charmbracelet / **bubbletea** Public

Code | Issues 53 | Pull requests 50 | Discussions | Actions | Projects

master | 33 Branches | 57 Tags | Go to file | Go to file | Code

caarlos0 da49e8f · 4 days ago | 805 Commits

.github	feat: update bubbl...	4 days ago
examples	feat: update bubbl...	4 days ago
tutorials	feat: update bubbl...	4 days ago
.gitattributes	chore: update exa...	8 months ago
.gitignore	docs: move tutori...	2 years ago
.golangci-s...	Add separate sets ...	3 years ago
.golangci.yml	Add separate sets ...	3 years ago
CONTRIBU...	docs: Add CONTRI...	2 years ago
LICENSE	docs: update license	9 months ago
README.md	chore(docs): add e...	last month
commands....	feat: add set-wind...	2 months ago
commands...	feat: export Batch...	2 years ago
exec.go	fix: pass actual std...	2 years ago
exec_test.go	test: exec returnin...	2 years ago
go.mod	feat: update bubbl...	4 days ago
go.sum	feat: update bubbl...	4 days ago
key.go	feat: bracketed pa...	4 days ago
key_sequen...	feat: bracketed pa...	4 days ago
key_test.go	feat: bracketed pa...	4 days ago
logging.go	chore: restrict logf...	7 months ago
logging_tes...	test: logging test	3 years ago

About

A powerful little TUI framework

#go #cli #golang #framework #functional #elm-architecture #tui #hacktoberfest

Readme
MIT license
Activity
Custom properties
22.5k stars
113 watching
674 forks
Report repository

Releases 31

v0.25.0 Latest
on Dec 12, 2023

+ 30 releases

Packages

No packages published

Used by 5.2k

+ 5,187

Contributors 103

+ 89 contributors

mouse.go	feat: extended Co...	2 months ago
mouse_test...	feat: extended Co...	2 months ago
nil_rendere...	feat: bracketed pa...	4 days ago
nil_rendere...	test: complete nil r...	2 years ago
options.go	feat: bracketed pa...	4 days ago
options_tes...	feat: bracketed pa...	4 days ago
renderer.go	feat: bracketed pa...	4 days ago
screen.go	feat: bracketed pa...	4 days ago
screen_test....	feat: bracketed pa...	4 days ago
signals_uni...	add support for z/...	4 days ago
signals_win...	fix: detect terminal...	2 years ago
standard_re...	feat: bracketed pa...	4 days ago
tea.go	feat: bracketed pa...	4 days ago
tea_test.go	feat: add generic e...	10 months ago
tty.go	feat: bracketed pa...	4 days ago
tty_unix.go	add support for z/...	4 days ago
tty_window...	feat: use Termenv....	2 years ago

Languages

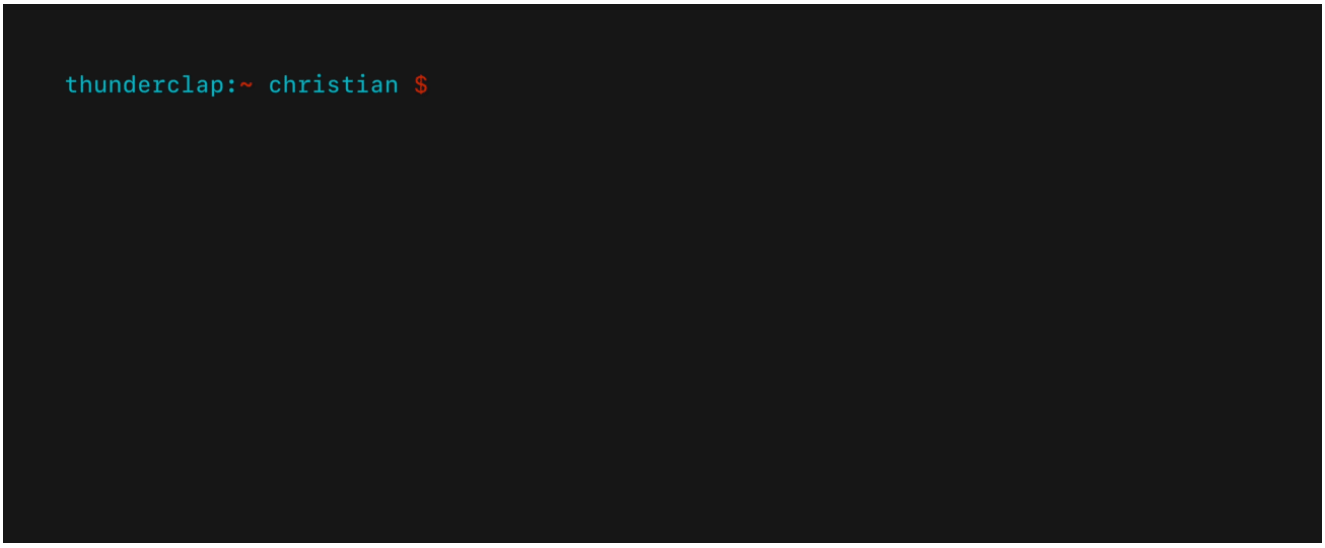
- Go 100.0%

README MIT license

Bubble Tea

release v0.25.0 reference build passing

The fun, functional and stateful way to build terminal apps. A Go framework based on The Elm Architecture. Bubble Tea is well-suited for simple and complex terminal applications, either inline, full-window, or a mix of both.

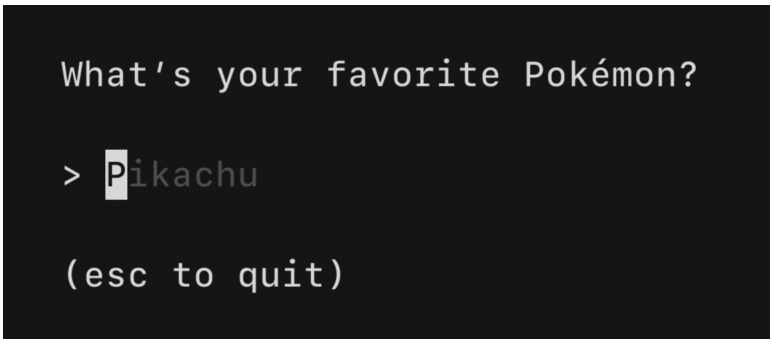


Bubble Tea is in use in production and includes a number of features and performance optimizations we've added along the way. Among those is a standard framerate-based renderer, a renderer for high-performance scrollable regions which works alongside the main renderer, and mouse support.

To get started, see the tutorial below, the examples, the docs, the video tutorials and some common resources.

By the way

Be sure to check out Bubbles, a library of common UI components for Bubble Tea.



Tutorial

Bubble Tea is based on the functional design paradigms of The Elm Architecture, which happens to work nicely with Go. It's a delightful way to build applications.

This tutorial assumes you have a working knowledge of Go.

By the way, the non-annotated source code for this program is available on GitHub.

Enough! Let's get to it.

For this tutorial, we're making a shopping list.

To start we'll define our package and import some libraries. Our only external import will be the Bubble Tea library, which we'll call `tea` for short.

```go
package main

import (
    "fmt"
    "os"

    tea "github.com/charmbracelet/bubbletea"
)
```

Bubble Tea programs are comprised of a **model** that describes the application state and three simple methods on that model:

- **Init**, a function that returns an initial command for the application to run.
- **Update**, a function that handles incoming events and updates the model accordingly.
- **View**, a function that renders the UI based on the data in the model.

The Model

So let's start by defining our model which will store our application's state. It can be any type, but a `struct` usually makes the most sense.

```go
type model struct {
    choices  []string           // items on the to-do list
    cursor   int                // which to-do list item our cursor is pointing at
    selected map[int]struct{}   // which to-do items are selected
}
```

Initialization

Next, we'll define our application's initial state. In this case, we're defining a function to return our initial model, however, we could just as easily define the initial model as a variable elsewhere, too.

```go
func initialModel() model {
        return model{
                // Our to-do list is a grocery list
                choices:  []string{"Buy carrots", "Buy celery", "Buy kohlrabi"},

                // A map which indicates which choices are selected. We're using
                // the  map like a mathematical set. The keys refer to the indexes
                // of the `choices` slice, above.
                selected: make(map[int]struct{}),
```

```
        }
}
```

Next, we define the `Init` method. `Init` can return a `Cmd` that could perform some initial I/O. For now, we don't need to do any I/O, so for the command, we'll just return `nil`, which translates to "no command."

```go
func (m model) Init() tea.Cmd {
    // Just return `nil`, which means "no I/O right now, please."
    return nil
}
```

The Update Method

Next up is the update method. The update function is called when "things happen." Its job is to look at what has happened and return an updated model in response. It can also return a `Cmd` to make more things happen, but for now don't worry about that part.

In our case, when a user presses the down arrow, `Update`'s job is to notice that the down arrow was pressed and move the cursor accordingly (or not).

The "something happened" comes in the form of a `Msg`, which can be any type. Messages are the result of some I/O that took place, such as a keypress, timer tick, or a response from a server.

We usually figure out which type of `Msg` we received with a type switch, but you could also use a type assertion.

For now, we'll just deal with `tea.KeyMsg` messages, which are automatically sent to the update function when keys are pressed.

```go
func (m model) Update(msg tea.Msg) (tea.Model, tea.Cmd) {
    switch msg := msg.(type) {

    // Is it a key press?
    case tea.KeyMsg:

        // Cool, what was the actual key pressed?
        switch msg.String() {

        // These keys should exit the program.
        case "ctrl+c", "q":
            return m, tea.Quit

        // The "up" and "k" keys move the cursor up
        case "up", "k":
            if m.cursor > 0 {
                m.cursor--
            }

        // The "down" and "j" keys move the cursor down
```

```
        case "down", "j":
            if m.cursor < len(m.choices)-1 {
                m.cursor++
            }

        // The "enter" key and the spacebar (a literal space) toggle
        // the selected state for the item that the cursor is pointing at.
        case "enter", " ":
            _, ok := m.selected[m.cursor]
            if ok {
                delete(m.selected, m.cursor)
            } else {
                m.selected[m.cursor] = struct{}{}
            }
        }
    }

    // Return the updated model to the Bubble Tea runtime for processing.
    // Note that we're not returning a command.
    return m, nil
}
```

You may have noticed that `ctrl+c` and `q` above return a `tea.Quit` command with the model. That's a special command which instructs the Bubble Tea runtime to quit, exiting the program.

The View Method

At last, it's time to render our UI. Of all the methods, the view is the simplest. We look at the model in its current state and use it to return a `string`. That string is our UI!

Because the view describes the entire UI of your application, you don't have to worry about redrawing logic and stuff like that. Bubble Tea takes care of it for you.

```
func (m model) View() string {
    // The header
    s := "What should we buy at the market?\n\n"

    // Iterate over our choices
    for i, choice := range m.choices {

        // Is the cursor pointing at this choice?
        cursor := " " // no cursor
        if m.cursor == i {
            cursor = ">" // cursor!
        }

        // Is this choice selected?
        checked := " " // not selected
        if _, ok := m.selected[i]; ok {
            checked = "x" // selected!
        }

        // Render the row
```

```
        s += fmt.Sprintf("%s [%s] %s\n", cursor, checked, choice)
    }

    // The footer
    s += "\nPress q to quit.\n"

    // Send the UI for rendering
    return s
}
```

All Together Now

The last step is to simply run our program. We pass our initial model to `tea.NewProgram` and let it rip:

```
func main() {
    p := tea.NewProgram(initialModel())
    if _, err := p.Run(); err != nil {
        fmt.Printf("Alas, there's been an error: %v", err)
        os.Exit(1)
    }
}
```

What's Next?

This tutorial covers the basics of building an interactive terminal UI, but in the real world you'll also need to perform I/O. To learn about that have a look at the Command Tutorial. It's pretty simple.

There are also several Bubble Tea examples available and, of course, there are Go Docs.

Debugging

Debugging with Delve

Since Bubble Tea apps assume control of stdin and stdout, you'll need to run delve in headless mode and then connect to it:

```
# Start the debugger
$ dlv debug --headless .
API server listening at: 127.0.0.1:34241

# Connect to it from another terminal
$ dlv connect 127.0.0.1:34241
```

Note that the default port used will vary on your system and per run, so actually watch out what address the first `dlv` run tells you to connect to.

Logging Stuff

You can't really log to stdout with Bubble Tea because your TUI is busy occupying that! You can, however, log to a file by including something like the following prior to starting your Bubble Tea program:

```
if len(os.Getenv("DEBUG")) > 0 {
        f, err := tea.LogToFile("debug.log", "debug")
        if err != nil {
                fmt.Println("fatal:", err)
                os.Exit(1)
        }
        defer f.Close()
}
```

To see what's being logged in real time, run `tail -f debug.log` while you run your program in another window.

Libraries we use with Bubble Tea

- Bubbles: Common Bubble Tea components such as text inputs, viewports, spinners and so on
- Lip Gloss: Style, format and layout tools for terminal applications
- Harmonica: A spring animation library for smooth, natural motion
- BubbleZone: Easy mouse event tracking for Bubble Tea components
- Termenv: Advanced ANSI styling for terminal applications
- Reflow: Advanced ANSI-aware methods for working with text

Bubble Tea in the Wild

For some Bubble Tea programs in production, see:

- AT CLI: execute AT Commands via serial port connections
- Aztify: bring Microsoft Azure resources under Terraform
- brows: a GitHub release browser
- Canard: an RSS client
- charm: the official Charm user account manager
- chezmoi: securely manage your dotfiles across multiple machines
- chtop: monitor your ClickHouse node without leaving the terminal
- circumflex: read Hacker News in the terminal
- clidle: a Wordle clone
- cLive: automate terminal operations and view them live in a browser
- container-canary: a container validator
- countdown: a multi-event countdown timer

- CRT: a simple terminal emulator for running Bubble Tea in a dedicated window, with optional shaders
- dns53: dynamic DNS with Amazon Route53. Expose your EC2 quickly, securely and privately
- eks-node-viewer: a tool for visualizing dynamic node usage within an eks cluster
- End Of Eden: a "Slay the Spire"-like, roguelite deck-builder game
- enola: find social media accounts by username across social networks
- flapioca: Flappy Bird on the CLI!
- fm: a terminal-based file manager
- fork-cleaner: clean up old and inactive forks in your GitHub account
- fztea: a Flipper Zero TUI
- gama: manage GitHub Actions from the terminal
- gambit: chess in the terminal
- gembro: a mouse-driven Gemini browser
- gh-b: a GitHub CLI extension for managing branches
- gh-dash: a GitHub CLI extension for PRs and issues
- gitflow-toolkit: a GitFlow submission tool
- Glow: a markdown reader, browser, and online markdown stash
- go-sweep: Minesweeper in the terminal
- gocovsh: explore Go coverage reports from the CLI
- got: a simple translator and text-to-speech app build on top of simplytranslate's APIs
- gum: interactivity and styling for shells and shell scripts
- hiSHtory: your shell history in context, synced, and queryable
- httpit: a rapid http(s) benchmark tool
- IDNT: a batch software uninstaller
- json-log-viewer: interactive viewer for JSON logs
- kboard: a typing game
- fractals-cli: a multiplatform terminal fractal explorer
- mc: the official MinIO client
- mergestat: run SQL queries on git repositories
- meteor: a highly customizable conventional commit message tool
- mods: AI on the CLI; built for pipelines
- Neon Modem Overdrive: a BBS-style TUI client for Discourse, Lemmy, Lobste.rs and Hacker News
- Noted: a note viewer and manager
- nom: an RSS reader and manager
- pathos: a PATH environment variable editor
- portal: secure transfers between computers
- redis-viewer: a Redis database browser
- redis_tui: a Redis database browser

- scrabbler: an automatic draw TUI for your duplicate Scrabble games
- sku: Sudoku on the CLI
- Slides: a markdown-based presentation tool
- SlurmCommander: a Slurm workload manager TUI
- Soft Serve: a command-line-first Git server that runs a TUI over SSH
- solitaire-tui: Klondike Solitaire for the terminal
- StormForge Optimize Controller: a tool for experimenting with application configurations in Kubernetes
- Storydb: a bash/zsh ctrl+r improved command history finder.
- STTG: a teletext client for SVT, Sweden's national public television station
- sttr: a general-purpose text transformer
- tasktimer: a dead-simple task timer
- termdbms: a keyboard and mouse driven database browser
- ticker: a terminal stock viewer and stock position tracker
- tran: securely transfer stuff between computers (based on portal)
- trainer: a Go concurrency coding interview simulator with learning materials
- Typer: a typing test
- typioca: a typing test
- tz: an scheduling aid for people in multiple time zones
- ugm: a unix user and group browser
- walk: a terminal navigator
- wander: a HashiCorp Nomad terminal client
- WG Commander: a TUI for a simple WireGuard VPN setup
- wishlist: an SSH directory

Feedback

We'd love to hear your thoughts on this project. Feel free to drop us a note!

- Twitter
- The Fediverse
- Discord

Acknowledgments

Bubble Tea is based on the paradigms of The Elm Architecture by Evan Czaplicki et alia and the excellent go-tea by TJ Holowaychuk. It's inspired by the many great *Zeichenorientierte Benutzerschnittstellen* of days past.

License

MIT

Part of Charm.