Building ~~spacehip~~ simple API with Go

Setup

To clone the project from GitHub, follow this link and copy the project url. Go back to your terminal window and execute following command:

```
git clone https://github.com/Line-39/go-micro-service.git
cd go-micro-service
ls -ahl
```

Project structure

If everything went right, you will see following output:

```
total 52K
drwxrwxr-x 3 ubot ubot 4.0K Apr  3 17:11 .
drwxrwxr-x 3 ubot ubot 4.0K Apr  3 17:00 ..
drwxrwxr-x 8 ubot ubot 4.0K Apr  3 17:10 .git
-rw-rw-r-- 1 ubot ubot  478 Apr  3 17:00 .gitignore
-rw-rw-r-- 1 ubot ubot  35K Apr  3 17:00 LICENSE
-rw-rw-r-- 1 ubot ubot 1.4K Apr  3 17:37 README.md
```

Project structure

If you see different output, make sure that you are on the branch `00-setup`, and switch to this branch otherwise:

```
git switch 00-setup
```

Alternative: create your own folder

In your *working directory* create the project folder, and change into it:

```
mkdir go-micro-service
cd go-micro-service
```

Initialize *git repository*, add `README.md`:

```
git init
echo -e "# Simple microservice with Golang\nAdd your description..." > README.md
```

Alternative: commit your changes

If you are working alone, on manually created project, commit your changes as shown below:

```
git add --all
git commit -m 'Initializing repository'
git switch -b 00-setup
```

Installation

If not yet installed, follow the official guide to install Go on your system.

Verify the installation:

```
go version
# go version go1.22.1 linux/amd64
```

You are ready to Go!

Setup the project

If you are working with clonned project, switch to the next branch (`02-first-programm`):

```
git switch 02-first-programm
```

Setup the project (continued)

Go *package* is a collection of functions, types, variables and constants defined in source files located at the same directory, functionally related and *visible* to each other.

Go *module* is a collection of one or more related *packages*. The `go.mod` located in the module directory, defines the *paths* for all packages used by module.

Go *repository* is composed from the different modules and can be compiled into the *application* providing the required functionality.

Read more about Go code organisation [here](#).

Setup the project (continued)

`go.mod` contains all the paths for your module. The naming convention for the modules, requires name of the module to be composed of the name of your organisation plus module parent directory plus module name. E.g.: `github.com/Line-39/go-micro-service`. Note that it is not required, and it is not required for the module to be published online - if you do not follow this convention, or you do not sotre your code at public repository, the module will be still available locally for other modules within your *workspace*.

Setup the project (continued)

The `main` *package* in Go contains `main()` function, which serves as an entry point for the programm. It takes no arguments, returns no values and is not called directly:

```
package main               // package declaration

import "fmt"               // imports

func main() {              // main() declaration
    /*
      the magick lives here 🪄🪄🪄
    */
}
```

First program

In the project directory, create a `main.go` file and open it in your online editor.

```
touch main.go
# vim main.go   # if you ran this command - burn your pc - your only path to freedom
code main.go
```

First program (continued)

Add following lines to the file you've just opened:

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, world 🪄🪄🪄")
}
```

First program (continued)

If your are working in VScode and you have Go extensions installed, you will notice VSCode complains that no packages found for the `main`. Ignore it for now.

First program (continued)

From you project directory, run folowing command from the terminal:

```
go run .
```

Unfortunataly it doesn't work. You should see following output in your console:

```
current directory outside modules listed in go.work or their selected dependencies
```

There are two particular problems that has to be solved yet.

First program (continued)

We are about to write a Go module, to combine useful functionality from different packages in order to create a tool we need. In order to do so, we need to specify the dependencies and the version of Go we are using for this specific module.

First program (continued)

As scary as it sounds, in practice we only need to run following command from the terminal in our projects directory:

```
go mod init github.com/line-39/go-microservice
# go: creating new go.mod: module github.com/line-39/go-microservice
# go: to add module requirements and sums: go mod tidy
```

This command creates a `go.mod` file in the current directory, and collects all the required information for us. Read more about modules [here].

First program (continued)

Inspect the `go.mod` file:

```
ls .
# go.mod  LICENSE  main.go  README.md

cat go.mod
# module github.com/line-39/go-microservice
#
# go 1.22.1
```

First program (continued)

We are going to run our program again:

```
go run .
# current directory is contained in a module that is not one of the workspace
# modules listed in go.work. You can add the module to the workspace using:
# go work use .
```

Still doesn't work! But we can see that output differs from what we've seen before.

First program (continued)

In a nutshell Go complains that the module we are trying to run is not associated with the *go workspace*. Go workspace is group of modules / packages defined in `go.work` file which should be located in the directory containing the *modules*. For the workspace initialization we use `go work init <directory to use>` command. You can read more about it here.

First program (continued)

Let say your Go code is organized this way:

```
go
└── src
    ├── github.com
    │   ├── line-39
    │   └── utubun
    └── seqquery.de
        └── s3
```

And you are going to use all the modules located under `github.com/...` and `sequery.de` dirs.

First program (continued)

From the parent directory containing all of your modules (in this specific case `src`), initialize go workspace with following command:

```
go work init .
```

First program (continued)

If you `tree` your parent directory, after that, you will get this output:

```
go
└── src
    ├── github.com
    │   ├── line-39
    │   └── utubun
    ├── go.code-workspace
    ├── go.work
    ├── go.work.sum
    └── seqquery.de
        └── s3
```

First program (continued)

In case you created your module within the directory with *initialized* `go.work`, assuming you are located in the *project directory* just add your module to the workspace as shown below:

```
go work use .
```

First program (continued)

Run the program one more time

```
go run .
# Hello, world 🪄🪄🪄
```

Finally, the magick works 🪄🪄🪄

First program (continued)

The `go run` compiles your programm on background, saves the binary to your `/temp` dir, and run it. You can build it yourself with `go build` command. Build and run your programm from the terminal:

```
go build .
ls
# go-microservice  go.mod  LICENSE  main.go  README.md
chmod 776 go-microservice
./go-microservice
# Hello, world 🪄🪄🪄
rm go-microservice
```

First API service

Now let's make a big jump from iconic "Hello, world!" app to the web server. Our service will respond to requests to it's `/` endpoint with simple "Hi there 👋" message.

We must consider three main components of our service:

1. A *handler* executing the logic of our app in response to the *specific* request;
2. A *router / servermux* which maps *URL patterns* to corresponding *handlers*;
3. A webserver listening to the requests;

First API service (continued)

```
package main

import (
    "log"
    "net/http"
)

func hello(w http.ResponseWriter, r http.Request) {
    w.Write([]byte("Hi there 👋"))
}

func main() {
    // create a new servermux
    mux := http.NewServeMux()
    // register hello() as a handler for "/" pattern
    mux.HandleFunc("/", hello)

    // log the service startup
    log.Print("starting service on :4000")

    // start http server on :4000
    err := http.ListenAndServe(":4000", mux)
    // log the error message if ListenAndServe() encounters error
    log.Fatal(err)
}
```

First API service (continued)

From the command line in your project folder run the command below:

```
go run .
```

If everything goes right, you will see the log message printed into your terminal, saying that service is starting on port 4000.

First API service (continued)

Open your browther at http://localhost:4000 to see the response from the service.

Alternatively, you can query the service with `curl`:

```
curl http://localhost:4000/
# Hi there 👋
```

Adding more endpoints

Switch to the new branch `04-first-api` by running `git switch 04-first-api` if you are working on clonned project, or modify yor `main.go` as shown on the next slide.

Adding new handlers

First, define new handlers (definitions go before `main()` definition):

```go
// ...

func hello(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hi there 👋"))
}

func viewData(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Display user data 📁"))
}

func uploadData(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Upload user data 📤"))
}
```

Registering new handlers

Second register your new handlers:

```go
func main() {
    // create a new servermux
    mux := http.NewServeMux()

    // register handlers
    mux.HandleFunc("/", hello)
    mux.HandleFunc("/data/view", viewData)
    mux.HandleFunc("/data/upload", uploadData)

    // ... the rest of the code
}
```

Run the service

Now, run the service locally, executing following command in your terminal.

```
go run .
# 1970/01/01 00:00:00 starting service on :4000
```

You are ready to query the API

Query the API (browser)

While service is running, open your browser and type the following addresses to see the response from the service:

- localhost:4000/ our `hello` endpoint;
- localhost:4000/data/view our `viewData` endpoint;
- localhost:4000/data/upload our `uploadData` endpoint;

Query the API (CLI)

Type following commands to see the response from the endpoints we just created.

```
curl -i localhost:4000/
curl -i localhost:4000/data/view
curl -i localhost:4000/data/upload
```

Restricting subtree path

Every path that deos not end with the trailing slash will be matched *exectly* by the router. However, any path with *trailing slash* is considered to be a *subtree path pattern*, and it matches for any path matching subtree pattern.

Run the service with `go run .`, and navigate to localhost:4000/foo. This endpoint does not exist in our `servermux` definition. But server responds with greetings, it is because *subtree path pattern* match. I.e localhost:4000/ will call `hello()` handler, sending the response to the user. In other words, trailing slash can be red as `/**` i.e. wildcard pattern.

Restricting subtree path

To *restrict subtree pattern matching* we can add a *special character* `{$}` to the end of the path, after the trailing slah. Modify your router definition as shown below and restart the server:

```
// ...
mux.HandleFunc("/{$}", hello)
// ...
```

Now restart the service, and navigate to localhost:4000/foo again, you should receive `404 page not found` response.

WIldcard patterns

The `net/http` `servemux` lets us to us whildcards in path patterns. Lets consider following scenario for our service:

1. All the data served based on specified user ID;
2. There are two types of data: *raw* data and *clean* data;

Lets change our `data/view` and `data/upload` routes:

```
// ...
mux.HandleFunc("/{user}/data/{datatype}/view", viewData)
mux.HandleFunc("/{user}/data/{datatype}/upload", uploadData)
// ...
```

Wildcard patterns (continued)

```
func viewData(w http.ResponseWriter, r *http.Request) {
    user := r.PathValue("user")
    if user == "" {
        http.NotFound(w, r)
        return
    }
    dtype := r.PathValue("datatype")
        if user == "" {
        http.NotFound(w, r)
        return
    }

    msg := fmt.Sprintf("📁 Display the %s data for user %s\n", dtype, user)
    w.Write([]byte(msg))

}
```

Wildcard patterns (continued)

Now we can restart our service. Let say we want to see **raw** data for the user **ubot**:

```
curl -i localhost:4000/ubot/data/raw/view
#HTTP/1.1 200 OK
#Date: Mon, 08 Apr 2024 14:42:04 GMT
#Content-Length: 40
#Content-Type: text/plain; charset=utf-8
#
#📁 Display the raw data for user ubot
```

Keep in mind, that user can send any kind of parameter as a wildcard. So checking the validity is entirely on you.

Wildcard patterns (continued)

Let's try another request

```
curl -i localhost:4000/jer/data/jobs/view
#HTTP/1.1 200 OK
#Date: Mon, 08 Apr 2024 14:44:53 GMT
#Content-Length: 40
#Content-Type: text/plain; charset=utf-8
#
#📁 Display the jobs data for user jer
```

Wildcard patterns (continued)

Be aware that patterns defined with *wildcard* might overlap. E.g. `user/view` and `user/{data}` requests overlap (incoming `user/view` request is a valid match for `user/{data}` pattern).

In such cases `servemux` applies following precedence rule: *The most specific pattern wins.*

Since `user/view` matches only *one specific* request, and `user/{data}` matches infinit amount of possible requests `user/view` will take precedent.

HTTP methods

We can introduce constraints, so our API responts *only* to the HTTP requests with *appropriate HTTP method*. To achive this we will edit the rout regisration in `main.go`:

```
// ...
// register handlers
mux.HandleFunc("GET /{$}", hello)
mux.HandleFunc("GET /{user}/data/{datatype}/view", viewData)
mux.HandleFunc("POST /{user}/data/{datatype}/upload", uploadData)
// ...
```

Now save your changes and restart the service to request API.

HTTP methods (continued)

From CLI try following querries

```
curl -i localhost:4000/                              # 200
curl -i -X POST localhost:4000/ubot/data/raw/view   # 405 Method Not Allowed
curl -i -X GET  localhost:4000/ubot/data/raw/view   # 200
curl -i -X GET  localhost:4000/ubot/data/raw/upload # 405 Method Not Allowed
curl -i -X POST localhost:4000/ubot/data/raw/upload # 200
```

All the requests with unappropriate HTTP methods were rejected.

HTTP methods (continued)

Using HTTP methods, we can simplify the pattern of our endpoints, getting rid of the last `view / upload` part, since it will be defined by the HTTP method we use.

```
// ...
// register handlers
mux.HandleFunc("GET /{$}", hello)
mux.HandleFunc("GET /{user}/data/{datatype}", viewData)
mux.HandleFunc("POST /{user}/data/{datatype}", uploadData)
// ...
```

HTTP methods (continued)

Modifying our request, we can see that the same endpoint is now responds differently depending on the HTTP method we are using.

```
curl -X GET localhost:4000/jer/data/raw
# 📁 Display the raw data for user jer

curl -X POST localhost:4000/jer/data/raw
# 📤 Upload the raw data for user jer
```

HTTP status codes

First time the `w.Write()` has been called it wrights `200 OK` to the response *header*. Write `201 Created` status code to the header, as shown below:

```
package main

// ...
func uploadData(w http.ResponseWriter, r *http.Request) {
    // ...
    w.WriteHeader(http.statusCreated)
    msg := fmt.Sprintf("📤 Upload the %s data for user %s\n", dtype, user)
    w.Write([]byte(msg))
}
```

Http status codes (continued)

Onece writen, header's status code *can not be changed*. So the status code has to be modified *before* any subsequent `w.Write()` call. For example, the code below returns warning, and status code remains to be `200 OK`:

```
func someHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.statusProcessing)

    if ok := true; ok {
        w.WriteHeader(http.statusAccepted)
    }
    w.Write([]byte("Hell yeah!"))
}
```

HTTP status codes (continued)

Run you service, and call `<user>/data/<datatype>` endpoint using `POST` request:

```
go run .
# 1970/01/01 00:00:00 starting service on :4000
# another terminal window
curl -Xi POST localhost:4000/jer/data/raw
#HTTP/1.1 201 Created
#Date: Thu, 01 Jan 1970 00:00:00 GMT
#Content-Length: 38
#Content-Type: text/plain; charset=utf-8
#
#📤 Upload the raw data for user jer
```

Modifying header map

We can modify header with `Header()`, `Add()`, `Set()`, `Del()`, `Get()`, `Values()` methods. For example, overwrite *content-type of the response, add the information about our server (key) name (value) for `/` handler:

```
func hello(w http.ResponseWriter, r *http.Request) {
	w.Header().Set("Content-Type", "application/json")
	w.Header().Add("Server", "Simple Go Service")
	w.Write([]byte(`{"message":"Hi there 👋"}`))
}
```

Modifying header map (continued)

Restart the service and call `/` endpoint:

```
go run .
curl -i localhost:4000/
#HTTP/1.1 200 OK
#Content-Type: application/json
#Server: Simple Go Service
#Date: Thu, 01 Jan 1970 00:00:00 GMT
#Content-Length: 29
#
#{"message":"Hi there 👋\n"}
```

Configuration settings: CLI arguments

Our service listens to the network address wich is hardcoded in `main()`. This is OK for early builds, but not a good way to configure the application.

```
// log the service startup
log.Print("starting service on :4000")

// start http server on :4000
err := http.ListenAndServe(":4000", mux)
```

The good idea is to make such things as network-address, name and version of the application, secrets etc configurable at runtime.

Configuration settings: CLI arguments (continued)

We can pass command-line arguments when starting the application:

```
go run . -addr=":4000" -name="Simple Go Microservice" -version="0.0.1"
```

We can access CLI arguments with `flag` package `String` function:

```go
func main() {
    // ...
    addr    := flag.String("addr", ":4000", "HTTP network address")
    name    := flag.String("name", "Simple Go Microservice", "The name of the app")
    version := flag.String("version", "0.0.0", "The version of the app")
    flag.Parse()
    // ...
}
```

Configuration settings: CLI arguments (continued)

It is important to understand that first we *define* our variables `addr`, `name` and `version` as new vars of *CLI flag type*. When latter we call `flag.Parse()` it parses CLI arguments and passes the values to this variables:

```
func main() {
    //...
    // log on startup
    log.Printf("Starting %s, version %s on %s", *addr, *name, *version)

    // start http server on :4000
    err := http.ListenAndServe(*addr, mux)
    //...
}
```

Configuration settings: CLI arguments (continued)

Modify `main()` and run the service without *any* command-line flags:

```
go run .
# 1970/01/01 00:00:01 starting Simbple Go Microservice, version 0.0.0 on :4000
```

Run the service with some flags:

```
go run . -addr="5000" -name="Einfacher Go-Service" -version="Infinity"
# 1970/01/01 00:00:01 starting Einfacher Go-Service, version Infinity on :5000
```

Configuration settings: CLI arguments (continued)

In several secanarios we would like to store the values received through the command line flags, in a single structure and access it during the run. Let's define a `config` structure in our `main.go`

```
type config struct {
    addr string
    name string
    version string
}
```

Configuration settings: CLI arguments (continued)

We can access CLI arguments with `flag` package `StringVar` function, and pass their values into predefined `config`:

```
func main() {
    // ...
    var cfg config

    flag.StringVar(&cfg.addr, "addr", ":4000", "HTTP network address")
    flag.String(&cfg.name, "name", "Simple Go Microservice", "The name of the app")
    flag.String(&cfg.version, "version", "0.0.0", "The version of the app")
    flag.Parse()
    // ...
}
```

Configuration settings: CLI arguments (continued)

It is important to understand that first we *define* our variables `addr`, `name` and `version` as new vars of *CLI flag type*. When latter we call `flag.Parse()` it parses CLI arguments and passes the values to this variables:

```go
func main() {
    //...
    // log on startup
    log.Printf("Starting %s, version %s on %s", cfg.name, cfg.version, cfg.addr)

    // start http server on :4000
    err := http.ListenAndServe(cfg.addr, mux)
    //...
}
```

Configuration settings: CLI arguments (continued)

Modify `main()` and run the service without *any* command-line flags:

```
go run .
# 1970/01/01 00:00:01 starting Simple Go Microservice, version 0.0.0 on :4000
```

Run the service with some flags:

```
go run . -name="Rocinante 🚀"
# 1970/01/01 00:00:01 starting Rocinante 🚀, version 0.0.0 on :4000
```

Configuration settings: CLI arguments (continued)

The help for command-line arguments is automatically created for us:

```
go run . -help
# Usage of /tmp/go-build2113936530/b001/exe/go-microservice:
#   -addr string
#         HTTP network address (default ":4000")
#   -name string
#         Service name (default "Simple Go Microservice")
#   -version string
#         Service version (default "0.0.0")
```

Environmental variables

In some scenarios we would need to use environmental variables for our application. For example, environmental variables can be usefull for our service configuration. Let's create a `.env` file and save our `addr`, `name` and `version` there:

```
ADDR=":4000"
NAME="Simple Go Microservice 🚀"
VERS="0.0.2"
```

Environmental variables (continued)

Load the variables to the Bash environment, and access their values:

```
source .env
echo $ADDR
# :4000
echo $NAME
# Simple Go Microservice 🚀
echo $VERS
# 0.0.2
```

Environmental variables (continued)

Run the service, and pass the environmental variables as a flag values

```
go run . -name="${NAME}" -version="${VERS}"
# 1970/01/01 00:00:01 starting Simple Go Microservice 🚀, version 0.0.2 on :4000
```

Makefile

There is a lot of typing already. Instead of sourcing the `.env` and passing the variables to `go run .`, and `go build .` we will write this once into the make file. Create `Makefile` in the root of the project:

```
touch Makefile
code Makefile
```

Makefile (continued)

Edit it as follows.

```
#!make
include .env
export

config:
	@echo -e 'Current config:\n-Name:\t\t${NAME}\n-Version:\t${VERS}\n-Address:\t${ADDR}\n'

run:
	@go run . -addr=${ADDR} -name=${NAME} -version=${VERS}

build:
	@go build

all:
	@$(MAKE) build
	@./go-microservice -addr=${ADDR} -name=${NAME} -version=${VERS}
```

Makefile (continued)

Save the changes and run following commands:

```
make config
# Current config:
# -Name:              "Simple Go Microservice 🚀"
# -Version:           "0.0.2"
# -Address:           ":4000"
make run
# 1970/01/01 00:00:00 starting Simple Go Microservice 🚀, version 0.0.2 on :4000
make all
# make[1]: Entering directory '/home/ubot/go/src/github.com/line-39/go-micro-service'
# make[1]: Leaving directory '/home/ubot/go/src/github.com/line-39/go-micro-service'
# 1970/01/01 00:00:00 starting Simple Go Microservice 🚀, version 0.0.2 on :4000
```

Accessing environment at run

There are scenarios, when we want to access environmental variables not when we start the application, but at runtime. Let's modify our code to do so. First we are going to modify the code for our `config` declaration:

```
// config model
type config struct {
    addr    string `default:":4000"`
    name    string `default:"Simple Go Microservice 🚀"`
    version string `0.0.1`
}
```

Accessing environment at run (continued)

Next, we define method on `config` type:

```go
func (cnf *config) init() {
    addr := os.Getenv("ADDR")
    if addr != "" { cnf.addr = addr }

    name := os.Getenv("NAME")
    if name != "" { cnf.name = name }

    vers := os.Getenv("VERS")
    if vers != "" { cnf.version = vers }
}
```

Accessing environment at run (continued)

Next, remove everything related to command-line flags parsing, and replace it with the following code:

```
func (cnf *config) init() {
    //...
    // declare config
    var cfg config

    // init config
    cfg.init()
    //...
}
```

Accessing environment at run (continued)

And restart your service.

```
make config
#Current config:
#-Name:              "Simple Go Microservice 🚀"
#-Version:           "0.0.3"
#-Address:           ":4000"
make run
# 1970/01/01 00:00:00 starting Simple Go Microservice 🚀, version 0.0.3 on :4000
```

Try to change the name of the service in `.env` file, and restart the service to see what happens.

Logger

Go *standard library* includes the `log/slog` package to create a *structured* log entries in a set format. The created log entry includes:

- A timestamp;
- The severity level (`Debug`, `Info`, `Warn`, `Error`);
- The log message (`string`);
- Optional number of key-value pairs for additional information;

Logger (continued)

We create a `slog` instance with `slog.New()` function, which takes a *structured log handler* as it's argument.

```
logHandler := slog.NewTextHandler(os.StdOut, nil)
logger := slog.New(loggerHandler)
```

Or, as a oneliner

```
logger := slog.New(slog.NewTextHandler(os.StdOut, nil))
```

We create the logger to log into standard output, whith the handler with no customized options.

Logger (continued)

Lets implement it in our application. First, in `main.go` `main()` body create an instance of `slog`:

```
logger := slog.New(slog.NewTextHandler(os.Stdout, nil))
```

Then update all the calls of `log`:

```
        // log the service startup
        logger.Info(fmt.Sprintf("Starting service %s v:%s  on %s", cfg.name, cfg.version, cfg.addr))

        // start http server on :4000
        err := http.ListenAndServe(cfg.addr, mux)
        // log the error message if ListenAndServe() encounters error
        logger.Error(err.Error())
        os.Exit(1)
```

Logger (continued)

Now run the service, and see the log message.

```
go run .
# time=1970-01-01T00:00:03.044+02:00 level=INFO msg="Starting service Simple Go Microservice 🚀 v:0.0.3  on :4000"
```

Initiate an error message executing `go run .` from another terminal:

```
go run .
# ime=1970-01-01T00:00:03.044+02:00 level=INFO msg="Starting service  v:  on "
# ime=1970-01-01T00:00:03.044+02:00 level=ERROR msg="listen tcp :80: bind: permission denied"
# exit status 1
```

Logger (continued)

The same way we did before we can log in *JSON* format. Change the logger definition, as shown below:

```
logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))
```

And restart the service:

```
go run .
# {"time":"2024-04-10T23:36:59.319554283+02:00","level":"INFO","msg":"Starting service Simple Go Microservice 🚀 v:0.0.3  on :4000"}
```

Logger (continued)

If we would like to redirect the log output to the file, we can specify the densination in *slog handler* definition. However, we can make it agnostic about the final destination. For example, calling `go run . >> log` we will append the logs to the `log` file for the latter processing.

```
go run . >> log
# in another terminal
cat log
# {"time":"2024-04-10T23:42:09.643528183+02:00","level":"INFO","msg":"Starting service Simple Go Microservice 🚀 v:0.0.3  on :4000"}
```

Logger (continued)

Loggers created by `slog.New()` are concurrency-safe: a single logger can be used accross multiple goroutines ignorring the race conditions.

Dependency injection

Wouldn't it be nice if we could use the same instance of our `logger` everywhere in our service, including our handler functions? This is where we need a *dependency injections*.

Let's update our code. First create a `app` embeding the `logger` and `config`:

```
// application model
type application struct {
	config *config
	logger *slog.Logger
}
```

Dependency injection (continued)

We convert our handler functions to be a *methods* against our `app`.

```
// handler
func (app *application) hello(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.Header().Add("Server", "Simple Go Service")

    app.logger.Info("endpoint: 'hello'", "method", r.Method, "uri", r.URL.RequestURI())

    w.Write([]byte(`{"message":"Hi there 👋"}`))
}
```

Dependency injection (continued)

We have to initialize our `app` in `main`:

```
// init logger
logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

// init applicatio
app := &application{
    logger: logger,
}
```

Dependency injection (continued)

And change our routs accordingly

```
// create a new servermux
mux := http.NewServeMux()

// register handlers
mux.HandleFunc("GET /{$}", app.hello)
//mux.HandleFunc("GET /{user}/data/{datatype}", app.viewData)
//mux.HandleFunc("POST /{user}/data/{datatype}", app.uploadData)
```

Dependency injection (continued)

Now if we run our service, and hit `/` endpoint, we will see logs printed by `hello` handler function: