
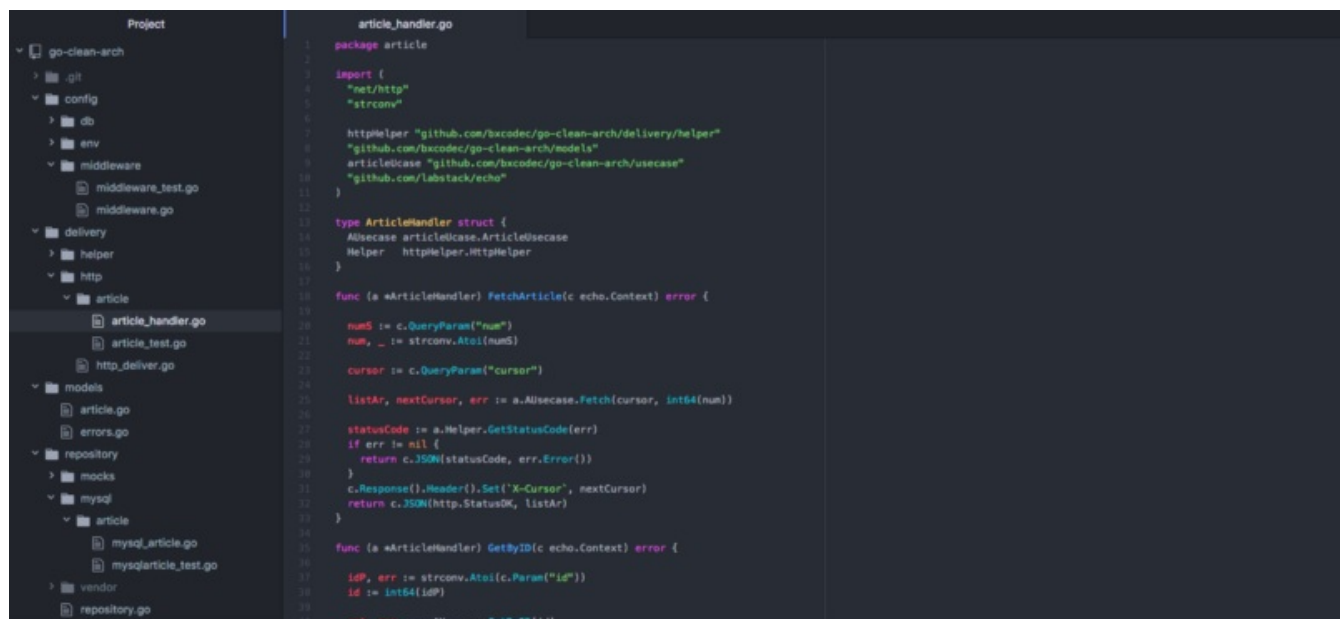


# Trying Clean Architecture on Golang

 [hackernoon.com/golang-clean-architecture-efd6d7c43047](https://hackernoon.com/golang-clean-architecture-efd6d7c43047)

After reading the uncle Bob's Clean Architecture Concept, I'm trying to implement it in Golang. This is a similar architecture that we used in our company, **Kurio - App Berita Indonesia**, but a little different structure. Not too different, same concept but different in folder structure.

You can look for a sample project here <https://github.com/bxcodec/go-clean-arch>, a sample CRUD management article.



Disclaimer :

I'm not recommending any library or framework used here. You could replace anything here, with your own or third party that has the same functions.

## 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.

More at <https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html>

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

If Uncle Bob's Architecture, has 4 layer :

- Entities
- Usecase
- Controller
- Framework & Driver

In my projects, I'm using 4 too :

- 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 gRPC 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 interface. 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](https://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 be mocked with mockery, based on the contract interface defined before

For mocking, I use mockery for go by vektra, which can be seen here <https://github.com/vektra/mockery>

## Repository Test

---

To test this layer, like I said before, I'm using a sql-mock to mock my query process. You can use like what I used here [github.com/DATA-DOG/go-sqlmock](https://github.com/DATA-DOG/go-sqlmock), or another that has similar function

```
func TestGetByID(t *testing.T) {
    db, mock, err := sqlmock.New()
    if err != nil {
        t.Fatalf("an error '%s' was not expected when opening a stub
            database connection", err)
    }

    defer db.Close()
    rows := sqlmock.NewRows([]string{
        "id", "title", "content", "updated_at", "created_at"}).
        AddRow(1, "title 1", "Content 1", time.Now(), time.Now())

    query := "SELECT id,title,content,updated_at, created_at FROM
        article WHERE ID = \?"

    mock.ExpectQuery(query).WillReturnRows(rows)

    a := articleRepo.NewMySQLArticleRepository(db)

    num := int64(1)

    anArticle, err := a.GetByID(num)

    assert.NoError(t, err)
    assert.NotNil(t, anArticle)
}
```

## Usecase Test

---

Sample test for Usecase layer, that depends on Repository layer.

Mockery will generate a mockup of repository layer for me. So I don't need to finish my Repository layer first. I can work finishing my Usecase first even my Repository layer not implemented yet.

## Delivery Test

---

Delivery test will depends on how you to deliver the data. If using http REST API, we can use httptest a builtin package for httptest in golang.

Because it's depends to Usecase, so we need a mock of Usecase . Same with Repository, i'm also using Mockery to mock my usecase, for delivery testing.

```
func TestGetByID(t *testing.T) {
    var mockArticle models.Article
    err := faker.FakeData(&mockArticle)
    assert.NoError(t, err)
    mockUCase := new(mockArticleUsecase)
    num := int(mockArticle.ID)
    mockUCase.On("GetByID", int64(num)).Return(&mockArticle, nil)
    e := echo.New()
    req, err := http.NewRequest(echo.GET, "/article/" +
        strconv.Itoa(int(num)), strings.NewReader(""))

    assert.NoError(t, err)

    rec := httptest.NewRecorder()
    c := e.NewContext(req, rec)
    c.SetPath("article/:id")
    c.SetParamNames("id")
    c.SetParamValues(strconv.Itoa(num))

    handler := articleHttp.ArticleHandler{
        AUseCase: mockUCase,
        Helper: httpHelper.HttpHelper{}
    }
    handler.GetByID(c)

    assert.Equal(t, http.StatusOK, rec.Code)
    mockUCase.AssertCalled(t, "GetByID", int64(num))
}
```

## Final Output and The Merging

---

After finish all layer and already passed on testing. You should merge into one system in main.go in root project.

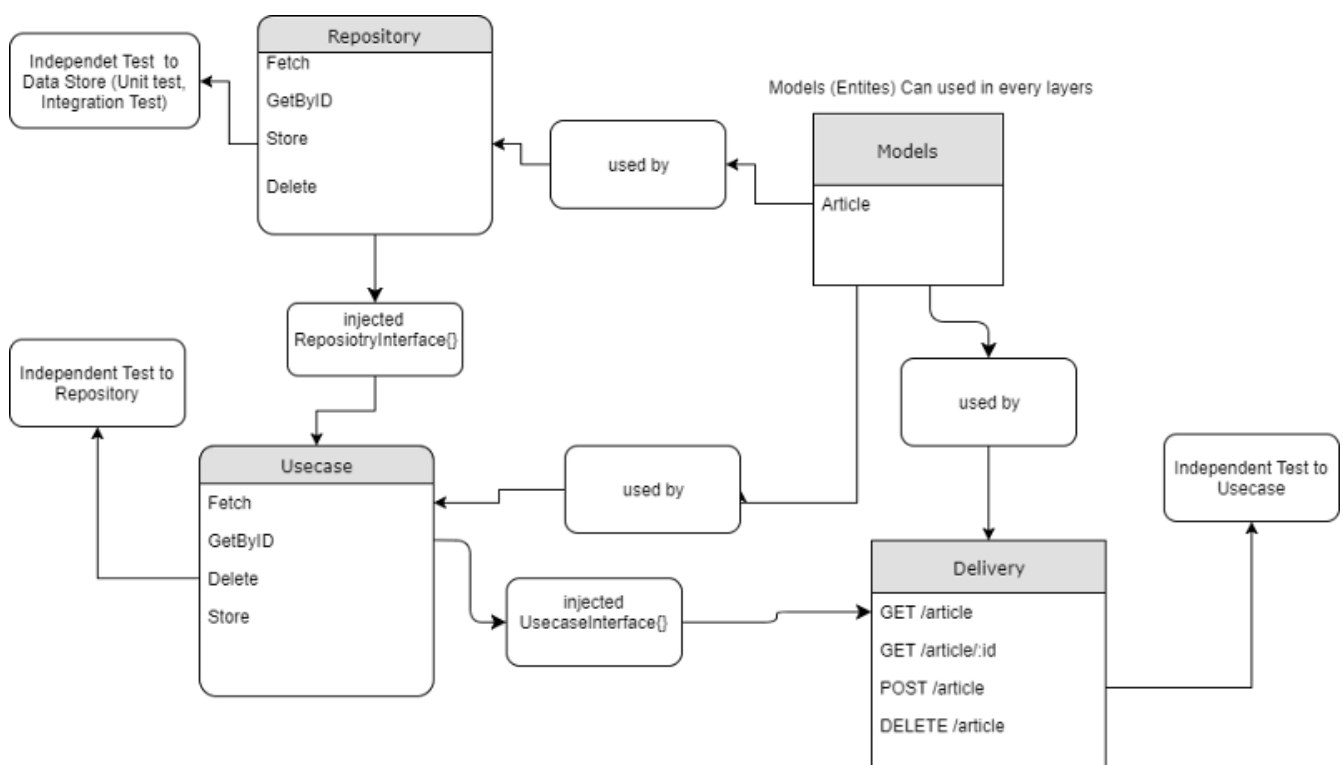
Here you will define, and create every needs to environment, and merge all layers into one.

Look for my main.go as example:

You can see, every layer merge into one with its dependencies.

Conclusion :

In short, if drawn in a diagram, can seen below



- Every library used here you can change by your own. Because the main point of clean architecture is : no matter your library, but your architecture is clean, and testable also independent
- This is how i organized my projects, you could argue, or agree, or maybe improve this for more better, just leave a comment and share this

## The Sample Projects

The sample project can seen here <https://github.com/bxcodec/go-clean-arch>

Library used for my project:

- Glide : for package management
- go-sqlmock from [github.com/DATA-DOG/go-sqlmock](https://github.com/DATA-DOG/go-sqlmock)
- Testify : for testing
- Echo Labstack (Golang Web Framework) for Delivery layer
- Viper : for environment configurations

Further Reading about Clean Architecture :

- Second part of this article: <https://hackernoon.com/trying-clean-architecture-on-golang-2-44d615bf8fdf>
- <https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html>
- <http://manuel.kiessling.net/2012/09/28/applying-the-clean-architecture-to-go-applications/> . Another version of Clean Architecture in Golang

*If you have a question , or need more explanation, or something, that I can not explain well here, you can ask me from my or email me. Thank you*