Go-Web Documentation

Release 0.3.x-beta

Roberto Ferro

Aug 25, 2021

TABLE OF CONTENTS

CHAPTER

ONE

WHY GO-WEB?

Go-Web adopts a "convention over configuration" approach similarly to frameworks like Laravel ,Symfony and Rails.

By following this principle, Go-Web reduces the need for "repetitive" tasks like explicitly binding routes, actions and middleware; while not problematic per se, programmers may want to adopt "ready-to-use" solutions, for instances if they want easily build complex services, aiming at a reduced "productive lag".

Programmers may want to use existing frameworks like Gin-Gonic and Go Buffalo, but Go-Web differs from them because of the aforementioned "convention over configuration" approach.

1.1 Installation

1.1.1 Standard installation

You can download and install Go-Web by following these steps:

- Download Go-Web release from GitHub
- Extract the content in you project root
- Rename the *config.yaml.example* file in the project root by removing the *.example* suffix
- Download all dependencies by executing *go mod download* in project root
- Execute *go run . show:commands* to see all available GWF command

1.1.2 Docker

Go-Web provides a docker-compose.yml file that allows developers to easily set up a new development environment: this requires both Docker and Docker-compose installed on the development system.

Note: The docker-compose.yml defines several services, i.e. it is configured for providing instances ofMySQL, Redis, MongoDB and ElasticSearch; if needed, instances of other services may be added by modifying the docker-compose.yml file.

1.2 Alfred

Alfred is the command-line interface included with Go-Web. It provides a number of helpful commands that can assist you while you build your application. You can compile *Alfred* by running *sudo make build* in your project root.

Usage: *./alfred –help*

Table 1: Alfred available commands

Commands	Descriptions
–help, -h	Shows help menu
–make-controller, -mC <controller_name>	Creates new Go-Web controller*
–make-command, -mCMD <command_name>	Creates new Go-Web command*
–make-model, -mM <model_name>	Creates new Go-Web model*
–make-migration, -mMDB <migration_name>	Creates new Go-Web SQL migration*
–make-middleware, -mMW <middleware_name>	Creates new Go-Web SQL middleware*
–make-job, -mJ <job_name>	Creates new Go-Web async job*
–show-route, -sR	Shows service routes*
–migrate-up, -mU	Executes migrations*
–migrate-rollback, -mR <steps>	Executes migrations rollback*
–generate-key, -gK	Generate new application key*

* *Run this command only in project root*

1.3 Architecture

The architecture of Go-Web uses few components, namely:

- HTTP Kernel
- Service Container
- Controllers
- Middleware
- Views

Go-Web uses the so-called kernel in conjunction with the Service Container, file routes.yml and dependency gorilla/mux to build the map that routes each incoming HTTP request to the appropriate method of a specific controller: after the initialization process, requests will be processed by the Go-Web black box. *Figure 1 illustrates this process.*

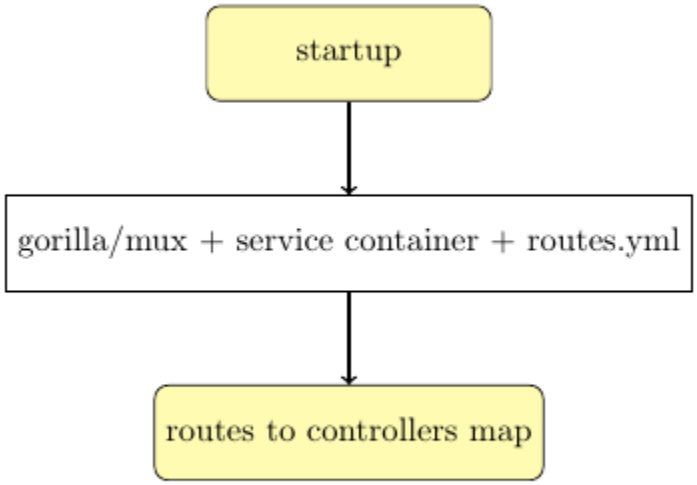
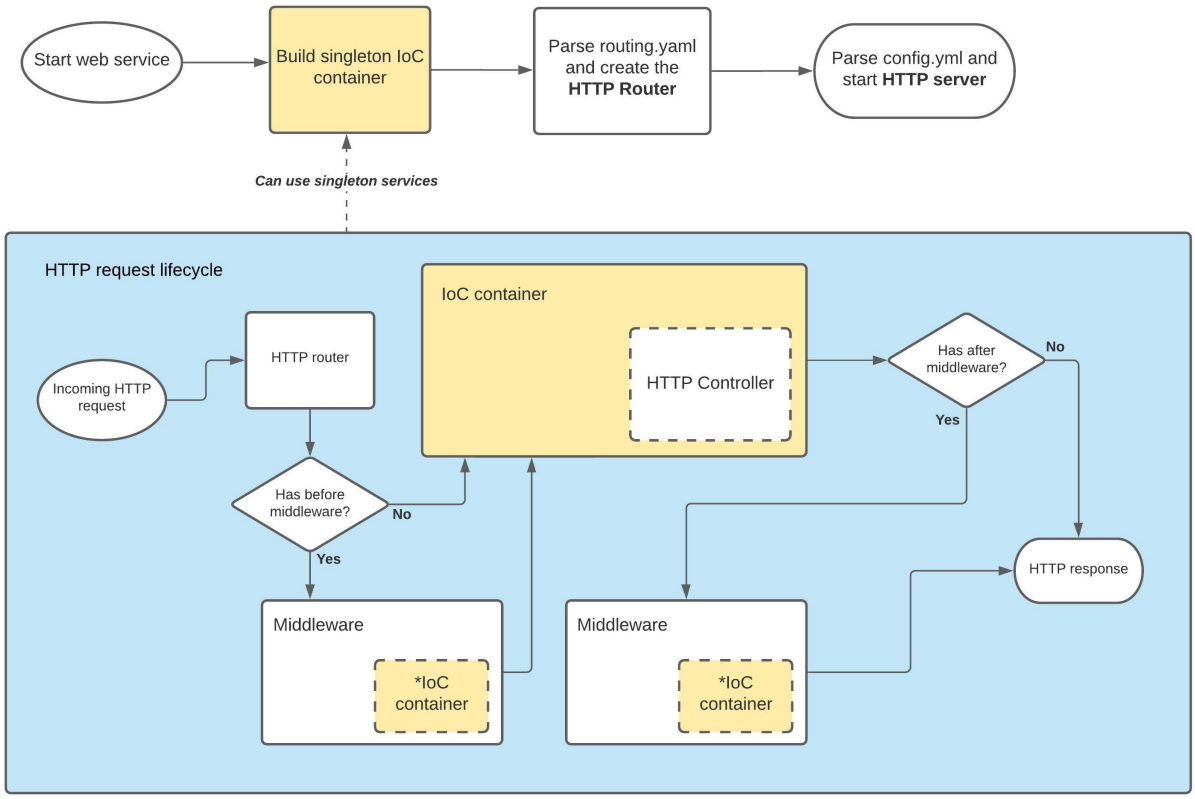


Figure 1: Go-Web workflow: initialization

The startup process is executed once and uses reflection so that the service container can inject relevant bits of code into controllers; this approach allows controllers to access any service defined in the service container, like databases or log systems, reducing redundant ("boilerplate") code. While resolving a route is done by gorilla/mux, the execution of the code associated to the same route is performed (or tunneled) by the Service Container, which injects dependencies into the end-point controller: before going through a controller, a request may be processed by one or more middlewares.

This workflow can be easily understood by looking at figure 2.



After being received by the Go-Web "black box", a request may follow workflow starting in entry points A or B, figure 2.

1.3.1 Service container

The service container (fig. 3) is the tool that manages class dependencies and performs dependency injection through DIG2. By default, Go-Web implements some services, specifically it leverages some libraries like gorilla/mux, gorm and more.

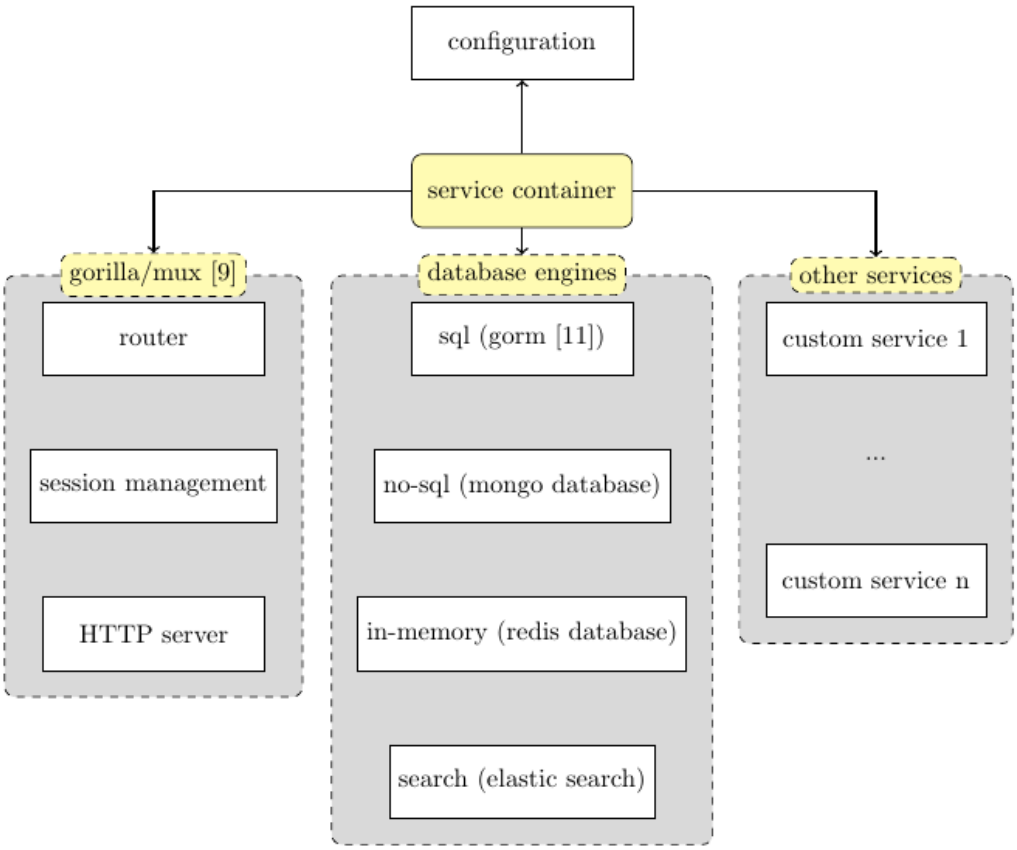


Figure 3: Go-Web services included in Service Container

As depicted in figure 3, the service container will register other services if correctly linked in the kernel: the process requires the implementation of such new services and further "registration" by adding them to Services array defined in Go-Web kernel: *go-web/app/kernel/kernel.go*

1.4 Configuration

Configuring a service is straightforward and developers can use config.yml.example as an example for further customization; Go-Web will look for file config.yml, thus the aforementioned config.yml.example can be copied with such name. The following listing demonstrates how a developer can configure both MySQL database and HTTP server.

```
database:
  driver: "mysql"
  host: "localhost"
  port: 3306
  database: "your_database_name"
  user: "<db_username>"
  password: "<db_password>"
server:
  name: "localhost"
  port: 8080
```

1.5 HTTP

Go-Web encapsulate Gorilla Mux to handles every HTTP requests. A *routing.yml* contains the definitions of every route.

1.5.1 Routing

Updating routes is simple and requires little changes to routing.yml file, which is located in the root folder of the project. The definition of a route is, in fact, straightforward and routes can be organized in groups.

A route is defined by:

- **path**
  - describes the URI of the route
  - a path is expressed as a string which could define parameters and supports regular expressions as gorilla/mux does
  - requests targeting undefined routes will cause a "client error" response with HTTP status 404 (not found)
  - example: *"/hello-world"*
- **action**
  - describes the destination of a route as a combination of a controller and one of its functions
  - an action is expressed as the string <controller name>@<function name>
  - if the action cannot be resolved (undefined controller or action), Go-Web will produce an error
  - example: **SampleController@Main**
- **method**
  - describes the HTTP verb supported by the route
  - **a method must be one of the verbs supported by HTTP, i.e.:**
    - GET
    - HEAD
    - POST
    - PUT
    - DELETE
    - CONNECT
    - OPTIONS
    - TRACE
    - PATCH
  - requests targeting an existing route with a wrong method (i.e. one that is not supported by the route) will cause a "client error" response with HTTP status 405 (method not allowed)
- **middleware (optionals)**
  - represents the ordered list of middlewares that will process the request received by the route before performing the route's action

  - the value of this property is a yml list of strings which must identify existing middlewares
  - example: Logging

- **descriptions (optionals)**
  - a string describing the purpose of the route
  - example: Returns JSON {"message": "Hello World"}

1.5.2 Controllers

Being a MVC framework, Go-Web encourages the use of controllers, i.e. containers of the business logic of the application. For instance, the controller named "SampleController" can be created by running command:

```
alfred -mC sample_controller
```

Go-Web will create the the .go file containing controller named "SampleController" in folder:

<go-web>/app/http/controller

The content of the newly created file will be:

Listing 1: Sample controller

```
package controller

import "github.com/RobyFerro/go-web-framework"

type SampleController struct{
    gwf.BaseController
}

// Main controller method
func (c *SampleController) Main(){
    // Insert your custom logic
}
```

When creating a controller, Go-Web will add to it the function Main, which could be expanded with some logic, as shown in listing 4; controllers can be extended by adding new public functions.

Listing 2: List 4: Sample controller

```
package controller

import (
    "github.com/RobyFerro/go-web-framework"
    "github.com/RobyFerro/go-web/exception"
)

type SampleController struct{
    gwf.BaseController
}

// Main controller method
func (c *SampleController) Main() {
```

(continues on next page)

(continued from previous page)

```
    _, err := c.Response.Write([]byte("Hello world")) if err != nil {
        exception.ProcessError(err)
    }
}
```

To gain access to everything underlying a Go-Web controller, including HTTP request and response, a controller needs to extend gwf.BaseController. Because the service container is used to "resolve" all controllers in Go-Web, developers can type- hint any of their dependency because they will be injected into the controller instance, as represented by the following code:

Listing 3: List 5: Dependency injection in controller

```
package controller

import (
    "github.com/RobyFerro/go-web-framework" "github.com/RobyFerro/go-web/database/model"
↪"github.com/jinzhu/gorm"
)

type SampleController struct{
    gwf.BaseController
}

// Main controller method
func (c *SampleController) Main(db *gorm.DB) {
    var user model.User
    if err := db.Find(&user).Error;err != nil {
        gwf.ProcessError(err)
    }
}
```

Note: both listings 4 and 5 includes a call to gwf.ProcessError(err); this is how Go-Web can handle errors, but developers may adopt another approach.

1.5.3 Middleware

Like controllers, a middleware can be created with command:

./goweb middleware:create <middleware name>

For instance, middleware named "Passthrough" can be created by running command:

./goweb middleware:create passthrough

After executing the command, the newly created middleware will be available in folder:

<go-web>/app/http/middleware

As described previously, middlewares can be used for pre/post processing requests.

Note: Check Gorilla Mux Middleware definition to more info about middlewares.

1.5.4 Authentication

By default, Go-Web provides two ways for authenticating users:

- JWT-based authentication
- basic (base) authentication

JWT Authentication

Commonly used to authenticate users thought mobile applications or a SPA, JWT authentication is implemented by function JWTAuthentication of controller AuthController or, in Go-Web terms, by endpoint **AuthController@JWTAuthentication**

The JSON structure used to represent credentials of a user must conform to JSON

```
{
    "username": <string, mandatory>,
    "password": <string, mandatory>
}
```

The result of a successful login attempt with this type of authentication is a HTTP response containing a JWT token. Resource access can be restricted only to authenticated users by adding middleware Auth to specific routes.

Basic authentication

Basic, or base, authentication is the simplest way to authenticate users for service access; this method is implemented by endpoint **AuthController@BasicAuth**

The base authentication requires the same data structure as JWT-based method and routes can be protected by using middleware BasicAuth.

1.6 Database

1.6.1 Models

In MVC frameworks models are responsible of the database interaction logic. Go-Web take advantage of GORM library to provide them.

To create a new model you can use its specific CLI command:

```
alfred -mM <model_name>
```

Models are located in: *<go-web>/database/model*

> **Warning:** After manually creating a model, developers may need to register it: to do so, the controller needs to be added to Models list defined in *<go-web>/register.go*

1.6.2 Migration

Migrations are like version control for your database, allowing your team to easily modify and share the application's database schema. Developers can creates new migration as follows:

```
alfred --mMDB <migration_name>
```

Developer can find its newly created migration files in <go-web>/database/migration directory.

1.6.3 Seeding

By implementing "Seed" method you'te able to seed your table with a "fake" data. Go-Web uses *https://github.com/brianvoe/gofakeit* as faker library.

See the below example:

```
// Execute model seeding
func (User) Seed(db *gorm.DB) {
        for i := 0; i < 10; i++ {
                password := gofakeit.Password(true, true, true, true, false, 32)
                encryptedPassword, _ := bcrypt.GenerateFromPassword([]byte(password), 14)

                user := User{
                        Name:     gofakeit.FirstName(),
                        Surname:  gofakeit.LastName(),
                        Username: gofakeit.Username(),
                        Password: string(encryptedPassword),
                }

                if err := db.Create(&user).Error; err != nil {
                        exception.ProcessError(err)
                }
        }
}
```

Seeder may be executed by running the following command:

```
./goweb database:seed <model_name>
```

This executes the specified model seeder. Omitting the model name the command will run every model seeder's.

1.7 CLI Interface

Go-Web has a built-in command-line interface that makes easy for developers to use the framework. Before using the CLI, the developer needs to compile the project by running command: *go build goweb.go*

After compiling the project, the CLI can be used to view all commands supported by Go-Web *./goweb show:commands*

The following listing table shows commands presented to the user by show:command:

Table 2: Alfred available commands

Commands	Descriptions
database:seed <model_name>	Executes seeder (all available models if is not specified)
show:commands	Shows all custom CLI commands
server:run	Run Go-Web server normally

1.7.1 Create custom commands

Go-Web command line interface (CLI) can be extended by running command

```
alfred -mCMD <command_name>
```

Before being available to Go-Web, commands must be registered in *register.go*. The following listing shows the registration of command Greetings:

> **Warning:** Command must be registered in the *register.go* file located in project root directory.

```
Commands = gwf.CommandRegister{
        List: map[string]interface{}{
                "queue:failed": &console.QueueFailed{},
                "queue:run":    &console.QueueRun{},
                "greetings":    &console.Greetings{}, // new
                // Here is where you've to register your custom commands
        },
}
```

The command registration Commands variable is used by Go-Web to recognize and list supported commands.

1.8 Asynchronous jobs

Go-web allows developers to create and schedule asynchronous jobs that will be dispatched in a queue.

Like controllers, models and other entities, a job can be created with the CLI by running command:

    ./goweb job:create <job name>

Go-Web uses Redis to manage queues and developers can handle jobs with functions Schedule and Execute.

The following listing illustrates an example of a Go-Web job:

```
data := job.MailStruct {
    Message: "Hello world",
    To: []string { "test@test.com", "test@test1.com" },
}

payload, _ := json.Marshal(data) j := job.Job {
    Name: "Send email"
    MethodName: "Mail"
    Params: job.Param { Name: "message", Payload: string(payload), Type: "int" },
}
```

(continues on next page)

(continued from previous page)

```
j.Schedule("default", c.Redis)
```

Once scheduled, a job can be run with CLI command:

./goweb queue:run <queue name>

The default queue can be run with command:

./goweb queue:run default

1.9 Front-End

1.9.1 Introduction

While Go-Web does not dictate which technology developers should use when building applications consuming APIs made with this Go-Web framework, it does provide foundations for building apps by suggesting the use of React and Redux.

Front-end assets and files can be found in folder **go-web/assets**

Specifically, the structure of the assets folder is simple and contains following sub folders:

- **js**
  - contains JavaScript files used by the front-end application;
- **css**
  - contains JavaScript files used by the front-end application;
- **view**
  - contains HTML views

Because Go-Web suggests using React and Redux, developers who want to use this stack must install appropriate tools on the development machine, like NodeJS and NPM ; this document will not cover React, Redux or other front-end related topics other than few "basic" concepts. For instance, the core single page application can be installed by running command

npm install

1.9.2 Views

Views are implemented by package http/template; Go-Web provides a simple helper to return a view from a controller, as reported in following example:

```
func (c *ViewController) Main() {
    helper.View("index.html", c.Response, nil)
}
```

Helper function View accepts three parameters:

- the view's path
- the HTTP response returned by the controller
- the interface used to "fill" the view

1.10 Compile and run

If you'd like to run Go-web in order to try your new implementation you can run the following command:

```
// Make sure to run this command in project root
make gwf-run
```

To compile the entire project you just need to run:

```
// Make sure to run this command in project root
make gwf-build
```

Then to start you can simply run of your Go-Web commands:

```
./goweb server:run // Run server normally
./goweb server:daemon // Run server as a daemon
```

Tip: Check *./goweb show:commands* to see all Go-Web available commands

The server will start listening on port defined in file config.yml.