**API Design**

**JSON Encoding & Decoding**

**First, We have to understand that the Encoding process doesn't actually mean that it translates types and returns a JSON representation of a type. The process that gives you the JSON representation is called the Marshaling process and could be done by calling the** [**json.Marshal function**](https://golang.org/pkg/encoding/json/#Marshal)**.**

**On the other hand, the** [**Encoding**](https://golang.org/pkg/encoding/json/#Encoder.Encode) **process means that we want to get the JSON encoding of any type and to write(encode) it on a stream that implements io.Writer interface. As we can see the func NewEncoder(w io.Writer) \*Encoder receives an io.Writer interface as a parameter and returns a \*json.Encoder object. When the method encoder.Encode() is being called, it does the Marshaling process and then writes the result to the io.Writer that we have passed when creating a new Encoder object. You could see the implementation of json.Encoder.Encode()** [**here**](https://golang.org/src/encoding/json/stream.go?s=5070:5117#L191)**.**

**So, if you asked who does do the encoding process to the http stream, the answer is the http.ResponseWriter. ResponseWriter implements the io.Writer interface and when the Encode() method is being called, the encoder will Marshal the object to a JSON encoding representation and then call the func Write([]byte) (int, error) which is a contract method of the io.Writer interface and it will do the writing process to the http stream.**

**In summary, I could say that Marshal and Unmarshal mean that we want to get the JSON representation of any type and vice-versa. While Encode means that we want to do the Marshaling process and then write(encode) the result to any stream object. And Decode means that we want to get(decode) a json object from any stream and then do the Unmarshaling process.**

**Authentication and Authorization**

**Authentication verifies the identity of a user or service, and authorization determines their access rights.**

**Authentication Approach:**

* **Basic Authentication**
* **Bearer Authentication**
* **Client Certificate (Common + Popular)**

**Bearer authentication (also called token authentication) is an** [**HTTP authentication scheme**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication) **that involves security tokens called bearer tokens. The name “Bearer authentication” can be understood as “give access to the bearer of this token.” The bearer token is a cryptic string, usually generated by the server in response to a login request. The client must send this token in the Authorization header when making requests to protected resources:**

**JWT (3 parts) -> No Encryption**

**HEADER:ALGORITHM & TOKEN TYPE**

**{**

**"alg": "HS256",**

**"typ": "JWT"**

**}**

**PAYLOAD:DATA**

**{**

**"sub": "1234567890",**

**"name": "John Doe",**

**"iat": 1516239022**

**}**

**VERIFY SIGNATURE**

**HMACSHA256(**

**base64UrlEncode(header) + "." +**

**base64UrlEncode(payload),**

**) secret base64 encoded**

**SHA-256 is used to convert user passwords into a hash value before storing them in a database. The hash values are matched to the user's input during login verification.**

**Authorization:**

**Curl Command:**

**LoginHandler:**

curl -X POST -H "Content-Type: application/json" -d '{"username": "admin", "password": "admin"}' http://localhost:8080/login

**VerifyJWT- Post**

curl -X POST -H "Content-Type: application/json" -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE3MDU2NTkwNzMsInVzZXJuYW1lIjoiYWRtaW4ifQ.vg\_AdGSml4h\_FwCmyWl9287F8rb6Y2inHmcAlsgIhb8" -d '{"name": "YourBookName", "authorList": ["Author1", "Author2"], "publishDate": "2022-01-01", "isbn": "1234567890"}' <http://localhost:8080/api/v1/books>

# Build the Go application for a Linux environment

RUN CGO\_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o book\_api .

* CGO\_ENABLED=0: Disables CGO (C Go) to ensure a statically linked binary. CGO is the bridge between Go and C, and disabling it results in a binary that doesn't depend on external C libraries.
* GOOS=linux: Specifies the target operating system as Linux.
* go build -a: Forces rebuilding of all packages, even if they are up-to-date. This ensures that the binary is rebuilt with the specified configurations.
* -installsuffix cgo: Appends "cgo" to the output directory during the build. This is commonly used to create separate output directories for binaries built with and without CGO.
* -o book\_api: Specifies the output name of the binary as "book\_api".
* .: Indicates the source directory where the Go code is located. In this case, it's the current directory.