Kwiscale Documentation

Release v1

Patrice Ferlet

November 13, 2015

Contents

Contents:

CHAPTER 1

Kwiscale Framework

Welcome to the kwiscale documentation.

1.1 What is Kwiscale

Kwiscale is a framework built following common model. It provides methods to create handlers holding HTTP verbs that are called following the client request.

Kwiscale can be used to create website, API or Websocket server.

Basically, Kwiscale offers a way to create website built on MVC.

1.2 What is not Kwiscale

Kwiscale is not a CMS, not a blog engine... It's a framework that may be used to create CMS or blog engine, or REST API server. If you need comparaison, Kwiscale is more like Symfony or Zend for PHP. But Kwiscale is made in Go.

1.3 Framework design

Kwiscale is a HTTP Handler framework. As WebApp2 for Python, the request process is working in this order:

- User call a route with HTTP Verb (GET, POST, HEAD...)
- Application fetch a *handler* that matches this route
- If the handler exists, application instanciate this handler and call the given HTTP verb as a method (Get(), Post()...)
- If route doesn't match, a HTTP 404 ERROR is sent to client

1.4 CLI

A Command Line Interface is provided to help application managment. The next documentation section delivers command to use along the devlopment process.

1.5 About Go conventions

You will notice that Kwiscale doesn't use the largely used handler function design that takes *http.ResponseWriter* and *http.Request*. Also, Kwiscale use a *complex* structure composition to *simulate* class/methods purpose. It's important to understand that choice.

The main goal of Kwiscale is to make web application development as easy as possible. Even if recommandation is to not follow classic "OOP" design, it's not prohibited to use some of interessing concepts comming from "OOP".

That's why we decided to implement methods that deals with ResponseWriter and Request internaly, letting developpers to use

```
h.WriteString("Hello")
//or
h.Render("mytemplate.html", context)
```

So, you will not find the *standard* and largely used:

```
func Get (w http.ResponseWriter, r *http.Request)
```

But you will be able to get this values if you really need them:

```
func (h *Handler) Get(){
    w := h.GetResponse()
    r := h.GetRequest()
}
```

1.6 Extensible

Kwiscale provides functions to plug some addons. For example you may use Pongo2 template engine or build your own Session Handler for "memcache".

1.7 What a strange name

kwiscale is a transformation of a french word: Quiscale that is a bird classification. The word is rarely used in french. So why that name ? That's simple. I was searching a name for the framework and, because I didn't find any idea, I used the "random page" link on Wikipedia website. After 10 clicks, I saw this name that I decided to keep.

CHAPTER 2

Getting Started

2.1 Prerequists

You have to install `go` and have set `$GOPATH` to point on a writable directory.

You need to set `$PATH` to append `$GOPATH/bin`.

An example `.bashrc` modification:

```
export GOPATH=~/goproject
export PATH=$GOPATH/bin:$PATH
```

After having set those variables, you **must** reset your shell. Restart your session or call:

```
source ~/.bashrc
```

It's recommanded to install `goimports` command that kwiscale CLI will try to call:

```
go get -u golang.org/x/tools/cmd/goimports
```

Important: If you don't install `goimports`, kwiscale CLI may have problem to generate a working main.go file.

2.2 Installation

Kwiscale is a standard Go package, so you may install it with the `go get` command.

Please, **don't use github url** but use the gopkg.in url that provides versionning. , the package is at gopkg.in/kwiscale

Installation is made by the following command:

```
go get gopkg.in/kwiscale/framework.v1
```

The version `v1` is the current version. To use master version, please use `v0` (while it's not recommanded either you need a specific feature that is not yet in next version).

At this time, kwiscale is installed and you can develop service.

You may install kwiscale cli:

```
go get gopkg.in/kwiscale/framework.v1/kwiscale
```

Right now, if you set `$GOPATH/bin` in your `$PATH`, the "kwiscale" command should work:

```
$ kwiscale
NAME:
   kwiscale - tool to manage kwiscale application

USAGE:
   kwiscale [global options] command [command options] [arguments...]

VERSION:
   0.0.1

COMMANDS:
   new      Generate resources (application, handlers...)
   generate Parse configuration and generate handlers, main file...
   help, h  Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --project "kwiscale-app" project name, will set \
             $GOPATH/src/[projectname] [$KWISCALE_PROJECT]
   --handlers "handlers"    handlers package name \
             [$KWISCALE_HANDLERS]
   --help, -h           show help
   --generate-bash-completion
   --version, -v        print the version
```

2.3 Basic application

You may create and modify application by using the `kwiscale` cli or manually.

2.3.1 With CLI

It's recommanded to use environment variables to not repeat paths in command. To create an application named "kwiscale-tutorial", please set this environment variable:

```
export KWISCALE_PROJECT=kwiscale-tutorial
```

Now, create application:

```
kwiscale new app
```

This command should create a directory named `$GOPATH/src/kwiscale-tutorial`.

Create a new handler to respond to the `/` route that is the "index":

```
kwiscale new handler index "/"
```

This command makes changes in `$GOPATH/src/kwiscale-tutorial`:

- it appends "/" route in `config.yml`
- it creates `handlers/index.go` containing `IndexHandler` and register call
- it creates or change `main.go` to add route to the "app"

You may now edit `$GOPATH/src/kwiscale-tutorial/handlers/index.go` to add "Get" method

```
package handlers

import (
    "gopkg.in/kwiscale/framework.v0"
)

func init() {
    kwiscale.Register(&IndexHandler{})
}

type IndexHandler struct{ kwiscale.RequestHandler }

// Add this method to serve
func (h *IndexHandler) Get() {
    h.WriteString("Hello world")
}
```

2.3.2 Manually

With config file

Create a project directory

```
mkdir -p $GOPATH/src/kwiscale-tutorial/handlers
cd $GOPATH/src/kwiscale-tutorial
```

Now create `config.yml`:

```
listen: :8000
session:
  name: kwiscale-tutorial
  secret: Change this to a secret passphrase
```

Edit `./handlers/index.go`:

```
package handlers

import (
    "gopkg.in/kwiscale/framework.v1"
)

func init(){
    kwiscale.Register(&IndexHandler{})
}

type IndexHandler struct{ kwiscale.RequestHandler }

// Add this method to serve
func (h *IndexHandler) Get() {
    h.WriteString("Hello world")
}
```

Now, create `main.go`:

```
package main

import (
```

```
    _ "kwiscale-tutorial/handlers"
    "gopkg.in/kwiscale/framework.v1"
)

func main(){
    app := kwiscale.NewAppFromConfigFile()
    app.ListenAndServe()
}
```

Note: `handlers` package is imported with an underscore here. As you can see, we don't use the package in `main.go` but `app` will register handlers itself. If the package is not imported, application will panic.

Without config file

Create a project directory

```
mkdir -p $GOPATH/src/kwiscale-tutorial/handlers
cd $GOPATH/src/kwiscale-tutorial
```

Edit `./handlers/index.go`:

```
package handlers

import (
    "gopkg.in/kwiscale/framework.v1"
)

func init(){
    // not mandatory but recommanded if you want
    // to use config.yml file later to map routes.
    kwiscale.Register(&IndexHandler{})
}

type IndexHandler struct{ kwiscale.RequestHandler }

// Add this method to serve
func (h *IndexHandler) Get() {
    h.WriteString("Hello world")
}
```

Create a `main.go` file:

```
package main

import (
    "kwiscale-tutorial/handlers"
    "gopkg.in/kwiscale/framework.v1"
)

func main(){
    // Create a new application (nil for default configuration)
    app := kwiscale.NewApp(nil)

    // Add a new route
    app.AddRoute("/", &handlers.IndexHandler{})

    // start service
```

```
    app.ListenAndServe()
}
```

2.4 Launch application

Go to the project path and launch:

```
go run main.go
```

By default, application listens ":8000" port. You may now open a browser and go to http://127.0.0.1:8000.

The page should display "Hello you", if not please check output on terminal

2.5 Adding routes and handlers

The CLI helps a lot to create handlers and routes.

But you may create handlers and routes yourself inside `config.yml` file and appending your handler package file in application.

2.5.1 Create handler with CLI:

```
kwiscale new handler user "/user/{username:.+}"
```

2.5.2 Create handler without CLI:

In `handlers` directory, append a new file named "user.go"

In `config.yml` you have to set new route if you didn't use CLI:

```
routes:
  /:
    handler: handlers.IndexHandler
  /user/{username:.+}:
    handler: handlers.UserHandler
```

2.5.3 Both CLI and manually:

Now append a method to respond to GET:

```
package handlers

import (
    "gopkg.in/kwiscale/framework.v1"
)

func init(){
    // Mandatory if you are using config.yml to
    // map routes and handlers.
    kwiscale.Register(&UserHandler{})
}
```

```
// Our new handler
type UserHandler struct { kwiscale.RequestHandler }

func (h *UserHandler) Get(){
    // "username" should be present in route definition,
    // see config.yml later
    name := h.Vars["username"]

    // write !
    h.WriteString("User name:" + name)
}
```

As you can see, the route can take a “username” that should respect regular expression ”.+” (at least one char). The “username” key in the route definition will set `handler.Vars["username"]` in UserHandler.

Right now, routes and handlers are defined, you may relaunch application and open http://127.0.0.1:8000/user/Foo to display “Hello Foo” in you browser.

CHAPTER 3

Developping with Kwiscale

3.1 Behind the scene

kwiscale is a web framework that uses GorillaToolkit. The main purpose is to allow developers to create handlers that serve reponses.

There are two Handlers types:

- RequestHandler to respond to HTTP requests (Get, Post, Put, Delete, Patch, Trace, Head)
- WebSocketHandler to serve websocket connection to client

Kwiscale proposes addon system to be able to plug template engines and session engines. By default you may be able to use the standard html/template package provided by Go and session by encrypted cookies provided by GorillaToolkit.

3.2 Project Structure

3.2.1 Recommandation is not obligation

The common structure we give here is not mandatory. You can prefer other file structure and project managment.

3.2.2 The standard Kwiscale structure

In a common usage, the following file structure is recommanded:

```
[projectpath]/
    main.go
    handlers/
        index.go
        [other name].go
        ...
    templates/
        index.html
        - common/
            footer.html
            header.html
            menu.html
        - home/
            main.go
```

```
    statics/
        - js/
            ...
        - css/
            ...
```

Note that "handlers" directory may contains subpackages. The goal is to classify HTTP handlers in the same directory. An example:

```
handlers/
    index.go
    user/
        auth.go
        register.go
        profile-edition.go
    cms/
        page.go
        edit.go
    blog/
        index.go
        ticket.go
```

3.3 Handler story

When a user calls a route, Kwiscale will find the corresponding handler in a stack. When a route matches, kwiscale app detect handler type and call a serie of methods (see *Handler story diagram*)

3.4 Serve static files

Important The static handler provided by kwiscale is provided for development and **not for the production**. It's not recommanded to let Kwiscale serve directoy web application, you'd rather use HTTP Server as nginx or Apache as reverse proxy. That way, the HTTP server will serve static files instead of using static handler provided by Kwiscale.

To serve static files (css, js, images, and so on) you may configure Kwiscale.App like this:

```
cfg := kwiscale.Config{
    StaticDir: "./statics",
}
app := kwiscale.NewApp(&cfg)
```

Kwiscale uses the directory name to serve files that resides inside. You can now hit URL http://127.0.0.1:8000/statics/...

Note that static handler doesn't make directory index. Hitting the static route without any filename will result on 404 Error.

3.5 URL Routing

Kwiscale make use of GorillaToolkit route system. This routing implementation allows you to set url parameters and to reverse an url from a handler name.

Example:

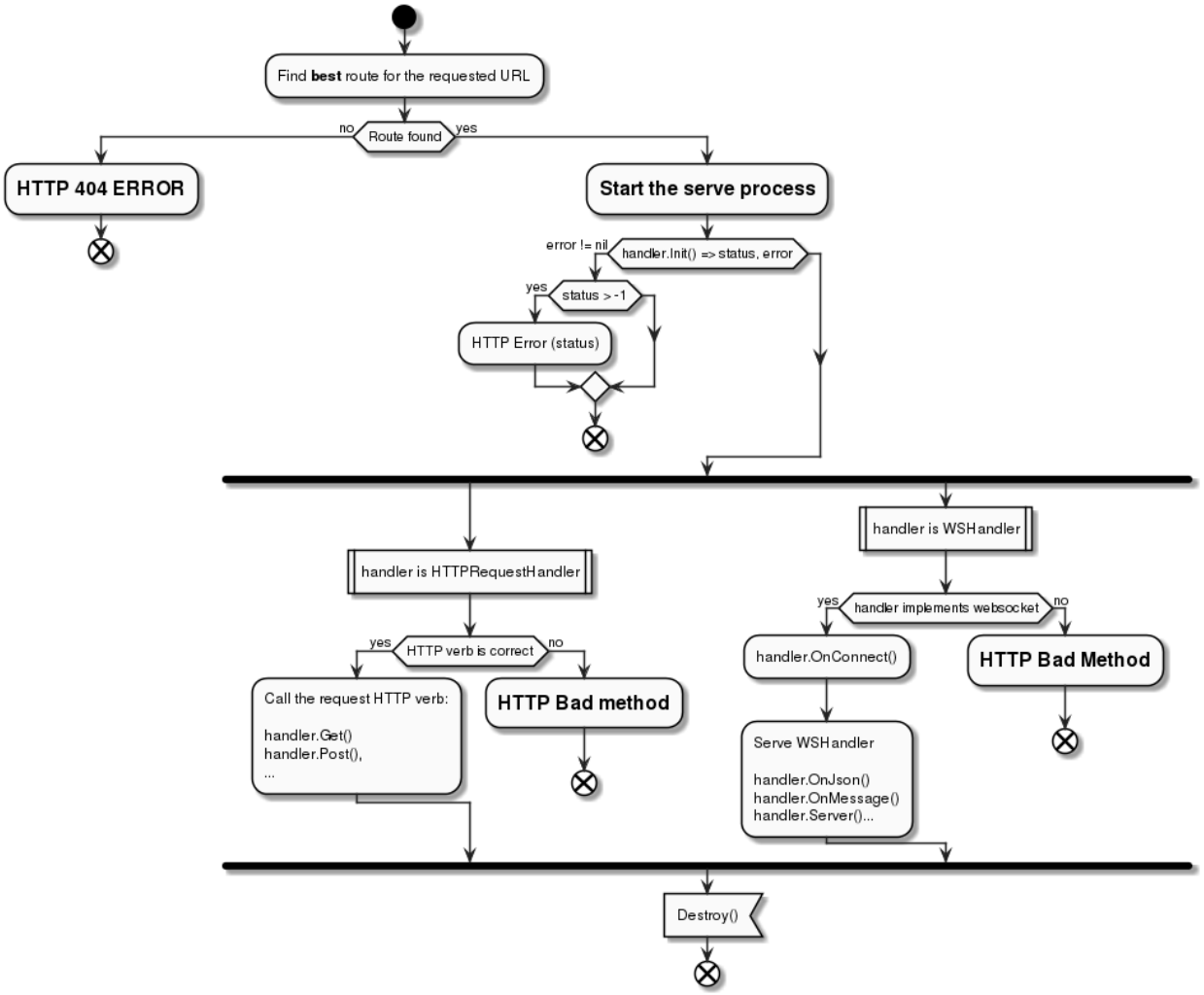

Fig. 3.1: Handler story diagram

```
type MyHandler struct { kwiscale.RequestHandler }

func (h *UserHandler) Get(){
    userid := h.Vars["userid"]
}

func main(){
    //...

    // Add a route that need an user id named "userid".
    // Route parameters are regular expression.
    app.AddRoute("/user/{userid:\d+}", UserHandler{})

    //...
}
```

The corresponding route could be "/user/123456", then in `Get()`, `userid` contains a string value: "123456".

To reverse an url, you need the name of the handler. The "kwiscale.App" can provide the named route and you may use `URL` to return the corresponding URL. Here is an example:

```
// Route /user/{userid:\d+}
url := myhandler.GetApp().GetRoute("main.UserHandler").URL("userid", "123456")


// If myhandler is the wanted handler
url := myhandler.GetURL("userid", "123456")
```

3.6 Named route

If you want to not use handler name based on reflected value, you may use `AddNamedRoute()` instead:

```
app.AddNamedRoute("/user/{userid:\d+}", UserHandler{}, "users")
```

So, to reverse URL:

```
// Route /user/{userid:\d+}
url := myhandler.GetApp().GetRoute("users").URL("userid", "123456")
```

CHAPTER 4

RequestHandler

4.1 Usage

RequestHandler handles HTTP verbs (Get, Post, Put, Delete, Head, Pathch, Trace, Option) as structure method.

It implements IBaseHandler, each HTTP verb is already implemented but returns a 404 Error by default. That way, you only have to create your own RequestHandler based type to implement the needed method.

4.2 Call story

When a client enter an URL, the framework finds the right handler to use. Then your own request handler is spawned (as a new instance) and a list of methods are called:

- `Init()` - you can override this method to initialize the response or reject client (usefull for authentification and authorisation check). This method should return an integer and a nil error to let handler continue. If error is not nil, the integer is used as status returne to the clien
- Http method - `Get()` or `Post()`, and so on
- `Destroy()` - Called after response is sent to client

You may override this methods. Note that `Init()` method must return integer status and an `error` (that should be `nil`) if you want to continue to serve with HTTP verb method.

Example:

```
type PrivateHandler struct { kwiscale.RequestHandler}

// Initialize - test is client is authenticated
func (h *PrivateHandler) Init(){
    isauth, ok := h.GetSession("auth")
    if !ok || !isauth.(bool) {
        return http.StatusForbidden, errors.New("Unauthaurized")
    }

    // authenticated user, we can continue
    return -1, nil
}

// When GET method happends.
func (h *HomeHandler) Get() {
    //...
```

```
}

// After reponse sent to the client.
func (h *HomeHandler) Destroy(){

}
```

This `PrivateHandler` can be used as a "parent" handler to privatize other handlers:

```
type AdminHandler { PrivateHandler }

// only if user is authenticated
func (ah *AdminHandler) Get(){
    //..
}
```

CHAPTER 5

Websocket Handler

5.1 Usage

WebSocketHandler will accept websocket connection and react on events. There are 3 ways to intercept client messages:

- on json message
- on text message
- serve in a loop

Using the URL path, WebSocketHandler provides way to send message in several form to :

- the connected client only
- the "room" clients
- the entire clients list connected to the server

Important Only one of `Serve()`, `OnJSON()` or `OnMessage()` method should be declared. If you declared more that one of this method, only one of those methods will be use. The priority order is:

1. Serve
2. OnJSON
3. OnMessage

5.2 Basic

The most common way to use websocket is to listen JSON message or text message. Then answer to the client.

To use JSON, you must implement WSJsonHandler, that means you should impement :

```
OnJSON (interface{}, error)
```

Example:

```
// A standard type to communicate
type Message struct {
    From string
    Message string
}
```

```
type MyWS struct { kwiscale.WebSocketHanlder}

func (w *MyWS) OnJSON(i interface{}, err error) {

    if err != nil {
        // an error occured
        return
    }

    // i is an interface{} type, you may cast type
    if i, ok := i.(Message); ok {
        //... work with message

        // Send response
        w.SendJSON(Message{
            From: "server",
            Message: "Hello",
        })
    }
}
```

If the `error` given as argument is not `nil`, that means that a problem occured with client connection. So the connection is probably closed. After the method returns, the connection will be removed. Client should reconnect itself to be able to communicate with the server.

To work with text message instead of JSON, you must implement WSStringHandler interface. That means you must implement

`OnMessage(string, err)`

Example:

```
type MyWS struct { kwiscale.WebSocketHanlder}

func (w *MyWS) OnMessage(s string, err error) {

    if err != nil {
        // an error occured
        return
    }

    // Send response as text
    w.SendText("Hello")
}
```

5.3 Serving WebSocket

You may implement your own server loop implementing `WSServer` interface, that means you may implement the method:

`Serve()`

The method should make a loop to read messages from client.

Example:

```
type MyWS struct {kwiscale.WebSocketHandler}
```

```
func (ws *MyWS) Serve() {
    conn := ws.GetConn();
    for {
        var i interface{}
        err := conn.ReadJSON(&i)
        if err != nil {
            break
        }

        // works with interface...

        // send message
        ws.SendJSON(map[string]string{
            "message" : "Hello !",
        })
    }
}
```

Using `Serve()` can be very usefull to make specific manipulation on connection or to customize some behaviours.

5.4 Rooms

In the following explanation, `XXX` shoud be replace by `JSON` or `Text`, respectivally to send JSON or string message. The complete list follows explanations.

Each websocket connection is kept in a named "room". A room is a compartimented list where resides connections. Each room is created using the websocket path given in url.

That could be very usefull if you want to create a chatroom with several channels.

For example, your website allows 2 routes to connect with websocket:

- '/chat/general'
- '/chat/administrators'

Then, in the handler, if you call one of the `SendXXXToThisRoom`method, each clients connected to the the route named "/chat/administrators" will receive the message, but **not** those that are only connected to "/chat/general".

To send message to the entire connected clients list, you may use one of the `SendXXXToAll()`.

Connected to another room, there is a way to send client to a specific room: `SendXXXToRoom(name string)`.

For JSON:

- `SendJSONToThisRoom(interface{})` to send json to this room
- `SendJSONToRoom(string, interface{})` to send json to a specific room
- `SendJSONToAll(interface{})` to send json to the entire clients list

For text:

- `SendTextToThisRoom(interface{})` to send text message to this room
- `SendTextToRoom(string, interface{})` to send text message to a specific room
- `SendTextToAll(interface{})` to send text message to the entire clients list

CHAPTER 6

Templates

kwiscale uses html/template from the built-in package of Go. You may use Pongo2 template engine using the kwiscale addon.

Kwiscale appends an override system based on a simple template comment that will allow you to reuse bases structure.

6.1 Built-in template engine

Create a template directory named "templates". Create a file named "templates/index.html" and append this content:

```
<!doctype html>
<html>
<head>
    <title>{{ .Title }}</title>
</head>
<body>
<div>
    {{ .Content }}
</div>
</body>
</html>
```

Then, in `main.go`:

```
package main

import (
    "gopkg.in/kwiscale/framework.v1"
)

type HomeHandler struct { kwiscale.RequestHandler }

func (h *HomeHandler) Get(){
    h.Render("index.html", map[string]string{
        "Title": "The title of the page",
        "Content" : "This is the content",
    })
}

func main(){
    app := kwiscale.NewApp(&kwiscale.Config{
        TemplateDir : "./templates",
```

```
    })
    app.AddRoute("/", &HomeHandler{})
    app.ListenAndServe()
}
```

6.2 Pongo2 template

Pongo2 is a template engine that is quasi compatible with Jinja2 (python) or Twig (PHP). Syntax is powerfull and designed to be easy to learn.

To use Pongo2 template, install addon:

```
go get gopkg.in/kwiscale/template-pongo2.v1
```

Create templates directory and set `templates/index.html`:

```
<!doctype html>
<html>
<head>
    <title>{% Title %}</title>
</head>
<body>
<div>
    {% Content %}
</div>
</body>
</html>
```

Then, in `main.go`:

```
package main

import (
    "gopkg.in/kwiscale/framework.v1"
    _ "gopkg.in/kwiscale/template-pongo2.v1"
)

type HomeHandler struct { kwiscale.RequestHandler }

func (h *HomeHandler) Get(){
    h.Render("index.html", map[string]string{
        "Title": "The title of the page",
        "Content" : "This is the content",
    })
}

func main(){
    app := kwiscale.NewApp(&kwiscale.Config{
        TemplateDir:    "./templates",
        TemplateEngine: "pongo2"
    })
    app.AddRoute("/", &HomeHandler{})
    app.ListenAndServe()
}
```

CHAPTER 7

Addons creation

Kwiscale provides extensibility for database, session and template engines.

7.1 Database Addons

7.1.1 Goal

Kwiscale aims to give a "database engine agnostic" database system to allows usage of a lot of database. To provide an ORM there are a lot of complexity to manage.

Kwiscale project refuses to reinvent the wheel and provides a simple interface to implement in addons.

Addons can:

- simply map some well known packages to the interface
- manage database itself

7.2 Template addons

7.2.1 Goal

Built-in template is based on "html/template" built-in package and doesn't need any dependency. But you may prefer to use other templates (eg. Pango2)

Kwiscale implements a template addons system to allows usage of other templates.

7.2.2 Build a template addon

Create a directory where you'll develop template addon. The package file should call `kwiscale.RegisterTemplateEngine()` function.

The package name is not important and will not be visible by developpers. But a common way to name the package is `kwiscaletemplate[name]`.

Commonly, you have to call this function in the `init()` function of your package.

```
package kwiscaletemplateexample

import(
    "gopkg.in/kwiscale/framework.v1"
)

func init(){
    kwiscale.RegisterTemplateEngine("example", MyTemplateEngine{})
}

// should implement kwiscale.Template interface
type MyTemplateEngine struct {
    //...
}
```

7.2.3 Interface

The interface to implement:

```
type Template interface {
 // Render method to implement to compile and run template
 // then write to RequestHandler "w" that is a io.Writer.
 Render(w io.Writer, template string, ctx interface{}) error

 // SetTemplateDir should set the template base directory
 SetTemplateDir(string)

 // SetOptions pass TplOptions to template engine
 SetTemplateOptions(TplOptions)
}
```

- Render(w, template, ctx) should write content in the writer "w". "template" is the template filename to use and "ctx" contains values to set in the template
- SetTemplateDir() should register where templates reside. The path comes from template.dir yaml value or Config.TemplateDir
- SetTemplateOptions() receive other configuration that comes from Config.TemplateOptions or templates.options yaml configuration. Some template engine may need some special configuration and they are provided that way

CHAPTER 8

Indices and tables

- genindex
- modindex
- search