
Beautiful packages in Go
Mat Ryer

Other talks taking place right now

I just hacked your app! by Marcus Placona

Tensorflow and deep learning without a PhD by Martin Görner

If you're here, you must really care about beautiful packages

Mat Ryer

Coding Go since before v1

BLOG matryer.com

BOOK Go Programming Blueprints: Second Edition

WORK graymeta.com - 100% Go

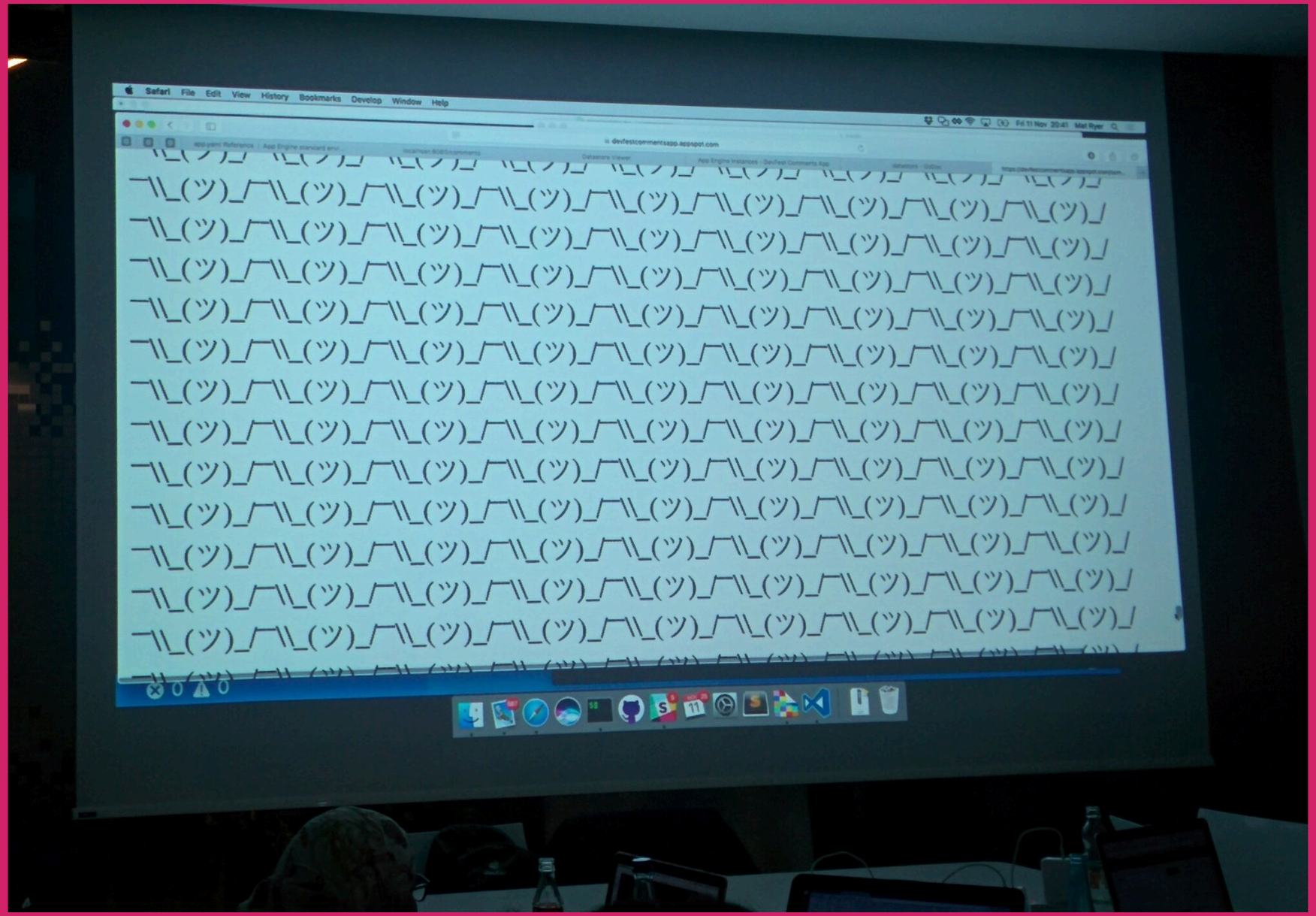
Not Banksy

Available for pre-order now

Workshop

Built a full API and deployed to Google App Engine

about 2 hours


Safari File Edit View History Bookmarks Develop Window Help
devfestcommentsapp.appspot.com
¯_(ツ)_/¯

¯_(ツ)_/¯

What is a package?

What is a package?

Folder full of `.go` (and hopefully `_test.go`) files

Can be imported into other projects

Has exported (public) and unexported (internal) stuff

Hopefully open-sourced

Not `package main` (that's a command)

Examples

Standard library is a load of packages

`fmt`, `log`, `http`, `os`, `filepath`, `errors`, `testing`

Open source

`glog`, `testify/assert`, `pkg/errors`, `gorilla/mux`, `protobuf/proto`

What is a beautiful package?

What is a beautiful package?

Elegant & simple

Obvious (maybe even boring)

You already know how to use it

Look forward to using it

SIMPLE!

QUIZ

What determines whether something is exported (public) or unexported (private)?

ANSWER

Items beginning with a capital letter are exported, otherwise not.

```
func Exported()   // exported
func unexported() // not exported
```

Moq

```
type Mocker struct
func New(src, packageName string) (*Mocker, error)
func (m *Mocker) Mock(w io.Writer, name ...string) error
func (m *Mocker) packageQualifier(pkg *types.Package) string
func (m *Mocker) extractArgs(sig *types.Signature, list *types.Tuple, nameFormat string) []*param
type obj struct
type method struct
func (m *method) Arglist() string
func (m *method) ArgCallList() string
func (m *method) ReturnArglist() string
type param struct
func (p param) String() string
func (p param) CallName() string
func (p param) TypeString() string
var moqTemplate
```

from github.com/matryer/moq

package moq

```
import "github.com/matryer/moq/package/moq"
```

Index

type Mocker

- func New(src, packageName string) (*Mocker, error)
- func (m *Mocker) Mock(w io.Writer, name ...string) error

Package Files

moq.go

type Mocker

```
type Mocker struct {
    // contains filtered or unexported fields
}
```

Mocker can generate mock structs.

func New

```
func New(src, packageName string) (*Mocker, error)
```

New makes a new Mocker for the specified package directory.

func (*Mocker) Mock

```
func (m *Mocker) Mock(w io.Writer, name ...string) error
```

Mock generates a mock for the specified interface name.

SIMPLE!

SUBLIMINAL


Mat Ryer
Go Programming
Blueprints
Second Edition
Build real-world, production-ready solutions in Go using cutting-edge technology and techniques
Packt>

How can we make our packages beautiful?

User centred design

"the needs, wants, and limitations of end users of a product, service or process are given extensive attention at each stage of the design process"

Personas, Scenarios, Use cases

User centred design

Packages have an interface (API)

They're used by humans

Why don't we apply the same kind of thinking?

Ask yourself...

Who is the user of the package?

What are they trying to do?

Why are they doing it?

Why are they using your package?

Are they building a PoC? Or is this part of a bigger system?

Narrow types are simpler

```
func WriteJSON(v interface{}, f *os.File) error
```

Can only write to file

```
func WriteJSON(v interface{}, w io.Writer) error
```

Now I can write to...

io.Writer

os.File
bytes.Buffer
http.ResponseWriter
encoding/zip

and whatever types our users have created

Create single method interfaces

If you're going to add an interface, try and keep them as tiny as possible

```
// EaseScore returns a score for how easy an interface
// is to implement. Lower is better.
func EaseScore(i Interface) int {
    return len(i.ExportedMethods)
}
```

If an interface has one method, it's very easy to implement.

...and you can do things like this:

```
type Handler interface {
    ServeHTTP(w http.ResponseWriter, r *http.Request)
}

type HandlerFunc func(w http.ResponseWriter, r *http.Request)

func (h HandlerFunc) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    return h(w, r)
}
```

...and even:

```
type StatusHandler int

func (s StatusHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    code := int(s)
    w.WriteHeader(code)
    io.WriteString(w, http.StatusText(code))
}
```

SIMPLE!

SUBLIMINAL


Mat Ryer
Go Programming
Blueprints
Second Edition
Build real-world, production-ready solutions in Go using cutting-edge technology and techniques
Packt

Obey my dog

GEHORCHEN MEINEM HUND

Context as the first argument

`context.Context` is part of the standard library

- Pass as the first argument
- If your package performs cancellable tasks
- If your code calls code that takes a context

QUIZ

In which version of Go did `context` join the standard library?

GO1.1

Leave concurrency to the user

Avoid this:

```
package thing

func DoAmazingThing() {
    go startDoingSomethingAmazing()
}
```

User won't necessarily know it's going to do this.

If they want to, the user can do this:

```
go thing.DoAmazingThing()
```

and they know what's going on

SIMPLE!

YOU ARE PROBABLY THE FIRST USER

Test driven development

Use the package before it even exists

Try different things

Aware of the API footprint from the very beginning

TIP: Put test code in different package

TIP: Use godoc early too

3 HARDEST THINGS...

The 3 hardest things in coding are:

1. Naming things
2. Invalidating the cache
3. Invalidating the cache

Naming things

Remember the package name is part of the API

```
package tea

func BrewTea(steeptime time.Duration) error {
    //...
}
```

Not nice from the outside:

```
err := tea.BrewTea(1 * time.Minute)
```

```go
package tea

func Brew(steeptime time.Duration) error {
    //...
}
```

Means we can call:

```go
err := tea.Brew(1 * time.Minute)
```

Make zero values useful

```
type Greeter struct {
    Format string
}

func (g Greeter) Greet(name string) string {
    if g.Format == "" {
        g.Format = "Hi %s"
    }
    return fmt.Sprintf(g.Format, name)
}
```

```go
func main() {
    var g Greeter
    fmt.Println(g.Greet("Natalie"))
}
// output: Hi Natalie
```

or

```go
func main() {
    g := Greeter{
        Format: "Hey there, %s",
    }
    fmt.Println(g.Greet("Natalie"))
}
// output: Hey there, Natalie
```

Avoid constructors if you can

```
b := tea.NewBrewer(2 * time.Minute)
```

vs

```
b := tea.Brewer{
    SteepTime: 2 * time.Minute,
}
```

Which one is clearer?

Follow conventions

Be similar to the standard library and other popular packages

Don't surprise people

Be obvious, not clever

QUIZ

What should you be instead of clever?

OBVIOUS

SIMPLE!

SHAMELESS


Mat Ryer
Go Programming
Blueprints
Second Edition
Build real-world, production-ready solutions in Go using cutting-edge technology and techniques
Packt

Computers can help too

github.com/alecthomas/gometalinter

goreportcard.com

Go fourth and write beautiful packages


CHEERS

Send me your packages on Twitter

@matryer