# Clean Architecture on Golang

Independent, Testable , and Clean

**Basic**

As we know the constraint before designing the Clean Architecture are :

1. Independent of Frameworks. The architecture does not depend on the existence of some library of feature laden software. This allows you to use such frameworks as tools, rather than having to cram your system into their limited constraints.
2. Testable. The business rules can be tested without the UI, Database, Web Server, or any other external element.
3. Independent of UI. The UI can change easily, without changing the rest of the system. A Web UI could be replaced with a console UI, for example, without changing the business rules.
4. Independent of Database. You can swap out Oracle or SQL Server, for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.
5. Independent of any external agency. In fact your business rules simply don’t know anything at all about the outside world.

So, based on this constraint, every layer must independent and testable.

In clean Architecture, has 4 layer :

* Entities
* Usecase
* Controller
* Framework & Driver

In most of the projects,following layers can be used :

* Models
* Repository
* Usecase
* Delivery

**Models**

Same as Entities, will used in all layer. This layer, will store any Object’s Struct and its method. Example : Article, Student, Book.  
Example struct :

Any entities, or model will stored here.

**Repository**

Repository will store any Database handler. Querying, or Creating/ Inserting into any database will stored here. This layer will act for CRUD to database only. No business process happen here. Only plain function to Database.

This layer also have responsibility to choose what DB will used in Application. Could be Mysql, MongoDB, MariaDB, Postgresql whatever, will decided here.

If using ORM, this layer will control the input, and give it directly to ORM services.

If calling microservices, will handled here. Create HTTP Request to other services, and sanitize the data. This layer, must fully act as a repository. Handle all data input - output no specific logic happen.

This Repository layer will depends to Connected DB , or other microservices if exists.

**Usecase**

This layer will act as the business process handler. Any process will handled here. This layer will decide, which repository layer will use. And have responsibility to provide data to serve into delivery. Process the data doing calculation or anything will done here.

Usecase layer will accept any input from Delivery layer, that already sanitized, then process the input could be storing into DB , or Fetching from DB ,etc.

This Usecase layer will depends to Repository Layer

**Delivery**

This layer will act as the presenter. Decide how the data will presented. Could be as REST API, or HTML File, or RPC whatever the delivery type.   
This layer also will accept the input from user. Sanitize the input and sent it to Usecase layer.

For my sample project, I’m using REST API as the delivery method.   
Client will call the resource endpoint over network, and the Delivery layer will get the input or request, and sent it to Usecase Layer.

This layer will depends to Usecase Layer.

**Communications Between Layer**

Except Models, each layer will communicate through inteface. For example, Usecase layer need the Repository layer, so how they communicate? Repository will provide an interface to be their contract and communication.

Example of Repository’s Interface

Usecase layer will communicate to Repository using this contract, and Repository layer **MUST**implement this interface so can used by Usecase

Example of Usecase’s Interface

Same with Usecase, Delivery layer will use this contract interface. And Usecase layer **MUST**implement this interface.

**Testing Each Layer**

As we know, clean means independent. Each layer testable even other layers doesn’t exist yet.

* Models Layer  
  This layer only tested if any function/method declared in any of Struct.  
  And can test easily and independent to other layers.
* Repository  
  To test this layer, the better ways is doing Integrations testing. But you also can doing mocking for each test. I’m using github.com/DATA-DOG/go-sqlmock as my helper to mock query process msyql.
* Usecase  
  Because this layer depends to Repository layer, means this layer need Repository layer for testing . So we must make a mockup of Repository  
  that mocked with mockery, based on the contract interface defined before.
* Delivery  
  Same with Usecase, because this layer depends to Usecase layer, means we need Usecase layer for testing. And Usecase layer also must mocked with mockery, based on the contract interface defined before