Benchmarks. Packages in Go

Session 13

Golang course by Exadel

17 Nov 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Agenda

- ▶️ Benchmarking
- ▶️ Packages in Go
- ▶️ Next time...

Benchmarking

Benchmarks

- ▶️ Documentation: std "testing" package (https://pkg.go.dev/testing#hdr-Benchmarks)
- ▶️ A culture of benchmarking
- ▶️ Easy to measure
- ▶️ Compiler optimisations may eliminate the function under test
- ▶️ Investigate code example: `code/benchexample/fib.go`

Source: "How to write benchmarks in Go" by Dave Cheney (https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go)

Packages in Go

Packages

- ✓ Packages define boundaries for compilation and reuse.
  - Go programs are constructed by linking together packages.
- ✓ Packages are directories containing source code.
  - An implementation may require that all source files for a package inhabit the same directory.
- ✓ A package in turn is constructed from one or more source files that together declare constants, types, variables and functions *belonging to the package* and which are accessible in all files of the same package.
- ✓ The unit of compilation is the **package**, not the file. !
- ✓ You import a whole package, not a type, or a symbol.
- ✓ Import paths should be globally unique, so use the path of your source repository as its base.

Package clause

- ✓ Each Go language source file must contain a package clause defining the package to which it belongs.
- ✓ The "*package name*" **must not** be the '_' blank identifier.
- ✓ A set of files sharing the same package name form the *implementation of a package*.
- ✓ This is a common practice to name package with the same name as the directory where the source code for package located.
  - But the directory name and package name can be different, for example:

```
import "google.golang.org/api/youtube/v3"

func main() {
    youtubeService, err := youtube.New(oauthHttpClient)
}
```

Import declarations (1/3)

- ✓ The interpretation of the "*Import Path*" is *implementation-dependent* but it is typically a substring of the full file name of the compiled package and may be relative to a repository of installed packages.

- ✓ An **import declaration** declares a dependency relation between the importing and imported package.
  - It is *illegal* for a package to import itself, directly or indirectly, or to directly import a package without referring to any of its exported identifiers

Import declarations (2/3)

- ✓ Different variations of importing declarations:
    - The most common way to import:

```go
import   "lib/math"     // can be referenced as: math.Sin
```

- This renaming is used to avoid package name clashing:

```go
import m "lib/math"     // can be referenced as: m.Sin
```

- This is similar to *Java's static imports*
    - Normally, this technique should be avoided.

```go
import . "lib/math"     // can be referenced as: Sin
```

Import declarations (3/3)

✓ Different variations of importing declarations:

- Import without a name:
  - Importing a package solely for its *side-effects* (**initialization**)
    - For example, this type of import can be used to import database drivers implementations

```
import _ "lib/math"     // can not be referenced, just imports
```

- Relative import paths is an import path beginning with `./` or `../`
  - Do not use on source-level!
  - The toolchain supports relative import paths as a shortcut *in two ways*:
    - a relative path can be used as a shorthand on the command line: like "`go test ./...`" to test all subdirectories.
    - if you are compiling a Go program not in a work space, you can use a relative path in an import statement in that program to refer to nearby code also not in a work space. This makes it easy to experiment with small multipackage programs outside of the usual work spaces, but such programs cannot be installed with "go install" (there is no work space in which to install them), so they are rebuilt from scratch each time they are built. To avoid ambiguity, Go programs cannot use relative import paths within a work space.

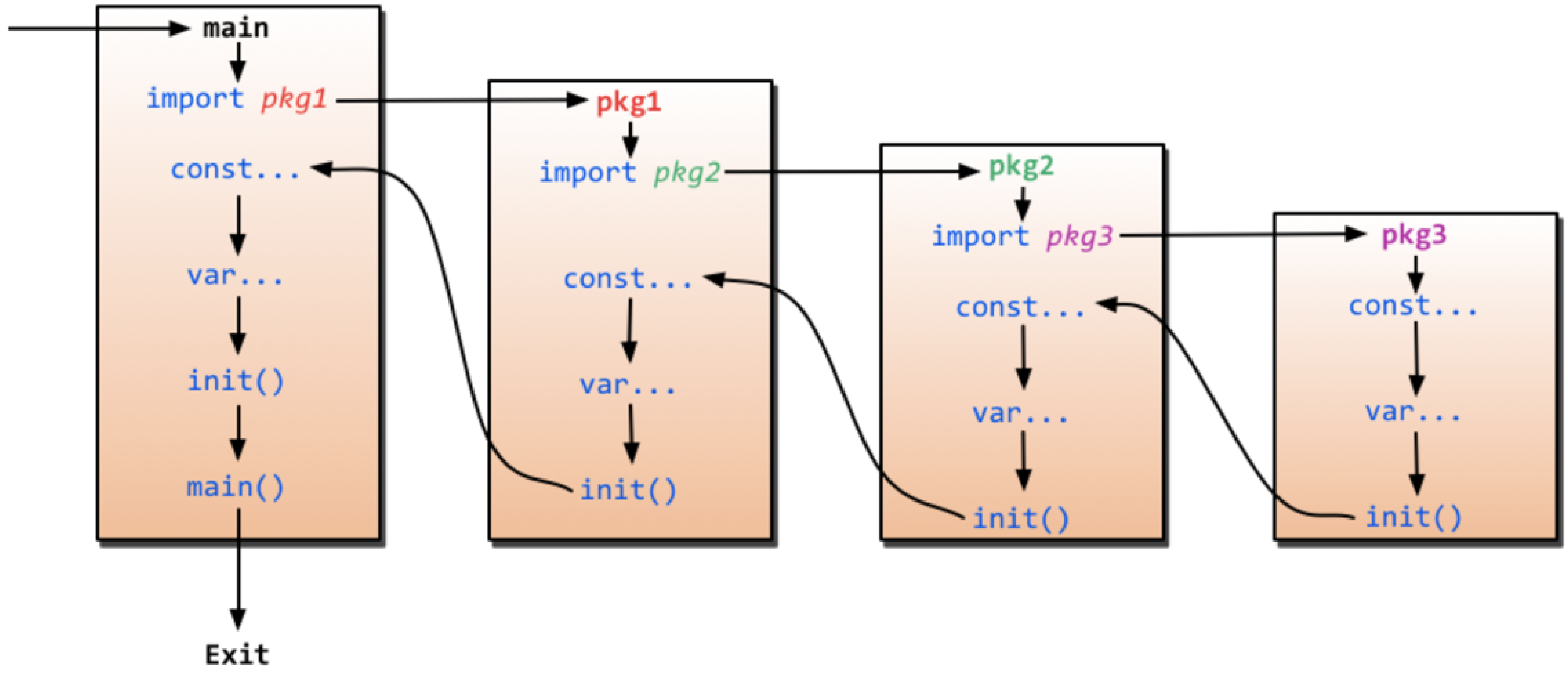
Package Paths

- ✓ A Go package has both a ***name*** and a ***path***.
- ✓ The package *name* is specified in the *package* statement of its source files;
  - client code uses it as the **prefix** for the package's exported names.
- ✓ Client code uses the *package path* when importing the package.
  - By convention, the **last** element of the package path is the package name:

```go
import (
    "fmt"                         // package fmt
    "os/exec"                     // package exec
    "golang.org/x/net/context"    // package context
)
```

- ✓ When renaming an imported package, the local name should follow the same guidelines as package names:
  - lower case, no under_scores or mixedCaps.

Initialization Flow



Source: "Flow of programs initialization in Go" by astaxie (https://astaxie.gitbooks.io/build-web-application-with-golang/content/en/02.3.html#main-function-and-init-function)

Package names (1/3)

✓ Good package names are short and clear. They are lower case, with *no under_scores or mixedCaps*.
  - They are often ***simple nouns***, like:
    - `time` (provides functionality for measuring and displaying time)
    - `list` (implements a doubly linked list)
    - `http` (provides HTTP client and server implementations)
  - Names should be in ***singular form***, not plural.

✓ A Go package may export several types and functions.

Package names (2/3)

- ✓ Abbreviate judiciously.
  - Package names *may be abbreviated* when the abbreviation is familiar to the programmer.
  - Widely-used packages often have compressed names:
    - *strconv* (string conversion)
    - *syscall* (system call)
    - *fmt* (formatted I/O)
- ✓ Don't steal good names from the user.
  - Avoid giving a package a name that is commonly used in client code.
  - For example, the buffered I/O package is called *bufio*, not *buf*, since *buf* is a good variable name for a buffer.
- ✓ A package name and its contents' names are coupled, since client code uses them together.

Package names (3/3)

✓ Avoid stutter.
  - The HTTP server provided by the **http** package is called **Server**, not **HTTPServer**. Client code refers to this type as **http.Server**, so there is no ambiguity.

✓ Simplify function names.
  - A function named **New** in package **pkg** returns a value of type **pkg.Pkg**.
  - This is a standard entry point for client code using that type

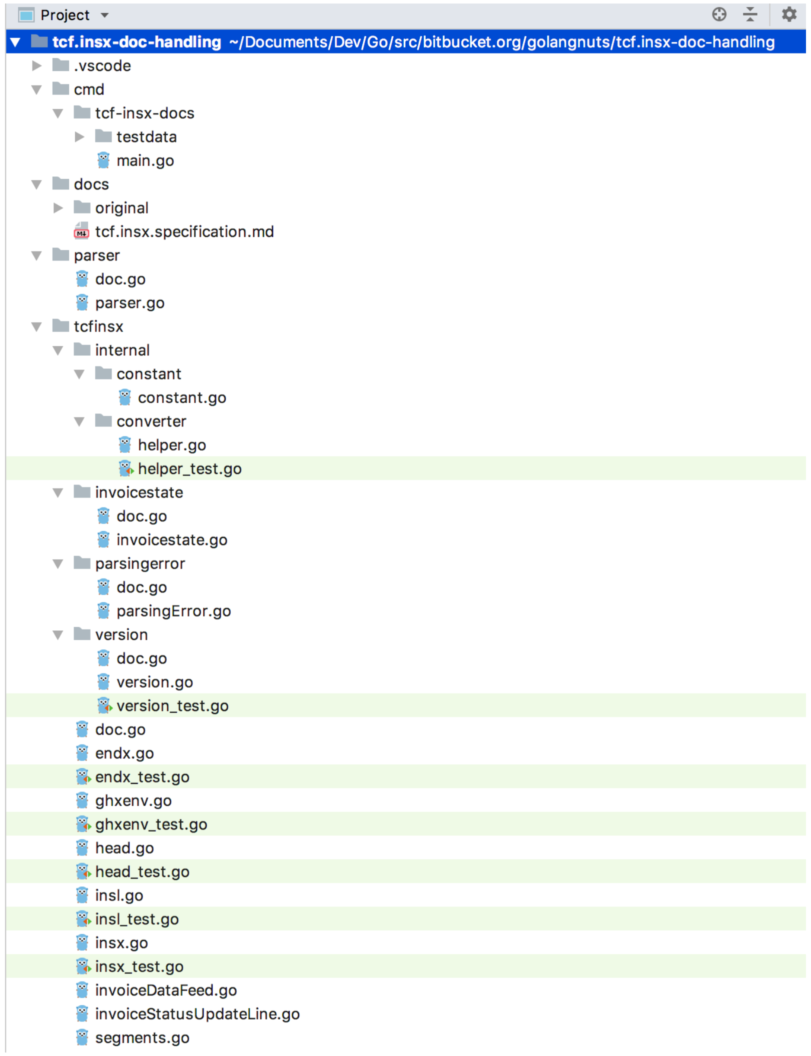
```
q := list.New()  // q is a *list.List
```

  - When a function returns a value of type **pkg.T**, where **T** is not **Pkg**, the function name *may include* **T** to make client code easier to understand.
  - A common situation is a package with multiple **New**-like functions

```
d, err := time.ParseDuration("10s")  // d is a time.Duration
elapsed := time.Since(start)         // elapsed is a time.Duration
ticker := time.NewTicker(d)          // ticker is a *time.Ticker
timer := time.NewTimer(d)            // timer is a *time.Timer
```

✓ If you cannot come up with a package name that's a meaningful prefix for the package's contents, the package abstraction boundary may be wrong.

My experience: bad package composition



```
Project
▼ tcf.insx-doc-handling ~/Documents/Dev/Go/src/bitbucket.org/golangnuts/tcf.insx-doc-handling
  ▶ .vscode
  ▼ cmd
    ▼ tcf-insx-docs
      ▶ testdata
        main.go
  ▼ docs
    ▶ original
      tcf.insx.specification.md
  ▼ parser
      doc.go
      parser.go
  ▼ tcfinsx
    ▼ internal
      ▼ constant
          constant.go
      ▼ converter
          helper.go
          helper_test.go
    ▼ invoicestate
        doc.go
        invoicestate.go
    ▼ parsingerror
        doc.go
        parsingError.go
    ▼ version
        doc.go
        version.go
        version_test.go
      doc.go
      endx.go
      endx_test.go
      ghxenv.go
      ghxenv_test.go
      head.go
      head_test.go
      insl.go
      insl_test.go
      insx.go
      insx_test.go
      invoiceDataFeed.go
      invoiceStatusUpdateLine.go
      segments.go
```

Internal Directories

✓ Code in or below a directory named "**internal**" is importable only by code in the directory tree rooted at the parent of "internal".

✓ Example:

```
/home/user/go/
    src/
        crash/
            bang/              (Go code in package `bang`)
                b.go
        foo/                   (Go code in package `foo` can import "foo/internal/baz")
            f.go
            bar/               (Go code in package `bar` can import "foo/internal/baz")
                x.go
            internal/
                baz/           (Go code in package `baz` can be imported only for Go source files in the
subtree rooted at `foo`)
                    z.go
            quux/              (Go code in package `quux` can import "foo/internal/baz")
                y.go
```

Vendor Directories

- ✓ Code below a directory named "**vendor**" is importable only by code in the directory tree rooted at the parent of "**vendor**", and only using an import path that omits the prefix up to and including the **vendor** element.

- ✓ Code in vendor directories deeper in the source tree shadows code in higher directories.
  - This is recursive behaviour.

- ✓ Within the subtree rooted at foo, an import of "crash/bang" resolves to "foo/vendor/crash/bang", not the top-level "crash/bang”:

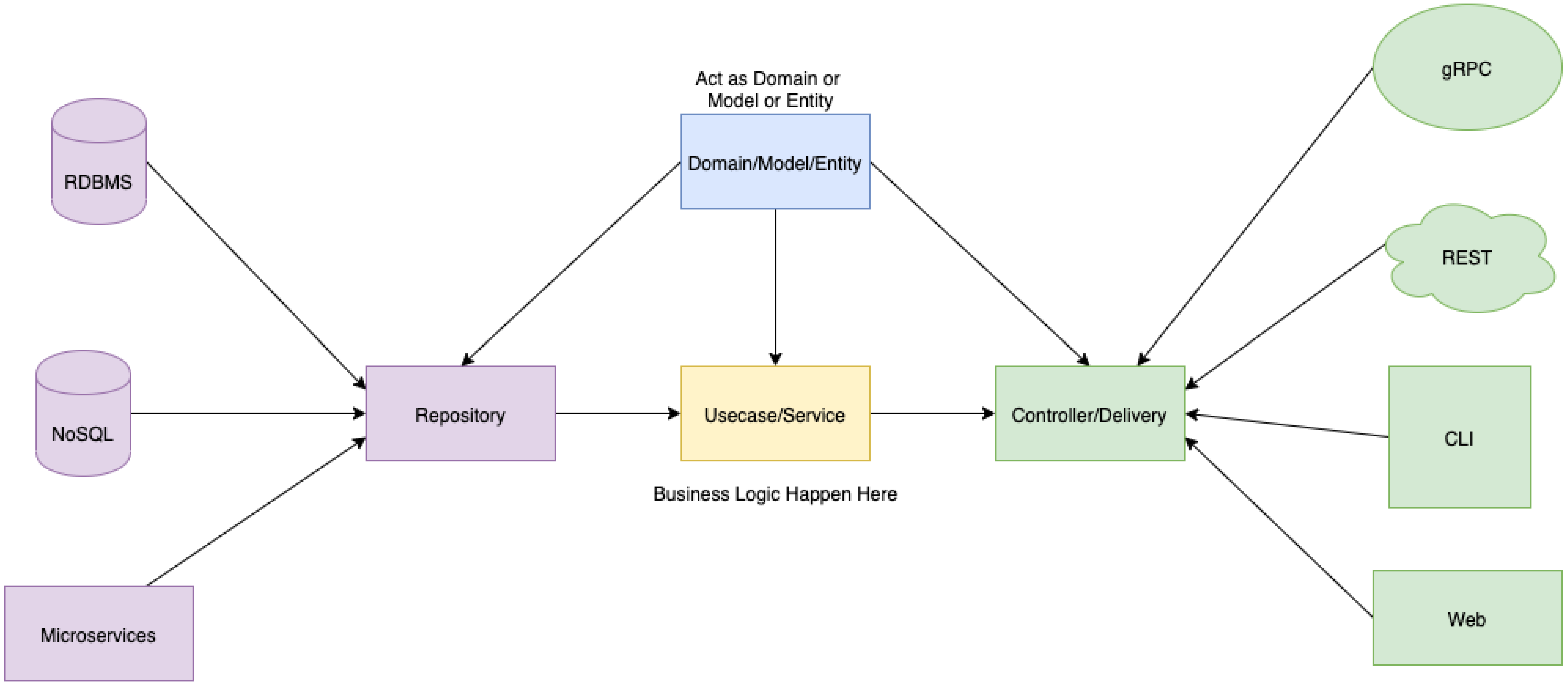
```
/home/user/go/
    src/
        crash/
            bang/               (Go code in package `bang`)
                b.go
        foo/                    (Go code in package `foo`)
            f.go
            bar/                (Go code in package `bar`)
                x.go
            vendor/             (Vendored folder)
                crash/
                    bang/       (code in `b.go` is imported as “crash/bang”, not as "foo/vendor/crash/bang")
                        b.go
                baz/            (code in `z.go` is imported as "baz", not as "foo/vendor/baz")
                    z.go
            quux/               (Go code in package `quux`)
                y.go
```

Circular dependencies

- ✓ When *two packages need to import each other*, it’s a smell that the packages need to be **merged** or **decoupled.**

- ✓ Go *doesn’t allow circular dependencies.*
  - So, if you have packages **A -> B -> C -> A** (*so-called transitive dependencies*)

Package decomposition example



Source: "https://github.com/bxcodec/go-clean-arch" (https://github.com/bxcodec/go-clean-arch)

Homework:

- ▶️ "How to write benchmarks in Go" by Dave Cheney (https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go)
- ▶️ "Package names" by Sameer Ajmani (https://go.dev/blog/package-names)
- ▶️ "Style guideline for Go packages" by Jaana Dogan (https://rakyll.org/style-packages/)
- ▶️ "Dealing With Import Cycles in Go" by Mantesh Jalihal (http://mantish.com/post/dealing-with-import-cycle-go/)
- ▶️ Play with sourcecode: "github.com/bxcodec/go-clean-arch" (https://github.com/bxcodec/go-clean-arch)

Next time...

▶️ Session14:

Modules in Go

- Adding new dependencies
- Go.mod anatomy
- Versioning strategies, V2+
- Bumping version of your own code
- Go Workspaces

Thank you

Golang course by Exadel

17 Nov 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)