



# Golang Programming

Modules and dependency management using go mod

# Where to Find The Code and Materials?

<https://github.com/iproduct/coursego>

# Agenda for This Session

- Modules in Go
- go.mod file
- Version selection
- Semantic import versioning
- Installing and activating module support
- Defining a module
- Upgrading and downgrading dependencies
- Main go mod commands
- Daily workflow.
- Hands-on: RESTful service using [gorilla/mux](#), [go-playground/validator.v9](#), [jwt-go](#), and MySQL ([go-sql-driver/mysql](#))
- Projects mentoring.



# Modules in Go



# Modules in Go

- Go projects use a wide variety of **dependency management** strategies:
  - Some projects store their entire **GOPATH** directory in a single **Git** repository.
  - Others simply rely on `go get` and expect fairly recent versions of dependencies to be installed in **GOPATH**.
  - Vendoring tools such as **dep**, **godep**, **govendor**, and **glide** are popular, but they have wide differences in behavior and don't always work well together.
- **Go's module system**, introduced in **Go 1.11**, provides an **official dependency management solution** built into the **go** command.

# Go Code Organization (1)

- Go programs are organized into packages - collection of source files in the same directory that are compiled together.
- A repository contains one or more modules
- A module is a collection of related Go packages that are released together.
- Go repository typically contains only one module, located at the root of the repository. A file named **go.mod** there declares the **module path**: the import **path prefix** for all packages within the module.
- The module contains the packages in the directory containing its **go.mod** file as well as subdirectories of that directory, **up to the next subdirectory containing another go.mod file** (if any).

## Go Code Organization (2)

- You don't need to publish your code to a remote repository. A module can be defined locally without belonging to a repository. However, it's a good habit to organize your code as if you will publish it someday.
- Each module's path not only serves as an [import path prefix](#) for its packages, but also indicates [where the go command should look to download it](#). For example, in order to download the module [golang.org/x/tools](#), the go command would consult the repository indicated by [https://golang.org/x/tools](#).
- A package's [import path](#) is its [module path](#) joined with its [subdirectory](#) within the module. E.g. the module [github.com/google/go-cmp](#) contains a package in the directory [cmp/](#). That package's import path is [github.com/google/go-cmp/cmp](#). Packages in the standard library do not have a module path prefix.

# go.mod file

```
module github.com/iproduct/coursego/modules
```

```
go 1.13
```

```
require (  
    github.com/dgrijalva/jwt-go v3.2.0+incompatible  
    github.com/go-playground/locales v0.13.0  
    github.com/go-playground/universal-translator v0.17.0  
    github.com/go-playground/validator/v10 v10.1.0  
    github.com/go-sql-driver/mysql v1.5.0  
    github.com/gorilla/mux v1.7.3  
    golang.org/x/crypto v0.0.0-20200208060501-ecb85df21340  
)
```



# go.sum file

```
github.com/leodido/go-urn v1.2.0 h1:hpXL4XnriNwQ/ABnpepYM/1vCLWNDfUNts8dX3xTG6Y=
github.com/leodido/go-urn v1.2.0/go.mod h1:+8+nEpDfqqsY+g338gtMEU0tuK+4dEMhiQEgpxp0KII=
github.com/pmezard/go-difflib v1.0.0 h1:4DBwDE0NGyQoBHBbLQYPwSUPoCMWR5BEzIk/f1lZbAQM=
github.com/pmezard/go-difflib v1.0.0/go.mod h1:iKH77koFhYxTK1pcRnkKkqfTogsbg7gZNVY4sRDYZ/4=
github.com/stretchr/objx v0.1.0/go.mod h1:HFkY916IF+rwdDfMAkV70twuqBVzrE8GR6GFx+wExME=
github.com/stretchr/testify v1.4.0 h1:2E4SXV/wtOkTonXsotYi4li6zVWxYlZuYNCXe9XRJyk=
github.com/stretchr/testify v1.4.0/go.mod h1:j7eGeouHqKxXV5pUuKE4zz7dFj8WfuZ+81PSLYec5m4=
golang.org/x/crypto v0.0.0-20190308221718-c2843e01d9a2/go.mod h1:djNgcEr1/C05ACkg1iLfiJU5Ep61QUkGW8qpdssI0+w=
golang.org/x/crypto v0.0.0-20200208060501-ecb85df21340 h1:K0cEaR10tFr7gdJV2GCKw80s5yED1u1a0qHjOAb6d2Y=
golang.org/x/crypto v0.0.0-20200208060501-ecb85df21340/go.mod h1:LzIPMQfyMNhhGPhUkY0s5KpL4U8rLKemX1yGLhDgUto=
golang.org/x/net v0.0.0-20190404232315-eb5bcb51f2a3/go.mod h1:t9HGtf8HONx5eT2rt7q6eTqICYqUVnKs3thJo3Qplg=
golang.org/x/sys v0.0.0-20190215142949-d0b11bdaac8a/go.mod h1:STP8DvDyc/dI5b8T5hshtkJs+E42TnysNCUPdjciGhY=
golang.org/x/sys v0.0.0-20190412213103-97732733099d/go.mod h1:h1NjWce9XRLGQEsW7wpKNCjG9DtNlClVuFLEZdDNbEs=
golang.org/x/text v0.3.0/go.mod h1:NqM8EUOU14njkJ3fqMW+pc6Ldnwhi/IjpwHt7yyuw0Q=
golang.org/x/text v0.3.2/go.mod h1:bEr9sfX3Q8Zfm5fL9x+3itogRgK3+ptLWKqgva+5dAk=
golang.org/x/tools v0.0.0-20180917221912-90fa682c2a6e/go.mod h1:n7NCudcB/nEzzVGmLbDWY5pfWTLqBcC2KZ6jyYvM4mQ=
gopkg.in/check.v1 v0.0.0-20161208181325-20d25e280405 h1:yhCVgyC4o1eVCA2tZl7eS0r+SDo693bJlVd1lGtEeKM=
gopkg.in/check.v1 v0.0.0-20161208181325-20d25e280405/go.mod h1:Co6ibVJAznAaIkqp8huTwlJQCZ016jof/cbN4VW5Yz0=
gopkg.in/yaml.v2 v2.2.2 h1:ZCJp+EgiOT7lHqUV2J862kp8Qj64Jo6az82+3Td9dZw=
gopkg.in/yaml.v2 v2.2.2/go.mod h1:hI93XBmqTisBFMUTm0b8Fm+jr3Dg1NNxqwp+5A1VGuI=
```

# go.mod file

- Module version is defined by source files tree, with a **go.mod** file in its root.
- When the **go** command is run, it looks in the **current directory** and then successive **parent directories** to find the **go.mod** marking the root of the **main (current) module**.
- The go.mod file itself is line-oriented, with `//` comments but no `/* */` comments. Each line holds a single directive, made up of a verb followed by arguments. For example:

```
module my/thing
go 1.12
require other/thing v1.0.2
require new/thing/v2 v2.3.4
exclude old/thing v1.2.3
replace bad/thing v1.4.5 => good/thing v1.4.5
```

# Verbs in go.mod

- **module**, to define the module path;
- **go**, to set the expected language version;
- **require**, to require a particular module at a given version or later;
- **exclude**, to exclude a particular module version from use; and
- **replace**, to replace a module version with a different module version.

# Semantic Versioning – e.g. v1.2.3

- **Major Versions** - All the versions within a particular major version should be backwards compatible with other minor and patch versions. Incrementing this typically tells other developers using your package that you have made some breaking changes to how your package works.
- **Minor Versions** - Developers tend to increment minor versions of their package or application when they have added new functionality, or new features to the package whilst maintaining backwards compatibility within the rest of the application.
- **Patch Versions** - Patch versions are typically used for general bug-fixes. If a developer notices a slight issue or bug within their application, they can fix the issue whilst again ensuring backwards compatibility and then increment the patch version by one to indicate new bug fixes.

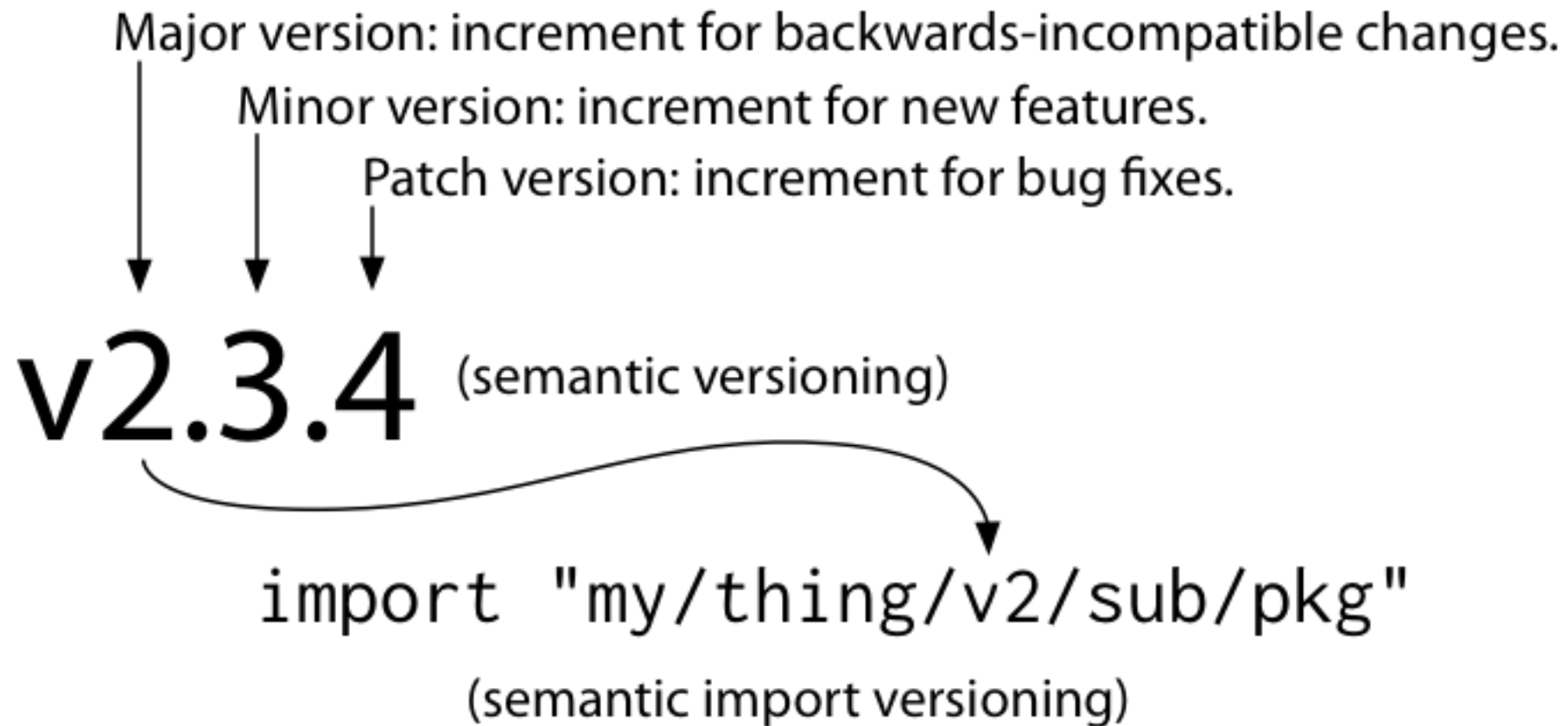


# Import Compatibility Rule in Go

If an **old package** and a **new package** have the **same import path**, the new package **must be backwards compatible** with the old package.

# Version selection – Semantic Import Versioning

- is an architecture for accessing distributed hypermedia web-services



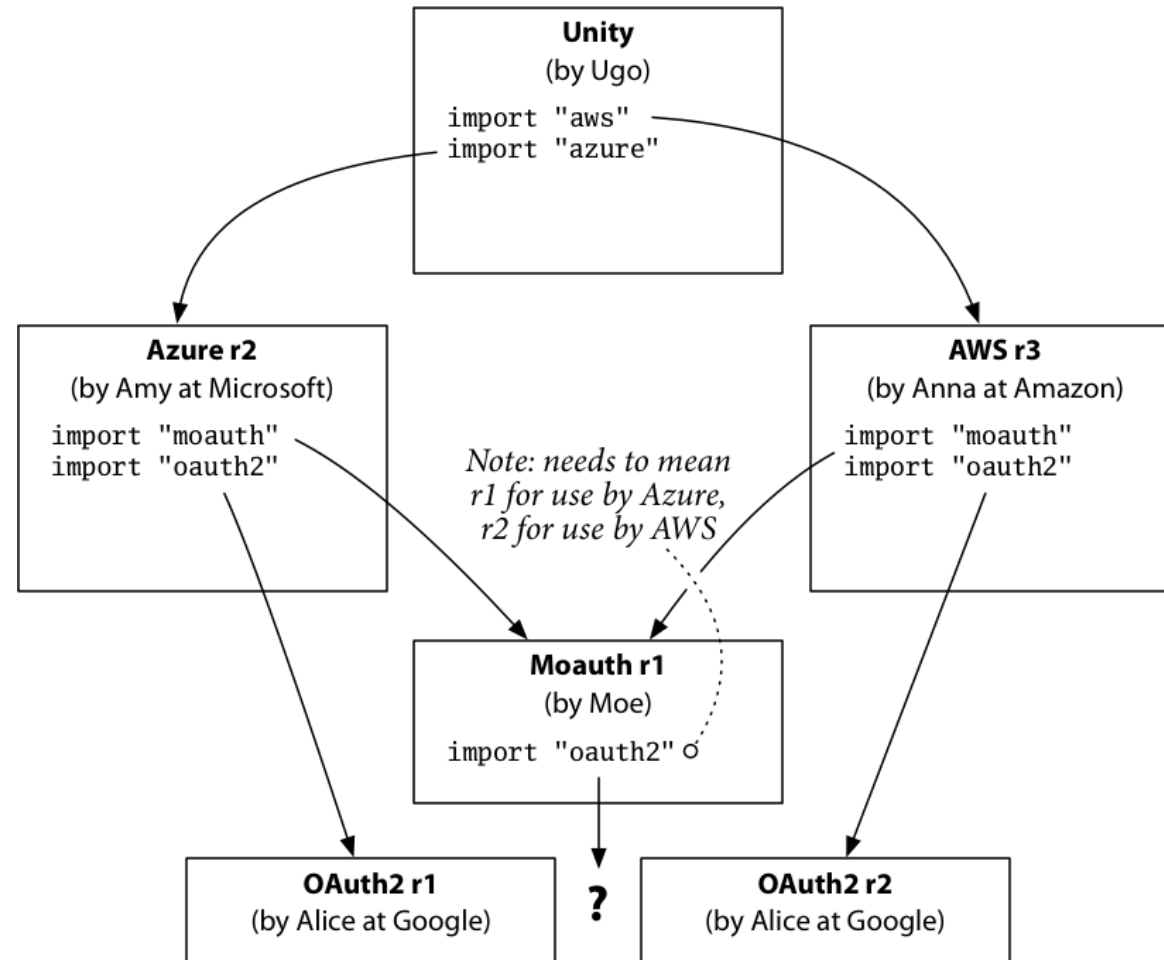
# Semantic import versioning (1)

<https://research.swtch.com/vgo-import>

- A module **my/thing** is imported as:
- **my/thing** for **v0** – the incompatibility period when breaking changes are expected and not protected against
- **my/thing** during **v1** – the first stable major version
- **my/thing/v2** for **v2** – we give it a new name, instead of redefining the meaning of the now-stable **my/thing**.

# Semantic import versioning (2)

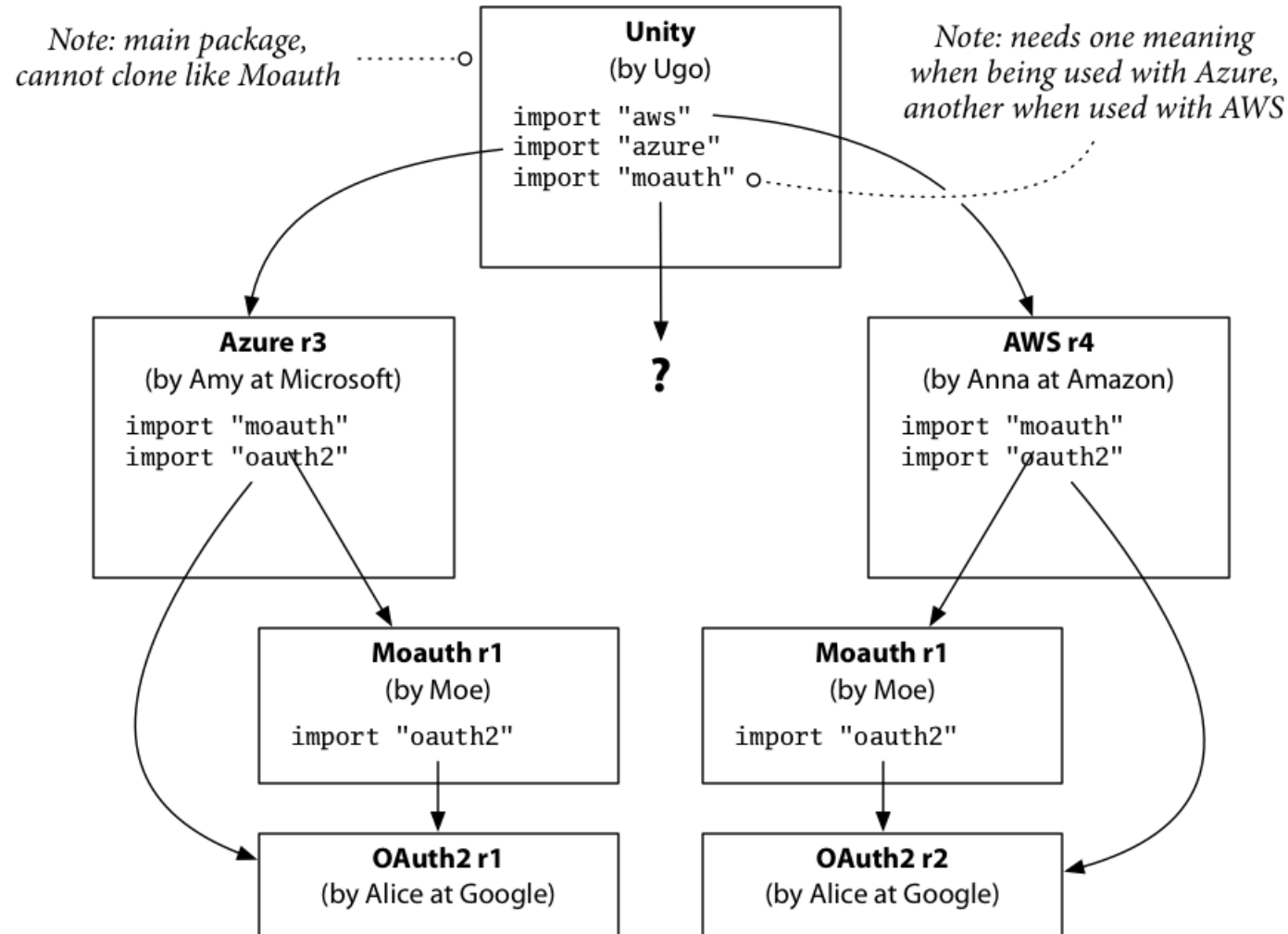
<https://research.swtch.com/vgo-import>





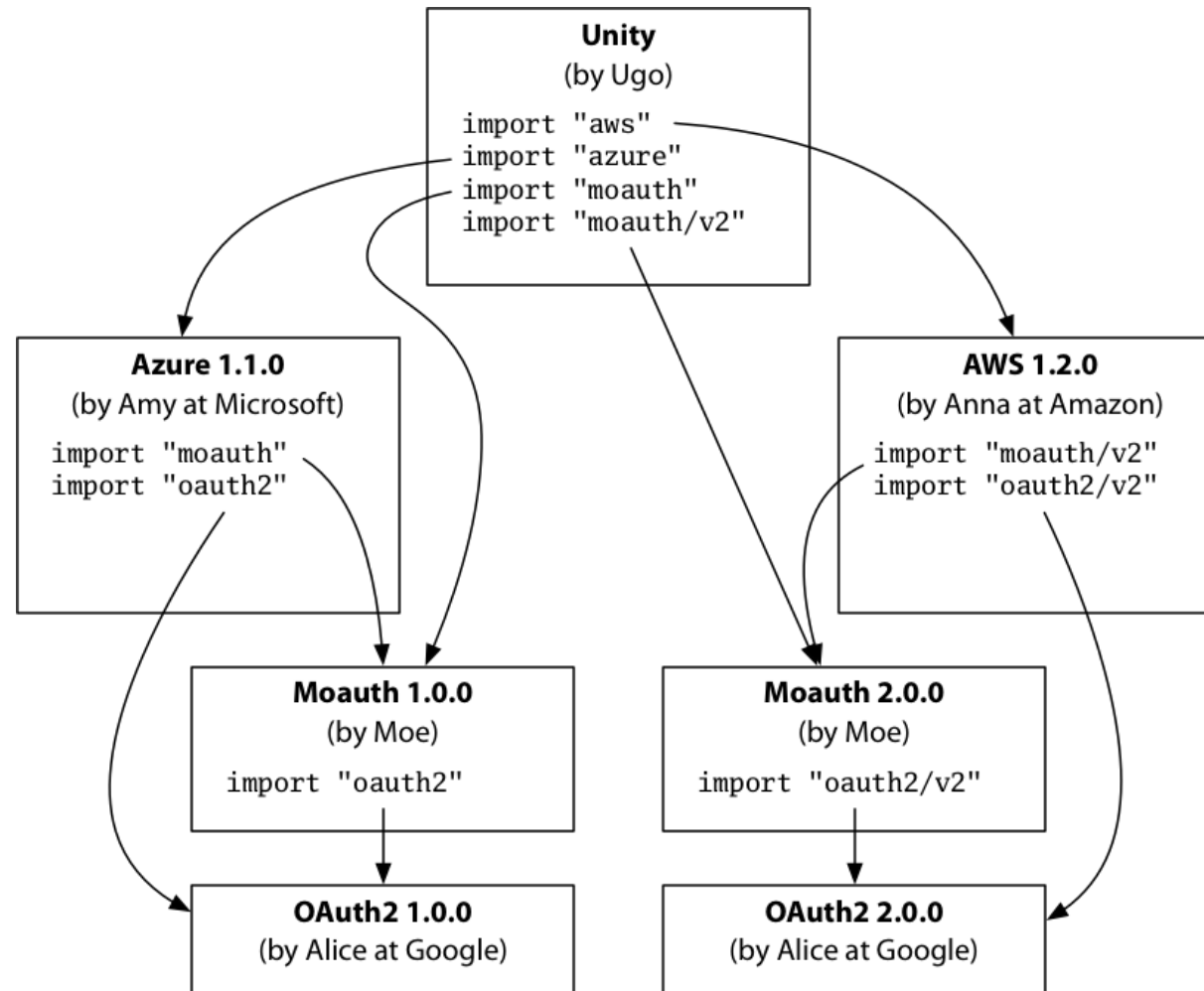
# Semantic import versioning (3)

<https://research.swtch.com/vgo-import>



# Semantic import versioning (4)

<https://research.swtch.com/vgo-import>



# go.mod file versions

```
module M
```

```
    require (  
        A v1  
        B v1.0.0  
        C v1.0.0  
        D v1.2.3  
        E dev  
    )
```

```
exclude D v1.2.3
```

# Installing and activating module support

- Go in some directory outside of `GOPATH` (e.g. home directory) and create a folder for our app called `module-demo` and set up my app there (instead of inside `GOPATH` we did before). Cache is in `$GOPATH/pkg/mod`

- Add dependency – e.g.:

```
package main
import (
    "fmt"
    "github.com/Pallinder/go-randomdata"
)
func main(){
    fmt.Println("Running the TestApp")
    fmt.Println(randomdata.SillyName())
}
```

- `go mod init /root/TestApp` → creates `go.mod` file
- `go mod tidy` → fetches and cleans-up the dependencies



# More go commands using module graph

- Because the module graph defines the meaning of import statements, any commands that load packages also use and therefore update **go.mod**, including:
  - go build
  - go get
  - go install
  - go list
  - go verify
  - go test
  - go mod graph,
  - go mod tidy,
  - go mod why
- Examples: `go mod why -m <module>`, `go mod vendor`, `go list -m all`, `go list -f='{{.Dir}}' -m all` – lists module directories

# Upgrading and downgrading dependencies

```
$ go get golang.org/x/text
go: finding golang.org/x/text v0.3.0
go: downloading golang.org/x/text v0.3.0
go: extracting golang.org/x/text v0.3.0
$ go test
PASS
ok      example.com/hello    0.013s
$
```

- The **golang.org/x/text** package has been upgraded to the **latest tagged version** (v0.3.0).
- The **go.mod** file has been updated to specify **v0.3.0** too.
- The **indirect** comment indicates a dependency is not used directly by this module, only indirectly by other module dependencies.
- See **go help modules** for details.

# Most used go mod commands

- **go mod init** – creates a new module, initializing the go.mod file that describes it.
- **go build** – go test, and other package-building commands add new dependencies to go.mod as needed.
- **go list -m all** – prints the current module's dependencies.
- **go get** – changes the required version of a dependency (or adds a new dependency).
- **go mod tidy** – removes unused dependencies.

# Hands-on: RESTful Service with Modules, TDD, Persistence, Validation & Security

- Projects @Github:
  - <https://github.com/iproduct/coursego/tree/master/modules>
  - <https://github.com/iproduct/coursego/tree/master/modules-auth>
- Used modules:
  - [gorilla/mux](#)
  - [go-playground/validator.v9](#)
  - [jwt-go](#)
  - MySQL: [go-sql-driver/mysql](#)



# Recommended Literature

- The Go Documentation - <https://golang.org/doc/>
- The Go Bible: Effective Go - [https://golang.org/doc/effective\\_go.html](https://golang.org/doc/effective_go.html)
- David Chisnall, *The Go Programming Language Phrasebook*, Addison Wesley, 2012
- Alan A. A. Donovan, Brian W. Kernighan, *The Go Programming Language*, Addison Wesley, 2016
- Nathan Youngman, Roger Peppé, *Get Programming with Go*, Manning, 2018
- Naren Yellavula, *Building RESTful Web Services with Go*, Packt, 2017

# Thank's for Your Attention!



Trayan Iliev

IPT – Intellectual Products & Technologies

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>