Add 2FA to your Application with the Verify API and Golang SDK

When you've worked hard at building a web application that offers real value to your users, it can be really disheartening to see it abused. Leaked credentials, fake signups ... there is always a small minority seeking to use your platform for their own nefarious purposes.

While it's next to impossible to prevent this from happening at some level, you can deter all but the most committed abusers by implementing two-factor-authentication (2FA).

What is 2FA?

2FA is an extra layer of protection that requires your user to provide something more than just a user name and password to use your service. This is typically access to a mobile device that can receive a security code that they then enter into your application as part of the registration or login process. Because your phone number is unique to you, it makes it much harder for scammers to impersonate existing users or sign up for multiple fake accounts.

Another good use case for 2FA is "step-up authentication". This is ideal for situations when a user

attempts to do something in your application which could have bad consequences if that user is not who they say they are. For example, if you have a banking application and your user attempts to add a new payee. In instances like this, 2FA provides some reassurance that they are legitimate.

The Vonage Verify API

The Vonage Verify API makes it really straightforward to implement 2FA in your applications and that's what we're hoping to demonstrate in today's post. We'll be building a simple site for a fictitious company called Acme Inc, which requires users to register using their mobile device to add some level of protection against fraudulent use.

And, to make things more exciting, we're going to be building this in our brand new (beta) Go SDK. This should please all you hardened Gophers out there and perhaps even encourage others to have their first experience of Go development? (Spoiler alert: I think you'll *love* Go!)

Let's get to it! (If you don't have time to code along, you can find the source code on GitHub).

Vonage API Account

To complete this tutorial, you will need a Vonage API account. If you don't have one already, you can sign up today and start building with free credit. Once you have an

account, you can find your API Key and API Secret at the top of the Vonage API Dashboard.

Create your project

Make a directory for your project and within it, run `go mod init`. This generates a `go.mod` file to manage dependencies.

```
mkdir go-2fa-example
cd go-2fa-example
go mod init go-2fa-example
```

Then, create a `main.go` file. This is where you'll write your application code:

We're going to use Go templates to build out our site. Let's create a directory for them (`tmpl`) and also for the CSS we'll use to style our pages (`static`):

Configure the Vonage Go SDK

We're going to be using the Verify API to require users to

prove that they own the device they want to register for our service with. The Verify API makes the whole two-factor authentication process really straightforward.

Not so long ago us Go developers would have had to make all the required API calls manually, but in these more enlightened times we have our shiny new Vonage Go SDK to make life easier, so let's get that all setup and configured.

To use the Go SDK we need to provide our API key and secret, which can be found in the Developer Dashboard. Now, we don't want to just put those credentials directly in our code, because anyone who has access to our key and secret can make API calls at our expense. So we'll configure those details in a `.env` file and use the `github.com/joho/godotenv` package to read those values into environment variables.

First, create the `.env` file in your project directory:

```
VONAGE_API_KEY=<YOUR_API_KEY>
VONAGE_API_SECRET=<YOUR_API_SECRET>
```

Replace `<YOUR_API_KEY>` and `<YOUR_API_SECRET>` with your own API key and secret.

Then, execute `go get` to install `dotenv` and the Vonage Go SDK:

```
go get github.com/joho/godotenv
go get github.com/vonage/vonage-go-sdk
```

In `main.go` write code to read your Vonage API credentials from the `.env` file and use them to instantiate an instance of the Vonage Go SDK `VerifyClient`:

```
package main

import (
    "log"
    "os"

    "github.com/joho/godotenv"
    "github.com/vonage/vonage-go-sdk"
)

var verifyClient *vonage.VerifyClient

func main() {
    err := godotenv.Load()
    if err != nil {
        log.Fatal("Error loading .env file")
        os.Exit(1)
    }

    apiKey := os.Getenv("VONAGE_API_KEY")
    apiSecret := os.Getenv("VONAGE_API_SECRET")

    auth := vonage.CreateAuthFromKeySecret(apiKey, apiSecre
    verifyClient = vonage.NewVerifyClient(auth)
```

```
}
```

Build the UI

To understand what pages our web app has to present to users, consider the workflow we are going to implement:

From this, you can see that we need the following pages:

- The **home page** (`/`): If the user is registered, we display their registered name and phone number here.
- The **registration page** (`/register`): Where the user can enter their name and phone number to register for our service.
- The **verification code** entry page (`/enter-code`): Where the user enters the PIN code sent to their phone by the Verify API.

We'll create these pages using Go templates. These make it easy to pass variables from code into the page and also have one template include others as "partials" so that you don't have to duplicate common content like headers, menus, and so on.

First, add the `html/template` module to the list of imports in `main.go`. Then, create the following templates in the `tmpl` directory:

The base page template

This is the base layout that all our templates will use. The `{{template "main" .}}` directive lets us populate the page body with content from the other templates:

```
<!-- base.layout.gohtml-->
{{define "base"}}
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Go Verify Demo</title>
  <link rel="stylesheet" href="/static/style.css">
  <link rel="preconnect" href="https://fonts.gstatic.com">
  <link href="https://fonts.googleapis.com/css2?family=Popp
</head>

<body>
  {{template "main" .}}
</body>

</html>
{{end}}
```

The home page template

When the user has successfully registered for the service, this page will display their name and phone number:

```
<!-- home.page.gohtml -->
{{template "base" .}}

{{define "main"}}
  <h1>Welcome to Acme Inc {{.Name}}!</h1>
  <p>Your registered phone number is: {{.Phone}}</p>
{{end}}
```

The registration page template

This template includes a form where the user can enter their name and phone number to register for the service:

```
<!-- register.page.gohtml -->
{{template "base" .}}

{{define "main"}}
  <h1>Acme inc.</h1>
  <p>Please register using your mobile/cell phone number fo
  <form action="/verify">
    <fieldset>
      <label for="name">Your name:</label>
      <input type="text" class="ghost-input" id="name" name
      <label for="phone_number">Your mobile number:</label>
      <input type="text" class="ghost-input" id="phone_numb
      <button type="submit" class="ghost-button">Register</
    </fieldset>
  </form>
{{end}}
```

The enter code page template

This page displays a form where the user enters the code they received from the Verify API:

```
<!-- entercode.page.gohtml -->
{{template "base" .}}

{{define "main"}}
  <h1>Acme inc.</h1>
  <p>Please enter the code you received by SMS:</p>
  <form action="/check-code">
    <fieldset>
      <label for="pin_code">Enter PIN:</label>
      <input type="text" class="ghost-input" id="pin_code"
    <button type="submit" class="ghost-button">Let's Go!</b
    </fieldset>
  </form>

{{end}}
```

Make it look pretty(ish)

Finally, let's add some CSS to make these pages look like we invested *some* effort in their creation. (I'm not a front-end person I'm afraid, so PRs are very welcome!)

Grab the `style.css` file from the Github repo and include it in your `static` directory.

Go needs to be able to serve that static CSS content from

your local filesystem, so you need to tell your application how to do that.

In `main.go`, first import the `net/http` module and then modify your `main()` function as follows:

```
func main() {
    err := godotenv.Load()
    if err != nil {
        log.Fatal("Error loading .env file")
        os.Exit(1)
    }

    apiKey := os.Getenv("VONAGE_API_KEY")
    apiSecret := os.Getenv("VONAGE_API_SECRET")

    auth := vonage.CreateAuthFromKeySecret(apiKey, apiSecre
    verifyClient = vonage.NewVerifyClient(auth)

    mux := http.NewServeMux()
    mux.Handle("/static/", http.StripPrefix("/static/", htt

}
```

Manage the user's session

We need to think about how we're going to tell if a user has registered for our service so that we can either log them in or redirect them to the registration page.

To do that, we're going to rely on session cookies. There's

a great module we can use to manage cookie and filesystem sessions in Go: the Gorilla sessions module.

To install it, run:

```
go get github.com/gorilla/sessions
```

Add it to your list of imports in `main.go` and create the session cookie. While you're there, also declare a `struct` called `UserData` that will store the name and phone number of a registered user. We'll pass this `struct` to our home page template so that we can display those details once the user has registered successfully:

```
package main

import (
    "html/template"
    "log"
    "net/http"
    "os"

    "github.com/gorilla/sessions"
    "github.com/joho/godotenv"
    "github.com/vonage/vonage-go-sdk"
)

var (

    key   = []byte("super-secret-key")
```

```
    store = sessions.NewCookieStore(key)
)


type UserData struct {
    Name  string
    Phone string
}

...
```

Define the routes

Now we must consider what routes we must define to manage our workflow. We're going to create the following routes:

- The **entry point** to our application (`/`). This will display the home page (if the user is registered), or redirect to the **`/register`** route (if they are not).
- The **registration** route (**`/register`**). This will display the registration form which, when submitted, will kick off the verification process by calling the **`/verify`** route.
- The **verification** route (**`/verify`**). This makes a Verify request to the Verify API to send a code to the user's mobile device. It then redirects to the **`/enter-code`** route.
- The **enter code** route (**`/enter-code`**) presents the

user with a form containing a text box where they can enter the PIN code they received. When they submit the form, we redirect to the **`/check-code`** route.

- The **check code** route (**`check-code`**). This matches the code they entered with the one that was sent by the Verify API. If it's a match we store the user's details in the session and redirect to the home page (`/`). If there's no match, we'll log an error to the console. (We could handle this more gracefully, but I'll leave that as an exercise for the reader!)
- The **clear** route (**`/clear`**). This deletes the session cookie so that you can "unregister" a registered user. Useful for testing!

Phew! That's quite a bit of code to write. Thankfully, it's all pretty straightforward.

First, define some handlers for the routes in your `main` function and start your server listening for incoming requests:

```
func main() {
    err := godotenv.Load()
    if err != nil {
        log.Fatal("Error loading .env file")
        os.Exit(1)
    }

    apiKey := os.Getenv("VONAGE_API_KEY")
    apiSecret := os.Getenv("VONAGE_API_SECRET")
```

```
    auth := vonage.CreateAuthFromKeySecret(apiKey, apiSecre
    verifyClient = vonage.NewVerifyClient(auth)

    mux := http.NewServeMux()
    mux.Handle("/static/", http.StripPrefix("/static/", htt
    mux.HandleFunc("/", home)
    mux.HandleFunc("/register", register)
    mux.HandleFunc("/verify", verify)
    mux.HandleFunc("/enter-code", enterCode)
    mux.HandleFunc("/check-code", checkCode)
    mux.HandleFunc("/clear", unregister)

    log.Println("Starting server on :5000")
    err = http.ListenAndServe(":5000", mux)
    log.Fatal(err)
}
```

The home route handler

Create the `home` handler above your `main` function. This handler is fired whenever a user visits the root of your web app at `/`.

It loads the session cookie and checks if the `registered` session value is set. If it is, it stores the cookie information in an instance of `UserData` and uses that to render the home page with the user's name and phone number.

If not, then it redirects to the `/register` route:

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

    session, _ := store.Get(r, "acmeinc-cookie")


    if auth, ok := session.Values["registered"].(bool); !ok

        http.Redirect(w, r, "/register", 302)
    }

    userData := UserData{
        Name:  fmt.Sprintf("%v", session.Values["name"]),
        Phone: fmt.Sprintf("%v", session.Values["phoneNumbe
    }

    files := []string{
        "./tmpl/home.page.gohtml",
        "./tmpl/base.layout.gohtml",
    }


    ts, err := template.ParseFiles(files...)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
        return
    }

    err = ts.Execute(w, userData)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
    }
```

```
}
```

The register route handler

The register route displays a form for the user to enter their name and phone number to register for the service.

First, add `fmt` to your list of imports. Then, create the `register` handler above your `main` function:

```
func register(w http.ResponseWriter, r *http.Request) {
    files := []string{
        "./tmpl/register.page.gohtml",
        "./tmpl/base.layout.gohtml",
    }

    ts, err := template.ParseFiles(files...)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
        return
    }

    err = ts.Execute(w, nil)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
    }
}
```

The verify route handler

When the user has filled in their registration details and clicked the "Register" button, the app calls the `/verify` route. It's here that we create the initial request to the Verify API to generate a code and send it to the user's phone.

When you submit a request to the Verify API, it returns a `request_id` to identify that particular request. You need this ID when you check the code that the user enters, so store it in a global variable:

```
package main

import (
    ...
)

var (
    ...
)


type UserData struct {
    ...
}

var verifyClient *vonage.VerifyClient
var requestID string
```

Initiating this request is very straightforward using the Go SDK. We call the `VerifyClient.Request` method, passing in the user's phone number, the text we want to appear in the verification SMS, and some options to customize the verification process.

In this example, the options we're specifying are:

- **`CodeLength`**: This is the length of the code we're sending to the user. It defaults to four digits, but we're generating a six-digit code.
- **`Lg`**: Specifies the language we're sending the verification SMS in, from the list of available languages.
- **`WorkflowID`**: The Verify API makes multiple attempts to send a verification code. How and when those attempts are made depends on your chosen workflow. We're using the workflow with an ID of **`4`**, which sends the code via SMS and then waits two minutes for the code to be verified. If it's not verified after that time it sends the code again and waits for a further three minutes before canceling the process. Other workflows use a mixture of SMS and voice calls with text-to-speech to deliver the verification code.

We're also going to store the information that our user entered in our session cookie.

Write the code for the `verify` route handler above your `main` function:

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

    session, _ := store.Get(r, "acmeinc-cookie")

    userName := r.URL.Query().Get("name")
    phoneNumber := r.URL.Query().Get("phone_number")
    session.Values["name"] = userName
    session.Values["phoneNumber"] = phoneNumber
    session.Save(r, w)
    log.Println("Verifying...." + userName + " at " + phone
    response, errResp, err := verifyClient.Request(phoneNum

    if err != nil {
        fmt.Printf("%#v\n", err)
    } else if response.Status != "0" {
        fmt.Println("Error status " + errResp.Status + ": "
    } else {
        requestID = response.RequestId
        fmt.Println("Request started: " + response.RequestI

        http.Redirect(w, r, "/enter-code", 302)
    }
}
```

The enter code route handler

If everything has gone to plan, the user will now receive an SMS containing a PIN code. Our `/enter-code` route will provide a form for the user to enter that code.

Add the `enterCode` route handler to your application:

```
func enterCode(w http.ResponseWriter, r *http.Request) {
    files := []string{
        "./tmpl/entercode.page.gohtml",
        "./tmpl/base.layout.gohtml",
    }

    ts, err := template.ParseFiles(files...)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
        return
    }

    err = ts.Execute(w, nil)
    if err != nil {
        log.Println(err.Error())
        http.Error(w, "Internal Server Error", 500)
    }
}
```

The check code route handler

When the user has entered the code they received via SMS, we need to make a Verify check request to ensure that the code they entered is the same as the one they were sent. We use the `VerifyClient.Check` method for that, passing in the ID of the request we made in the `verify` route handler and the code they entered into our page.

If the code matches, we set the `registered` session cookie value to `true` and redirect to the home page. If not, we'll log an error message to the console:

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

    session, _ := store.Get(r, "acmeinc-cookie")
    pinCode := r.URL.Query().Get("pin_code")
    response, errResp, err := verifyClient.Check(requestID,

    if err != nil {
        fmt.Printf("%#v\n", err)
    } else if response.Status != "0" {
        fmt.Println("Error status " + errResp.Status + ": "
    } else {
        fmt.Println("Request complete: " + response.Request

        session.Values["registered"] = true
        session.Save(r, w)
        http.Redirect(w, r, "/", 302)
    }
}
```

The clear route handler

This will be useful for testing. It lets you reset the user by making a request to `http://localhost:5000/clear`, which deletes the session cookie.

Create the `unregister` route handler as follows:

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

    session, _ := store.Get(r, "acmeinc-cookie")
    session.Options.MaxAge = -1
    session.Save(r, w)
    http.Redirect(w, r, "/", 302)
}
```

Try it out!

You're all done! Now it's time to test your application.

1. Launch the app by running `go run main.go`
2. Visit http://localhost:5000 in your browser
3. Enter your name and phone number, including the dialing code but omitting any leading zeroes. For example, the UK mobile phone number `07700900001` should be entered as `447700900001`:
4. You'll receive an SMS containing a six-digit code:
5. Enter that code into the app:
6. You are registered and sent to the home page:

   ![Signed in at the home page]("Signed in at the home page")

What next?

Hopefully, you've seen how easy it is to use the Verify API and Go SDK to enable two-factor authentication in your apps. In fact, the whole verification process only required a few lines of code! The bulk of the work here was creating and managing the UI.

Note: If you didn't manage to follow along, you can find the full source code for this tutorial on GitHub.

Think this demo could use some improvements? I agree! These are the ones that occur to me:

- **The not-so-graceful exit when the user enters the wrong code**. Instead, you could redirect the user to the enter code page and simultaneously check the progress of the request, maybe even cancelling an in-progress request after two many failed attempts.
- **The format in which users are required to enter their phone numbers**. The Vonage APIs expect phone numbers to be in E.164 format, but there's no reason why you should inflict that on your users! You could have them enter their local number and country from a drop-down list and use the Vonage Number Insight API to convert it into the right format. That would also provide a handy check that the number the user entered is a legitimate one.

We hope you're as excited about our new Go SDK as we

are. To find out more about it, and the Verify API, check out the following resources:

- The Vonage Go SDK package
- Go SDK usage examples
- The Verify API documentation
- The Verify API reference