Important: It's critical to note when gotext walks your code it actually only looks for calls to message.Printer.Printf(), Fprintf() and Sprintf() — basically the three methods that end with an f. It ignores all other methods such as Sprint() or Println(). You can see this behavior in the gotext implementation here. OK, let's put this into action and call go generate on our translations.go file. In turn, this will execute the gotext command that we included at the top of that file.

$ go generate ./internal/translations/translations.go de-DE: Missing entry for "Welcome!". fr-CH: Missing entry for "Welcome!". Cool, this looks like we're getting somewhere. We've got some useful feedback to indicate that we are missing the necessary German and French translations for our "Welcome!" message.

If you take a look at the directory structure for your project, it should now look like this:

. ├── cmd │ └── www │ ├── handlers.go │ └── main.go ├── go.mod ├── go.sum └── internal └── translations ├── catalog.go ├── locales │ ├── de-DE │ │ └── out.gotext.json │ ├── en-GB │ │ └── out.gotext.json │ └── fr-CH │ └── out.gotext.json └── translations.go We can see that the go generate command has automatically generated an internal/translations/catalog.go file for us (which we'll look at in a minute), and a locales folder containing out.gotext.json files for each of our target languages.

Let's take a look at the internal/translations/locales/de-DE/out.gotext.json file:

File: internal/translations/locales/de-DE/out.gotext.json { "language": "de-DE", "messages": [{ "id": "Welcome!", "message": "Welcome!", "translation": "" }] } In this JSON file, the relevant BCP 47 language tag is defined at the top of the file, followed by a JSON array of the messages which require translation. The message value is the text for translation in the source language, and the (currently empty) translation value is where we should enter appropriate German translation.

It's important to emphasize that you don't edit this file in place. Instead, the workflow for adding a translation goes like this:

You generate the out.gotext.json files containing the messages which need to be translated (which we've just done). You send these files to a translator, who edits the JSON to include the necessary translations. They then send the updated files back to you. You then save these updated files with the name messages.gotext.json in the folder for the appropriate language. For demonstration purposes, let's quickly simulate this workflow by copying the out.gotext.json files to messages.gotext.json files, and updating them to include the translated messages like so:

$ cp internal/translations/locales/de-DE/out.gotext.json internal/translations/locales/de-DE/messages.gotext.json $ cp internal/translations/locales/fr-CH/out.gotext.json internal/translations/locales/fr-CH/messages.gotext.json File: internal/translations/locales/de-DE/messages.gotext.json { "language": "de-DE", "messages": [{ "id": "Welcome!", "message": "Welcome!", "translation": "Willkommen!" }] } File: internal/translations/locales/fr-CH/messages.gotext.json { "language": "fr-CH", "messages": [{ "id": "Welcome!", "message": "Welcome!", "translation": "Bienvenue !" }] } If you like, you can also take a look at the out.gotext.json file for our en-GB source language. You'll see that the translation value for the message has been auto-filled for us.

File: internal/translations/locales/en-GB/messages.gotext.json { "language": "en-GB", "messages": [{ "id": "Welcome!", "message": "Welcome!", "translation": "Welcome!", "translatorComment": "Copied from source.", "fuzzy": true }] } The next step is to run our go generate command again. This time, it should execute without any warning messages about missing translations.

$ go generate ./internal/translations/translations.go Now it's a good time to take a look at the internal/translations/catalog.go file, which is automatically generated for us by the gotext update command. This file contains a message catalog, which is — very roughly speaking — a mapping of messages and their relevant translations for each target language.

Let's take a quick look inside the file:

File: internal/translations/catalog.go // Code generated by running "go generate" in golang.org/x/text. DO NOT EDIT.

package translations

```
import ( "golang.org/x/text/language" "golang.org/x/text/message"
"golang.org/x/text/message/catalog" )

type dictionary struct { index []uint32 data string }

func (d *dictionary) Lookup(key string) (data string, ok bool) { p, ok := messageKeyToIndex[key] if
!ok { return "", false } start, end := d.index[p], d.index[p+1] if start == end { return "", false }
return d.data[start:end], true }

func init() { dict := map[string]catalog.Dictionary{ "de_DE": &dictionary{index: de_DEIndex, data:
de_DEData}, "en_GB": &dictionary{index: en_GBIndex, data: en_GBData}, "fr_CH":
&dictionary{index: fr_CHIndex, data: fr_CHData}, } fallback := language.MustParse("en-GB") cat,
err := catalog.NewFromMap(dict, catalog.Fallback(fallback)) if err != nil { panic(err) }
message.DefaultCatalog = cat }

var messageKeyToIndex = map[string]int{ "Welcome!\n": 0, }

var de_DEIndex = []uint32{ // 2 elements 0x00000000, 0x00000011, } // Size: 32 bytes

const de_DEData string = "\x04\x00\x01\n\f\x02Willkommen!"

var en_GBIndex = []uint32{ // 2 elements 0x00000000, 0x0000000e, } // Size: 32 bytes

const en_GBData string = "\x04\x00\x01\n\t\x02Welcome!"

var fr_CHIndex = []uint32{ // 2 elements 0x00000000, 0x00000010, } // Size: 32 bytes

const fr_CHData string = "\x04\x00\x01\n\v\x02Bienvenue !"

// Total table size 143 bytes (0KiB); checksum: 385F6E56 I don't want to dwell on the details here,
because it's OK for use to treat this file as something of a 'black box', and — as warned by the
comment at the top of the file — we shouldn't make any changes to it directly.
```

But the most important thing to point out is that this file contains an init() function which, when called, initializes a new message catalog containing all our translations and mappings. It then sets this as the default message catalog by assigning it to the message.DefaultCatalog global variable.

When we call one of the message.Printer functions, the printer will lookup the relevant translation from the default message catalog for printing. This is really nice, because it means that all our translations are stored in memory at runtime, and any lookups are very fast and efficient.

So, if we take a step back for a moment, we can see that the gotext update command that we're using with go generate actually does two things. One — it walks the code in our cmd/www application and extracts the necessary strings for translation into the out.gotext.json files; and two — it also parses any messages.gotext.json files (if present) and updates the message catalog accordingly.

The final step in getting this working is to import the internal/translations package in our cmd/www/handlers.go file. This will ensure that the init() function in internal/translations/translations.go is called, and the default message catalog is updated to be the one containing our translations. Because we won't actually be referencing anything in the internal/translations package directly, we'll need to alias the import path to the blank identifer _ to prevent the Go compiler from complaining.

Go ahead and do that now:

File: cmd/www/handlers.go package main

import ("net/http"

```
// Import the internal/translations package, so that its init()
// function is called.
_ "bookstore.example.com/internal/translations"

"golang.org/x/text/language"
"golang.org/x/text/message"
```

)

func handleHome(w http.ResponseWriter, r *http.Request) { locale := r.URL.Query().Get(":locale")

```
var lang language.Tag

switch locale {
case "en-gb":
```

```
    lang = language.MustParse("en-GB")
case "de-de":
    lang = language.MustParse("de-DE")
case "fr-ch":
    lang = language.MustParse("fr-CH")
default:
    http.NotFound(w, r)
    return
}

p := message.NewPrinter(lang)
p.Fprintf(w, "Welcome!\n")
```

} Alright, let's try this out! When your restart the application and try making some requests, you should now see the "Welcome!" message translated into the appropriate language.

$ curl localhost:4018/en-GB Welcome!

$ curl localhost:4018/de-de Willkommen!

$ curl localhost:4018/fr-ch Bienvenue ! Using variables in translations Now that we've got the basic translations working in our application, let's move on to something a bit more advanced and look at how to manage translations with interpolated variables in them.

To demonstrate, we'll update the HTTP response from our handleHome() function to include a "{N} books available" line, where {N} is an integer containing the number of books in our imaginary bookstore.

File: cmd/www/handlers.go package main

...

func handleHome(w http.ResponseWriter, r *http.Request) { locale := r.URL.Query().Get(":locale")

```
var lang language.Tag

switch locale {
case "en-gb":
    lang = language.MustParse("en-GB")
```

```
case "de-de":
    lang = language.MustParse("de-DE")
case "fr-ch":
    lang = language.MustParse("fr-CH")
default:
    http.NotFound(w, r)
    return
}

// Define a variable to hold the number of books. In a real application
// this would probably be retrieved by making a database query or
// something similar.
var totalBookCount = 1_252_794

p := message.NewPrinter(lang)
p.Fprintf(w, "Welcome!\n")

// Use the Fprintf() function to include the new message in the HTTP
// response, with the book count as in interpolated integer value.
p.Fprintf(w, "%d books available\n", totalBookCount)
```

} Save the changes, then use go generate to output some new out.gotext.json files. You should see warning messages for the new missing translations like so:

$ go generate ./internal/translations/translations.go de-DE: Missing entry for "{TotalBookCount} books available". fr-CH: Missing entry for "{TotalBookCount} books available". Let's take a look at the de-DE/out.gotext.json file:

File: internal/translations/locales/de-DE/out.gotext.json { "language": "de-DE", "messages": [{ "id": "Welcome!", "message": "Welcome!", "translation": "Willkommen!" }, { "id": "{TotalBookCount} books available", "message": "{TotalBookCount} books available", "translation": "", "placeholders": [{ "id": "TotalBookCount", "string": "%[1]d", "type": "int", "underlyingType": "int", "argNum": 1, "expr": "totalBookCount" }] }] } The first thing to point out here is that the translation for our "Welcome!" message has been persisted across the workflow and is already present in the out.gotext.json file. This is obviously really important, because it means that when we send the file to the translator they won't need to provide the translation again.

The second thing is that there is now an entry for our new message. We can see that this has the form "{TotalBookCount} books available", with the (capitalized) variable name from our Go code being used as the placeholder parameter. You should keep this in mind when writing your code, and try to use sensible and descriptive variable names that will make sense to your translators. The placeholders array also provides additional information about each placeholder value, the most useful part probably being the type value (which in this case tells the translator that the TotalBookCount value is an integer).

So the next step is to send these new out.gotext.json files off to a translator for translation. Again, we'll simulate that here by copying them to messages.gotext.json files and adding the translations like so:

$ cp internal/translations/locales/de-DE/out.gotext.json internal/translations/locales/de-DE/messages.gotext.json $ cp internal/translations/locales/fr-CH/out.gotext.json internal/translations/locales/fr-CH/messages.gotext.json File: internal/translations/locales/de-DE/messages.gotext.json { "language": "de-DE", "messages": [{ "id": "Welcome!", "message": "Welcome!", "translation": "Willkommen!" }, { "id": "{TotalBookCount} books available", "message": "{TotalBookCount} books available", "translation": "{TotalBookCount} Bücher erhältlich", "placeholders": [{ "id": "TotalBookCount", "string": "%[1]d", "type": "int", "underlyingType": "int", "argNum": 1, "expr": "totalBookCount" }] }] } File: internal/translations/locales/fr-CH/messages.gotext.json { "language": "fr-CH", "messages": [{ "id": "Welcome!", "message": "Welcome!", "translation": "Bienvenue !" }, { "id": "{TotalBookCount} books available", "message": "{TotalBookCount} books available", "translation": "{TotalBookCount} livres disponibles", "placeholders": [{ "id": "TotalBookCount", "string": "%[1]d", "type": "int", "underlyingType": "int", "argNum": 1, "expr": "totalBookCount" }] }] } Make sure that both messages.gotext.json files are saved, and then run go generate to update our message catalog. This should run without any warnings.

$ go generate ./internal/translations/translations.go When you restart the cmd/www application and make some HTTP requests again, you should now see the new translated messages like so:

$ curl localhost:4018/en-GB Welcome! 1,252,794 books available

$ curl localhost:4018/de-de Willkommen! 1.252.794 Bücher erhältlich

$ curl localhost:4018/fr-ch Bienvenue ! 1 252 794 livres disponibles Now this is really cool. As we'll as the translations being applied by our message.Printer, it's also smart enough to output the interpolated integer value with the correct number formatting for each language. We can see here that our en-GB locale uses the "," character as a thousands separator, whereas de-DE uses "." and fr-CH uses the whitespace " ". A similar thing is done for decimal separators too.

Dealing with pluralization This is working nicely, but what happens if there is only 1 book available in our bookstore? Let's update the handleHome() function so that the totalBookCount value is 1:

File: cmd/www/handlers.go package main

...

func handleHome(w http.ResponseWriter, r *http.Request) { locale := r.URL.Query().Get(":locale")

```
var lang language.Tag

switch locale {
case "en-gb":
    lang = language.MustParse("en-GB")
case "de-de":
    lang = language.MustParse("de-DE")
case "fr-ch":
    lang = language.MustParse("fr-CH")
default:
    http.NotFound(w, r)
    return
}

// Set the total book count to 1.
var totalBookCount = 1

p := message.NewPrinter(lang)
p.Fprintf(w, "Welcome!\n")
p.Fprintf(w, "%d books available\n", totalBookCount)
```

} (I know this is a bit of a tenuous example, but it helps illustrate Go's pluralization functionality without much extra code, so bear with me!)

You can probably imagine what happens when we restart the application and make a request to localhost:4018/en-gb now.

$ curl localhost:4018/en-gb Welcome! 1 books available That's right, we see the message "1 books available", which isn't correct English because of the plural noun books. It would be better if this message read 1 book available or — even better — One book available instead.

Happily, it's possible for us to specify alternative translations based on the value of an interpolated variable in our messages.gotext.json files.

Let's start by demonstrating this for our en-GB locale. If you're following along, copy the en-GB/out.gotext.json file to en-GB/messages.gotext.json:

$ cp internal/translations/locales/en-GB/out.gotext.json internal/translations/locales/en-GB/messages.gotext.json And then update it like so:

File: internal/translations/locales/en-GB/messages.gotext.json { "language": "en-GB", "messages": [{ "id": "Welcome!", "message": "Welcome!", "translation": "Welcome!", "translatorComment": "Copied from source.", "fuzzy": true }, { "id": "{TotalBookCount} books available", "message": "{TotalBookCount} books available", "translation": { "select": { "feature": "plural", "arg": "TotalBookCount", "cases": { "=1": { "msg": "One book available" }, "other": { "msg": "{TotalBookCount} books available" } } } }, "placeholders": [{ "id": "TotalBookCount", "string": "%[1]d", "type": "int", "underlyingType": "int", "argNum": 1, "expr": "totalBookCount" }] }] } Now, rather than the translation value being a simple string we have set it to a JSON object that instructs the message catalog to use different translations depending on the value of the TotalBookCount placeholder. The key part here is the cases value, which contains the translations to use for different values of the placeholder. The supported case rules are:

Case Description "=x" Where x is an integer that equals the value of the placeholder "<x" Where x is an integer that is larger than the value of the placeholder "other" All other cases (a bit like default in a Go switch statement) Note: If you look at the documentation for the golang.org/x/text/feature/plural package (which is what gotext uses behind the scenes when generating the message catalog), you'll see that it also mentions the case rules "zero", "one", "two", "few", and "many". However, these rules aren't supported for all possible target languages, and you may get an error like gotext: generation failed: error: plural: form "many" not supported for language "de-DE" if you try to use them. It seems to be safer to stick with the three case rules

in the table above. Additionally, it's important to be aware that the range of allowed values for x in the "=x" and "<x" case rules is 0 to 32767. Trying to use something outside of that range will result in an error. There's an open issue about these behaviors here. Let's complete work this by updating the messages.gotext.json files for our de-DE and fr-CH languages to include the appropriate pluralized variations, like so:

File: internal/translations/locales/de-DE/messages.gotext.json { "language": "de-DE", "messages": [{ "id": "Welcome!", "message": "Welcome!", "translation": "Willkommen!" }, { "id": "{TotalBookCount} books available", "message": "{TotalBookCount} books available", "translation": { "select": { "feature": "plural", "arg": "TotalBookCount", "cases": { "=1": { "msg": "Ein Buch erhältlich" }, "other": { "msg": "{TotalBookCount} Bücher erhältlich" } } } }, "placeholders": [{ "id": "TotalBookCount", "string": "%[1]d", "type": "int", "underlyingType": "int", "argNum": 1, "expr": "totalBookCount" }] }] } File: internal/translations/locales/fr-CH/messages.gotext.json { "language": "fr-CH", "messages": [{ "id": "Welcome!", "message": "Welcome!", "translation": "Bienvenue !" }, { "id": "{TotalBookCount} books available", "message": "{TotalBookCount} books available", "translation": { "select": { "feature": "plural", "arg": "TotalBookCount", "cases": { "=1": { "msg": "Un livre disponible" }, "other": { "msg": "{TotalBookCount} livres disponibles" } } } }, "placeholders": [{ "id": "TotalBookCount", "string": "%[1]d", "type": "int", "underlyingType": "int", "argNum": 1, "expr": "totalBookCount" }] }] } Once those files are saved, use go generate again to update the message catalog:

$ go generate ./internal/translations/translations.go And if you restart the web application and make some HTTP requests, you should now see the appropriate message for 1 book:

$ curl localhost:4018/en-GB Welcome! One book available

$ curl localhost:4018/de-de Willkommen! Ein Buch erhältlich

$ curl localhost:4018/fr-ch Bienvenue ! Un livre disponible If you like, you can revert the totalBookCount variable back to a larger number...

File: cmd/www/handlers.go package main

...

func handleHome(w http.ResponseWriter, r *http.Request) { ...

```
// Revert the total book count.
var totalBookCount = 1_252_794

p := message.NewPrinter(lang)
p.Fprintf(w, "Welcome!\n")
p.Fprintf(w, "%d books available\n", totalBookCount)
```

} And when you restart the application and make another request, you should see the "other" version of our message:

$ curl localhost:4018/de-de Willkommen! 1.252.794 Bücher erhältlich Creating a localizer abstraction In the final part of this article we're going to create a new internal/localizer package which abstracts all our code for dealing with languages, printers and translations.

If you're following along, go ahead and create a new internal/localizer directory containing a localizer.go file.

$ mkdir -p internal/localizer $ touch internal/localizer/localizer.go At this point, your project structure should look like this:

. ├── cmd │ └── www │ ├── handlers.go │ └── main.go ├── go.mod ├── go.sum └── internal ├── localizer │ └── localizer.go └── translations ├── catalog.go ├── locales │ ├── de-DE │ │ ├── messages.gotext.json │ │ └── out.gotext.json │ ├── en-GB │ │ ├── messages.gotext.json │ │ └── out.gotext.json │ └── fr-CH │ ├── messages.gotext.json │ └── out.gotext.json └── translations.go And then add the following code to the new localizer.go file:

File: internal/localizer/localizer.go package localizer

import (// Import the internal/translations so that it's init() function // is run. It's really important that we do this here so that the // default message catalog is updated to use our translations // *before* we initialize the message.Printer instances below. _ "bookstore.example.com/internal/translations"

```
"golang.org/x/text/language"
"golang.org/x/text/message"
```

)

// Define a Localizer type which stores the relevant locale ID (as used // in our URLs) and a (deliberately unexported) message.Printer instance // for the locale. type Localizer struct { ID string printer *message.Printer }

// Initialize a slice which holds the initialized Localizer types for // each of our supported locales. var locales = []Localizer{ { // Germany ID: "de-de", printer: message.NewPrinter(language.MustParse("de-DE")), }, { // Switzerland (French speaking) ID: "fr-ch", printer: message.NewPrinter(language.MustParse("fr-CH")), }, { // United Kingdom ID: "en-gb", printer: message.NewPrinter(language.MustParse("en-GB")), }, }

// The Get() function accepts a locale ID and returns the corresponding // Localizer for that locale. If the locale ID is not supported then // this returns `false` as the second return value. func Get(id string) (Localizer, bool) { for _, locale := range locales { if id == locale.ID {

```
        return locale, true
    }
}

return Localizer{}, false
```

}

// We also add a Translate() method to the Localizer type. This acts // as a wrapper around the unexported message.Printer's Sprintf() // function and returns the appropriate translation for the given // message and arguments. func (l Localizer) Translate(key message.Reference, args ...interface{}) string { return l.printer.Sprintf(key, args...) } Note: Notice here that we're initializing a single message.Printer for each locale at startup, and these will be used concurrently by our web application handlers. Although the golang.org/x/text/message documentation doesn't say that message.Printer is safe for concurrent use, I checked with Marcel van Lohuizen (the lead developer of the golang.org/x/text packages) and he confirmed that message.Printer is intended to be used concurrently and is concurrency safe (so long as access to any write destination is synchronized). Next let's update the cmd/www/handlers.go file to use our new Localizer type, and — while we're at it — let's also make our handleHome() function render an additional "Launching soon!" message.

File: cmd/www/handlers.go package main

import ("fmt" // New import "net/http"

```
"bookstore.example.com/internal/localizer" // New import
```

)

func handleHome(w http.ResponseWriter, r *http.Request) { // Initialize a new Localizer based on the locale ID in the URL. l, ok := localizer.Get(r.URL.Query().Get(":locale")) if !ok { http.NotFound(w, r) return }

```
var totalBookCount = 1_252_794

// Update these to use the new Translate() method.
fmt.Fprintln(w, l.Translate("Welcome!"))
fmt.Fprintln(w, l.Translate("%d books available", totalBookCount))

// Add an additional "Launching soon!" message.
fmt.Fprintln(w, l.Translate("Launching soon!"))
```

} It's worth pointing out that our use of the Translate() method here isn't just some syntactic sugar. You might remember earlier that I wrote the following warning:

It's critical to note when gotext walks your code it actually only looks for calls to message.Printer.Printf(), Fprintf() and Sprintf() — basically the three methods that end with an f. It ignores all other methods such as Sprint() or Println().

By having all our translations go through the Translate() method — which uses Sprintf() behind-the-scenes — we avoid the scenario where you accidentally use a method like Sprint() or Println() and gotext doesn't extract the message to the out.gotext.json files.

Let's try this out and run go generate again:

$ go generate ./internal/translations/translations.go de-DE: Missing entry for "Launching soon!". fr-CH: Missing entry for "Launching soon!". So this is really smart. We can see that gotext has been clever enough to walk our entire codebase and identify what strings need to be translated, even when we abstract the message.Printer.Sprintf() call to a helper function in a different package. This is awesome, and one of the things that I really appreciate about the gotext tool.

If you're following along, please go ahead and copy the out.gotext.json files to message.gotext.json files, and add the necessary translations for the new "Launching soon!" message. Then remember

to run go generate again and restart the web application.

When you make some HTTP requests again now, your responses should look similar to this:

$ curl localhost:4018/en-gb Welcome! 1,252,794 books available Launching soon!

$ curl localhost:4018/de-de Willkommen! 1.252.794 Bücher erhältlich Bald verfügbar!

$ curl localhost:4018/fr-ch Bienvenue ! 1 252 794 livres disponibles Bientôt disponible ! Additional information Conflicting routes At this start of this post I'd deliberately didn't recommending using httprouter, despite it being an excellent and popular router. This is because using a dynamic locale as the first part of a URL path is likely to result in conflicts with other application routes which don't require a locale prefix, like /static/css/main.css or /admin/login. The httprouter package doesn't allow conflicting routes, which makes using it awkward in this scenario. If you do want to use httprouter, or want to avoid conflicting routes in your application, you could pass the locale as a query string parameter instead like /category/travel?locale=gb.

If you enjoyed this article, you might like to check out my recommended tutorials list or check out my books Let's Go and Let's Go Further, which teach you everything you need to know about how to build professional production-ready web applications and APIs with Go.

Filed under:golang tutorial

Revision #1
Created 2022-07-23 07:12:58 UTC by gasick
Updated 2023-04-16 19:36:18 UTC by gasick