Fyne

A fine talk by Nick White

For golang Bristol++

2022-07-20

https://github.com/nickjwhite/fynetalk-golangbristol2022

Graphical User Interface Toolkit

Fyne is a GUI toolkit. It provides functions and structures to help with drawing windows, popups, buttons, scrollbars, progress bars, colour pickers, etc.

Other options are things like wxWidgets, Gtk, Qt, Cocoa, and whatever Windows' native toolkit is called. There are Go packages that can hook into these libraries, which link to the appropriate compiled libraries using CGo.

Difference with most other toolkits

Fyne is written in Go from the ground up.

It doesn't need to link to anything apart from the system's OpenGL library, which means easier building and distributing of executables.

This also makes reading the code behind Fyne much more pleasant and easy.

There's at least one other mature GUI toolkit built with Go called *Gio*.

Targets

Fyne works cross platform, on Linux, BSD, Mac and Windows, as well as on iOS and Android.

I haven't used it for mobile development at all, but it's one of their core focuses, so it probably works!

Documentation

The fyne project has quite good documentation, check out
https://fyne.io

My experience

I wrote an OCR program in Go, with a lovely command line interface.

Unfortunately not everyone finds the command line lovely.

So I tried using Fyne to add an optional GUI to it, and it worked great!

You can see the tool at

- https://rescribe.xyz/rescribe
- `git clone https://rescribe.xyz/bookpipeline`

For this talk the relevant parts are the `cmd/rescribe` directory, particularly `gui.go` and `makefile`.

I am no Fyne expert, though!

Basics

Fyne is split into several packages under `fyne.io/fyne/v2`

The two packages you will almost certainly need are:

- `fyne.io/fyne/v2/app`
- `fyne.io/fyne/v2/widget`

`app` manages creating and managing the window(s)

`widget` is the base package for all widgets, which are the main interface elements: buttons, labels, input boxes, etc.

Hello world

From the official documentation:
https://developer.fyne.io/started/hello

```
package main

import (
    "fyne.io/fyne/v2/app"
    "fyne.io/fyne/v2/widget"
)

func main() {
    a := app.New()
    w := a.NewWindow("Hello World")

    w.SetContent(widget.NewLabel("Hello World!"))
    w.ShowAndRun()
}
```

Widgets

There are plenty of widgets, which are generally quite flexible.

You create a new widget by doing something like this:

- `progressBar := widget.NewProgressBar()`
- `button := widget.NewButton("Label", func() {})`

To make the progress bar tick up by 10% each second you could do something like this:

```
go func() {
    i := 0.0
    for range time.Tick(time.Second) {
        progressBar.SetValue(i)
        if i < 1 {
            i += 0.1
        }
    }
}
```

Widget definition

go doc fyne.io/fyne/v2/ Widget:

```
type Widget interface {
        CanvasObject

        // CreateRenderer returns a new WidgetRenderer for
        // This should not be called by regular code, it is
        CreateRenderer() WidgetRenderer
}
```

CanvasObject is another interface, that won't fit onto a slide, but you can look it up with `go doc fyne.io/fyne/v2/ CanvasObject`. It's simpler than you'd think!

A canvas is basically just a general drawing area, and a fundamental interface that fyne uses.

As this is all idiomatic, straightforward Go, we can easily write alternative implementations that fulfil these interfaces, to create new widgets or other weird things.

Containers & Layouts

The `fyne.io/fyne/v2/containers` and `fyne.io/fyne/v2/layout` packages.

In hello world we used w.SetContent(widget) to put one widget in a window.

Containers are used to include multiple widgets, and layouts are used to determine how they're laid out.

Containers are a simple structure that contains multiple canvases, like widgets or other containers.

Layouts are structures that can be associated with containers that specify how to lay out the widgets, e.g. in rows, columns, a grid, etc.

Containers & Layouts example

```
btn1 := widget.NewButton("One", func() {})
btn2 := widget.NewButton("Two", func() {})
btns := container.New(layout.NewHBoxLayout(), btn1, btn2)
w.SetContent(btns)
```

Dialogue Boxes

The `fyne.io/fyne/v2/dialog` package.

This is used for popup boxes, file opening / saving dialogues, showing errors, etc.

Two options to use them:

- `d := dialog.New*; d.Show()`
- `dialog.Show*`

Dialogue Box examples

```
dialog.ShowError(fmt.Errorf("Oh no!"), w)

dialog.ShowInformation("OCR Complete", "Oh yes!", w)

d := dialog.NewFileOpen(func(uri fyne.URIReadCloser, err er
    if err != nil || uri == nil {
        return
    }
    uri.Close()
    dir.SetText(uri.URI().Path())
    dirIcon.SetResource(theme.DocumentIcon())
    myWindow.SetContent(fullContent)
    gobtn.Enable()
}, myWindow)
d.SetFilter(storage.NewExtensionFileFilter([]string{".pdf"]
d.Resize(fyne.NewSize(740, 600))
d.Show()
```

Testing

Fyne has a cool looking unit test system, using the standard go test tools, so you can create gui_test.go and use ‘go test .’

I haven’t actually used it, because I was playing fast and loose, but it looks great and you should.

There’s good documentation for it here:
https://developer.fyne.io/started/testing

Compiling & deploying

Basic go tools all work out of the box, `go build`, `go install`, etc.

Optional tool called `fyne` that can build things in a more end-user friendly way.

- Add an icon
- Flag the executable is a GUI app to the OS
- Package it nicely as a .app or whatever

Examples:

```
fyne package -os darwin -icon myapp.png
fyne package -os windows -icon myapp.png
```

Cross compiling

To compile for other platforms is a bit tricky, because of the need to link to the OpenGL libraries for each platform.

Some helpful tips are here:
https://developer.fyne.io/started/cross-compiling

Linux cross-crompiling to Windows is easy. Install mingw (`apt install mingw-w64`), then cross compile and package up like this:

```
CC="x86_64-w64-mingw32-gcc" CGO_ENABLED=1 GOOS=windows \
  GOARCH=amd64 go build -o $@ .

CC="x86_64-w64-mingw32-gcc" fyne package \
  --name Rescribe --exe build/windows/rescribe-bin.exe \
  --os windows --icon icon.png
```

Cross compiling continued

Linux cross-compiling to macOS is more annoying. You can look at the `makefile` in the rescribe repository if you want to see a full example. In short, it involves:

- Installing osxcross (which is a pain)
- Cross-compiling binaries separately for both amd64 & arm64 darwin targets
- Combining these into one binary with osxcross' `lipo` tool
- Running the fyne package tool to create a .app

There's also a docker thing called fyne-cross that I haven't tried but potentially makes this a bit easier.

Fin

- Q&A
- Demo
- Rescribe GUI code exploration