Multiple step build technique

Since golang has the ability to provide bin file by compiling the source code. We can very easily utilize this property to prepare small containers.

When FROM golang: any-version is done, it accumulates all functionalities and properties required for preparing a go environment.

But in an application, there is no need of everything rather a very small amount of resources is reqd. This is where multi step build comes in. After preparing the bin file of the source code, we use a very minimal base to build again. But this time we use the bin file as source. Hence, only the properties reqd. as per the bin file are accumulated making the app extremely light weight.

A sample ex →

```
FROM golang:latest AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o
                                    ecommerceApp ./cmd/main.go
                                    or *.go

FROM alpine:latest
WORKDIR /root/
COPY --from=builder /app/ecommerceApp .
COPY /cmd/.env .env
RUN apk --no-cache add ca-certificates
EXPOSE 10000
CMD ["./ecommerceApp"]
```

After a significant discussion on docker, let's move back to where we left, middlewares.

Middlewares

Here, we are going to set up only one middleware, auth for allowing authorization.

The MVP here is ParseJWT function of auth.go.
It takes the token string and parse it to convert to byte token.jwt.ParseWithClaims function is utilized here.
This fn is a bit complex and weird but it is what it is.
It returns the token and error if parsing fails.
Then, we extract the claims from the token returned above and check for its type by comparing it with OurAppClaims which is our custom claim with std. claims plus user's info like email, id, etc. The type assertion returns the claims and a bool value ok. If everything is ok till now, returning the claims. (Before returning expiry date is checked if it is before the current time or not if it's before, then it's expired and gives error)

Now comes the role of middleware.go
It has a fn → Authorization.
It takes out the token from cookies.
Then it checks if it's empty. If it exists then the parse method is used to parse the token and take out the claims.
If claims are taken out then it simply means token is valid and claims consists of the user's info. Then set the gin's context to include the user's data. Then it next to the next fn in line with the data passed in.

In the later fn., utilize the passed info. to hit the route of authorized sections.

Now, to add middleware fn to the routes, we can add them directly. Or we can group the routes that require the same middleware, using Group fn of gin engine.

for ex → forgot password, add items and view products can be grouped together.

```
P := r.Group("/")
P.Use(middleware.Authorization())
P.POST("/forgotpassword", forgotp())
P.POST("/additems", additems())
:
and so on.
```