Pemrograman Golang Web dan API

Aditya Rizky Ramadhan

Table of contents

1 Clean Architecture **2**
- 1.1 Pengertian Clean Architecture . . . 2
- 1.2 Penerapan Clean Architecture . . . 2

2 REST API **3**
- 2.1 Pengertian API . . . 3
- 2.2 Pengertian REST . . . 3
- 2.3 Why use RESTful API? . . . 4

3 Pengenalan Gin Gonic (Framework Golang untuk REST API) **4**
- 3.1 Apa itu Gin Gonic? . . . 4
- 3.2 Mengapa menggunakan Gin? . . . 4
- 3.3 Perbedaan Gin dengan HTTP Server Golang Standar . . . 5

4 Middleware **6**
- 4.1 Pengertian . . . 6
- 4.2 Kenapa menggunakan Middleware? . . . 6
- 4.3 Cara Kerja Middleware . . . 7
- 4.4 Contoh Middleware . . . 8

5 Authorisasi dan Authentikasi **9**
- 5.1 JWT JSON Web Token . . . 9
  - 5.1.1 Anatomi . . . 9
  - 5.1.2 Flow . . . 10
  - 5.1.3 JWT implementation . . . 11
- 5.2 OAuth . . . 16
  - 5.2.1 Pengertian . . . 16
  - 5.2.2 Why OAuth? . . . 17
  - 5.2.3 Basic Flow . . . 17
  - 5.2.4 Implementasi OAuth . . . 18

1 Clean Architecture

1.1 Pengertian Clean Architecture

Clean Architecture adalah desain software yang memisahkan elemen-elemen menjadi beberapa lapisan (layer). Tujuan penting dari Clean Architecture adalah memisahkan antara layer business logic dan layer presentasi (delivery mechanism).

Keuntungan Clean Architecture:

- **Highly Testable:** kita dapat membuat test cases untuk setiap layer guna menentukan di mana error terjadi.
- **Framework Independent:** Clean architecture tidak menggunakan framework atau tergantung pada framework tertentu, jadi kita bisa mengubah framework kapan saja.
- **Database Independent:** memungkinkan kita mengganti database tanpa membuat perubahan besar pada source code, misalnya dari SQL ke database NoSQL.
- **UI Independent:** UI framework ada di layer terluar dan kita dapat mengubahnya dengan mudah karena sudah terpisah.

1.2 Penerapan Clean Architecture

MVC

Model Clean Architecture yang paling sering digunakan adalah model MVC (Model View Controller), yaitu sebuah pola arsitektur dalam membuat sebuah aplikasi dengan cara memisahkan kode menjadi tiga bagian yang terdiri dari

- **Model:** bagian yang berhubungan dengan data, biasanya berisi class yang berhubungan dengan database.
- **View:** bagian yang berhubungan dengan tampilan, biasanya berisi file HTML, CSS, dan Javascript.
- **Controller:** bagian yang berhubungan dengan logika aplikasi, biasanya berisi class yang mengatur alur aplikasi.

Layer MVC

./app

  controller

    userController.go

  model

    user.go

repository

userRepo.go

view

userView.go

Ini adalah tata letak yang paling umum dalam implementasi MVC di mana kita memisahkan resource berdasarkan layer MVC, biasanya digunakan pada aplikasi yang belum terlalu kompleks.

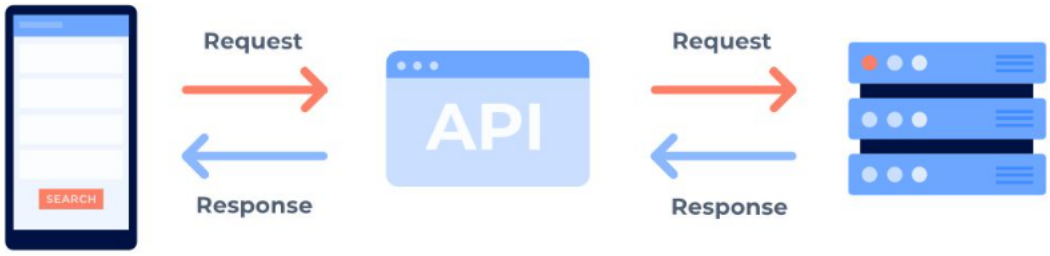
2 REST API

2.1 Pengertian API

API (Application Programming Interface) merupakan interface yang dapat menghubungkan satu aplikasi dengan aplikasi lainnya. Berperan sebagai perantara antar berbagai aplikasi berbeda, baik dalam satu platform yang sama atau lintas platform.

Perumpamaan yang bisa digunakan untuk menjelaskan API adalah seorang pelayan di restoran. Tugas pelayan tersebut adalah menghubungkan tamu restoran (client) dengan juru masak (server).

Jadi, tamu cukup memesan makanan sesuai daftar menu yang ada dan pelayan memberitahukannya ke juru masak. Nantinya, pelayan akan kembali ke tamu tadi dengan masakan yang sudah siap sesuai pesanan.



2.2 Pengertian REST

REST (REpresentational State Transfer) API adalah arsitektur perangkat lunak yang di dalamnya mendefinisikan aturan-aturan untuk membuat web service.

API dapat dikatakan “RESTful” jika memiliki fitur berikut :

- **Client-Server:** REST memisahkan antara client dan server, sehingga keduanya dapat berkembang secara independen.
- **Stateless:** Setiap request dari client ke server harus berisi semua informasi yang diperlukan untuk memahami request tersebut.
- **Cacheable:** Server harus menandai data sebagai cacheable atau non-cacheable.

2.3 Why use RESTful API?

- **Skalabilitas:** Sistem yang menerapkan REST API dapat menskalakan secara efisien karena REST mengoptimalkan interaksi client-server. Statelessness menghapus beban server karena server tidak perlu menyimpan informasi request client di masa lalu. Dan pembuatan cache yang dikelola dengan baik di sisi client. Semua fitur ini mendukung skalabilitas tanpa menyebabkan gangguan komunikasi yang mengurangi performa.
- **Fleksibilitas:** Layanan web RESTful menyederhanakan dan memisahkan berbagai komponen server sehingga masing-masing bagian dapat berkembang secara mandiri. Perubahan platform atau teknologi pada aplikasi server tidak mempengaruhi aplikasi client.
- **Independensi:** Implementasi API menggunakan REST bersifat independen terhadap teknologi yang digunakan. Kita dapat menulis baik aplikasi client dan server dalam berbagai bahasa pemrograman tanpa mempengaruhi desain API. Kita juga dapat mengubah teknologi mendasar di kedua sisi tanpa mempengaruhi komunikasi.

3 Pengenalan Gin Gonic (Framework Golang untuk REST API)

3.1 Apa itu Gin Gonic?

Gin adalah framework yang dikembangkan oleh komunitas Golang. Gin dibuat untuk memudahkan pengembangan aplikasi web. Sebelumnya kita sudah belajar tentang HTTP server Golang standar, dengan menggunakan Gin kita dapat membuat aplikasi web dengan lebih cepat dan mudah karena memiliki banyak fitur yang dapat memudahkan pengembangan aplikasi web. Dokumentasi resmi dari Gin dapat dilihat di sini.

3.2 Mengapa menggunakan Gin?

Cara kerja Gin sama seperti HTTP server Golang standar, hanya saja Gin memiliki beberapa kelebihan dibandingkan dengan HTTP server Golang standar, yaitu:

Jika menggunakan HTTP server, tidak ada built-in support di Golang untuk mengatasi routing berdasarkan regular expression atau “pattern”.

- Gin memiliki performa yang lebih baik dan efisien dibandingkan dengan HTTP server Golang standar, karena Gin menggunakan routing yang lebih cepat dan memiliki arsitektur yang lebih baik untuk memproses request.
- Gin menyediakan fitur middleware, yang memudahkan pengembangan aplikasi web dengan menambahkan logika pada tahap-tahap tertentu dalam proses request-response (ini akan dijelaskan di materi selanjutnya).
- Gin memiliki komunitas yang aktif dan membantu, memudahkan pengembangan aplikasi web dan mempercepat proses pemecahan masalah.

3.3 Perbedaan Gin dengan HTTP Server Golang Standar

Kode HTTP server Golang standar:

```
package main

import (
    "fmt"
    "net/http"
)

func Handler(w http.ResponseWriter, r *http.Request) {
    // handle only get request
    if r.Method != http.MethodGet {
        w.WriteHeader(http.StatusMethodNotAllowed)
        return
    }

    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, `{"message": "Hello World"}`)
}

func main() {
    mux := http.NewServeMux()

    mux.HandleFunc("/hello", Handler)

    http.ListenAndServe(":8080", mux)
}
```

Kode HTTP server menggunakan Gin:

```
package main

import (
```

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

func main() {
    r := gin.Default()

    r.GET("/hello", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{ "message": "Hello World" })
    })

    r.Run(":8080")
}
```

Dari kedua kode di atas, kita dapat melihat perbedaan antara HTTP server Golang standar dan Gin. Gin memiliki fitur yang lebih lengkap dan mudah digunakan, sehingga memudahkan pengembangan aplikasi web.

Untuk latihan menggunakan golang dapat dilihat di sini

4 Middleware

4.1 Pengertian

Middleware adalah sekumpulan logika yang berjalan di antara request dan response dalam aplikasi web. Dalam Golang, middleware dapat diterapkan pada sebuah web server untuk memproses serangkaian request sebelum diarahkan ke handler utama. Dalam satu aplikasi bisa terdapat lebih dari satu middleware, sehingga request akan dijalankan melalui beberapa middleware secara berurutan sebelum sampai ke handler utama.

```
request => middleware 1 => middleware 2 => middleware ... => handler => response
```

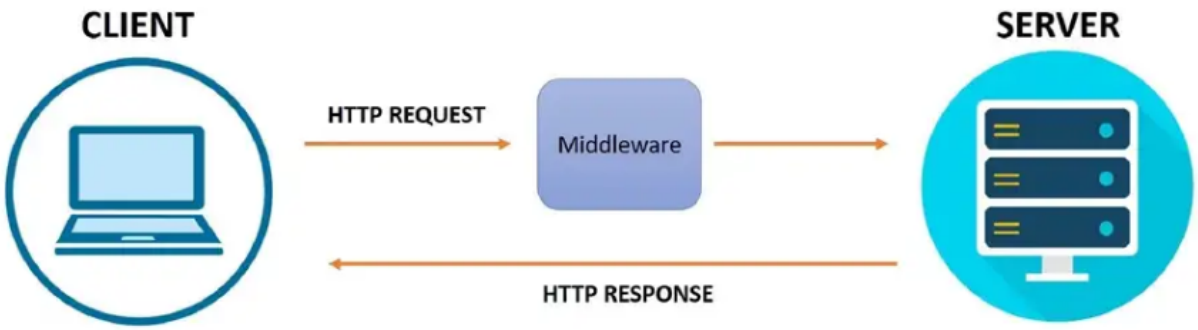
4.2 Kenapa menggunakan Middleware?

Middleware memudahkan web server untuk mengelola & memproses request dan response dengan lebih efisien dan efektif. Middleware juga dapat memfasilitasi integrasi antara aplikasi web dengan sistem lain seperti database, sistem pembayaran, dan lainnya. Oleh karena itu, middleware di web server sangat penting untuk memastikan bahwa aplikasi web berjalan dengan baik dan aman.

Middleware juga dapat meng-handle beberapa kasus sejenis menjadi satu. Misalnya, kita memiliki beberapa route yang membutuhkan autentikasi. Kita bisa membuat middleware untuk melakukan autentikasi dan menambahkan middleware tersebut ke setiap route yang membutuhkan autentikasi. Dengan begitu, kita tidak perlu menulis ulang kode autentikasi untuk setiap route yang membutuhkan autentikasi.

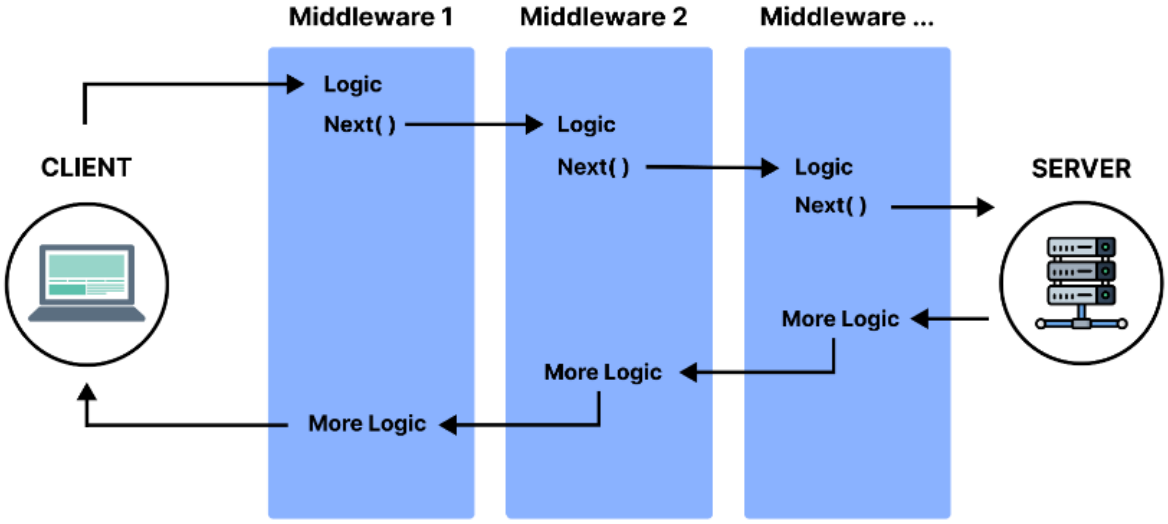
Secara umum, middleware sering digunakan untuk kasus autentikasi dan otorisasi. Middleware juga dapat digunakan untuk melakukan tugas seperti enkripsi, caching, kompresi, integrasi dengan sistem lain, manajemen sesi, dan load balancing.

4.3 Cara Kerja Middleware



Arsitektur middleware terdiri dari tiga komponen utama: client, web server, dan middleware:

- Client: Client (Klien) adalah perangkat yang mengirimkan permintaan ke web server, misalnya browser web.
- Server: Server atau Web Server adalah sistem yang menerima permintaan dari klien dan mengirimkan respons yang sesuai.
- Middleware: Middleware adalah perantara antara klien dan web server yang memproses permintaan dari klien, memvalidasi autentikasi dan otorisasi, dan mengirimkan respons yang sesuai. Middleware juga dapat menambahkan fitur tambahan seperti enkripsi, caching, dan kompresi.



Untuk middleware yang lebih dari 1 (middleware chain), klien mengirimkan permintaan ke middleware pertama. Middleware pertama memproses permintaan dan mengirimkan permintaan yang valid ke middleware kedua. Middleware kedua memproses permintaan dan mengirimkan permintaan yang valid ke middleware ketiga. Dan seterusnya sampai middleware terakhir.

Middleware terakhir mengirimkan permintaan yang valid ke web server. Web server mengirimkan respons kembali ke middleware terakhir, dan kemudian berlanjut sampai ke middleware awal. Terakhir, hasil response dari middleware awal di kirimkan ke klien.

Pada setiap middleware, dapat berisi logic sebelum diarahkan ke middleware selanjutnya. Logic ini disebut middleware logic before. Kemudian, middleware juga dapat berisi logic setelah middleware mendapatkan response dari web server. Logic ini disebut middleware logic after. Untuk lebih jelasnya dapat dilihat pada gambar di atas.

4.4 Contoh Middleware

```
package main

import (
    "github.com/gin-gonic/gin"
    "net/http"
)

func main() {
    r := gin.Default()

    // Middleware
```

```
    r.Use(func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{ "message": "Middleware Logic Before" })
        c.Next()
        c.JSON(http.StatusOK, gin.H{ "message": "Middleware Logic After" })
    })

    r.GET("/hello", func(c *gin.Context) {
        c.JSON(http.StatusOK, gin.H{ "message": "Hello World" })
    })

    r.Run(":8080")
}
```

Pada contoh kode di atas, kita membuat middleware yang berisi logic sebelum dan sesudah request diarahkan ke handler. Logic sebelum request diarahkan ke handler berada di dalam blok `c.Next()`, sedangkan logic setelah request diarahkan ke handler berada di luar blok `c.Next()`.

Untuk latihan menggunakan golang dapat dilihat di sini

5 Authorisasi dan Authentikasi

5.1 JWT JSON Web Token

JWT (JSON Web Token) adalah jenis token yang digunakan untuk mengidentifikasi user yang telah login pada aplikasi. Token ini berisi informasi tentang user dengan format JSON yang dienkripsi menggunakan algoritma tertentu dan dapat dibaca oleh server untuk memverifikasi identitas user dengan mudah tanpa harus menyimpan informasi sensitif mengenai login di server.

Banyak developer memilih menggunakan JWT dalam aplikasi mereka karena memudahkan pengelolaan user session, proses autentikasi dan otorisasi user lebih mudah diimplementasikan, dan membuat verifikasi lebih cepat dan efisien.

Biasanya token di kirim oleh client melalui header Authorization menggunakan skema Bearer yang akan terlihat seperti berikut:

```
Authorization: Bearer <token>
```

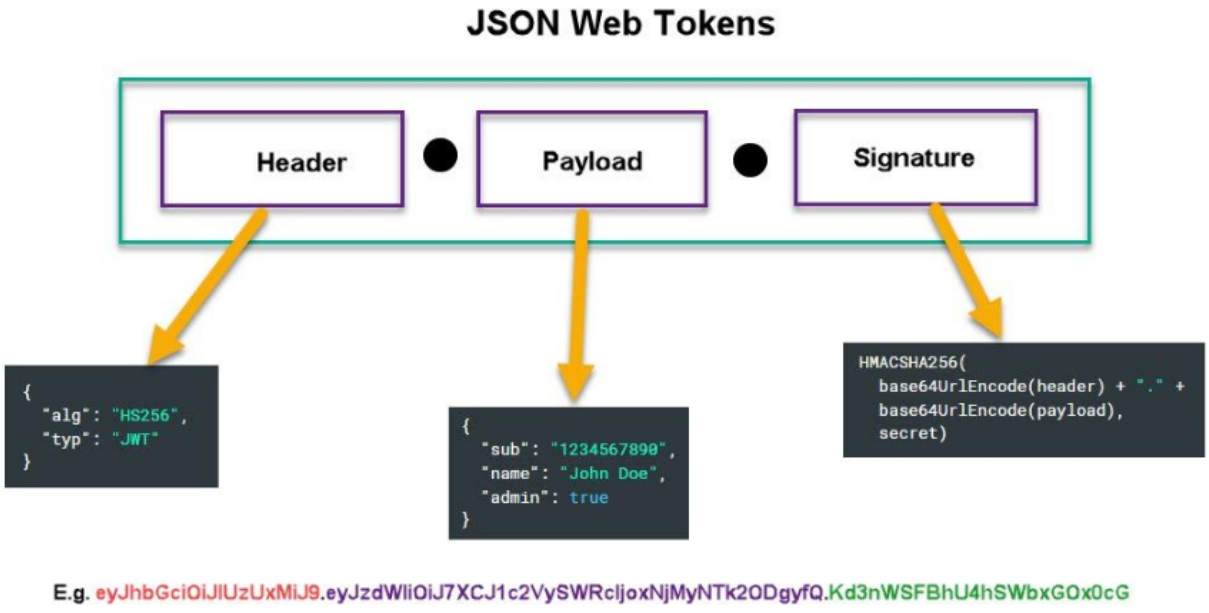
Selain itu, token juga bisa disimpan menggunakan cookie.

5.1.1 Anatomi

Anatomi JWT terdiri dari tiga bagian yang dipisahkan oleh titik (.), yaitu:

- Header: Bagian pertama dari JWT yang berisi informasi tentang algoritma enkripsi yang digunakan dan tipe token.

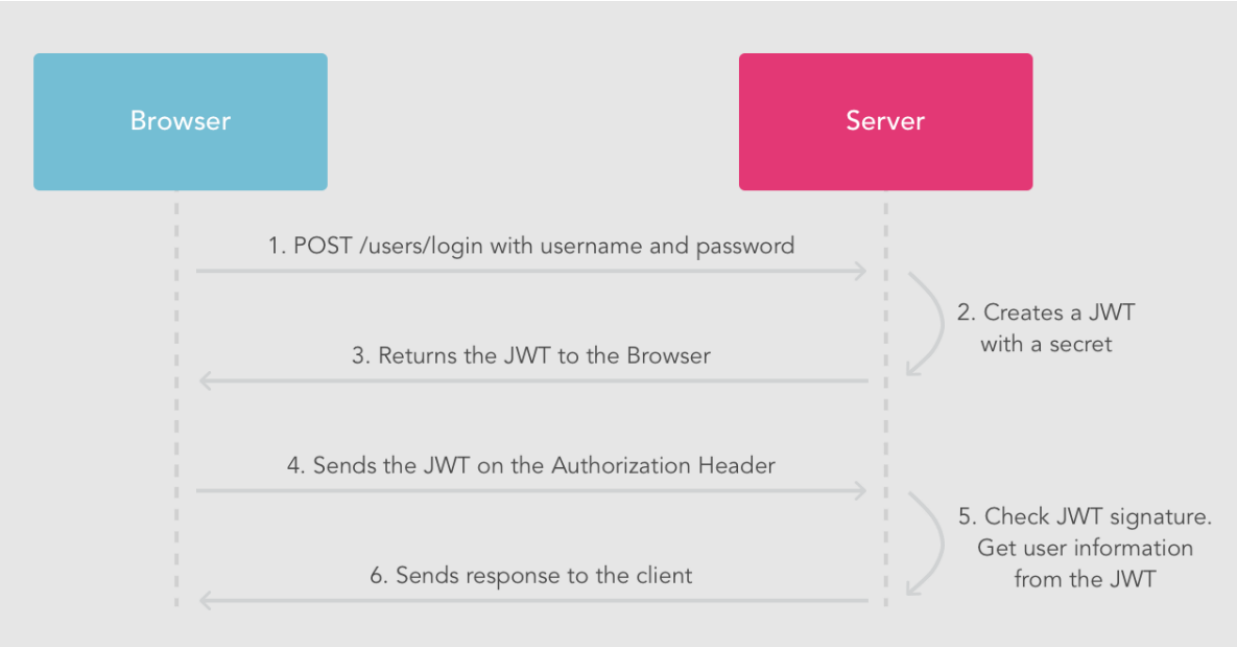
- Payload: Bagian kedua dari JWT yang berisi informasi user yang dienkripsi.
- Signature: Bagian terakhir dari JWT yang berisi tanda tangan digital yang digunakan untuk memverifikasi keaslian token.



5.1.2 Flow

Flow penggunaan JWT pada browser dan server dibagi menjadi beberapa tahap:

- Autentikasi: user melakukan login dengan mengirimkan informasi pengidentifikasi (seperti username dan password) ke server. Jika informasi pengidentifikasi benar, server akan mengeluarkan JWT yang berisi informasi user.
- Penyimpanan JWT: JWT yang diterima dari server disimpan dalam browser, biasanya dalam browser storage seperti local storage atau cookie.
- Autentikasi pada request: Setiap kali user mengirimkan request ke server, JWT disertakan dalam request sebagai header autentikasi. Server mengecek validitas JWT dan mengekstrak informasi user dari token.



5.1.3 JWT implementation

Untuk mengimplementasikan JWT menggunakan Golang kamu bisa melihat dokumentasi JWT Libraries dalam bahasa pemrograman Go. Di sana akan ditampilkan beberapa library yang tersedia untuk digunakan dalam aplikasi Go untuk menangani JWT, seperti mengeluarkan, memvalidasi, dan membuat JWT. Bisa dilihat di sini

```
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "time"

    "github.com/golang-jwt/jwt/v4"
)

// JWT key: dipakai untuk membuat signature JWT token.
var jwtKey = []byte("secret-key")

// Data key: user, password, role yang bisa digunakan untuk mengakses API
var users = map[string]*User{
    "aditira": {
        Password: "password1",
```

```
        Role:     "admin",
    },
    "dito": {
        Password: "password2",
        Role:     "student",
    },
}

type User struct {
    Password string
    Role     string
}

// Struct untuk membaca request body JSON
type Credentials struct {
    Password string `json:"password"`
    Username string `json:"username"`
}

// Struct Claims digunakan sebagai object yang akan di encode atau di parse oleh JWT
// jwt.StandardClaims ditambahkan sebagai embedded type untuk memudahkan proses encoding, parsing
type Claims struct {
    Username string `json:"username"`
    Role     string `json:"role"`
    jwt.StandardClaims
}


func main() {
    fmt.Println("Starting Server at port :8080")
    log.Fatal(http.ListenAndServe(":8080", Routes()))
}

func Routes() *http.ServeMux {
    mux := http.NewServeMux()

    mux.HandleFunc("/login", func(w http.ResponseWriter, r *http.Request) {
        var creds Credentials
        // JSON body diconvert menjadi credential struct & return bad request ketika terjadi kesa
        err := json.NewDecoder(r.Body).Decode(&creds)
        if err != nil {
            w.WriteHeader(http.StatusBadRequest)
            return
        }
```

```
    // Cek apakah username dan password ada dan sesuai dengan yang ada di Data Key & return u
    expectedPassword, ok := users[creds.Username]
    if !ok || expectedPassword.Password != creds.Password {
        w.WriteHeader(http.StatusUnauthorized)
        return
    }

    // Deklarasi expiry time untuk token jwt
    expirationTime := time.Now().Add(5 * time.Minute)
    // Buat claims berisi data username dan role yang akan kita embed ke JWT
    claims := &Claims{
    Username: creds.Username,
    Role:     expectedPassword.Role,
    StandardClaims: jwt.StandardClaims{
            // expiry time menggunakan time millisecond
            ExpiresAt: expirationTime.Unix(),
        },
    }

    // Buat token menggunakan encoded claim dengan salah satu algoritma yang dipakai
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)

    // Buat JWT string dari token yang sudah dibuat menggunakan JWT key yang telah dideklaras
    tokenString, err := token.SignedString(jwtKey)
    if err != nil {
        // return internal error ketika ada kesalahan saat pembuatan JWT string
        w.WriteHeader(http.StatusInternalServerError)
        return
    }

    // Set token string ke dalam cookie response
    http.SetCookie(w, &http.Cookie{
        Name:    "token",
        Value:   tokenString,
        Expires: expirationTime,
    })
})

// halaman /admin hanya dapat diakses oleh user yang sudah diotentikasi dan memiliki role adm
mux.HandleFunc("/admin", func(w http.ResponseWriter, r *http.Request) {
    // Ambil token dari cookie yang di kirim ketika request
    c, err := r.Cookie("token")
    if err != nil {
        if err == http.ErrNoCookie {
            // return unauthorized ketika token kosong
```

```
            w.WriteHeader(http.StatusUnauthorized)
            return
        }
        // return bad request ketika field token tidak ada
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    // Ambil value dari cookie token
    tknStr := c.Value

    // Deklarasi variable claims yang akan kita isi dengan data hasil parsing JWT
    claims := &Claims{}

    // parse JWT token ke dalam claims
    tkn, err := jwt.ParseWithClaims(tknStr, claims, func(token *jwt.Token) (interface{}, erro
        return jwtKey, nil
    })
    if err != nil {
        if err == jwt.ErrSignatureInvalid {
            // return unauthorized ketika ada kesalahan saat parsing token
            w.WriteHeader(http.StatusUnauthorized)
            return
        }
        // return bad request ketika field token tidak ada
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    //return unauthorized ketika token sudah tidak valid (biasanya karena token expired)
    if !tkn.Valid {
        w.WriteHeader(http.StatusUnauthorized)
        return
    }

    //return unauthorized ketika role user tidak sesuai dengan role admin
    if claims.Role != "admin" {
        w.WriteHeader(http.StatusUnauthorized)
        return
    }

    // return data dalam claims, yaitu username yang telah didefinisikan di variable claims
    w.Write([]byte(fmt.Sprintf("Welcome Admin %s!", claims.Username)))
})
```

```
// halaman /profile dapat diakses oleh user baik admin maupun bukan selama terotentikasi.
mux.HandleFunc("/profile", func(w http.ResponseWriter, r *http.Request) {
    // Ambil token dari cookie yang di kirim ketika request
    c, err := r.Cookie("token")
    if err != nil {
        if err == http.ErrNoCookie {
            // return unauthorized ketika token kosong
            w.WriteHeader(http.StatusUnauthorized)
            return
        }
        // return bad request ketika field token tidak ada
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    // Ambil value dari cookie token
    tknStr := c.Value

    // Deklarasi variable claims yang akan kita isi dengan data hasil parsing JWT
    claims := &Claims{}

    //parse JWT token ke dalam claims
    tkn, err := jwt.ParseWithClaims(tknStr, claims, func(token *jwt.Token) (interface{}, erro
        return jwtKey, nil
    })
    if err != nil {
        if err == jwt.ErrSignatureInvalid {
            // return unauthorized ketika ada kesalahan ketika parsing token
            w.WriteHeader(http.StatusUnauthorized)
            return
        }
        // return bad request ketika field token tidak ada
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    //return unauthorized ketika token sudah tidak valid (biasanya karena token expired)
    if !tkn.Valid {
        w.WriteHeader(http.StatusUnauthorized)
        return
    }

    // return data dalam claims, yaitu username yang telah didefinisikan di variable claims
    w.Write([]byte(fmt.Sprintf("Welcome %s!", claims.Username)))
})
```

```
    return mux
}
```

Untuk mencoba kode di atas menggunakan Postman, kita dapat mengikuti langkah-langkah berikut:

- Jalankan server dengan menjalankan perintah go run nama_file.go pada terminal.
- Buka Postman dan buat request baru dengan metode POST. Isikan alamat URL dengan http://localhost:8080/login
- Pada tab body, pilih raw dan pilih JSON(application/json) sebagai format yang digunakan.
- Isikan body dengan format JSON berikut {“username”:“aditira”, “password”:“password1”}
- Klik tombol send untuk mengirimkan request.
- Jika login berhasil, server akan mengeluarkan token JWT yang diterima dalam bentuk cookie. Anda dapat menggunakan token ini untuk mengirimkan permintaan ke halaman yang dibatasi oleh otorisasi.
- Buat request baru dengan metode GET, isikan alamat URL dengan http://localhost:8080/admin
- Klik tombol send untuk mengirimkan request.
- Jika token yang digunakan valid, server akan mengirimkan respon yang sesuai. Jika tidak, server akan mengembalikan kode respon 401 Unauthorized.

5.2 OAuth

5.2.1 Pengertian

OAuth (Open Authorization) adalah protokol otentikasi yang memungkinkan aplikasi untuk mengakses layanan atau data milik user tanpa memberikan informasi login (seperti username dan password) ke aplikasi tersebut.

OAuth digunakan oleh banyak layanan web populer, seperti Google, Facebook, Twitter, dan lainnya, untuk mengizinkan aplikasi lain untuk mengakses data user yang tersimpan di layanan tersebut.

5.2.2 Why OAuth?

OAuth memungkinkan user untuk mengontrol hak akses yang diberikan ke aplikasi dan mudah untuk membatalkannya jika diperlukan. Selain itu, OAuth juga memungkinkan aplikasi untuk mengakses data dari berbagai layanan tanpa harus mengimplementasikan otentikasi yang berbeda untuk setiap layanan.

Misalnya jika seorang user ingin menghubungkan aplikasi foto miliknya dengan akun Instagram-nya, dia dapat menggunakan OAuth untuk memberikan aplikasi foto akses ke akun Instagram-nya tanpa harus memberikan kredensial akun Instagram-nya ke aplikasi foto tersebut. User juga dapat mengontrol hak akses yang diberikan ke aplikasi dan membatalkannya jika diperlukan.

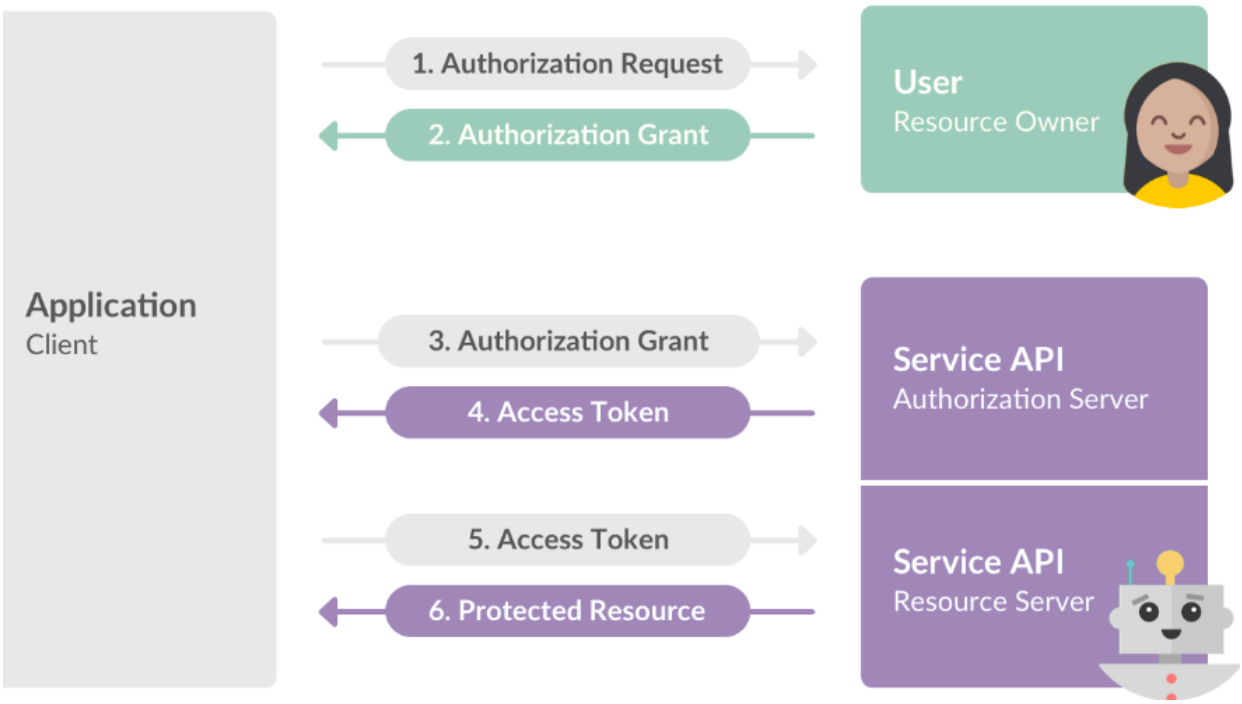
Contoh lain adalah ketika user ingin mengirim email dari aplikasi email tertentu menggunakan akun Google-nya, user dapat menggunakan OAuth untuk memberikan akses ke aplikasi tersebut tanpa harus memberikan password Google-nya.

5.2.3 Basic Flow

Basic flow dari OAuth adalah sebagai berikut:

- Aplikasi client mengirim permintaan otentikasi ke server OAuth.
- User diminta untuk masuk dengan akun yang sesuai.
- Setelah masuk, server OAuth mengarahkan user ke halaman izin yang menjelaskan hak akses yang diminta oleh aplikasi client.
- User menyetujui atau menolak izin yang diminta.
- Jika izin diterima, server OAuth mengirim token akses ke aplikasi client.
- Aplikasi client menggunakan token akses untuk mengakses data dari server layanan.
- Server layanan mengecek token akses dan memberikan akses ke data user jika token valid.

Itu adalah alur dasar dari OAuth, dan mungkin ada variasi dari implementasi ke implementasi. Namun prinsip dasarnya sama, yaitu mengizinkan aplikasi untuk mengakses data pribadi user dengan aman tanpa memberikan kredensial user tersebut ke aplikasi.



5.2.4 Implementasi OAuth

```
package main

import (
    "fmt"
    "net/http"

    "golang.org/x/oauth2"
    "golang.org/x/oauth2/google"
)

var (
    googleOauthConfig = &oauth2.Config{
        RedirectURL:  "http://localhost:8080/callback",
        ClientID:     "YOUR_CLIENT_ID",
        ClientSecret: "YOUR_CLIENT_SECRET",
        Scopes:       []string{"https://www.googleapis.com/auth/userinfo.profile", "https://www.g
        Endpoint:     google.Endpoint,
    }
    // Some random string, random for each request
    oauthStateString = "random"
)
```

```
func main() {
    http.HandleFunc("/", handleMain)
    http.HandleFunc("/login", handleGoogleLogin)
    http.HandleFunc("/callback", handleGoogleCallback)
    fmt.Println(http.ListenAndServe(":8080", nil))
}

func handleMain(w http.ResponseWriter, r *http.Request) {
    var htmlIndex = `<html><body>
        <a href="/login">Google Log In</a>
        </body></html>`

    fmt.Fprintf(w, htmlIndex)
}

func handleGoogleLogin(w http.ResponseWriter, r *http.Request) {
    url := googleOauthConfig.AuthCodeURL(oauthStateString)
    http.Redirect(w, r, url, http.StatusTemporaryRedirect)
}

func handleGoogleCallback(w http.ResponseWriter, r *http.Request) {
    state := r.FormValue("state")
    if state != oauthStateString {
        fmt.Printf("invalid oauth state, expected '%s', got '%s'\n", oauthStateString, state)
        http.Redirect(w, r, "/", http.StatusTemporaryRedirect)
        return
    }

    code := r.FormValue("code")
    token, err := googleOauthConfig.Exchange(oauth2.NoContext, code)
    if err != nil {
        fmt.Println("Code exchange failed with '%s'\n", err)
        http.Redirect(w, r, "/", http.StatusTemporaryRedirect)
        return
    }

    response, err := http.Get("https://www.googleapis.com/oauth2/v2/userinfo?access_token=" + tok

    defer response.Body.Close()
    contents, err := ioutil.ReadAll(response.Body)
    fmt.Fprintf(w, "Content: %s\n", contents)
}
```