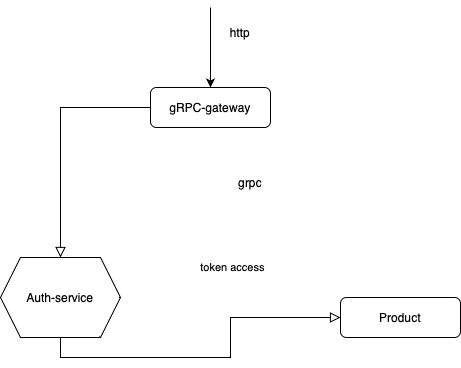
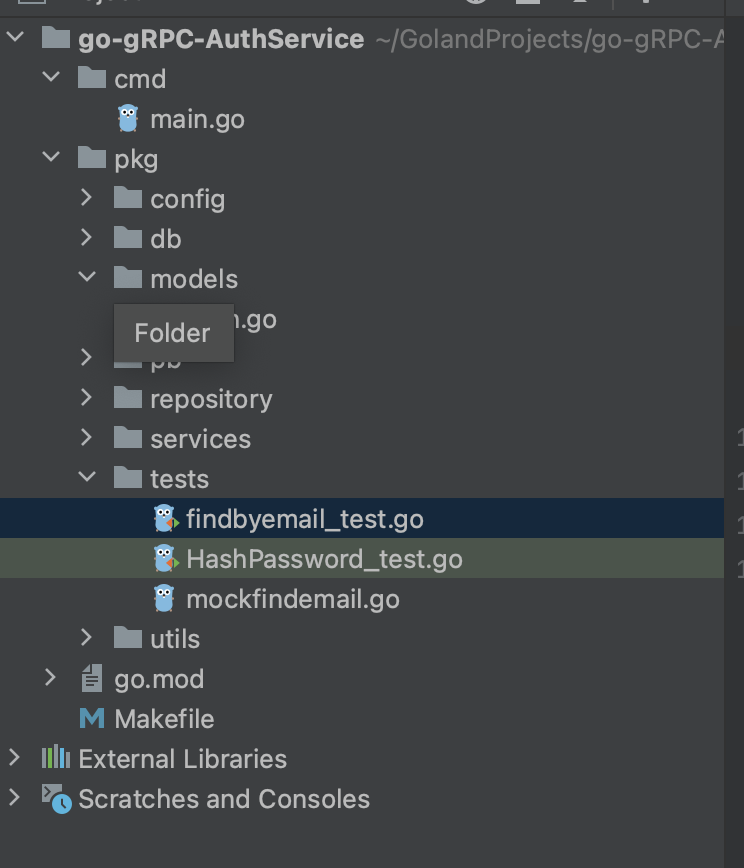
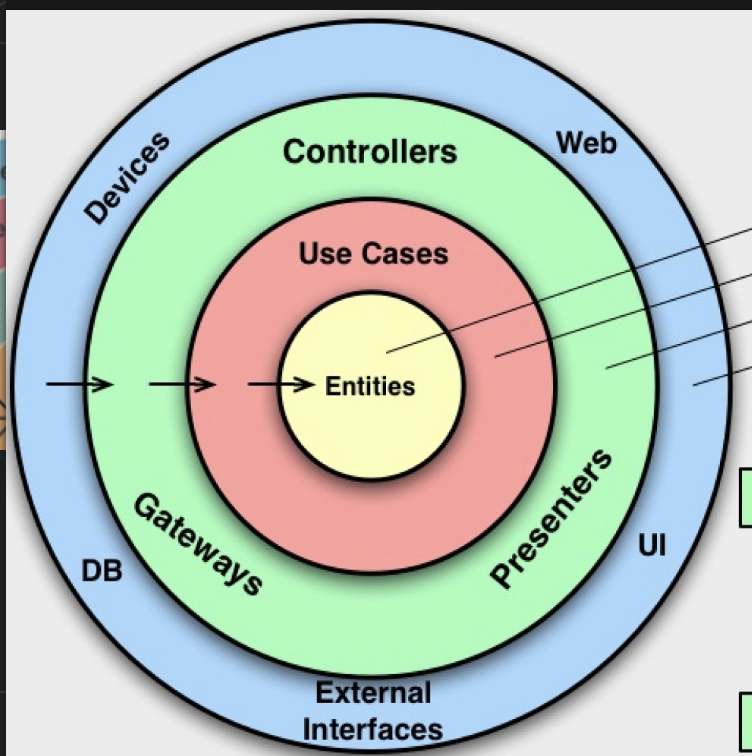
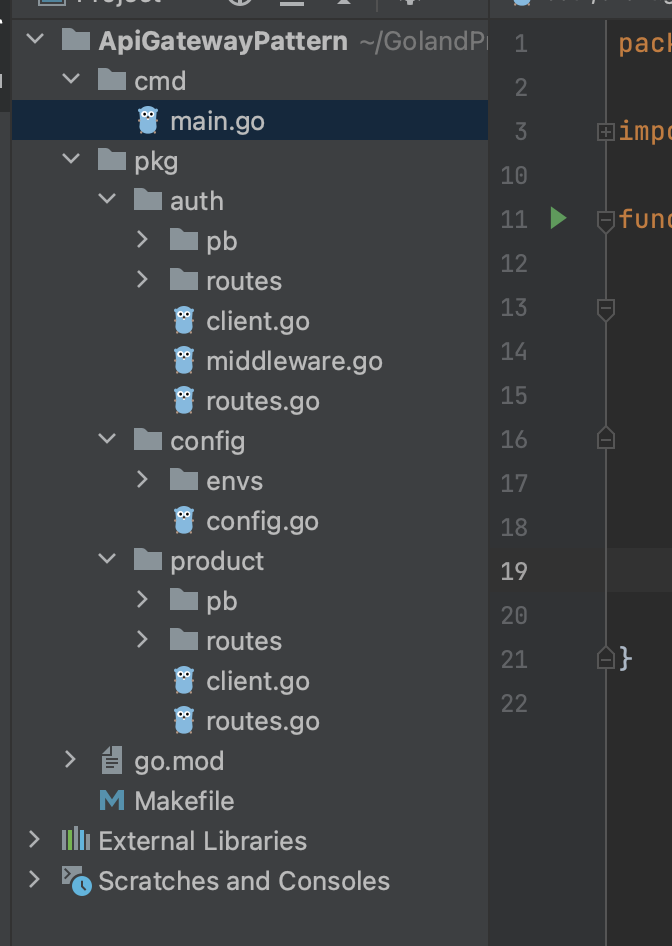
**API DOCUMENTATION**



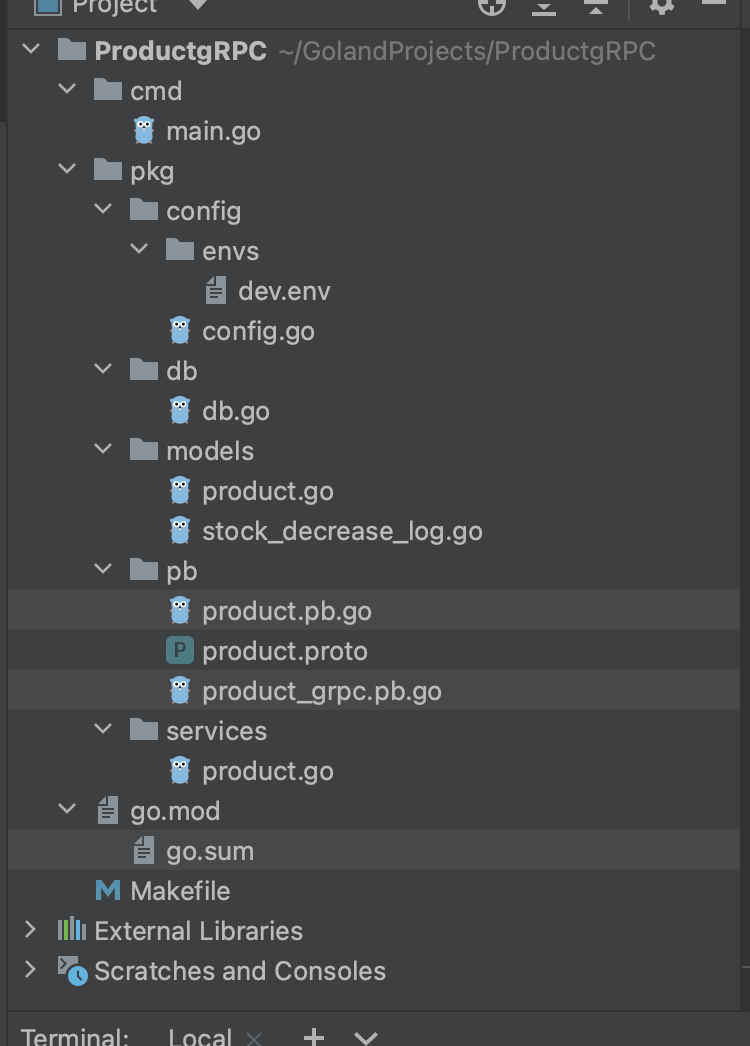
So basicly gRPC-Gateway acts as a powerful intermediary between the authorization service and the product service, enabling seamless communication and integration between the two. Acting as a translation layer, the gRPC-Gateway allows RESTful clients to interact with the gRPC-based authorization and product services. It handles the conversion of RESTful API requests into gRPC requests and forwards them to the respective services. This abstraction layer simplifies the process of exposing gRPC services to REST clients, promoting interoperability and easing the integration of different systems. By leveraging gRPC-Gateway, developers can harness the benefits of gRPC's performance and efficiency while accommodating the needs of RESTful clients, ensuring secure and efficient interactions between the authorization service and the product service.

So here is the clean architecture of Authorization microservice. So basicly 

But also there are in transport layer is grpc proto files.

So there are my grpcgateway that just routing for another two services.  
My proto document  
syntax = "proto3";  
  
package auth;  
  
option go\_package = "./pkg/auth/pb";  
  
service AuthService {  
 rpc Register(RegisterRequest) returns (RegisterResponse) {}  
 rpc Login(LoginRequest) returns (LoginResponse) {}  
 rpc Validate(ValidateRequest) returns (ValidateResponse) {}  
}  
  
// Register  
  
message RegisterRequest {  
 string email = 1;  
 string password = 2;  
}  
  
message RegisterResponse {  
 int64 status = 1;  
 string error = 2;  
}  
  
// Login  
  
message LoginRequest {  
 string email = 1;  
 string password = 2;  
}  
  
message LoginResponse {  
 int64 status = 1;  
 string error = 2;  
 string token = 3;  
}  
  
// Validate  
  
message ValidateRequest { string token = 1; }  
  
message ValidateResponse {  
 int64 status = 1;  
 string error = 2;  
 int64 userId = 3;  
}

package auth  
  
import (  
 "context"  
 "github.com/SmagulLK/APIGateway/pkg/auth/pb"  
 "github.com/gin-gonic/gin"  
 "strings"  
)  
  
type AuthMiddleware struct {  
 service \*ServiceClient  
}  
  
func InitAuthMiddleware(service \*ServiceClient) \*AuthMiddleware {  
 return &AuthMiddleware{  
 service: service,  
 }  
}  
func (c \*AuthMiddleware) AuthRequire(ctx \*gin.Context) {  
 authorization := ctx.Request.Header.Get("authorization")  
 if authorization == "" {  
 ctx.AbortWithStatus(401)  
 return  
 }  
 token := strings.Split(authorization, "Bearer ")  
 if len(token) != 2 {  
 ctx.AbortWithStatus(401)  
 return  
 }  
 res, err := c.service.Client.Validate(context.Background(), &pb.ValidateRequest{  
 Token: token[1],  
 })  
 if err != nil || res.Status != 200 {  
 ctx.AbortWithStatus(502)  
 return  
 }  
 ctx.Set("user\_id", res.UserId)  
 ctx.Next()  
  
}



Clean architecture is a software architectural pattern that promotes separation of concerns and maintainability in microservices-based systems. It emphasizes the organization of code into distinct layers, each with a specific responsibility and level of abstraction. Clean architecture helps to create modular and testable microservices by enforcing boundaries and minimizing dependencies between components.

In the context of microservices, clean architecture typically consists of the following layers:

Presentation Layer: This layer handles the interaction with external clients, such as web or mobile applications. It includes components responsible for handling requests, performing validations, and returning appropriate responses. The presentation layer should be agnostic to the underlying technologies and frameworks.

Application Layer: The application layer contains business logic and orchestrates the flow of data and operations within the microservice. It defines the use cases and interacts with the domain layer for processing business-specific rules. This layer is independent of frameworks and libraries, focusing solely on implementing the business requirements.

Domain Layer: At the core of the microservice, the domain layer encapsulates the domain models, entities, and business rules. It represents the essential concepts and behaviors of the business domain, free from any infrastructure or technology-specific details. The domain layer should be framework-agnostic and easily testable.

Infrastructure Layer: The infrastructure layer handles external dependencies, such as databases, external services, or message queues. It includes implementation details like data access, external API integrations, and other infrastructure concerns. The infrastructure layer should be isolated, allowing for easy replacement or modification of external dependencies without affecting the core business logic.

Clean architecture principles emphasize loose coupling and high cohesion between layers. It allows for independent development, testing, and deployment of microservices, as each layer has well-defined responsibilities and boundaries. The separation of concerns enables teams to work on different layers simultaneously, promoting scalability and maintainability.

By adhering to clean architecture principles, microservices gain flexibility and modularity. Developers can easily replace or upgrade components within a microservice without affecting other parts of the system. Additionally, the architecture facilitates unit testing, as the core business logic resides in the domain layer, which is decoupled from infrastructure dependencies.

In summary, clean architecture provides a structured approach to designing microservices, ensuring separation of concerns, maintainability, and scalability. It promotes code reusability, testability, and long-term system evolution by establishing clear boundaries and encapsulating business logic within the domain layer.