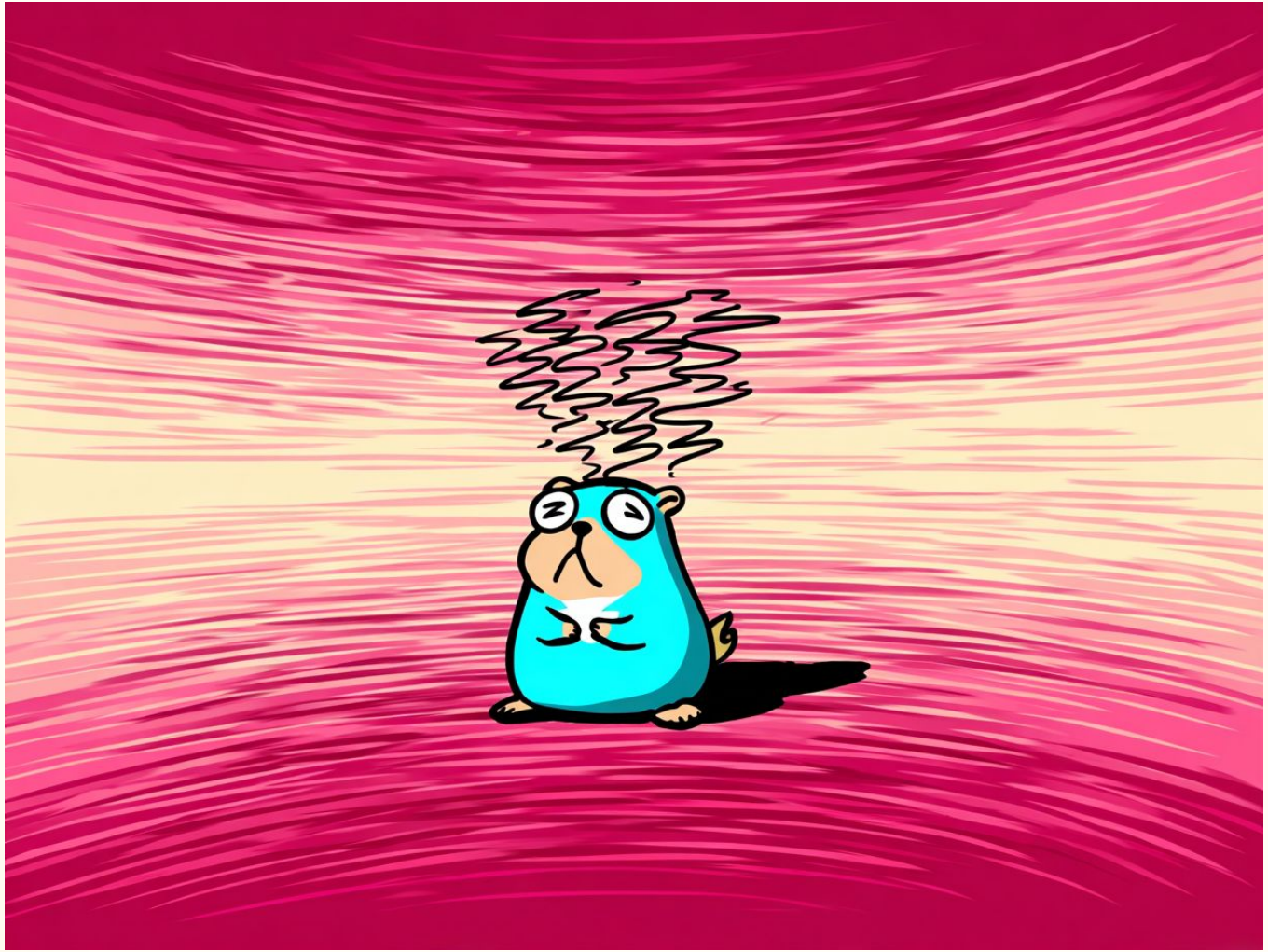
Code generation for 10x productivity - no AI

Andrea Medda Campus
Member of Technical Staff **@Gitpod**



The myth of the 10x engineer

Code generation

Code generation is the process of automatically creating source code, bytecode, or machine code from higher-level specifications, templates, or other code representations.

Language	Features
Java	Annotation Processing (compile time), bytecode manipulation (run time), Template engines, ...
Rust	Macros (compile time), ...
JavaScript	Dynamic execution (run time), Transpilation (build time), ...

Benefits

- Development Speed
- Standardisation and Consistency
- Easy to maintain
- Ideal for scaffolding (API, DB, Testing, ...)

Code Generation + GoLang = 🧡

- Go generate
- Rich packages: go/ast, go/parser, text/template...
- Myriad of tools to be used to be immediately effective

Plan

01 A bit about go generate history

02 Setting up

03 Tools + Hands on

04 Break!

05 Build our own tool

06 Hands on

07 Wrapping up

A bit of history

//go:generate stringer –type=Pill

- Introduced with Go 1.4 (2014)
- Philosophy
  - Programs that write other programs
  - Explicit and not automatic (not during go build)
  - Simplicity
  - Agnostic

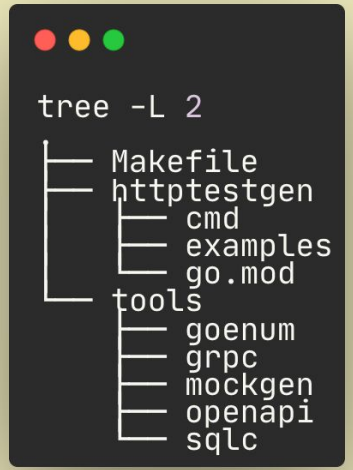
go generate ./...

Techie prep

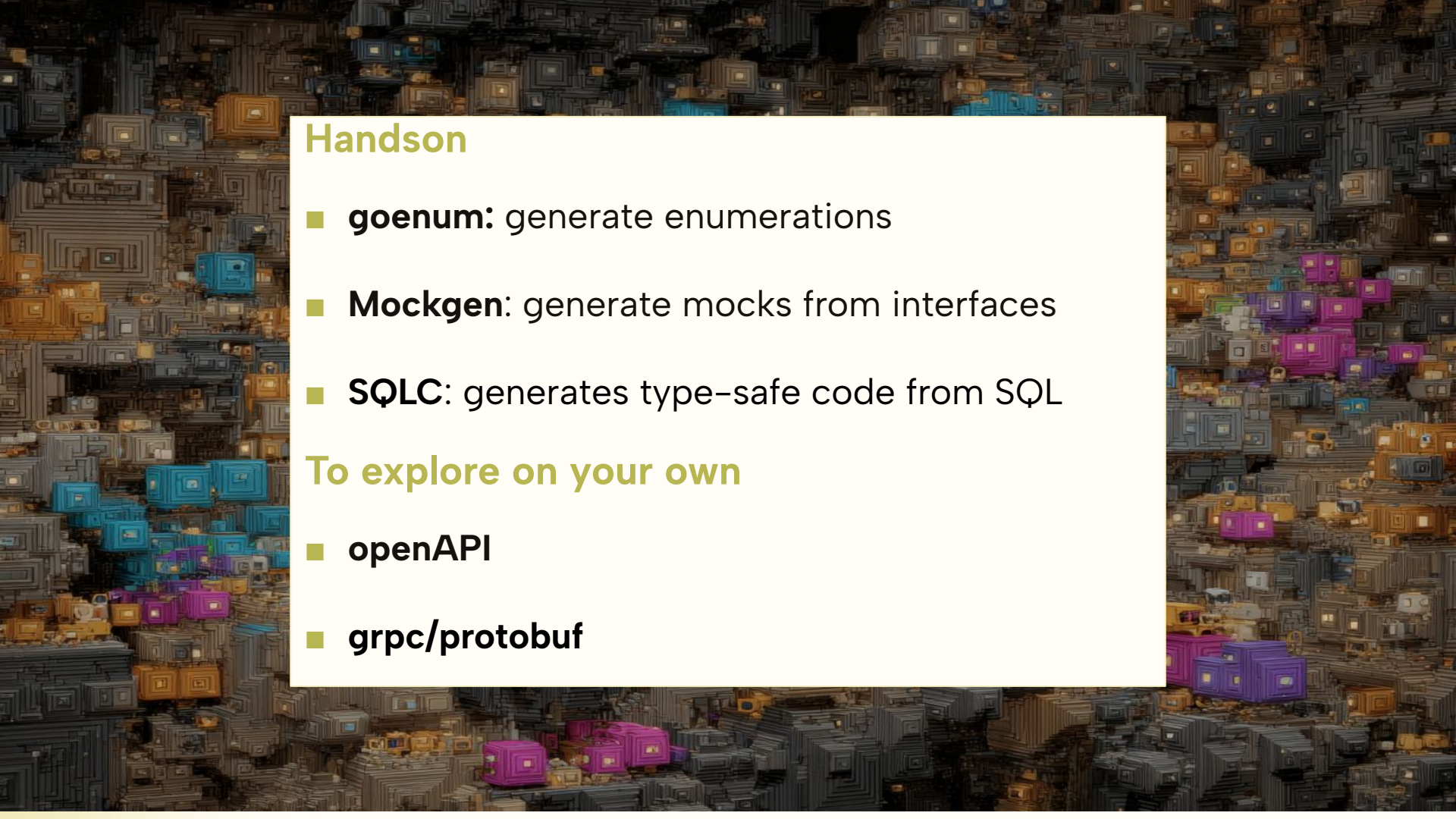
- Connect to the Wi-Fi
- Go ~v1.24 available
- A code editor / IDE
- Git available

Hands on prep

git clone https://github.com/andream16/gophercon-tutorial.git



tools/goenum



Handson

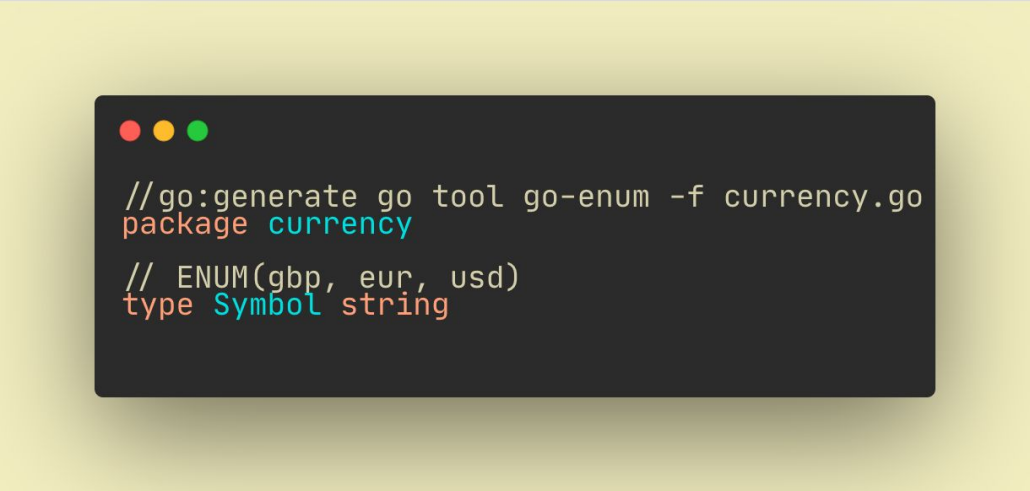
- **goenum:** generate enumerations
- **Mockgen**: generate mocks from interfaces
- **SQLC**: generates type-safe code from SQL

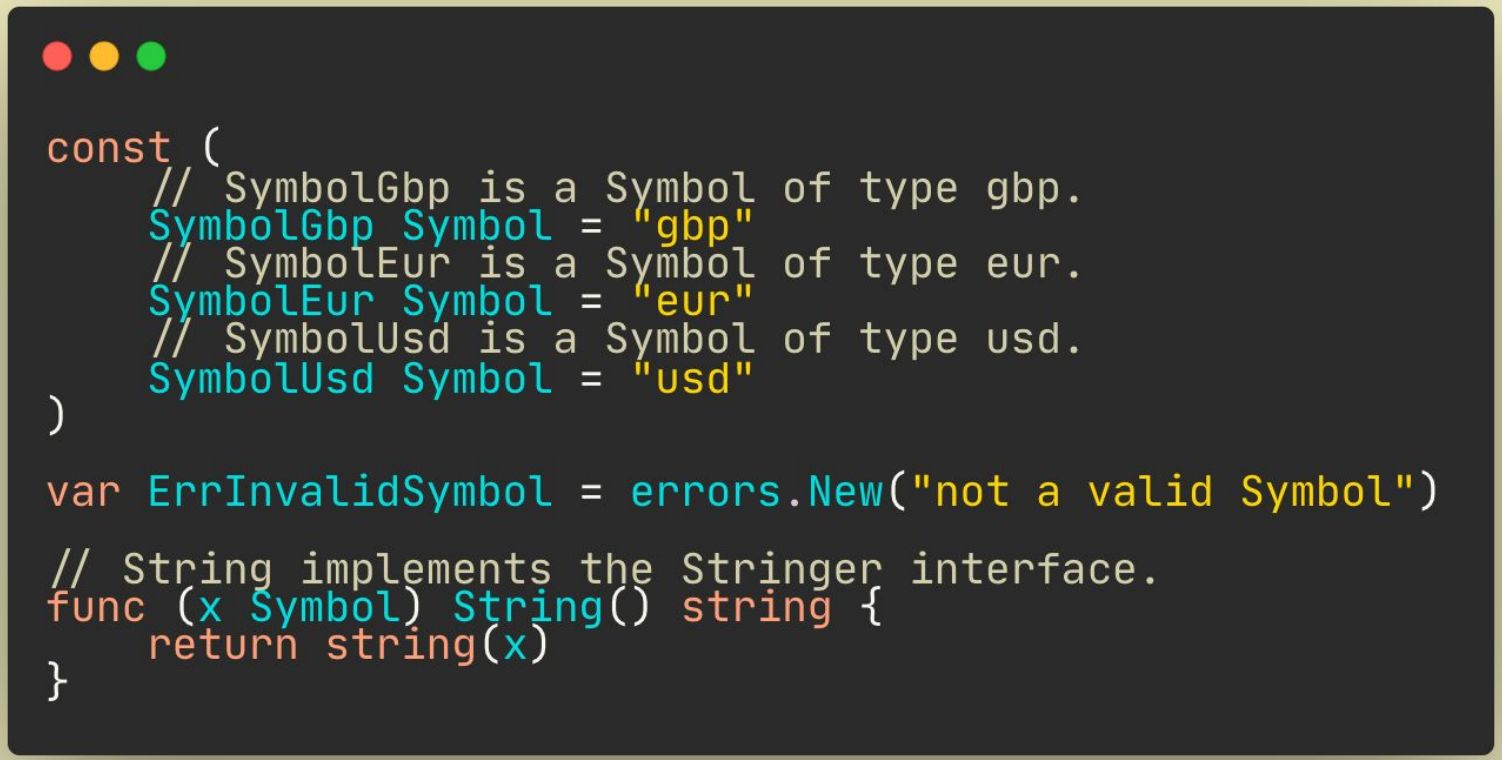
To explore on your own

- **openAPI**
- **grpc/protobuf**

Goenum - github.com/abice/go-enum

Generates enumerations from go types.

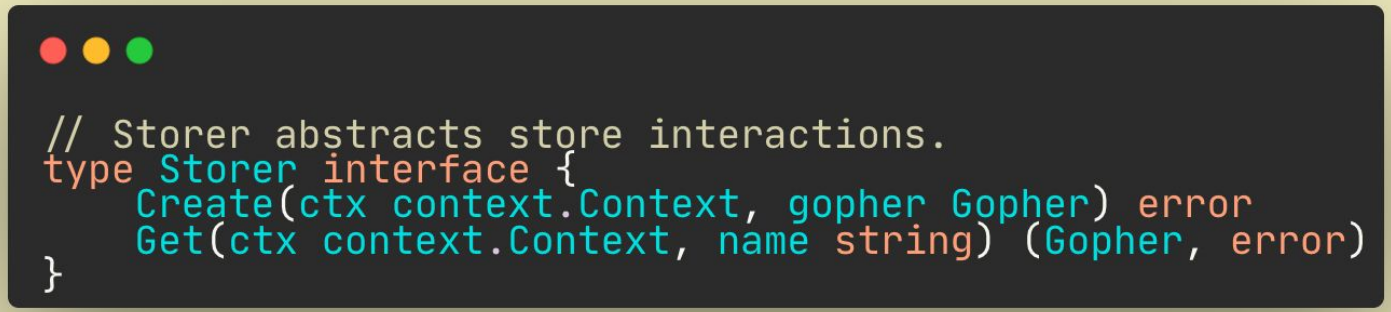




```
const (
    // SymbolGbp is a Symbol of type gbp.
    SymbolGbp Symbol = "gbp"
    // SymbolEur is a Symbol of type eur.
    SymbolEur Symbol = "eur"
    // SymbolUsd is a Symbol of type usd.
    SymbolUsd Symbol = "usd"
)

var ErrInvalidSymbol = errors.New("not a valid Symbol")

// String implements the Stringer interface.
func (x Symbol) String() string {
    return string(x)
}
```

Gomock - github.com/uber-go/mock

Generates mocks from Interface definitions

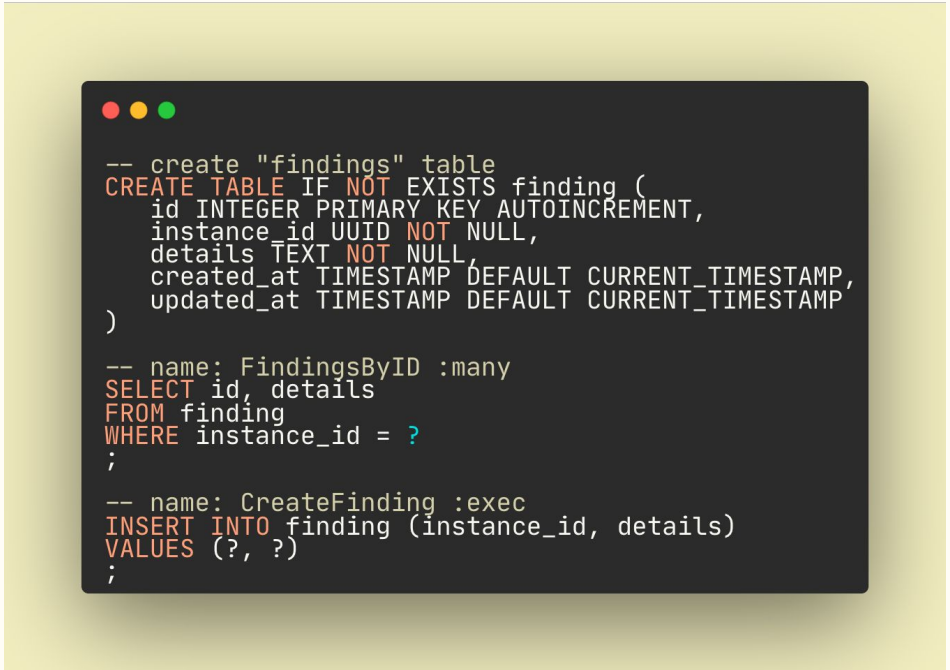


```go
// Storer abstracts store interactions.
type Storer interface {
    Create(ctx context.Context, gopher Gopher) error
    Get(ctx context.Context, name string) (Gopher, error)
}
```

```
var (
    ctx, cancel = context.WithTimeout(context.Background(), time.Second)
    ctrl        = gomock.NewController(t)
    g           = gopher.Gopher{
        Name:  "Ella",
        Color: gopher.BlueColor,
    }
    mockStorer = NewMockStorer(ctrl)
)
defer cancel()

manager, err := gopher.NewManager(mockStorer)
if err ≠ nil {
    t.Fatalf("could not create gopher manager: %v", err)
}

...

t.Run("it errors when something unexpected happens", func(t *testing.T) {
        mockStorer.
            EXPECT().
            Get(ctx, g.Name).
            Return(gopher.Gopher{}, errors.New("something bad happened"))

        if err := manager.Create(
            ctx,
            g,
        ); err == nil {
            t.Fatal("expected an error on get, got none")
        }
    })
```

SQLC - github.com/sqlc-dev/sqlc

Generates type-safe code from SQL



```sql
-- create "findings" table
CREATE TABLE IF NOT EXISTS finding (
   id INTEGER PRIMARY KEY AUTOINCREMENT,
   instance_id UUID NOT NULL,
   details TEXT NOT NULL,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)

-- name: FindingsByID :many
SELECT id, details
FROM finding
WHERE instance_id = ?
;

-- name: CreateFinding :exec
INSERT INTO finding (instance_id, details)
VALUES (?, ?)
;
```

```go
import sqlc "gophercon-tutorial/tools/sqlc/complete/store/sqlc/gen"

type store struct {
  db      *sql.DB
  queries *sqlc.Queries
}

func (s store) ReadFindings(ctx context.Context, id string) ([]Finding, error) {
    rows, err := s.queries.FindingsByID(ctx, id)
    if err != nil {
        return nil, err
    }

    var findings = make([]Finding, 0, len(rows))
    for _, row := range rows {
        findings = append(
            findings,
            Finding{
                ID:         row.ID,
                InstanceID: id,
                Details:    json.RawMessage(row.Details),
            },
        )
    }

    return findings, nil
}
```

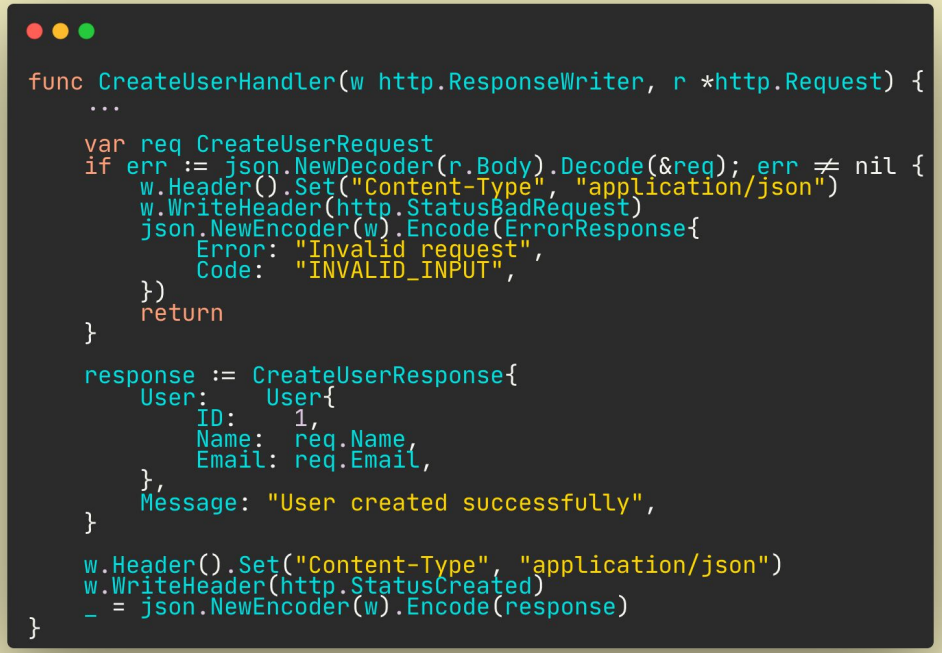
Go generate + Go tool

Go 1.24 Introduced a tool directive in go.mod that allows modules to track executable dependencies.

The key unification happens in how go generate can now leverage go tool. When you run go generate on a file with a //go:generate go tool directive, it will use the version of the tool listed in your go.mod file.

10 Minutes Break!

httptestgen



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

    var req CreateUserRequest
    if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
        w.Header().Set("Content-Type", "application/json")
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode(ErrorResponse{
            Error: "Invalid request",
            Code:  "INVALID_INPUT",
        })
        return
    }

    response := CreateUserResponse{
        User:    User{
            ID:    1,
            Name:  req.Name,
            Email: req.Email,
        },
        Message: "User created successfully",
    }

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)
    _ = json.NewEncoder(w).Encode(response)
}
```

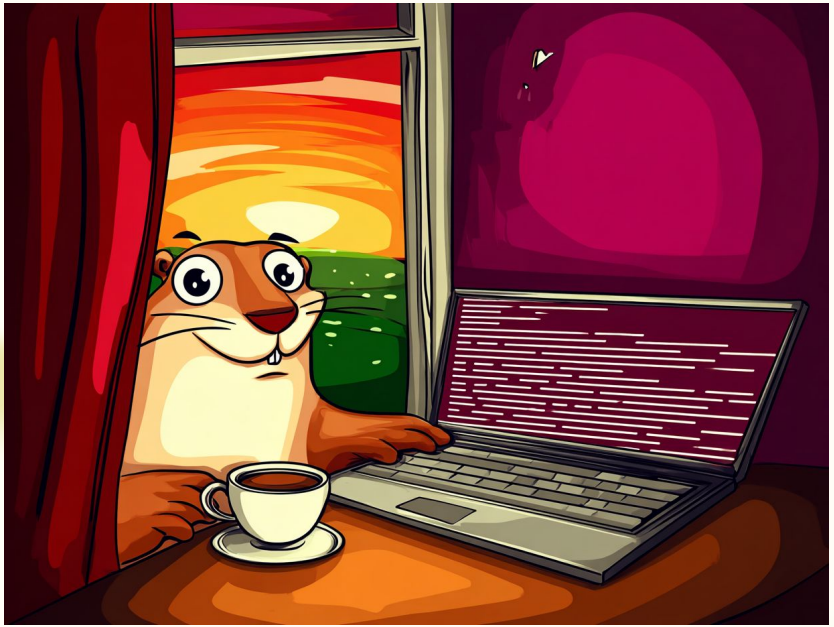
- Testing handlers can be tedious
- Let’s write a tool that generates unit tests for them from a specification

CLI and Specification

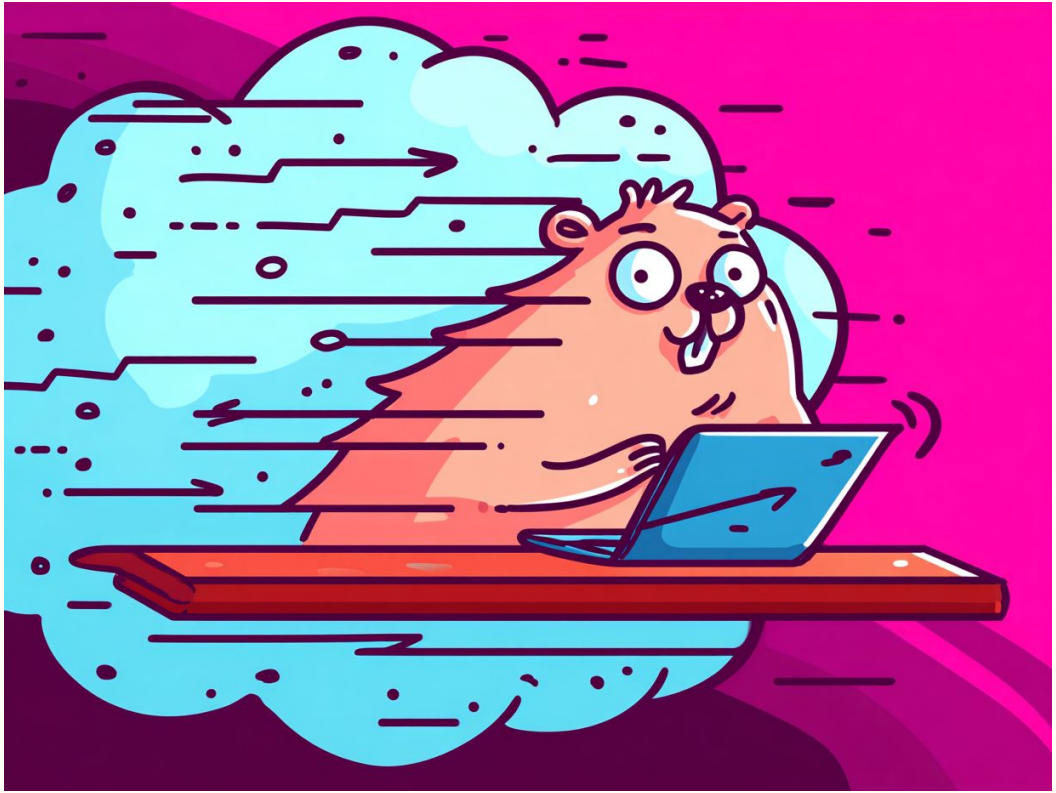
```
//go:generate httptestgen \
//  -input=handler.go \
//  -output=handler_test.go
//  testcases=testdata/testcases.json
```

Let’s build!

- **CLI**: to interact with the tool
- **Parser**: to parse specification and source code
- **Generator**: to template the resulting tests



Extending httptestgen with headers support



The ugly parts

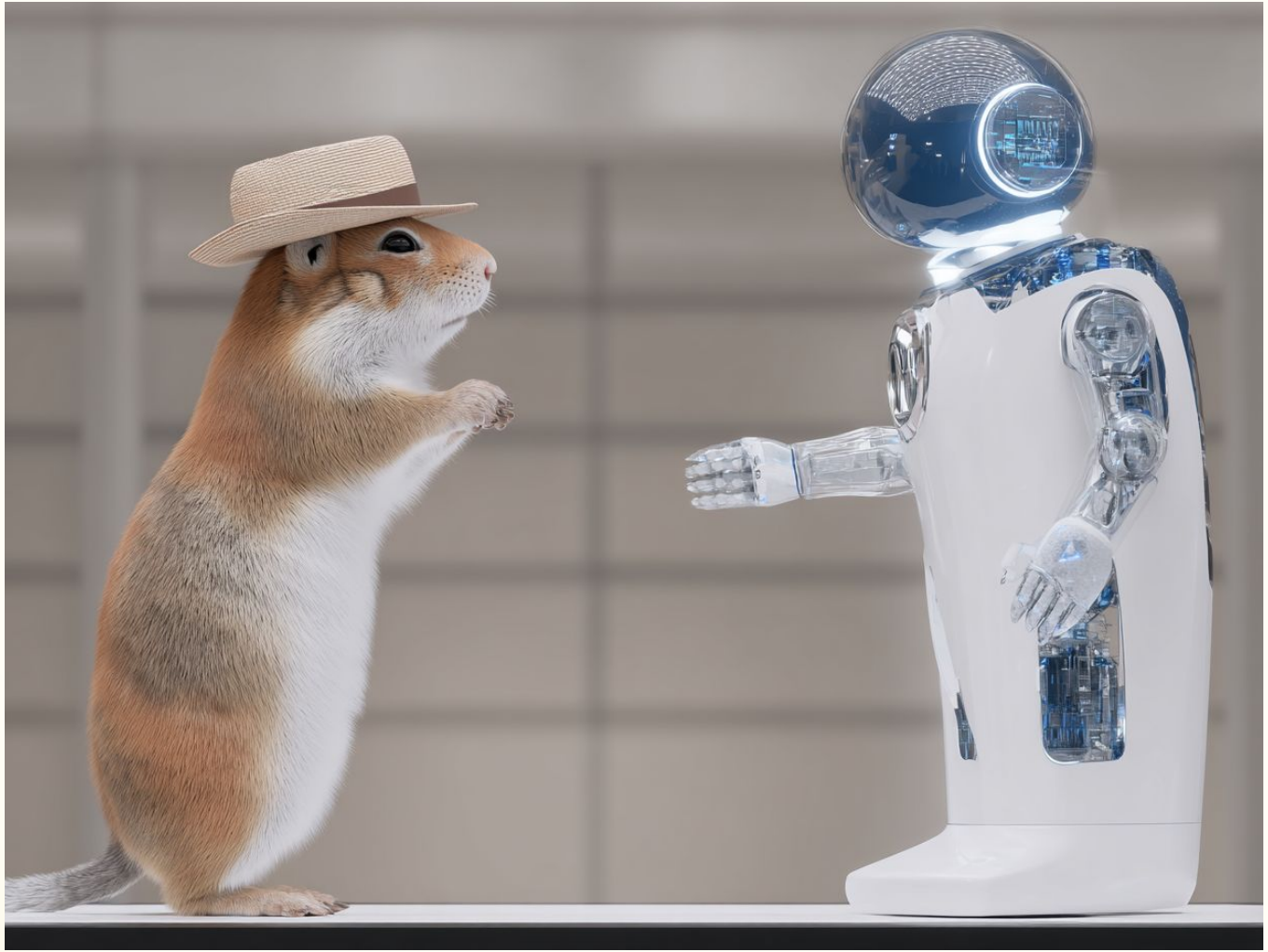
- Accepting generated code
- Understanding and Debugging
- Maintenance and Security
- Review tools

What we learned

- Go generate and generated code can be strong allies to boost productivity
- Go’s ecosystem and tools are perfect for code generation
- Understanding generated code and writing tools to generate it is doable
- We need to be aware of some drawbacks

Other applications

- Templates can be used to generate anything
  - JIRA tickets
  - Emails
  - …
- AST let’s you process and understand Go source code:
  - Security
  - Linting
  - …



Where to go next

- Play around with more tools
- Familiarise with Protobuf and OpenAPI (examples in this repo)
- Be a 10x engineer 😎

- linkedin.com/in/andream16
- github.com/andream16
- me@andreamedda.com

Contacts

Andrea Medda Campus
Member of Technical Staff
@Gitpod


Thank you