Login and Register Example with JWT using Golang

Yusuf Syaifudin

September 22, 2018

Contents

1 apidoc/static.go

```go
package apidoc

import (
	"net/http"
	"strings"

	"github.com/yusufsyaifudin/go-bindata-assetfs"
)

// see this tutorial https://medium.com/@erinus/go-my-way-day-3-9c9b420ed43e
type apidocGoBindata struct {
	FileSystem http.FileSystem
}

func (apidocGoBindata *apidocGoBindata) Open(name string) (http.File, error) {
	return apidocGoBindata.FileSystem.Open(name)
}

func (apidocGoBindata *apidocGoBindata) Exists(prefix string, filepath string) bool {
	var err error
	var url string
	url = strings.TrimPrefix(filepath, prefix)
	if len(url) < len(filepath) {
		_, err = apidocGoBindata.FileSystem.Open(url)
		if err != nil {
			return false
		}
		return true
	}
	return false
}

func Static() *apidocGoBindata {
	var fs *assetfs.AssetFS
	fs = &assetfs.AssetFS{
		Asset:     Asset,
		AssetDir:  AssetDir,
		AssetInfo: AssetInfo,
	}
	return &apidocGoBindata{fs}
}
```

2 cmd/go-jwt-login-example/main.go

```
package main

import (
	"os"
	"os/signal"
	"syscall"

	"github.com/golang-migrate/migrate"
	"github.com/namsral/flag"
	"github.com/rs/zerolog/log"
	"github.com/yusufsyaifudin/go-jwt-login-example/pkg/auth"
	"github.com/yusufsyaifudin/go-jwt-login-example/pkg/db"
	"github.com/yusufsyaifudin/go-jwt-login-example/server"
)

var serverSecretKey = flag.String("secret-key", "ndjsHJUTUI8uok", "Server␣secret␣key")
var listenAddress = flag.String("listen-address", "localhost:8000", "Address␣to␣bind")
var dbUrl = flag.String("db-url", "postgres://postgres:postgres@localhost:5432/go-users?
    sslmode=disable", "Connection␣string␣to␣postgres")
var dbDebug = flag.Bool("db-debug", true, "Whether␣to␣show␣sql␣debug␣or␣not")
var logger = log.With().Str("pkg", "main").Logger()

// @title Authentication System
// @version 3.0
// @description This is a documentation for Authentication System
// @termsOfService http://example.com

// @contact.name API Support
// @contact.url http://www.example.com
// @contact.email contact.us@example.com

// @license.name Apache 2.0
// @license.url http://www.apache.org/licenses/LICENSE-2.0.html

// @host localhost:8000
// @BasePath /api/v1
func main() {
	flag.Parse()

	dbConfig := &db.Config{
		ConnectionString: *dbUrl,
		Debug:            *dbDebug,
	}

	dbConnection, query, err := db.NewGoPgQuery(dbConfig)
	defer dbConnection.Close()
	if err != nil {
		logger.Error().Err(err).Msg("database␣connection␣fail")
		return
	}

	if err := query.Migrate(); err != nil && err != migrate.ErrNoChange {
		logger.Error().Err(err).Msg("migration␣fail")
	}

	srv := &server.Config{
		ListenAddress:   *listenAddress,
		ServerSecretKey: *serverSecretKey,
		DB:              query,
		Auth:            auth.NewJwtAuth(),
	}

	var apiErrChan = make(chan error, 1)
	go func() {
		logger.Info().Msgf("running␣api␣at␣%s", *listenAddress)
		apiErrChan <- srv.Run()
	}()

	// to gracefully shutdown the server
	var signalChan = make(chan os.Signal, 1)
	signal.Notify(signalChan, os.Interrupt, syscall.SIGTERM)
	select {
	case <-signalChan:
		logger.Info().Msg("got␣an␣interrupt,␣exiting...")
		srv.Shutdown()
	case err := <-apiErrChan:
		if err != nil {
			logger.Error().Err(err).Msg("error␣while␣running␣api,␣exiting..."
			    )
		}
	}

}
```

3 internal/app/user/handler.go

```go
package user

import (
	"github.com/yusufsyaifudin/go-jwt-login-example/pkg/auth"
	"github.com/yusufsyaifudin/go-jwt-login-example/pkg/db"
)

type HandlerConfig struct {
	ServerSecretKey string
	DB              db.Query
	Auth            auth.Auth
}

func NewUserHandler(serverSecretKey string, db db.Query, auth auth.Auth) *HandlerConfig {
	return &HandlerConfig{
		ServerSecretKey: serverSecretKey,
		DB:              db,
		Auth:            auth,
	}
}
```

4 internal/app/user/helper.go

```go
package user

import (
	"golang.org/x/crypto/bcrypt"
)

// HashPassword will hash the user password using bcrypt algorithm with cost 10
func HashPassword(password string) (string, error) {
	bytes, err := bcrypt.GenerateFromPassword([]byte(password), bcrypt.DefaultCost)
	return string(bytes), err
}

// CheckPasswordHash will check two different hash is same or not
func CheckPasswordHash(password, hash string) bool {
	err := bcrypt.CompareHashAndPassword([]byte(hash), []byte(password))
	return err == nil
}
```

5 internal/app/user/login.go

```
package user

import (
        "context"
        "fmt"
        "strings"
        "time"

        "github.com/yusufsyaifudin/go-jwt-login-example/internal/pkg/model"
        "github.com/yusufsyaifudin/go-jwt-login-example/pkg/auth"
        "github.com/yusufsyaifudin/go-jwt-login-example/pkg/http"
)

/**
 * @api {post} /user/login Login
 * @apiVersion 1.0.0
 * @apiName Login
 * @apiGroup User
 *
 * @apiDescription User login
 *
 * @apiParam (Request body) {String} username Username of registered user
 * @apiParam (Request body) {String} password User password
 */
func (handler *HandlerConfig) LoginUserHandler(ctx context.Context, req http.Request)
    http.Response {
        form := &struct {
                Username string `json:"username" form:"username"`
                Password string `json:"password" form:"password"`
        }{}

        if err := req.Bind(form); err != nil {
                return http.NewJsonResponse(500, map[string]interface{}{
                        "error": map[string]interface{}{
                                "message": fmt.Sprintf("fail when binding the payload: %s
                                    ", err.Error()),
                        },
                })
        }

        if strings.TrimSpace(form.Username) == "" {
                return http.NewJsonResponse(400, map[string]interface{}{
                        "error": map[string]interface{}{
                                "message": "username cannot be empty",
                        },
                })
        }

        if strings.TrimSpace(form.Password) == "" {
                return http.NewJsonResponse(400, map[string]interface{}{
                        "error": map[string]interface{}{
                                "message": "password cannot be empty",
                        },
                })
        }

        // check user in database
        user := &model.User{}
        handler.DB.Raw(user, "SELECT * FROM users WHERE username = ? LIMIT 1", form.
            Username)
        if user == nil || user.ID == 0 {
                return http.NewJsonResponse(404, map[string]interface{}{
                        "error": map[string]interface{}{
```

```
                        "message": "user␣not␣found",
                    },
                })
        }

        if !CheckPasswordHash(form.Password, user.Password) {
                return http.NewJsonResponse(401, map[string]interface{}{
                        "error": map[string]interface{}{
                                "message": "wrong␣password",
                        },
                })
        }

        // if found, then check hashing password
        tokenPayload := &auth.Payload{
                ID:         fmt.Sprintf("%d", user.ID),
                Username:   user.Username,
                IssuedAt:   time.Now().Unix(),
                NotBefore:  time.Now().Unix(),
                ExpiredAt:  time.Now().Add(5 * time.Hour).Unix(),
        }

        accessToken, err := handler.Auth.GenerateToken(tokenPayload, handler.
            ServerSecretKey)
        if err != nil {
                return http.NewJsonResponse(422, map[string]interface{}{
                        "error": map[string]interface{}{
                                "message": fmt.Sprintf("fail␣generating␣access␣token:␣%s"
                                    , err.Error()),
                        },
                })
        }

        return http.NewJsonResponse(200, map[string]interface{}{
                "access_token": accessToken,
                "user": map[string]interface{}{
                        "id":             user.ID,
                        "name":           user.Name,
                        "username":       user.Username,
                        "registered_at": user.CreatedAt.Unix(),
                },
        })
}
```

6 internal/app/user/middleware.go

```
package user

import (
        "bytes"
        "context"
        "fmt"
        "io/ioutil"
        "strings"

        "github.com/yusufsyaifudin/go-jwt-login-example/internal/pkg/model"
        "github.com/yusufsyaifudin/go-jwt-login-example/pkg/http"
)

/**
 * @apiDefine MiddlewareAuthTokenCheck
 * @apiHeader {String} Authorization Must using Bearer access token.
 * @apiHeaderExample {json} Header-Example:
 *     {
 *       "Authorization": "Bearer your-access-token"
 *     }
 *
 * @apiParamExample {json} Request-Example:
 *     {
 *       "access_token": "your-access-token"
 *     }
 */
func (handler *HandlerConfig) MiddlewareAuthTokenCheck(next http.Handler) http.Handler {
        return func(parent context.Context, req http.Request) http.Response {
                var accessToken string

                // get access token from header
                headerAuthorization := req.RawRequest().Header.Get("Authorization")
                headerAuthorization = strings.TrimSpace(headerAuthorization)

                headerPart := strings.Split(headerAuthorization, "␣")
                if len(headerPart) < 2 {
                        headerPart = []string{"", ""}
                }

                if strings.ToLower(headerPart[0]) == "bearer" {
                        accessToken = headerPart[1]
                }

                // if not exist on header, try using body parameter
                if accessToken == "" {
                        body, err := ioutil.ReadAll(req.RawRequest().Body)
                        if err != nil {
                                return http.NewJsonResponse(500, map[string]interface{}{
                                        "error": map[string]interface{}{
                                                "message": fmt.Sprintf("%s:␣%s", "error␣
                                                    when␣reading␣the␣request␣body", err.
                                                    Error()),
                                        },
                                })
                        }

                        // copy twice to make sure body can be re-binding after
                            middleware
                        body1 := ioutil.NopCloser(bytes.NewBuffer(body))
                        body2 := ioutil.NopCloser(bytes.NewBuffer(body))

                        var form struct {
                                AccessToken string `json:"access_token" form:"
                        access_token"`
                }

                // binding the data using body 1
                req.RawRequest().Body = body1
                req.Bind(&form)

                accessToken = form.AccessToken

                // set copied body to raw request body again
                req.RawRequest().Body = body2
        }

        jwtPayload, err := handler.Auth.ValidateToken(accessToken, handler.
            ServerSecretKey)
        if err != nil {
                return http.NewJsonResponse(403, map[string]interface{}{
                        "error": map[string]interface{}{
                                "message": fmt.Sprintf("%s:␣%s", "error␣when␣
                                    validating␣access␣token", err.Error()),
                        },
                })
        }

        // jwtPayload.ID
        sqlGetUser := `SELECT * FROM users WHERE id = ? LIMIT 1;`

        // check user in database
        user := &model.User{}
        handler.DB.Raw(user, sqlGetUser, jwtPayload.ID)
        if user == nil || user.ID == 0 {
                return http.NewJsonResponse(401, map[string]interface{}{
                        "error": map[string]interface{}{
                                "message": "cannot␣continue␣this␣request␣since␣
                                    user␣is␣not␣found␣with␣this␣token",
                        },
                })
        }

        req.SetUser(user)

        // run the wrapped handler
        return next(parent, req)
    }
}
```

7 internal/app/user/profile.go

```go
package user

import (
        "context"

        "github.com/yusufsyaifudin/go-jwt-login-example/pkg/http"
)

/**
 * @api {get} /user/profile Profile
 * @apiVersion 1.0.0
 * @apiName Get Profile
 * @apiGroup User
 *
 * @apiDescription Get user profile, based on authentication header.
 *
 * @apiHeader {String} Authorization Authorization value, using format ‘Bearer {user-jwt-
     access-token}.
 */
func (handler *HandlerConfig) ProfileUserHandler(ctx context.Context, req http.Request)
    http.Response {
        user := req.User()

        return http.NewJsonResponse(200, map[string]interface{}{
                "user": map[string]interface{}{
                        "id":            user.ID,
                        "name":          user.Name,
                        "username":      user.Username,
                        "registered_at": user.CreatedAt.Unix(),
                },
        })
}
```

8 internal/app/user/register.go

```go
package user

import (
        "context"
        "fmt"
        "strings"
        "time"

        "github.com/yusufsyaifudin/go-jwt-login-example/internal/pkg/model"
        "github.com/yusufsyaifudin/go-jwt-login-example/pkg/auth"
        "github.com/yusufsyaifudin/go-jwt-login-example/pkg/http"
)

/**
 * @api {post} /user/register Register
 * @apiVersion 1.0.0
 * @apiName Register
 * @apiGroup User
 *
 * @apiDescription User register. This also return authentication token for the first
    time.
 *
 * @apiParam (Request body) {String} name Name of this user
 * @apiParam (Request body) {String} username Username of the user. This should be unique
    .
 * @apiParam (Request body) {String} password User password
 */
func (handler *HandlerConfig) RegisterUserHandler(ctx context.Context, req http.Request)
    http.Response {
        form := &struct {
                Name     string `json:"name" form:"name"`
                Username string `json:"username" form:"username"`
                Password string `json:"password" form:"password"`
        }{}

        if err := req.Bind(form); err != nil {
                return http.NewJsonResponse(500, map[string]interface{}{
                        "error": map[string]interface{}{
                                "message": fmt.Sprintf("fail␣when␣binding␣the␣payload:␣%s
                                    ", err.Error()),
                        },
                })
        }

        if strings.TrimSpace(form.Name) == "" {
                return http.NewJsonResponse(400, map[string]interface{}{
                        "error": map[string]interface{}{
                                "message": "name␣cannot␣be␣empty",
                        },
                })
        }

        if strings.TrimSpace(form.Username) == "" {
                return http.NewJsonResponse(400, map[string]interface{}{
                        "error": map[string]interface{}{
                                "message": "username␣cannot␣be␣empty",
                        },
                })
        }

        if strings.TrimSpace(form.Password) == "" {
                return http.NewJsonResponse(400, map[string]interface{}{
                        "error": map[string]interface{}{
```

```
                        "message": "password␣cannot␣be␣empty",
                },
        })
}

// check if user already exist
user := &model.User{}
handler.DB.Raw(user, "SELECT␣*␣FROM␣users␣WHERE␣username␣=␣?␣LIMIT␣1", form.
    Username)
if user != nil && user.ID != 0 {
        return http.NewJsonResponse(400, map[string]interface{}{
                "error": map[string]interface{}{
                        "message": "user␣with␣this␣username␣already␣registered",
                },
        })
}

passwordHash, err := HashPassword(form.Password)
if err != nil {
        return http.NewJsonResponse(422, map[string]interface{}{
                "error": map[string]interface{}{
                        "message": fmt.Sprintf("fail␣when␣hashing␣password:␣%s",
                            err.Error()),
                },
        })
}

var sqlInsertUser = `
        INSERT INTO users (name, username, password) VALUES (?, ?, ?) ON CONFLICT
            (username) DO UPDATE SET updated_at = now() RETURNING *;
`

// insert to db user in database
err = handler.DB.Raw(user, sqlInsertUser, form.Name, form.Username, passwordHash)
if err != nil {
        return http.NewJsonResponse(422, map[string]interface{}{
                "error": map[string]interface{}{
                        "message": fmt.Sprintf("fail␣inserting␣user␣into␣db:␣%s",
                             err.Error()),
                },
        })
}

// Check password hash is different or not with body json data, if different, it
    may because attacking.
// If still the same, it may because race condition in request (2 or more request
     at one time)
if !CheckPasswordHash(form.Password, user.Password) {
        return http.NewJsonResponse(401, map[string]interface{}{
                "error": map[string]interface{}{
                        "message": "wrong␣password",
                },
        })
}

// if found, then check hashing password
tokenPayload := &auth.Payload{
        ID:         fmt.Sprintf("%d", user.ID),
        Username:   user.Username,
        IssuedAt:   time.Now().Unix(),
        NotBefore:  time.Now().Unix(),
        ExpiredAt:  time.Now().Add(5 * time.Hour).Unix(),
}

accessToken, err := handler.Auth.GenerateToken(tokenPayload, handler.
    ServerSecretKey)
```

```
        if err != nil {
                return http.NewJsonResponse(422, map[string]interface{}{
                        "error": map[string]interface{}{
                                "message": fmt.Sprintf("fail␣generating␣access␣token:␣%s"
                                    , err.Error()),
                        },
                })
        }

        return http.NewJsonResponse(200, map[string]interface{}{
                "access_token": accessToken,
                "user": map[string]interface{}{
                        "id":            user.ID,
                        "name":          user.Name,
                        "username":      user.Username,
                        "registered_at": user.CreatedAt.Unix(),
                },
        })
}
```

9 internal/pkg/model/user.go

```go
package model

import "time"

// User is a data structure that resemble column in database
type User struct {
	ID        int64     `json:"id"`
	Name      string    `json:"name"`
	Username  string    `json:"username"`
	Password  string    `json:"password"`
	CreatedAt time.Time `json:"created_at"`
	UpdatedAt time.Time `json:"updated_at"`
}
```

10 pkg/auth/auth.go

```
package auth

// Payload is a data carried by JWT token
type Payload struct {
	ID        string `json:"id"`       // required, id of this user
	Username  string `json:"username"` // required, name of this user
	IssuedAt  int64  `json:"iss"`      // token creation date, epoch time in seconds
	    value (10 character)
	NotBefore int64  `json:"nbf"`      // token valid start date, if token used
	    before this time, it will contain error, epoch time in seconds value (10
	    character)
	ExpiredAt int64  `json:"exp"`      // token expiration date, epoch time in
	    seconds value (10 character)
}

// Auth is an higher abstraction level of authorization method.
// ValidateToken method: to check if a token is valid or not, and
// GenerateToken method: to generate token based on jwt payload
// By this interface, you can easily change the JWT 3rd party library if it doesn't meet
    your needs.
type Auth interface {
	GenerateToken(payload *Payload, secretKey string) (token string, err error)
	ValidateToken(token string, secretKey string) (payload *Payload, err error)
}
```

11 pkg/auth/auth$_j wt.go$

```go
package auth

import (
	"encoding/json"
	"fmt"
	"strings"
	"time"

	"github.com/dgrijalva/jwt-go"
)

// JwtPayload extends the base auth Payload, so all Payload property can be read here
type JwtPayload struct {
	Payload
}

// Jwt will implements Auth interface using library github.com/dgrijalva/jwt-go.
type Jwt struct {
}

// NewJwtAuth is like a class implementing interface Auth
func NewJwtAuth() (auth Auth) {
	auth = &Jwt{}
	return
}

// Valid is a method required by jwt.Claims set (github.com/dgrijalva/jwt-go).
// Inside this function, we will do any validation checking.
func (jwtPayload JwtPayload) Valid() error {
	if strings.TrimSpace(jwtPayload.ID) == "" {
		return fmt.Errorf("id␣must␣contains␣value")
	}

	if strings.TrimSpace(jwtPayload.Username) == "" {
		return fmt.Errorf("name␣must␣contains␣value")
	}

	if len(fmt.Sprintf("%d", jwtPayload.IssuedAt)) != 10 {
		return fmt.Errorf("iat␣must␣in␣epoch␣time␣contained␣10␣character␣length")
	}

	if jwtPayload.IssuedAt > time.Now().Unix() {
		return fmt.Errorf("token␣used␣before␣issued")
	}

	if len(fmt.Sprintf("%d", jwtPayload.NotBefore)) != 10 {
		return fmt.Errorf("nbf␣must␣in␣epoch␣time␣contained␣10␣character␣length")
	}

	if jwtPayload.NotBefore < time.Now().Unix() {
		return fmt.Errorf("token␣is␣not␣valid␣yet")
	}

	if len(fmt.Sprintf("%d", jwtPayload.ExpiredAt)) != 10 {
		return fmt.Errorf("exp␣must␣in␣epoch␣time␣contained␣10␣character␣length")
	}

	if jwtPayload.ExpiredAt <= time.Now().Unix() {
		return fmt.Errorf("token␣is␣expired")
	}

	return nil
}
```

```
// GenerateToken will generate jwt token using inputted payload
func (authJwt *Jwt) GenerateToken(payload *Payload, secretKey string) (token string, err
    error) {
        jwtToken := jwt.New(jwt.SigningMethodHS256)

        // generate token using HS256
        jwtToken.Header = map[string]interface{}{
                "alg": jwt.SigningMethodHS256.Name,
        }

        jwtToken.Claims = JwtPayload{
                Payload: *payload,
        }

        token, err = jwtToken.SignedString([]byte(secretKey)) // sign with secret key
        return
}

// ValidateToken implements validating jwt token using secret key and return payload
func (authJwt *Jwt) ValidateToken(token string, secretKey string) (payload *Payload, err
    error) {
        jwtToken, err := jwt.Parse(token, func(t *jwt.Token) (interface{}, error) {
                // hmacSampleSecret is a []byte containing your secret, e.g. []byte("
                    my_secret_key")
                return []byte(secretKey), nil
        })

        if err != nil {
                return nil, err
        }

        // Don't forget to validate the alg is what you expect. We use HMAC algorithm.
        // _, ok := jwtToken.Method.(*jwt.SigningMethodHMAC)
        // if !ok {
        //      err = fmt.Errorf("unexpected signing method: %v", jwtToken.Header["alg"])
        //      return nil, err
        // }

        claims, ok := jwtToken.Claims.(jwt.MapClaims)
        if !ok {
                err = fmt.Errorf("token␣is␣not␣valid␣payload")
                return
        }

        if claims.Valid() != nil {
                return nil, claims.Valid()
        }

        // convert type jwt.MapClaims to type Payload
        mapClaimBytes, err := json.Marshal(claims)
        if err != nil {
                err = fmt.Errorf("payload␣cannot␣be␣marshalled")
                return nil, err
        }

        payload = &Payload{}
        err = json.Unmarshal(mapClaimBytes, payload)
        if err != nil {
                err = fmt.Errorf("payload␣cannot␣be␣unmarshalled")
                return nil, err
        }

        // build payload based on jwt payload
        return payload, nil
}
```

12 pkg/auth/auth$_{jwt_test.go}$

```
package auth_test

import (
        "testing"
        "time"

        "crypto/ecdsa"
        "crypto/elliptic"
        "crypto/rand"
        "log"

        "github.com/dgrijalva/jwt-go"
        "github.com/smartystreets/goconvey/convey"
        "github.com/yusufsyaifudin/go-jwt-login-example/pkg/auth"
)

func TestGenerateAndValidateTokenAtBestCondition(t *testing.T) {
        t.Parallel()

        secretKey := "abc"
        authJwt := auth.NewJwtAuth()

        convey.Convey("Generate␣and␣validate␣token", t, func() {

                convey.Convey("When␣all␣value␣is␣good", func() {
                        inputPayload := &auth.Payload{
                                ID:        "1",
                                Username:  "John␣Doe",
                                IssuedAt:  time.Now().Unix(),
                                NotBefore: time.Now().Unix(),
                                ExpiredAt: time.Now().Add(2 * time.Minute).Unix(),
                        }

                        jwtToken, err := authJwt.GenerateToken(inputPayload, secretKey)
                        convey.So(err, convey.ShouldBeNil)

                        outputPayload, err := authJwt.ValidateToken(jwtToken, secretKey)
                        convey.So(err, convey.ShouldBeNil)
                        convey.So(outputPayload, convey.ShouldResemble, inputPayload)
                })

                convey.Convey("When␣secret␣key␣is␣different", func() {
                        inputPayload := &auth.Payload{
                                ID:        "1",
                                Username:  "John␣Doe",
                                IssuedAt:  time.Now().Unix(),
                                NotBefore: time.Now().Unix(),
                                ExpiredAt: time.Now().Add(2 * time.Minute).Unix(),
                        }

                        jwtToken, err := authJwt.GenerateToken(inputPayload, secretKey)
                        convey.So(err, convey.ShouldBeNil)

                        outputPayload, err := authJwt.ValidateToken(jwtToken, "cba")
                        convey.So(err, convey.ShouldNotBeNil)
                        convey.So(err.Error(), convey.ShouldResemble, "signature␣is␣
                            invalid")
                        convey.So(outputPayload, convey.ShouldNotResemble, inputPayload)
                })

        })
}
```

```
func TestNewJwtAuth(t *testing.T) {
        t.Parallel()

        convey.Convey("Test␣initiating␣new␣jwt␣auth", t, func() {
                convey.Convey("Should␣return␣JWT␣struct␣and␣implements␣Auth␣interface",
                    func() {
                        var jwtAuth auth.Auth
                        jwtAuth = auth.NewJwtAuth()
                        convey.So(jwtAuth, convey.ShouldNotBeNil)
                        convey.So(jwtAuth, convey.ShouldEqual, &auth.Jwt{})
                })
        })
}

func TestJwt_ValidateToken(t *testing.T) {
        t.Parallel()

        convey.Convey("Validate␣token", t, func() {
                secretKey := "abc"
                authJwt := auth.NewJwtAuth()

                // This occurred if GenerateToken function, at some point becouse wrong
                    logic, returns wrong signing method
                convey.Convey("When␣signing␣method␣is␣different", func() {
                        // https://stackoverflow.com/a/51472209/5489910
                        key, err := ecdsa.GenerateKey(elliptic.P256(), rand.Reader)
                        if err != nil {
                                log.Fatal(err)
                        }

                        claims := &jwt.StandardClaims{
                                ExpiresAt: 15000,
                                Issuer:    "test",
                        }

                        token := jwt.NewWithClaims(jwt.SigningMethodES256, claims)

                        tokenString, err := token.SignedString(key)
                        convey.So(err, convey.ShouldBeNil)
                        convey.So(token, convey.ShouldNotBeNil)

                        // test the function
                        jwtPayload, err := authJwt.ValidateToken(tokenString, secretKey)
                        convey.So(jwtPayload, convey.ShouldBeNil)
                        convey.So(err.Error(), convey.ShouldResemble, "key␣is␣of␣invalid␣
                            type")
                })

                convey.Convey("Token␣is␣expired", func() {
                        tokenString := "eyJhbGciOiJIUzI1NiJ9.
                            eyJpZCI6IjEiLCJuYW1lIjoiSm9obiBEb2UiLCJpc3MiOjE1MzY0OTA0MDgsIm5iZiI6MTUzNjQ5
                            .Qz8gVmKS6v75S8TLcyteTOH3J5_6EO6R0f6h9OmhBJ0"
                        jwtPayload, err := authJwt.ValidateToken(tokenString, secretKey)
                        convey.So(jwtPayload, convey.ShouldBeNil)
                        convey.So(err, convey.ShouldNotBeNil)
                })

        })
}
```

13 pkg/db/config.go

```go
package db

type Config struct {
        ConnectionString string
        Debug            bool
}
```

14 pkg/db/query.go

```
package db

type Query interface {
        Raw(dst interface{}, sql string, args ...interface{}) (err error)
        Exec(sql string, args ...interface{}) (err error)
        Migrate() error
}
```

15 pkg/db/query$_{g}opg.go$

```
package db

import (
	"time"

	"fmt"

	"github.com/go-pg/pg"
	"github.com/golang-migrate/migrate"
	_ "github.com/golang-migrate/migrate/database/postgres"
	"github.com/golang-migrate/migrate/source/go_bindata"
	"github.com/rs/zerolog/log"
	"github.com/yusufsyaifudin/go-jwt-login-example/assets/migrations"
)

var logger = log.With().Str("pkg", "db").Logger()
var goPgConnection *pg.DB

// NewGoPgQuery will create new connection and returns 3 output,
// 1. connection to database, this should not be used other than to close the connection
// 2. implementation of db.Query interface where you can query with opened connection
// 3. error if any error occurred
func NewGoPgQuery(config *Config) (dbConn *pg.DB, query Query, err error) {
	dbOptions, err := pg.ParseURL(config.ConnectionString)
	if err != nil {
		return
	}

	dbOptions.PoolSize = 10
	dbOptions.IdleTimeout = time.Duration(5) * time.Second

	dbConn = pg.Connect(dbOptions)
	goPgConnection = dbConn

	if config.Debug {
		dbConn.OnQueryProcessed(func(event *pg.QueryProcessedEvent) {
			query, err := event.FormattedQuery()
			if err != nil {
				log.Printf("error␣when␣log␣query,␣%s", err.Error())
				return
			}

			elapsedTime := float64(time.Since(event.StartTime).Nanoseconds())
			     / float64(1000000)
			logger.Debug().
				Str("elapsedTime", fmt.Sprintf("%0.2f␣ms", elapsedTime)).
				Str("query", query).
				Msg("")
		})
	}

	// using implemented interface
	query = &QueryGoPg{
		config: config,
	}
	return
}

// QueryGoPg implements Query interface with github.com/go-pg/pg connection
type QueryGoPg struct {
	config *Config
}
```

```go
// Raw will query to Postgres using raw sql and map the result into dst.
func (q *QueryGoPg) Raw(dst interface{}, sql string, args ...interface{}) (err error) {
        _, err = goPgConnection.Query(dst, sql, args...)
        return
}

// Exec will do query to Postgres without returning values
func (q *QueryGoPg) Exec(sql string, args ...interface{}) (err error) {
        _, err = goPgConnection.Exec(sql, args...)
        return
}

func (q *QueryGoPg) Migrate() error {
        s := bindata.Resource(
                migrations.AssetNames(),
                func(name string) ([]byte, error) {
                        return migrations.Asset(name)
                },
        )

        d, err := bindata.WithInstance(s)
        if err != nil {
                logger.Error().Msgf("bindata␣instance␣fail␣%s", err.Error())
                return err
        }

        m, err := migrate.NewWithSourceInstance(
                "go-bindata",
                d,
                q.config.ConnectionString,
        )
        if err != nil {
                logger.Error().Msgf("fail␣do␣migration␣to␣host␣%s␣=>␣%s", q.config.
                    ConnectionString, err.Error())
                return err
        }

        // run your migrations and handle the errors above of course
        // if migration error, it will flag as dirty, and you must run it manually
        version, dirty, _ := m.Version()
        if dirty {
                err := m.Force(int(version))
                if err != nil {
                        return err
                }

                return m.Up()
        }

        return m.Up()
}
```

16 pkg/http/handler.go

```
package http

import "context"

type (
	Handler    func(context.Context, Request) Response
	Middleware func(Handler) Handler
)
```

17 pkg/http/midleware.go

```go
package http

import "context"

// Implementing this idea https://hackernoon.com/simple-http-middleware-with-go-79
    a4ad62889b
func ChainMiddleware(mw ...Middleware) Middleware {
        return func(final Handler) Handler {
                return func(ctx context.Context, req Request) Response {
                        last := final
                        for i := len(mw) - 1; i >= 0; i-- {
                                last = mw[i](last)
                        }

                        // last middleware
                        return last(ctx, req)
                }
        }
}
```

18 pkg/http/request$_{gin.go}$

```go
package http

import (
	"context"
	"net/http"

	"github.com/gin-gonic/gin"
	"github.com/yusufsyaifudin/go-jwt-login-example/internal/pkg/model"
)

// WrapGin wraps a Handler and turns it into gin compatible handler
// This method should be called with a fresh ctx
func WrapGin(parent context.Context, handler Handler) gin.HandlerFunc {
	return func(ginContext *gin.Context) {
		// create span
		ctx := context.Background()
		defer ctx.Done()

		// create request and run the handler
		var req = newGinRequest(ginContext)
		resp := handler(ctx, req)

		if resp == nil {
			ginContext.JSON(http.StatusInternalServerError, map[string]
			    interface{}{
				"error": map[string]interface{}{
					"code":    "internal_server_error",
					"message": "nil␣response",
					"data":    nil,
				},
			})
			return
		}

		// get the body first
		body, err := resp.Body()
		if err != nil {
			ginContext.JSON(http.StatusInternalServerError, map[string]
			    interface{}{
				"error": map[string]interface{}{
					"code":    "internal_server_error",
					"message": err.Error(),
					"data":    nil,
				},
			})
			return
		}
		// then write header
		for k, v := range resp.Header() {
			for _, h := range v {
				ginContext.Writer.Header().Add(k, h)
			}
		}

		ginContext.Writer.Header().Add("Content-Type", resp.ContentType())
		ginContext.Writer.WriteHeader(resp.StatusCode())

		// the last is writing the body
		ginContext.Writer.Write(body)
	}
}

type ginRequest struct {
        context *gin.Context
        user    *model.User
}

func newGinRequest(context *gin.Context) (request Request) {
        request = &ginRequest{
                context: context,
        }
        return
}

func (ginRequest *ginRequest) Bind(out interface{}) error {
        return ginRequest.context.Bind(out)
}

func (ginRequest *ginRequest) GetParam(key string) string {
        return ginRequest.context.Param(key)
}

func (ginRequest *ginRequest) Header() http.Header {
        return ginRequest.context.Request.Header
}

func (ginRequest *ginRequest) ContentType() string {
        return ginRequest.context.ContentType()
}

func (ginRequest *ginRequest) RawRequest() *http.Request {
        return ginRequest.context.Request
}

func (ginRequest *ginRequest) User() *model.User {
        return ginRequest.user
}

func (ginRequest *ginRequest) SetUser(user *model.User) {
        ginRequest.user = user
}
```

19 pkg/http/request.go

```go
package http

import (
	"net/http"

	"github.com/yusufsyaifudin/go-jwt-login-example/internal/pkg/model"
)

type Request interface {
	ContentType() string
	Bind(out interface{}) error
	GetParam(key string) string
	RawRequest() *http.Request
	User() *model.User // get the current user
	SetUser(user *model.User)
}
```

20 pkg/http/response.go

```go
package http

import "net/http"

type Response interface {
	StatusCode() int
	Body() ([]byte, error)
	Header() http.Header
	ContentType() string
}
```

21 pkg/http/response$_{j}son.go$

```
package http

import (
        "encoding/json"
        "net/http"
)

type jsonResponse struct {
        statusCode int
        data       interface{}
        header     http.Header
        next       bool
}

func NewJsonResponse(statusCode int, data interface{}) (response Response) {
        response = &jsonResponse{
                statusCode: statusCode,
                data:       data,
                header:     http.Header{},
        }
        return
}

func (jsonResponse *jsonResponse) StatusCode() int {
        return jsonResponse.statusCode
}

func (jsonResponse *jsonResponse) Body() ([]byte, error) {
        b, err := json.Marshal(jsonResponse.data)
        if err != nil {
                return nil, err
        }
        return b, nil
}

func (jsonResponse *jsonResponse) Header() http.Header {
        return jsonResponse.header
}

func (jsonResponse *jsonResponse) ContentType() string {
        return "application/json;␣charset=utf-8"
}
```

22 server/logger.go

```
package server

import (
	"fmt"
	"io"
	"time"

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

// Logger instances a Logger middleware that will write the logs to gin.DefaultWriter.
// By default gin.DefaultWriter = os.Stdout.
func Logger() gin.HandlerFunc {
	return LoggerWithWriter(gin.DefaultWriter)
}

// LoggerWithWriter instance a Logger middleware with the specified writter buffer.
// Example: os.Stdout, a file opened in write mode, a socket...
func LoggerWithWriter(out io.Writer, notlogged ...string) gin.HandlerFunc {
	var skip map[string]struct{}

	if length := len(notlogged); length > 0 {
		skip = make(map[string]struct{}, length)

		for _, path := range notlogged {
			skip[path] = struct{}{}
		}
	}

	return func(c *gin.Context) {
		// Start timer
		start := time.Now()
		path := c.Request.URL.Path
		raw := c.Request.URL.RawQuery

		// Process request
		c.Next()

		// Log only when path is not being skipped
		if _, ok := skip[path]; !ok {
			// Stop timer
			end := time.Now()
			latency := end.Sub(start)

			clientIP := c.ClientIP()
			method := c.Request.Method
			statusCode := c.Writer.Status()
			comment := c.Errors.ByType(gin.ErrorTypePrivate).String()

			if raw != "" {
				path = path + "?" + raw
			}

			logger.Info().
				Str("requestTime", end.Format("2006/01/02␣-␣15:04:05")).
				Int("code", statusCode).
				Str("latency", fmt.Sprintf("%13v", latency)).
				Str("clientIp", clientIP).
				Str("method", method).
				Str("path", path).
				Msg(comment)
		}
	}
}
```

23 server/server.go

```
package server

import (
	"context"

	"github.com/gin-contrib/static"
	"github.com/gin-gonic/gin"
	"github.com/rs/zerolog/log"
	"github.com/yusufsyaifudin/go-jwt-login-example/apidoc"
	"github.com/yusufsyaifudin/go-jwt-login-example/internal/app/user"
	"github.com/yusufsyaifudin/go-jwt-login-example/pkg/auth"
	"github.com/yusufsyaifudin/go-jwt-login-example/pkg/db"
	"github.com/yusufsyaifudin/go-jwt-login-example/pkg/http"
)

var logger = log.With().Str("pkg", "server").Logger()
var stopped = false

type Config struct {
	ListenAddress   string
	ServerSecretKey string
	DB              db.Query
	Auth            auth.Auth
}

// Run will run the server and return error if error occurred.
func (config *Config) Run() error {
	parentCtx := context.Background()
	defer parentCtx.Done()

	gin.SetMode(gin.ReleaseMode)
	router := gin.New()
	router.Use(Logger())

	// api documentation
	router.Use(static.Serve("/", apidoc.Static()))

	// to gracefully shutdown the server
	router.Use(func(ctx *gin.Context) {
		// if it's the case then don't receive anymore requests
		if stopped {
			ctx.Status(503)
			return
		}

		ctx.Next()
	})

	router.NoRoute(func(ctx *gin.Context) {
		response := map[string]interface{}{
			"error": map[string]interface{}{
				"message": "route␣not␣found",
			},
		}

		ctx.JSON(404, response)
		ctx.Abort()
	})

	router.NoMethod(func(ctx *gin.Context) {
		response := map[string]interface{}{
			"error": map[string]interface{}{
				"message": "method␣for␣this␣route␣not␣found",
                },
            }

            ctx.JSON(404, response)
            ctx.Abort()
    })

    userHandler := user.NewUserHandler(config.ServerSecretKey, config.DB, config.Auth
        )
    protectedMiddleware := http.ChainMiddleware(userHandler.MiddlewareAuthTokenCheck)

    userGroup := router.Group("/api/v1/user")
    userGroup.POST("/login", http.WrapGin(parentCtx, userHandler.LoginUserHandler))
    userGroup.POST("/register", http.WrapGin(parentCtx, userHandler.
        RegisterUserHandler))
    userGroup.GET("/profile", http.WrapGin(parentCtx, protectedMiddleware(userHandler
        .ProfileUserHandler)))

    // for debugging purpose
    for _, routeInfo := range router.Routes() {
            logger.Debug().
                    Str("path", routeInfo.Path).
                    Str("handler", routeInfo.Handler).
                    Str("method", routeInfo.Method).
                    Msg("registered␣routes")
    }

    return router.Run(config.ListenAddress)
}

// Shutdown this package
func (config *Config) Shutdown() {
    logger.Info().Msg("not␣receiving␣requests␣anymore")
    stopped = true
}
```