

# Gin Framework

Gin est un framework web écrit en Go (Golang).

- [Introduction à Gin](#)
- [Installation](#)
- [Live reload](#)
- [Restful API Server](#)
- [Static server](#)
- [Reverse proxy](#)

# Introduction à Gin

[Repo github des exercices](#)

## Présentation de Gin

[Gin](#) est un framework web HTTP écrit en Go.

Il dispose d'une API de type [Martini](#), mais avec des performances jusqu'à 40 fois plus rapides que Martini. Si vous avez besoin de performances époustouflantes, procurez-vous du Gin (le framework hein !).

Gin simplifie de nombreuses tâches de codage associées à la création d'applications Web, y compris les services Web.

## Fonctionnalités

[Documentation Gin](#)

- Rapide
- Prise en charge des middlewares
  - Exemple : Logger, Authorization, GZIP ...
- Pas de crash
  - Possède un catcheur `panic` interne pour intercepter les erreurs et empêcher l'arrêt de notre API
- JSON Validation
- Gestion d'erreur
  - On peut gérer manuellement les erreurs interceptées

## Gin VS Node

Gin est INCROYABLEMENT rapide comparé à des concurrents de tous les jours.

Le fait qu'il soit codé en Go permet d'obtenir un binaire complet pesant ~10Mo et contenant notre serveur API au complet. Comparé à Node qui doit inclure toutes ses librairies, c'est beaucoup moins !

Untitled

Untitled

# Docker

On peut créer des images docker de notre API gin ULTRA légères, puisque le binaire est standalone, une image alpine suffit [\[1\]](#)

```
ARG GO_VERSION=1.18

FROM golang:${GO_VERSION}-alpine AS builder

RUN apk update && apk add --no-cache alpine-sdk git

WORKDIR /api

COPY go.mod .
COPY go.sum .
RUN go mod download

COPY . .
RUN go build -o ./app ./main.go

FROM alpine:latest

RUN apk update && apk add --no-cache ca-certificates

WORKDIR /api
COPY --from=builder /api/app .

EXPOSE 8080

ENTRYPOINT [ "./app" ]
```

On peut faire plus simple, mais je vous montre une version très optimisée d'un Dockerfile pour faire tourner une app go dans un environnement ultra léger.

# Installation

Dans le cadre de cette explication, j'utiliserais l'IDE Goland, donc il se peut que certaines choses soient simplifiées par l'IDE, et d'autres que je doive faire spécifiquement par rapport à cet IDE

## Nouveau projet

Tout d'abord, nous allons créer un nouveau projet Go.

Untitled

Dans l'environnement, j'ai spécifié `GOPROXY=direct`  
C'est très important, car cela va nous permettre d'inclure des librairies externes (Gin)

Vérifier dans vos paramètres Go / Go Modules que l'option **Enable Go Modules integration** est bien coché et ressemble à ça :

Untitled

## Ajout des dépendances

Pour ajouter les dépendances nécessaires à Gin, nous devons ouvrir un terminal à la racine du projet (que ne doit contenir pour le moment que le fichier `go.mod`) et taper la commande suivante.

```
go get -u github.com/gin-gonic/gin
```

Cela va télécharger et indiquer dans notre fichier `go.mod` les dépendances nécessaires au fonctionnement de **Gin** :

Untitled

## Création d'un serveur basique

Pour tester que tout va bien, nous allons créer le fichier `main.go` à la racine du projet et le remplir ainsi :

```
package main

import "github.com/gin-gonic/gin"

func main() {
    r := gin.Default()

    r.GET("/", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "hello world",
        })
    })

    r.Run(":8080")
}
```

Testons ce petit code rapidement avec la commande :

```
go run main.go
```

Untitled

Et voilà 🎉 ! Vous avez votre premier serveur Gin qui est en marche !

Rendons-nous avec notre navigateur sur l'adresse [localhost:8080](http://localhost:8080) pour admirer la superbe réponse

Untitled

Untitled

# Live reload

Recompiler notre code à chaque fois que l'on change notre code, arrêter le serveur et le relancer...

Tout ça est long et fastidieux ! Surtout pendant le développement !

En nodeJS certains se souviendront de nodemon qui permettait de surveiller les changements dans nos fichiers, et les recompiler à la volée.

En Go il existe différents outils permettant de faire cela, nous allons utiliser [Air](#)

## Air installation

Pour installer Air en tant qu'exécutable reconnu par notre machine, il suffit de faire la commande suivante :

```
go install github.com/cosmtrek/air@latest
```

Ensuite, nous allons "pimper" un peu la configuration de cet outil pour avoir un peu de couleur ☺.

Pour ce faire, nous allons créer le fichier `.air.conf` à la racine de notre projet et le remplir ainsi :

```
# .air.conf
# Config file for [Air](https://github.com/cosmtrek/air) in TOML format

# Working directory
# . or absolute path, please note that the directories following must be under root.
root = "."
tmp_dir = "tmp"

[build]
# Just plain old shell command. You could use `make` as well.
cmd = "go build -o ./tmp/main ." # replace by main.exe if on windows !
# Binary file yields from `cmd`.
bin = "tmp/main" # replace by main.exe if on windows !
# Customize binary.
# Watch these filename extensions.
include_ext = ["go", "tpl", "tmpl", "html"]
```

```
# Ignore these filename extensions or directories.
exclude_dir = ["assets", "tmp", "vendor", "frontend/node_modules"]
# Watch these directories if you specified.
include_dir = []
# Exclude files.
exclude_file = []
# It's not necessary to trigger build each time file changes if it's too frequent.
delay = 1000 # ms
# Stop to run old binary when build errors occur.
stop_on_error = true
# This log file places in your tmp_dir.
log = "air_errors.log"

[log]
# Show log time
time = false

[color]
# Customize each part's color. If no color found, use the raw app log.
main = "magenta"
watcher = "cyan"
build = "yellow"
runner = "green"

[misc]
# Delete tmp directory on exit
clean_on_exit = true
```

**Attention à la ligne 11 et 13 !**

Désormais, il suffit de lancer la commande `air` à la racine du projet pour le lancer et surveillez les changements de code :

Untitled

Au moindre changement, on pourra voir que le script le détecte et recompile aussitôt

Untitled

# Restful API Server

## Simple Server

[GitHub repo](#)

Pour utiliser **Gin**, il suffit d'importer `github.com/gin-gonic/gin` au niveau de son fichier main et de créer une variable qui va contenir notre fameux routeur.

```
package main

import "github.com/gin-gonic/gin"

func main() {
    r := gin.Default()
}
```

Il faut ensuite lui définir des routes sur lesquels il va écouter. Dans notre exemple, nous ferons 2 GET qui renvoient un JSON facilement grâce à la librairie **Gin**.

```
package main

import "github.com/gin-gonic/gin"

func main() {
    r := gin.Default()

    r.GET("/", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "Hello World!",
        })
    })

    r.GET("/ping", func(c *gin.Context) {
        cIndentedJSON(200, gin.H{
            "message": "pong",
        })
    })
}
```



```
}
```

Enfin, il faut lui dire de se lancer et d'écouter sur un port spécifique grâce à une dernière ligne.

```
package main

import "github.com/gin-gonic/gin"

func main() {
    r := gin.Default()

    r.GET("/", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "Hello World!",
        })
    })

    r.GET("/ping", func(c *gin.Context) {
        cIndentedJSON(200, gin.H{
            "message": "pong",
        })
    })

    r.Run(":9090")
}
```

Dans ce code, nous :

- Initialisons un routeur Gin en utilisant [Default](#).
- Utilisons le [GET](#), pour associer la méthode HTTP `GET` et le chemin `/, /ping` à une fonction contenant le contexte de la requête
- Utilisons `c.JSON` ou bien `c.IndentedJSON` permettent de simplement convertir une structure (en l'occurrence `gin.H` qui permet d'en créer une à la volée)

Il suffit alors de lancer notre magnifique app avec la commande `go run main.go` et aller tester nos routes.

Untitled

Untitled

Aussi simple que ça.

# Simple music server

[GitHub repo](#)

Pour complexifier un peu plus les choses, on va refaire la même chose, mais avec quelques structures et un peu de découpage. L'objectif étant de servir une API de gestion d'une liste d'album de musique (Très originale oui).

On va essayer de faire du pseudo MVC et l'architecture de notre application sera la suivante :

```
controllers/  
| controller.go  
| command.go  
| query.go  
data/  
| albums.go  
models/  
| album.go  
main.go  
go.mod  
go.sum
```

## Models

Nous allons dans un premier temps définir la structure d'un album. Il faut déclarer dans le fichier `models/album.go` la structure suivante :

```
package models  
  
type Album struct {  
    ID      string `json:"id"`  
    Title   string `json:"title"`  
    Artist  string `json:"artist"`  
    Price   float64 `json:"price"`  
}
```

Les balises telles que `json:"artist"` spécifient le nom d'un champ lorsque le contenu de la structure est sérialisé en JSON.

Sans eux, le JSON utiliserait les noms de champs des propriétés, avec la majuscule, ce qui n'est pas très courant (pour rappel, en go, la majuscule, en début de variable ou fonction, permet de définir sa visibilité en dehors de son package) .

# Data

On va ensuite déclarer une liste d'albums qui nous serviront de “base de données” pour notre application.

Remplissons dans le fichier `data/albums.go` de cette manière :

```
package data

import "gin-form/simple_music_api/models"

var Albums = []models.Album{
    {
        ID:      "1",
        Title:   "Taste of you",
        Artist:  "Rezz",
        Price:   1.99,
    },
    {
        ID:      "2",
        Title:   "Go",
        Artist:  "Google",
        Price:   9999,
    },
    {
        ID:      "3",
        Title:   "C#",
        Artist:  "Microsoft",
        Price:   -1,
    },
}
```

# Controllers

Occupons-nous de la partie controllers désormais !

Dans un premier temps, nous allons écrire nos fonctions servant à récupérer les données uniquement (en mode CQS tu connais).

Le fichier `controllers/query.go` va contenir deux fonctions :

- Une permettant de récupérer la liste des albums
- Une autre permettant de récupérer un seul album par son **ID**

La première partie du fichier ressemblera simplement à ça

```
package controllers

import (
    "gin-form/simple_music_api/data"
    "github.com/gin-gonic/gin"
    "net/http"
)

func getAlbums(c *gin.Context) {
    c.IndentedJSON(http.StatusOK, data.Albums)
}
```

Dans ce code, nous :

- Écrivons une fonction `getAlbums` qui prend un `gin.Context` en paramètre.
  - `gin.Context` est la partie la plus importante de Gin. Il prend en charge la requête, les détails, la validation et sérialisation JSON, et plus encore.
- Appelons la fonction `c.IndentedJSON` afin de sérialiser notre tableau `data.Albums` en JSON indenté proprement.
- Utilisons une librairie interne à go `net/http` pour récupérer le code HTTP voulu (200). On pourrait écrire directement 200 à la main, mais maintenant vous savez que cette librairie existe ☐☐

La deuxième méthode est un peu plus complexe et permet de récupérer un album parmi ceux existants avec son ID, qui sera passé dans le chemin de la requête (`/album/:id`).

```
func getAlbumByID(c *gin.Context) {
    id := c.Param("id")

    for _, album := range data.Albums {
        if album.ID == id {
            c.IndentedJSON(http.StatusOK, album)
        }
    }
}
```

```

    return
  }
}

c.IndentedJSON(http.StatusNotFound, gin.H{"error": "Album not found"})
}

```

Nous allons maintenant nous occuper du fichier `controllers/command.go` qui contiendra notre fonction permettant d'ajouter un album à notre liste.

```

package controllers

import (
    "gin-form/simple_music_api/data"
    "gin-form/simple_music_api/models"
    "github.com/gin-gonic/gin"
    "net/http"
)

func addAlbum(c *gin.Context) {
    var newAlbum models.Album

    if err := c.BindJSON(&newAlbum); err != nil {
        c.IndentedJSON(400, gin.H{
            "message": "Invalid JSON",
            "error":   err.Error(),
        })
        return
    }

    data.Albums = append(data.Albums, newAlbum)
    c.IndentedJSON(http.StatusCreated, newAlbum)
}

```

Dans cette fonction nous :

- Déclarons une variable `newAlbum` de type `Album`
- Utilisons la méthode fournie par **Gin** `c.BindJSON` qui va tenter de parser le body de notre requête dans notre structure en se basant sur le format décrit plus haut (les fameux `json:"artist"`).
  - Si, on n'y parvient pas, on renvoie un code erreur avec un message et stoppons l'exécution de la méthode.

- Ajoutons notre nouvel album à notre tableau
- Renvoyons un code de Création et l'album qui vient d'être enregistré

---

Les fonctions écrites plus hautes sont privées, il va falloir donc faire quelque chose pour qu'elles puissent être utilisées par notre routeur se trouvant dans le fichier `main.go`.

Ça sera le but de notre fichier `controllers/controller.go` qui va s'occuper de faire notre routage :

```
package controllers

import "github.com/gin-gonic/gin"

func SourceControllers(router*gin.Engine) {
    []router.GET("/albums", getAlbums)
    []router.GET("/albums/:id", getAlbumByID)
    []router.POST("/albums", addAlbum)
}
```

---

Nous pouvons enfin relier tout ça à notre routeur principal dans le fichier `main.go`

```
package main

import (

    []"gin-form/simple_music_api/controllers"
    []"github.com/gin-gonic/gin"
)

func main() {
    []router := gin.Default()

    []controllers.SourceControllers(router)

    []router.Run(":8080")
}
```

On peut maintenant aller tester tout ça ☐☐

Untitled

Untitled

Untitled

Untitled

Untitled

GGWP 

# Static server

[GitHub repo](#)

Dans certains cas, on souhaite juste héberger un site statique.

On pourrait se tourner vers apache ou nginx mais ce n'est pas ce que nous recherchons ☹️

Il est possible assez facilement grâce à **Gin** de rendre accessible notre site statique.

Pour cette démonstration, je possède l'architecture suivante :

```
static/  
| assets/  
| | ...  
| index.html  
| script.js  
| ...  
main.go  
go.mod  
go.sum
```

Mon site dans le dossier `static` est une application Angular (avec Angular router pour l'exemple haha) compilé en version de production.

Pas besoin d'aller très loin, notre fichier `main.go` ressemblera à ça :

```
package main  
  
import "github.com/gin-gonic/gin"  
  
func main() {  
    r := gin.Default()  
  
    r.Static("/", "./static")  
  
    r.Run(":8080")  
}
```

Pas besoin de détailler, les fonctions parlent d'elles même.



Si je me rends sur mon site, on voit que tout va BIEN :

Untitled

Je me rends sur une autre page de mon site en lançant un combat, et là aussi tout va BIEN :

Untitled

SAUF QUE !

Si je décide de rafraîchir ma page, avec cet URL là, et bien j'obtiens une belle 404...

Untitled

Cela vient du fait que le router va chercher bêtement un fichier au chemin

`/fight/charizard/blastoise` dans notre dossier `static` alors qu'il devrait passer ce chemin à notre application Angular, c'est un problème récurrent avec les applications web.

Il existe heureusement une solution, il suffit de dire à **Gin** que s'il ne trouve pas le chemin en question dans l'arborescence de dossier, il doit alors interroger l'application Angular, qui se chargera elle-même de renvoyer une erreur 404 si le chemin n'existe effectivement pas.

```
package main

import "github.com/gin-gonic/gin"

func main() {
    r := gin.Default()

    r.Static("/", "./static")

    r.NoRoute(func(c *gin.Context) {
        c.File("./static/index.html")
    })

    r.Run(":8080")
}
```

Et là, si on rafraîchit, on retrouve notre beau combat dans notre arène ☺

# Reverse proxy

## Reverse proxy simple

[GitHub repo](#)

Le reverse proxy est quelque chose de majoritairement utilisé aujourd'hui.

Dans beaucoup de cas d'utilisation, on utilise des outils tels que Nginx, Apache, Caddy uniquement pour faire du reverse proxy.

Mais avec **Gin**, on peut coder ça soit même !

### Contexte :

- Notre serveur **Gin** écoute en local sur le port 8080
- Mon serveur portainer tourne en local et écoute sur le port 9000
- Je veux qu'en me connectant sur mon serveur **Gin**, ce dernier me fasse un reverse proxy sur mon serveur portainer.

Dans mon fichier `main.go`, nous allons déclarer l'URL de mon reverse proxy ainsi qu'une méthode `proxy` qui sera la méthode utilisée par **Gin**

```
package main

import "github.com/gin-gonic/gin"

const reverseServerAddr = "http://127.0.0.1:9000"

func proxy(c *gin.Context) {

}

func main() {
    □
}
```

Nous dire à notre routeur **Gin**, que TOUTES les requêtes, et ce, peu importe la méthode, doit utiliser notre fameuse méthode `func proxy(c *gin.Context)`.

```
func main() {
    r := gin.Default()

    r.Any("/any", proxy)

    r.Run(":8080")
}
```

- `router.Any` signifie que quelle que soit la méthode (GET, POST, PUT, ...) utilisé, elle sera pris en charge.
- `/*any` est une expression indiquant à **Gin** que la route peut être n'importe quoi

Nous allons maintenant nous attaquer à la méthode `func proxy(c *gin.Context)` qui va dans un premier temps parser notre URL de destination (la variable `reverseServerAddr`) avec la librairie `net/url` :

```
func proxy(c *gin.Context) {

    proxy, err := url.Parse(reverseServerAddr)
    if err != nil {
        fmt.Printf("Error parsing reverse proxy address: %s\n", err)
        cIndentedJSON(http.StatusInternalServerError, gin.H{
            "message": "Error parsing reverse proxy address",
            "error":  err.Error(),
        })
        return
    }

}
```

On gère bien évidemment le cas d'erreur où on n'arriverait pas à parser correctement cette URL et on gère le renvoi d'une erreur au client, on arrête également la méthode avec `return`.

La variable `proxy` sera du type `*url.URL`.

Il suffit ensuite d'extraire la requête de notre contexte `c *gin.Context` et modifier son chemin ainsi que son protocole par celui de notre `proxy`.

```
func proxy(c *gin.Context) {
    ...
}
```

```
req := c.Request
req.URL.Scheme = proxy.Scheme
req.URL.Host = proxy.Host
}
```

Notre requête est prête, nous allons maintenant l'exécuter et récupérer son retour

```
func proxy(c *gin.Context) {
    ...

    transport := http.DefaultTransport
    resp, err := transport.RoundTrip(req)
    if err != nil {
        fmt.Printf("Error making request: %s\n", err)
        cIndentedJSON(http.StatusInternalServerError, gin.H{
            "message": "Error making request",
            "error":   err.Error(),
        })
        return
    }
}
```

Là encore, si une erreur survient, on la dirige correctement et on met fin à l'exécution de la méthode.

- `http.DefaultTransport`
- `transport.RoundTrip(req)`

Ce sont des méthodes de la librairie `net/http` et permettent d'exécuter une seule requête web.

Maintenant que la requête a été effectuée et que son retour est récupéré, il faut maintenant la donner à notre réponse et notre reverse proxy sera complet.

```
func proxy(c *gin.Context) {
    ...

    for headerKey, headerValues := range resp.Header {
        for _, headerValue := range headerValues {
```

```
    c.Header(headerKey, headerValue)
  }
}
defer resp.Body.Close()
bufio.NewReader(resp.Body).WriteTo(c.Writer)
return
}
```

Ce que nous faisons ici est :

- Nous récupèrerons l'en-tête de notre réponse et les passons à notre contexte (notre vraie réponse).
- Nous passons le body de notre réponse à celui de notre retour aussi.

---

C'est parti pour tester tout ça !

On lance notre application et on se rend sur notre adresse `localhost:8080`

Untitled

Untitled

Untitled

# Load Balancer

[Surprise ....](#)