Web service with Go and Go-Web

A Golang MVC framework for building web services.
Go-Web v0.3.x-beta - https://go-web.dev

Author: Roberto Ferro

Bergamo, Italy
2020-03-20

Contents

1 Why Go-Web?

Go-Web adopts a "convention over configuration" approach similarly to frameworks like Laravel [1], Symfony [2] and Rails [3].

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" which could be experienced when approaching a new technology.

Programmers may want to use existing frameworks like Gin-Gonic [4] and Go Buffalo [5], but Go-Web differs from them because of the aforementioned "convention over configuration" approach.

2 Installation

You can install Go-Web as follows[1]:

```
mkdir <project_folder>
go mod init <your_project_name>
go get github.com/RobyFerro/go-web@v0.1.0-alpha
cp $GOPATH/bin/go-web /usr/bin/goweb
sudo go-web install <project_directory>
sudo chown -R user:group <project_directory>
sudo chmod -R +w <project_directory>
```

Listing 1: Installation commands

If you're using Visual Studio Code you can install Go-Web framework with it's specific extension: `https://marketplace.visualstudio.com/items?itemName=Roberto.go-web`

3 Architecture

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

- the HTTP *kernel*
- the Service Container3.1
- Controllers
- Middlewares
- Views

Go-Web uses the so-called *kernel* in conjunction with the Service Container, file `routes.yml` and dependency `gorilla/mux`[9] 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.

[1] The installation and update process will improve in the future release

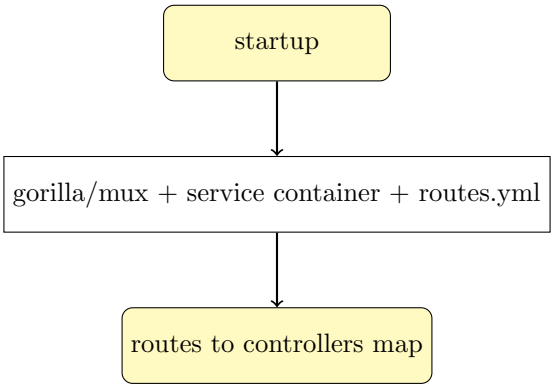

Figure 1: Go-Web workflow: initialization

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.

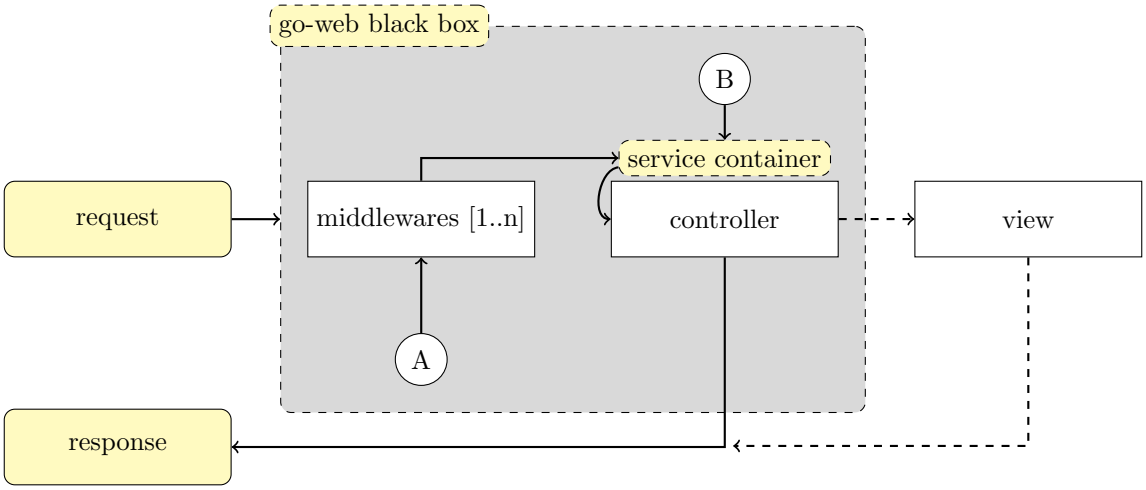

Figure 2: Go-Web workflow: request processing

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

3.1 Service container

The service container (fig. 3) is the tool that manages class dependencies and performs dependency injection through `DIG`[2].

By default, Go-Web implements some services, specifically it leverages some libraries like gorilla/mux ([9]), gorm ([11]) and more.

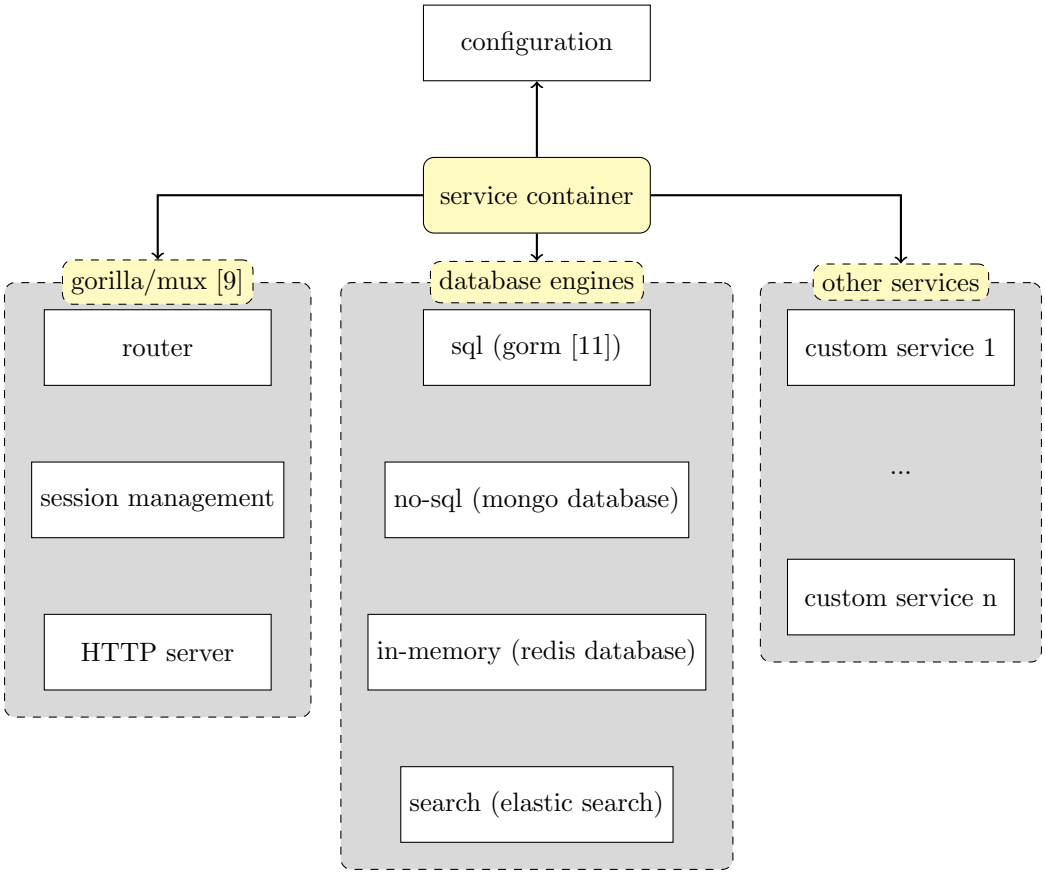

Figure 3: Go-Web services included in Service Container

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`

[2]See Uber DIG - https://godoc.org/go.uber.org/dig.

4 Development environment setup

Go-Web provides a `docker-compose.yml` file that allows developers to easily set up a new development environment: this requires both Docker [7] and Docker-compose [8] installed on the development system.

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

5 Service 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
```

Listing 2: config.yml configuration

6 The command line 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 (see 1):

```
./goweb show:commands
```

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

Command	Description
migration:create ⟨name⟩ migration:up migration:rollback ⟨steps⟩ database:seed ⟨name⟩	Creates a new migration Executes migrations Rolls migrations back Executes the database seeder
model:create ⟨name⟩ controller:create ⟨name⟩ middleware:create ⟨name⟩ cmd:create ⟨name⟩	Creates a new database model Creates a new controller Creates a new middleware Creates a new command
server:run server:daemon	Runs Go-Web server Runs Go-Web server as a daemon
install	Installs Go-Web
show:route show:commands	Shows active Go-Web routes Shows Go-Web commands list
job:create ⟨name⟩ queue:failed queue:run ⟨name⟩	Creates a new asynchronous job Moves failed jobs into a queue Runs a specific queue

Table 1: Go-Web commands

7 Creating a controller

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:

```
./goweb controller:create sample
```

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:

```
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
}
```

Listing 3: Sample controller

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.

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

Listing 4: SampleController@Main returns "Hello World"

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 listing 5:

```
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)
    }
}
```

Listing 5: Dependency injection in controller

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.

7.1 Registering a controller

After creating a controller, developers need to register it into Go-Web kernel: to do so, the controller needs to be added to `Controllers` list defined in

`⟨go-web⟩/app/kernel/conf.go`

7.2 Binding controllers to routes

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 ([9]
  - 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` (*optional*):
  - 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`
- `description` (*optional*):
  - a string describing the purpose of the route
  - example: Returns JSON {`‘‘message’’: ‘‘Hello World’’`}

8 Running the HTTP server

After creating a controller, registering it in the kernel and creating the binding with a route, the developer can run the server to ensure that the solution works; running a server can be done by running Go-Web like

```
./goweb server:run
```

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

9 Creating a middleware

Like controllers (section 7), 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 in sub section 7.2, middlewares can be used for pre-processing requests received by routes defined in file `routing.yml`.

10 Creating 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" },
}

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

Listing 6: Async job

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
```

11 Creating CLI commands

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

```
./goweb cmd:create <name>
```

Before being available to Go-Web, commands must be registered in the console kernel at

```
⟨go-web⟩/app/console/kernel.go
```

The following listing shows the registration of command `Greetings`:

```
var (
    Commands = map[string]interface{} {
        /* Default commands */
        "migration:up":       &command.MigrationUp{},
        "migration:create":   &command.MigrationCreate{},

        /* ---------------------------------------  *
         * Other default commands                   *
         * ---------------------------------------  */

        "job:create":         &command.JobCreate{},
        "install":            &command.Install{},
        /* Default commands */

        /* Registration of Greetings command */
        "greetings":          &command.Greetings{},
    }
)
```

Listing 7: Commands register

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

12 Database handling

12.1 Create new model

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:

```
./goweb model:create <model name>
```

Models are located in `<go-web>/database/model` directory.

12.2 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:

```
./goweb migration:create <migration_name>
```

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

12.3 Seeding

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

```
// 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)
		}
	}
}
```

Listing 8: Seeding

13 Authentication

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

- JWT-based authentication
- basic (base) authentication

13.1 JWT-based authentication

Commonly used to authenticate users thought mobile applications or a SPA, JWT [15] 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>
}
```

Listing 9: Credential structure

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.

13.2 Basic (base) 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 (see listing 9) and routes can be protected by using middleware `BasicAuth`.

14 More than APIs: React and Redux scaffolding

14.1 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 all style sheet files used by the front-end application, like *SCSS*, *SASS*, *LESS* and other
- `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
```

14.2 Using 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)
}
```

Listing 10: View helper

Helper function `View` accepts three parameters:

- the view's path
- the HTTP response returned by the controller
- the interface used to "fill" the view[3]

[3] See `http/template` package documentation

15 About the future

Go-Web is currently under active development and aims to become a valid, *Go* alternative to well known frameworks like Rails and Laravel.

16 Appendix

16.1 Out of CLI command “show:commands”

```
  ________                    __      __         ___.
 /  _____/   ____           /  \    /  \  ____  \_ |__
/   \  ___  /  _ \   ______ \   \/\/   /_/ __ \  | __ \
\    \_\  \(  <_> ) /_____/  \        / \  ___/  | \_\ \
 \______  / \____/            \__/\  /   \___  > |___  /
        \/                        \/        \/      \/
Go-Web CLI tool - Author: roberto.ferro@ikdev.eu
+-----------------------------+--------------------------------+
|          COMMAND            |          DESCRIPTION           |
+-----------------------------+--------------------------------+
| migration:up                | Execute migration              |
| migration:create <name>     | Create new migration files     |
| server:daemon               | Run Go-Web server as a daemon  |
| middleware:create <name>    | Create new middleware          |
| show:route                  | Show active Go-Web routes      |
| show:commands               | Show Go-Web commands list      |
| job:create <name>           | Create new async job           |
| install                     | install Go-Web                 |
| queue:failed                | Move failed jobs into a queue  |
| queue:run <name>            | Run a specific queue           |
| database:seed <name>        | Execute database seeder        |
| controller:create <name>    | Create new controller          |
| cmd:create <name>           | Create new command             |
| migration:rollback <steps>  | Rollback migrations            |
| server:run                  | Run Go-Web server              |
| model:create <name>         | Create new database model      |
+-----------------------------+--------------------------------+
```

References

[1] Laravel is a free, open-source PHP web framework, created by Taylor Otwell and intended for the development of web applications following the model–view–controller (MVC) architectural pattern and based on Symfony. Some of the features of Laravel are a modular packaging system with a dedicated dependency manager, different ways for accessing relational databases, utilities that aid in application deployment and maintenance, and its orientation toward syntactic sugar.
`https://laravel.com/`

[2] Symfony is a PHP web application framework and a set of reusable PHP components/libraries. It was published as free software on October 18, 2005 and released under the MIT license.
`https://symfony.com/`

[3] Ruby on Rails, or Rails, is an MVC server-side web application framework written in Ruby.
`https://rubyonrails.org/`

[4] One of the major Golang web framework
`https://gin-gonic.com/`

[5] Another Golang web framework
`https://gobuffalo.io/en/`

[6] Simple comparison between Golang and other major languages.
`https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go.html`

[7] Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. All containers are run by a single operating-system kernel and are thus more lightweight than virtual machines.
`https://www.docker.com/`

[8] Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services.
`https://docs.docker.com/compose/`

[9] Package mux implements a request router and dispatcher. The name mux stands for "HTTP request multiplexer". Like the standard http.ServeMux, mux.Router matches incoming requests against a list of registered routes and calls a handler for the route that matches the URL or other conditions.
`https://www.gorillatoolkit.org/pkg/mux`

[10] A reflection based dependency injection toolkit for Go.
`https://github.com/uber-go/dig`

[11] An ORM library for Go lang.
`https://gorm.io/`

[12] An elegant wrapper around Webpack.
`https://laravel-mix.com/`

[13] Node.js is an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside of a browser.
`https://nodejs.org/`

[14] Npm (originally short for Node Package Manager) is a package manager for the JavaScript programming language.
`https://www.npmjs.com/`

[15] JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties.
`https://jwt.io/`