Securing Go Application

LAB 10

Learning Outcomes

After completing this lab session you should be able to

- Insure input validation of form data
- Manage Session
- Implement authentication and authorization functionality for your application
- Preventing CSRF attack

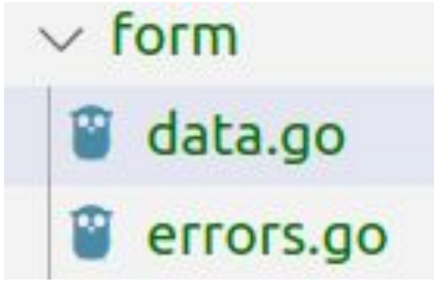
The full sample code is found in the following link

https://github.com/betsegawlemma/web-prog-go-sample

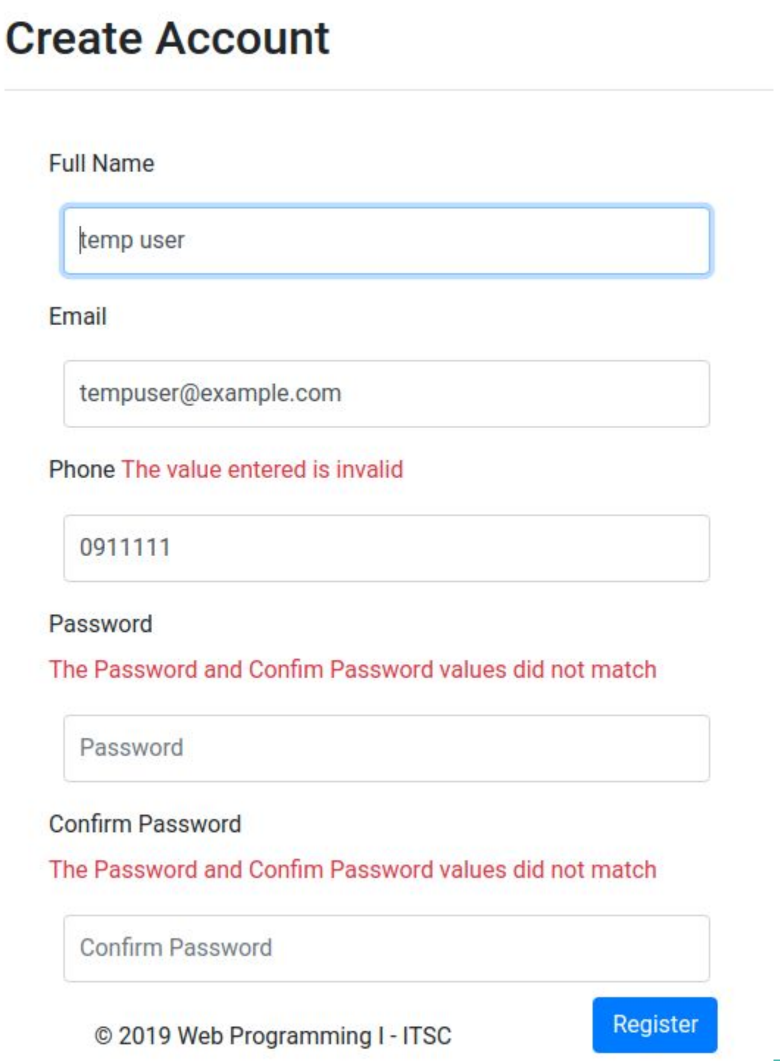
Input Validation

Let us see how to do the validation shown in the right

We have a package named **`form`**



form
data.go
errors.go



Create Account

Full Name

temp user

Email

tempuser@example.com

Phone The value entered is invalid

0911111

Password

The Password and Confim Password values did not match

Password

Confirm Password

The Password and Confim Password values did not match

Confirm Password

© 2019 Web Programming I - ITSC

Input Validation

```go
// Input represents form input
// values and validation error
type Input struct {
   Values  url.Values
   VErrors ValidationErrors
   CSRF    string
}
```

Checking minimum input character length

```go
// MinLength checks if a given minium length is satisfied
func (inVal *Input) MinLength(field string, d int) {
	value := inVal.Values.Get(field)
	if value == "" {
		return
	}
	if utf8.RuneCountInString(value) < d {
		inVal.VErrors.Add(field, fmt.Sprintf("This field is
		too short (minimum is %d characters)", d))
	}
}
```

Managing Errors

```go
// ValidationErrors represents input validation errors
type ValidationErrors map[string][]string
// Add method to add error messages for a given field to the map
func (ve ValidationErrors) Add(field, message string) {
    ve[field] = append(ve[field], message)
}
// Get method to retrieve the first error message for a given field
func (ve ValidationErrors) Get(field string) string {
    ves := ve[field]
    if len(ves) == 0 {
        return ""
    }
    return ves[0]
}
```

Checking input validation in side your handler

```
// Signup handles the GET/POST /signup requests
func (uh *UserHandler) Signup(w http.ResponseWriter, r *http.Request) {
      ….
     // Validate the form contents
      singnUpForm := form.Input{Values: r.PostForm, VErrors: form.ValidationErrors{}}
      singnUpForm.Required("fullname", "email", "password", "confirmpassword")
      singnUpForm.MatchesPattern("email", form.EmailRX)
      singnUpForm.MatchesPattern("phone", form.PhoneRX)
      singnUpForm.MinLength("password", 8)
      singnUpForm.PasswordMatches("password", "confirmpassword")
      singnUpForm.CSRF = token
      // If there are any errors, redisplay the signup form.
      if !singnUpForm.Valid() {
          uh.tmpl.ExecuteTemplate(w, "signup.layout", singnUpForm)
          return
      }
}
```

Displaying Error and data on the form

```
<form class="form-account" method="POST" action="/admin/users/new">
       <input type="hidden" name="_csrf" value="{{ .CSRF }}" />
       <div class="col-auto">
           <label for='fullname' class='col-form-label'>Full Name</label>
           {{with .VErrors.Get "fullname"}}
           <label class="text-danger">{{.}}</label>
           {{end}}
           <input type='text' class='form-control' name='fullname'
               value='{{ .Values.Get "fullname" }}' >
       </div>
  ....
```

Managing Session

We will use JWT to generate and validate tokens

The generated JWT token will be stored on the client as a cookie value

It will also be stored on the server side

For generating and validating the JWT token, we need randomly generated signing key

For identifiying client session we also use randomly generated strings

Session Struct

```go
//Session represents login user session
type Session struct {
    ID         uint
    UUID       string `gorm:"type:varchar(255);not null"`
    Expires    int64  `gorm:"type:varchar(255);not null"`
    SigningKey []byte `gorm:"type:varchar(255);not null"`
}
```

We have the CRUD to manage the session in the database

Getting Go’s implementation of JWT

```
go get "github.com/dgrijalva/jwt-go"
```

Generating JWT Token

```go
// CustomClaims specifies custom claims
type CustomClaims struct {
   Email string `json:"email"`
   jwt.StandardClaims
}

// Generate generates jwt token
func Generate(signingKey []byte, claims jwt.Claims) (string, error) {
   tn := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
   signedString, err := tn.SignedString(signingKey)
   return signedString, err
}
```

Validating JWT token

```go
// Valid validates a given token
func Valid(signedToken string, signingKey []byte) (bool, error) {
    token, err := jwt.ParseWithClaims(signedToken, &CustomClaims{},
            func(token *jwt.Token) (interface{}, error) {
        return signingKey, nil
    })
    if _, ok := token.Claims.(*CustomClaims); !ok || !token.Valid {
        return false, err
    }
    return true, nil
}
```

Generate Random Bytes

```go
// GenerateRandomBytes returns securely generated random bytes.
func GenerateRandomBytes(n int) ([]byte, error) {
   mrand.Seed(time.Now().UnixNano())
   b := make([]byte, n)
   _, err := rand.Read(b)
   if err != nil {
       return nil, err
   }
   return b, nil
}
```

Generate URL safe string

```go
// GenerateRandomString returns a URL-safe, base64 encoded
// securely generated random bytes.
func GenerateRandomString(s int) (string, error) {
    b, err := GenerateRandomBytes(s)
    return base64.URLEncoding.EncodeToString(b), err
}
```

Generating Random String for Session ID

```
// GenerateRandomID generates random id for a session
func GenerateRandomID(s int) string {
   mrand.Seed(time.Now().UnixNano())
   const letterBytes =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
   b := make([]byte, s)
   for i := range b {
       b[i] = letterBytes[mrand.Int63()%int64(len(letterBytes))]
   }
   return string(b)
}
```

Creating Session Cookie

```go
// Create creates and sets session cookie
func Create(claims jwt.Claims, sessionID string, signingKey []byte, w
http.ResponseWriter) {
	signedString, err := rtoken.Generate(signingKey, claims)
	if err != nil {
		fmt.Println(err)
		w.WriteHeader(http.StatusInternalServerError)
	}
	c := http.Cookie{
		Name:     sessionID,
		Value:    signedString,
		HttpOnly: true,
	}
	http.SetCookie(w, &c)
}
```

Saving session cookie in the database

```
// Login handles the GET/POST /login requests
func (uh *UserHandler) Login(w http.ResponseWriter, r *http.Request) {
    ...
        claims := rtoken.Claims(usr.Email, uh.userSess.Expires)
        session.Create(claims, uh.userSess.UUID, uh.userSess.SigningKey, w)
        newSess, errs := uh.sessionService.StoreSession(uh.userSess)
        if len(errs) > 0 {
            loginForm.VErrors.Add("generic", "Failed to store session")
            uh.tmpl.ExecuteTemplate(w, "login.layout", loginForm)
            return
        }
        uh.userSess = newSess
```

Note: the session object is passed from the main function with random sesssion id and signing key

Authentication and Authorization

To easily implement this feature we need to use middlewares

Middleware is a common approach to organize shared functionality (for example securing some routes)

Go’s Middleware Pattern

```go
func SomeMddleware(next http.Handler) http.Handler {
   fn := func(w http.ResponseWriter, r *http.Request) {
       // TODO: Execute our middleware logic here...
       next.ServeHTTP(w, r)
   }
   return http.HandlerFunc(fn)
}
```

Go’s Middleware Pattern (Short form)

```go
func SomeMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r
    *http.Request) {
        // TODO: Execute our middleware logic here...
        next.ServeHTTP(w, r)
    })
}
```

Authentication

```go
// Authenticated checks if a user is authenticated to access a given route
func (uh *UserHandler) Authenticated(next http.Handler) http.Handler {
   fn := func(w http.ResponseWriter, r *http.Request) {
       ok := uh.loggedIn(r) ──────────────────────────────► Next Slide
       if !ok {
           http.Redirect(w, r, "/login", http.StatusSeeOther)
           return
       }
       ctx := context.WithValue(r.Context(), ctxUserSessionKey, uh.userSess)
       next.ServeHTTP(w, r.WithContext(ctx))
   }
   return http.HandlerFunc(fn)
}
```

Here, **Context** is used to store user session on requests so that it will be used in subsequent requests

Checking if a user is logged in

```go
func (uh *UserHandler) loggedIn(r *http.Request) bool {
    if uh.userSess == nil {
        return false
    }
    userSess := uh.userSess
    c, err := r.Cookie(userSess.UUID)
    if err != nil {
        return false
    }
    ok, err := session.Valid(c.Value, userSess.SigningKey)
    if !ok || (err != nil) {
        return false
    }
    return true
}
```

Next Slide

Validating User Cookie Session

```go
// Valid validates client cookie value
func Valid(cookieValue string, signingKey []byte) (bool, error) {
   valid, err := rtoken.Valid(cookieValue, signingKey)
   if err != nil || !valid {
       return false, errors.New("Invalid Session Cookie")
   }
   return true, nil
}
```

Using the Authenticated middleware

The `/admin` route, for instance, should only be accessed by authenticated users.

To ensure this you can wrap the handler as shown below

```
http.Handle("/admin",uh.Authenticated(http.HandlerFunc(mh.Admin)))
```

After doing this if they user tries to access `/admin` route before login, the user will be redirected to the login page

Authorization

We will use role based authorization by defining user permissions as shown in the example table below

Routes	Allowed Roles	Allowed Methods
/login	USER, ADMIN	GET, POST
/contact	USER, ADMIN	GET, POST
/admin	ADMIN	GET, POST

Defining Permissions

```go
type permission struct {
   roles    []string
   methods  []string
}

type authority map[string]permission
```

```go
var authorities = authority{
    "/contact": permission{
        roles:    []string{"USER"},
        methods:  []string{"GET", "POST"},
    },
    "/login": permission{
        roles:    []string{"USER"},
        methods:  []string{"GET", "POST"},
    },
    "/admin": permission{
        roles:    []string{"ADMIN"},
        methods:  []string{"GET", "POST"},
    },
}
```

Check if a user has permission

```
// HasPermission checks if a given role has permission func
HasPermission(path string, role string, method string) bool {
    if strings.HasPrefix(path, "/admin") {
        path = "/admin"
    }
    perm := authorities[path]
    checkedRole := checkRole(role, perm.roles)
    checkedMethod := checkMethod(method, perm.methods)
    if !checkedRole || !checkedMethod {
        return false
    }
    return true
}
```

Check allowed role

```
func checkRole(role string, roles []string) bool {
    for _, r := range roles {
        if strings.ToUpper(r) == strings.ToUpper(role) {
            return true
        }
    }
    return false
}
```

Check allowed method

```go
func checkMethod(method string, methods []string) bool {
    for _, m := range methods {
        if strings.ToUpper(m) == strings.ToUpper(method) {
            return true
        }
    }
    return false
}
```

Authorized middleware

We will use middleware pattern to check for proper authorization

Authorized middleware

```go
// Authorized checks if a user has proper authority to access a give route
func (uh *UserHandler) Authorized(next http.Handler) http.Handler {
   return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        if uh.loggedInUser == nil {
          http.Error(w, http.StatusText(http.StatusUnauthorized),
          http.StatusUnauthorized)
             return
        }
        roles, errs := uh.userService.UserRoles(uh.loggedInUser)
        if len(errs) > 0 {
          http.Error(w, http.StatusText(http.StatusUnauthorized),
          http.StatusUnauthorized)
             return
        }
```

Authorized middleware

```
    for _, role := range roles {
        permitted := permission.HasPermission(r.URL.Path, role.Name,
        r.Method)
          if !permitted {
            http.Error(w, http.StatusText(http.StatusUnauthorized),
            http.StatusUnauthorized)
              return
          }
      }
               next.ServeHTTP(w, r)
   })
}
```

Using the Authorized middleware

You can chain the Authorized wrapper after the Authenticated middleware as shown below

```
http.Handle("/admin",
uh.Authenticated(uh.Authorized(http.HandlerFunc(mh.Admin))))
```

Now if authenticated but unauthorized (non-admin user) tries to access the `/admin` route, the user will get unauthorized reply

Preventing CSRF attacks

We will use JWT to generate and validate csrf tokens

We need token signing key, for this we will use the same random byte generation function used for session management

Generate token for CSRF

```go
// CSRFToken Generates random string for CSRF
func CSRFToken(signingKey []byte) (string, error) {
   tn := jwt.New(jwt.SigningMethodHS256)
   signedString, err := tn.SignedString(signingKey)
   if err != nil {
       return "", err
   }
   return signedString, nil
}
```

Validate CSRF token

```
// ValidCSRF checks if a given csrf token is valid
func ValidCSRF(signedToken string, signingKey []byte) (bool, error)
{
    token, err := jwt.Parse(signedToken,
                    func(token *jwt.Token) (interface{}, error) {
        return signingKey, nil
    })
    if err != nil || !token.Valid {
        return false, err
    }
    return true, nil
}
```

Hidden Input for CSRF token

Here is an example on the form used for creating categories

```
<form method="POST" action="/admin/categories/new" enctype="multipart/form-data">
    <input type="hidden" name="_csrf" id="_csrf" value="{{ .CSRF }}" />
     <div class="form-group row">
        ...
```

The same is done for every form, login, logout, signup, user, category, ...

Inserting the csrf token

```
// AdminCategoriesNew hanlde requests on route /admin/categories/new
func (ach *AdminCategoryHandler) AdminCategoriesNew(w http.ResponseWriter, r
*http.Request) {
    token, err := rtoken.CSRFToken(ach.csrfSignKey)
    if err != nil {
        http.Error(w, http.StatusText(http.StatusInternalServerError),
http.StatusInternalServerError)
    }
    if r.Method == http.MethodGet {
        newCatForm := struct {
            Values  url.Values
            VErrors form.ValidationErrors
            CSRF    string
        }{ Values:  nil, VErrors: nil, CSRF:    token }
        ach.tmpl.ExecuteTemplate(w, "admin.categ.new.layout", newCatForm)
    }
```

Reading and validating the CSRF value

You can have separate middleware for this or you can do the validation inside Authorized middleware

The following code shows the latter case

Reading and validating the CSRF value

Just add the following code above the `next.ServeHttp(w, r)` line inside the `Authorized` middleware

```
if r.Method == http.MethodPost {
    ok, err := rtoken.ValidCSRF(r.FormValue("_csrf"), uh.csrfSignKey)
    if !ok || (err != nil) {
         http.Error(w, http.StatusText(http.StatusUnauthorized),
         http.StatusUnauthorized)
        return
    }
}
next.ServeHTTP(w, r)
```