Project overview:

This is a simple banking API implemented in REST architecture.

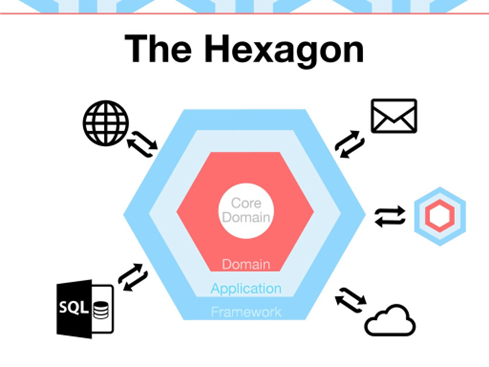
I used hexagonal design pattern in order to make application robust, scalable and flexible.

Project Stack:

* GO language.
* Gorillamux library as server multiplexer.
* Uber-go/
* zap as internal application logger.
* MySQL as database.
* Sql driver for connecting application to Mysql Service.
* Sqlx as an additional extension for interacting with database.

Brief explanation about design pattern:

Alistair Cockburn developed hexagonal architecture, or ports and adapters architecture, and used it for designing software applications. We insert inputs and outputs on the edge of our design with hexagonal architecture. It allows us to isolate the central logic of the application. Since our inputs and outputs are on edge, we can switch their handlers without affecting our core code.

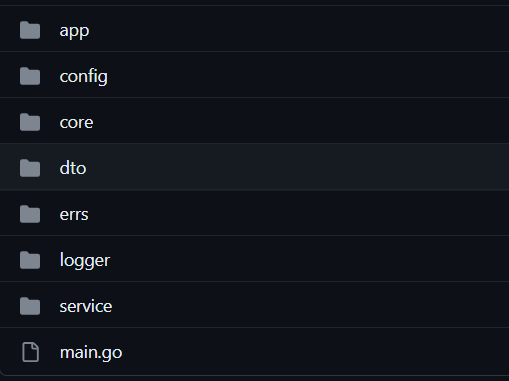


For more information about hexagonal design pattern, visit the link below:

* <https://www.mitrais.com/news-updates/hexagonal-architecture-design-pattern/>

Code Base Tutorial:

Before getting deep in code base and read about code behavior, let’s have a brief look at project structure:



Picture above is a snap shot from /src directory. I intent to give you simple explanation about each folder’s purpose.

* main.go

The entry point to application is this file. This file invokes the start function which is inside the app package and start the application on specified port.

* app

app package contains two main components of API. First, api handlers which are functions matched to their specified routes. Their purpose is parsing the incoming requests by passing them to their determined services. Second, start function which is responsible for initializing basic configs, injecting environment variables, connecting application to database service, setting routes, wiring up the service and repository or primary and secondary ports.

And at the end, making server alive by putting a request listener on specified port.

* config

This package maintains functions that are used by start function in order to check environment variables existence, initializing appConfig variable which is used by application in lots of parts, generating database source name and server base address.

* core

Core maintains secondary ports, secondary adapters and database entity schemas which are implemented using golanguage structs.

* dto

Dto package maintains data transferable object schemas which is implemented by golanguage structs to representing data that are fetched from database in user side.

* errs

Errs package is a library package that unify the error messages and customizing their behavior. This behavior are like NotFoundError, UnexpectedDatabaseError and…

* logger

Logger package is another library package which maintains logger behavior implemented by go-uber/zap third party package.

This package customizing common log messages such Error, Debug, Info and…

* service

service package maintains all primary ports and adapters. All the request validating is happens here before converting the dto to core object and pass them to repositories.

Technical Tutorial on API Behavior

After Start function get invoked by main function, variable appConf located in config package, get initiated and being ready to be used by exported getters implemented inside same package.

This variable contains value of injected environment variables like SERVER\_ADDRESS, SERVER\_PORT, DB\_USERNAME and…

During initiation, existence of this variables are checked properly and in case of their absence, program will panic.

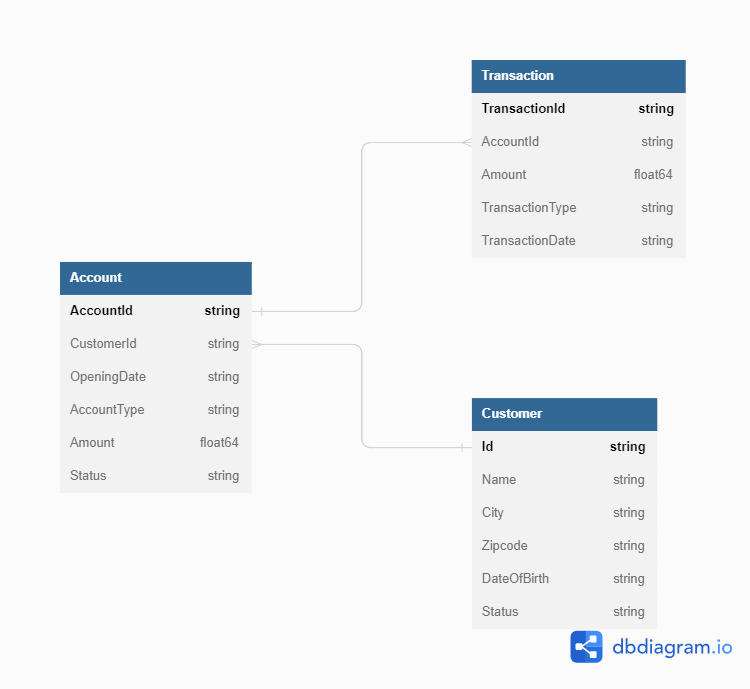
After this phase, program will make a database connection and make sure we are connected to database (using getNewPullConnection inside app package).

After that, program will make new router using gorillamux library for registering routes.

Before going further, let’s talk about entities and database tables. This version of program has three entities:

* Customer
* Account
* Transaction

Diagram below shows Schema of Banking Database precisely:



It is good to mention that no ORM is used inside this API and all of database stuff is handled by writing and shooting native SQL commands.

Corresponded structs of each entity is implemented in core directory, alongside secondary ports and adapters and help us to interact with database tables properly.

Let’s continue from where we stopped.

After establishing connection to database, it is time to connect our primary and secondary ports to each other. We do this by instantiating our secondary adapters (CustomerRepositoryDb and AccountRepositoryDb) and passing them into Primary adapters (DefaultCustomerService and DefaultAccountService) these adapters are connected to each other through their ports. ports are simple interfaces that contains set of rules for accepting adapters.

At the end, each service is passed to its corresponding request handler.

Here is an overview of the process that each request goes through to get to the core application to fetch the appropriate response:

Each request will be fetched by determined handler which is passed to specified route. Each handler will invoke proper function from its primary adapter which is a service embedded in handler struct. After validating the request and converting it to core object data, request is ready to be passed to secondary adapter throughout specified port.

Each secondary adapter has responsibility of getting the request, parsing it, and shoot the appropriate SQL command to database, fetching the result and send it back to sender service. Each service will send the incoming response back to handler and after handling error in each phase of this journey, handlers will write the response into w which is a ResponseWriter object and is a representation of the response that is going back to client.

Routes and Features:

* Resource: customer
* HTTP Method: POST
* Path: /customers
* Name: CreateCustomer

You can save new a customer to database by sending request to this path.

You have to provide data below in request body.

* name string
* date\_of\_birth string
* city string
* zipcode string
* status string

Expected response is Json formatted text with field customer\_id which is a string that contains newly inserted customer id.

* Resource: customer
* HTTP Method: GET
* Path: /customers
* Name: GetCustomers

Query parameters: status

Fetch all records of customer table. This request is filterable by status which can be “active” or “inactive”.

* Resource: customer
* HTTP Method: GET
* Path: /customers/:customer\_id
* Name: GetCustomerById

Select single record of customer table using customer\_id provided in path as path variable.

* Resource: account
* HTTP Method: POST
* Path: /customers/:customer\_id/accounts
* Name: CreateAccount

Saving new account for specific customer that its id is provided in path as path variable.

You have to provide data below inside request body

* CustomerId (use the path variable) string
* AccountType (“saving” or “checking”) string
* Amount (must be a positive value) float64

Response is a json formatted text with field account\_id which is the newly inserted account id.

* Resource: account
* HTTP Method: GET
* Path: /customers/:customer\_id/accounts/:account\_id
* Name: GetAccountById

Selecting specific account associated to specific customer using their Ids which are provided in path as path variables.

* Resource: transaction
* HTTP Method: POST
* Path: /customers/:customer\_id/accounts/:account\_id
* Name: NewTransaction

You can submit new transaction using this route. By submitting new transaction, affected account will be updated too.

In case of new transaction is “withdrawal” type, before submitting new transaction, associated account balance will be checked.

Transaction amount must be positive value.

You have to provide data below in request body in order to submitting transaction:

* account\_id (provided in path) string
* amount float64
* transaction\_type (“deposit” or “withdrawal”) string
* transaction\_date string
* customer\_id (provided in path) string
* Resource: transaction
* HTTP Method: GET
* Path: /customers/:customer\_id/accounts/:account\_id/transactions
* Name: NewTransaction
* Qurey params: transactionType

Fetch all the existing transactions associated to account. Result of this query is filterable by transactionType which can be “deposit” and “withdrawal”.