BAB 3
PELAKSANAAN KERJA MAGANG

3.1 Kedudukan dan Organisasi

Pada kegiatan magang sebagai *Backend developer intern* di PT Omni Digitama Internusa, pimpinan tertinggi di departemen *tech* dipegang oleh Bapak Ronny Winoto selaku Chief Technology Officer. Tim *Backend* PT Omni Digitama Internusa dipimpin oleh Bapak Rusmin Subiakto yang menjabat sebagai *Senior Backend Lead* dan berada di bawah kepemimpinan Bapak Ronny Winoto. Pada departemen *Backend* terdapat 6 *Backend Lead*, salah satunya adalah Bapak Joseph Kondar Halomoan yang memimpin tim Shoped (Shopee dan Tokopedia) dan sekaligus menjadi *supervisor* atau *mentor* selama periode magang berlangsung. Bapak Joseph Kondar Halomoan memimpin 4 orang *Backend Engineer* dan 1 *Backend developer intern*.

Selama periode magang, *intern* dipercayakan kepada salah satu *Backend Engineer* yaitu Bapak Ainur Riza Abdul Haris Ramadhan yang dipercayakan untuk membimbing *intern* namun tidak menutup kemungkinan untuk meminta bimbingan kepada Bapak Joseph Kondar Halomoan jika diperlukan.

Penugasan diberikan oleh Bapak Ainur setelah melewati diskusi dengan Bapak Joseph. Tugas diberikan melalui *backlog* pada *platform* Jira dan tanggung jawab penyelesaian tugas akan langsung diberikan kepada *intern*. Untuk memudahkan jalur komunikasi antar rekan tim, discord digunakan sebagai media utama untuk berkomunikasi antar rekan satu tim dan dengan tim lainnya seperti dengan tim *Quality Assurance* dan *Product Manager*.

3.2 Tugas yang Dilakukan

Selama enam bulan periode kerja magang, tugas yang diberikan adalah sebagai berikut.

1. Melakukan *refactor* kode yang menangani *error message*
2. Meningkatkan efisiensi waktu dari *service show log bulk status*
3. Meningkatkan efisiensi waktu dari *service dashboard special price*
4. Meningkatkan cakupan kode dalam *unit test* untuk repositori *wrapper*

5. Meningkatkan cakupan kode dalam *unit test* untuk repositori *product*
6. Meningkatkan cakupan kode dalam *unit test* untuk repositori *order*
7. Memindahkan fitur *resolve hold price* dari repositori *order* V1 ke V2

3.3 Uraian Pelaksanaan Magang

Uraian pelaksaan magang yang dilakukan dapat dilihat pada Tabel 3.1.

Tabel 3.1. Uraian pelaksanaan praktik kerja magang

Minggu Ke	Pekerjaan yang dilakukan
1	– *Instalasi* dan *setup development environment* – Perkenalan repositori yang digunakan – Pembelajaran mandiri terhadap konsep-konsep di *bahasa Go*
2	– Penugasan *refactor* kode yang menangani *error message* – Pembelajaran mandiri terhadap repositori *wrapper-vendor* – Melakukan *setup environment* seperti OS yang kompatibel dan VPN – Melakukan *self test* dan *debugging*
3	– Menerapkan *refactor* yang sesuai – Memperbaiki kode agar lebih bersih – Melakukan *deploy* ke *server staging* dan melakukan *testing*
Lanjut pada halaman berikutnya	

Tabel 3.1 (lanjutan)

Minggu Ke	Pekerjaan yang dilakukan
4	– Penugasan meningkatkan efisiensi waktu dari *service show log bulk status* – Mempelajari repositori untuk tugas baru – Melakukan *development* dan *debugging service show log bulk status*
5	– Penugasan meningkatkan efisiensi waktu dari *service special price* – Melakukan *development* dan *debugging* pada OMS dan repositori *product* – Melakukan *self test* dan *testing* dengan QA
6	– Penugasan meningkatkan cakupan kode dengan *unit test* pada repositori *wrapper, product* dan *order* – Pembelajaran mandiri terhadap repositori *wrapper* – Melakukan *development unit test*
7	– Menyelesaikan *development unit test* pada repositori *wrapper* – Melanjutkan *development unit test* pada repositori *product* – Mempelajari *business logic* pada repositori *product*

Lanjut pada halaman berikutnya

Tabel 3.1 (lanjutan)

Minggu Ke	Pekerjaan yang dilakukan
8	Melakukan *development* dan *debugging* pada *unit test* repositori *product*
9	– Menyelesaikan *development unit test* pada repositori *product* – Melanjutkan *development unit test* pada repositori *order* – Mempelajari *business logic* pada repositori *order*
10	Menyelesaikan *development* dan *debugging* pada *unit test* repositori *order*
11	Menyelesaikan *development* dan *debugging* pada *unit test* repositori *order*
12	Menyelesaikan *development* dan *debugging* pada *unit test* repositori *order*
13	Menyelesaikan *development* dan *debugging* pada *unit test* repositori *order*

Lanjut pada halaman berikutnya

Tabel 3.1 (lanjutan)

Minggu Ke	Pekerjaan yang dilakukan
14	Menyelesaikan *development* dan *debugging* pada *unit test* repositori *order*
15	– Penugasan memindahkan fitur *resolve hold price* dari repositori V1 ke V2 – Mempelajari *business logic* pada fitur *resolve hold price*
16	– Melakukan *development* – Melakukan *self test*
17	– Menyelesaikan penugasan dan melakukan *self test* – Melakukan *testing* dengan QA

3.3.1 Arsitektur Sistem Back-End dan Design Pattern

Design arsitektur yang digunakan pada sistem *Backend* adalah bersifat *microservices* dimana layanan dikelompokkan sesuai dengan tujuannya. Seperti layanan *order* akan dikelompokkan di repositori *order*, layanan lain seperti *re-order* tidak akan ada di repositori *order*. Bahasa yang digunakan adalah bahasa Go dengan *framework* Echo. *Database* yang digunakan ada 2 yaitu SQL dan NoSQL, *database* SQL menggunakan MySQL dan *database* NoSQL menggunakan MongoDB dan Elastic Search. MySQL digunakan untuk menyimpan data yang lebih statis seperti data *customer*, *order*, *shipment* dan lain-lain, MongoDB digunakan untuk menyimpan data yang lebih dinamis seperti harga produk, data *cart*, data *stock*, dan lain-lain. Elastic Search juga digunakan sebagai *database* namun sekaligus sebagai *search engine* bagi pelanggan agar dapat mengakses data lebih cepat.

Komunikasi antar layanan menggunakan 2 metode yaitu *Restful* API dan *Event Driven*. Komunikasi dengan *Restful* API adalah metode komunikasi secara sinkronus menggunakan HTTP *requests*.Sedangkan *Event Driven* adalah metode komunikasi secara asinkronus menggunakan NSQ dan Amazon SQS (*Simple Queueing System*). Selain itu terdapat sistem *batch* yang digunakan untuk *update* data secara serentak, seperti *update promo price*.

Design pattern yang digunakan adalah *layered architecture* atau bisa disebut dengan *n-tier architecture* [4]. Arsitektur ini memungkinkan untuk memisahkan beberapa *layer* yang memiliki fungsi berbeda. Terdapat 3 *layer* yang ada di sistem *Backend* Ruparupa yaitu:

- *Presentation Layer*, yaitu *layer* yang berfungsi untuk menerima data *request* dari HTTP *request*, memvalidasi data, melakukan pemanggilan servis disertai data *request* lalu mengirimkan *response* yang berisikan kode status, data dan pesan *error*.
- *Business Layer*, yaitu *layer* yang berfungsi sebagai *layer* yang menangani segala logika bisnis.
- *Repository/Database Layer*, yaitu *layer* yang berfungsi untuk formulasi dan eksekusi *query* seperti melakukan operasi CRUD (*Create, Read, Update, Delete*).

3.3.2 Melakukan *refactor* kode yang menangani *error message*

Salah satu faktor-faktor yang menjadi perhatian utama dalam sistem atau *software* yang dibangun adalah produktivitas, efisiensi dan kualitas. *Key Performance Index* atau KPI dapat menjadi salah satu indikator yang digunakan untuk mengawasi faktor-faktor tersebut. Pada sistem *Backend* Ruparupa, digunakan *tool* pengawasan KPI yang bernama New Relic, New Relic dapat memudahkan pengawasan secara *live* bagaimana keadaan dan *traffic* yang sedang terjadi di sistem. Kode status pada HTTP/HTTPS *response* menjadi salah satu indikator seberapa produktif sebuah sistem. Oleh karena itu, diberikanlah penugasan untuk menyesuaikan kode status untuk HTTP/HTTPS *response* yang ada di sistem. Sebelumnya kode status untuk HTTP/HTTPS *response* belum terlalu diperhatikan, kebanyakan kode status yang diberikan adalah 500 (*Internal Server Error*) yang berarti *server* menemukan suatu kondisi yang mengakibatkan ketidakmampuan

untuk memberikan *response* yang memuaskan kepada *request* yang diminta. Kode status 500 adalah salah satu indikator bahwa sistem kita tidaklah produktif, hal ini yang dapat menyebabkan KPI suatu sistem menurun. Contoh kasus, pada sistem *internal* ruparupa, terdapat fungsi yang bernama SyncStockShopee yang berfungsi untuk melakukan sinkronisasi *stock* produk yang dijual di *vendor* Shopee. Dalam fungsi tersebut ada proses yang melakukan *request* API yang berisikan *request* untuk mengambil *availability stock* suatu produk kepada *server* Shopee. Adanya suatu kondisi tertentu pada *server internal* Shopee, mengakibatkan sistem *internal* Shopee mengirim response dengan *message* "*failed get availability stock*". Terdapat potongan kode bertanggung jawab untuk mengeset kode *status* apa yang harus diberi berdasarkan *response* dari sistem *internal* Shopee. Dalam kode tersebut *message* "*failed get availability stock*" tidak diperhitungkan yang menyebabkan masuk ke kondisional *else* yang mengeset kode *status* 500. Hal ini menyebabkan menurunnya KPI sistem *internal* ruparupa, padahal kesalahan tersebut bukanlah dari pihak sistem *internal* ruparupa melainkan dari sistem *internal* Shopee. Penugasan ini ditujukan untuk minimalisir kejadian tersebut dengan membuat kondisional *message response* dari pihak Shopee dan Tokopedia dapat mencakup *message-message error* yang lebih beragam.

```
if len(errs) > 0 {
  logger.LogError(
    fmt.Errorf("failed sync stock shopee, error : %v", errs),
    ctx.Value(constants.REQUEST_ID).(string),
    nil,
  )

  var statusCode int
  for _, v := range errs {
    if v.Message == "failed to get product shopee" {
      statusCode = http.StatusNotFound
    } else {
      statusCode = http.StatusInternalServerError
    }
  }

  return dtos.Response{
    Code:    statusCode,
    Message: "failed sync stock shopee",
    Error:   errs,
    Data:    nil,
```

```
  }
}
```

Kode 3.1: Potongan kode sebelum dilakukan optimisasi

Kode 3.1 adalah potongan kode kondisional pada fungsi SyncStockShopee yang bertugas untuk mengset kode *status* apa yang harus dikirim berdasarkan *error message*. Pertama program akan cek apakah *array* errs kosong, jika tidak kosong maka akan dikirim *error log*. Lalu *array* errs diiterasi untuk melihat apakah ada *error message* yang berisikan kalimat "*failed to get product shopee*", jika ada maka statusCode diset kode 404, jika tidak maka statusCode diset 500. Setelah itu akan dikirim *response* yang berisi *Code, Message, Error* dan Data. Dapat dilihat bahwa minimnya *error message* yang dapat dicakup. Oleh karena itu, dilakukan *refactor* kode yang dapat mencakup *error message* yang lebih beragam.

```
if len(errs) > 0 {
  logger.LogError(
    fmt.Errorf("failed sync stock shopee, error : %v", errs),
    ctx.Value(constants.REQUEST_ID).(string),
    nil,
  )

  var statusCode int
  for _, v := range errs {
    statusCode = helpers.ResponseStatusCode(v.Error, constants.
    Shopee)
  }

  return dtos.Response{
    Code:    statusCode,
    Message: "failed sync stock shopee",
    Error:   errs,
    Data:    nil,
  }
}
```

Kode 3.2: Potongan kode sesudah dilakukan optimisasi


```
var statusCodeMap = map[string]map[string]int{
  constants.Shopee: {
    "failed to get product shopee":           http.StatusNotFound,
    "Item_id is not found":                   http.StatusNotFound,
    "Update stock failed, please try later.": http.
    StatusUnprocessableEntity,
```

```
    "status is abnormal":                          http.
    StatusUnprocessableEntity,
    "failed get availability stock":                http.
    StatusUnprocessableEntity,
    "error_internal":                               http.
    StatusUnprocessableEntity,
    "(reserve stock number)":                       http.
    StatusUnprocessableEntity,
    "failed to get access token":                   http.
    StatusUnprocessableEntity,
  },
  constants.Tokopedia: {
    "Update Tokopedia Stock Failed": http.StatusForbidden,
    "Too Many Request":              http.StatusTooManyRequests,
    "ErrRateLimitExceeded":          http.StatusTooManyRequests,
    "failed get availability stock": http.
    StatusUnprocessableEntity,
    "Warehouse Data Not Found":      http.
    StatusUnprocessableEntity,
  },
}

func ResponseStatusCode(err string, vendor string) int {
  if vendorMap, ok := statusCodeMap[vendor]; ok {
    for message, code := range vendorMap {
      if strings.Contains(err, message) {
        return code
      }
    }
  }

  return http.StatusInternalServerError
}
```

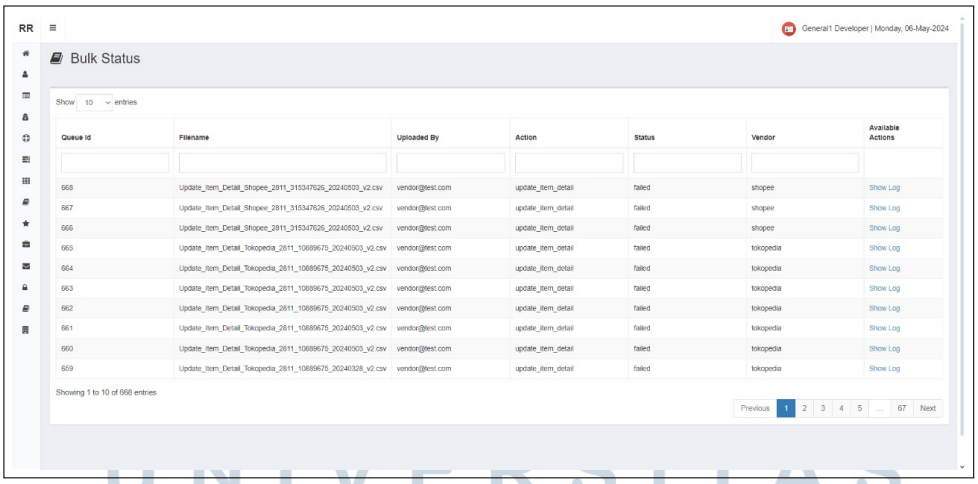
Kode 3.3: Potongan kode pada *package helpers*

Kode 3.2 adalah potongan kode kondisional baru yang telah diimplementasikan untuk menentukan kode *status* dipanggil fungsi yang bernama ResponseStatusCode yang ada di *package helpers*. Fungsi ResponseStatusCode yang ada di *package helpers* dapat dilihat pada Kode 3.3. Di fungsi tersebut akan dilakukan pengecekan apakah nama *vendor* ada di *map* statusCodeMap atau tidak, jika ada maka akan dilakukan komparasi err *message* dari Shopee dengan *string error message* yang ada di *map*. Ketika ditemukan kecocokan maka diambil kode

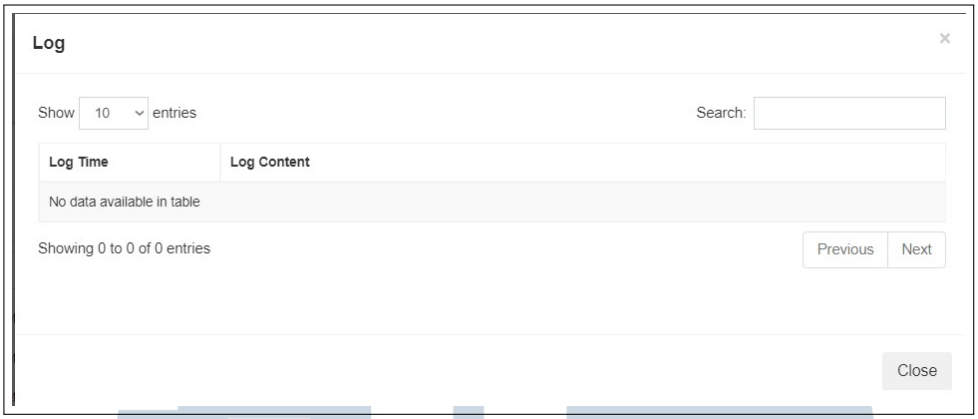
status yang sudah ditentukan pada *map*. Namun jika tidak ditemukan kecocokan maka kode *status* yang akan di *set* adalah StatusInternalServerError atau 500.

3.3.3 Meningkatkan Efisiensi Waktu dari *Service Show Log Bulk Status*

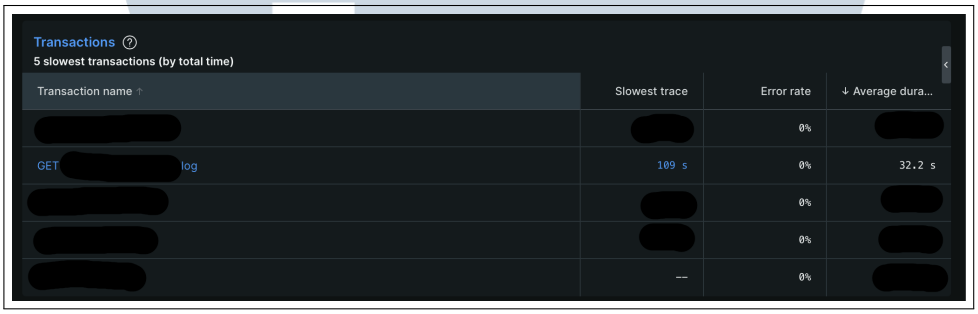
Pada *website internal* ruparupa terdapat *menu bulk status* seperti Gambar 3.1 yang berisikan *status* semua produk ketika dilakukan proses *add, update* dan *delete* ke pihak *vendor* dengan csv. Terdapat *menu show log* yang menampilkan *detail* dari proses CRUD terhadap barang di *vendor* seperti Gambar 3.2. Di dalam menu tersebut akan ditampilkan *log time* dan *log content*, *log content* memuat informasi *detail* tentang *status* CRUD beserta informasi tentang produknya. Pada mulanya, *menu show log* mengambil semua data tentang log dari proses CRUD pada *bulk update* tersebut. Banyaknya *log* yang ada, menyebabkan *response time* pada *service* tersebut mencapai lebih dari 60 detik (Gambar 3.3) dimana *response time* tersebut sangatlah lambat dan tidak produktif. Oleh karena itu, tugas yang diberikan adalah mengurangi *response time* dari *service* tersebut dengan mengimplementasikan *pagination*, melakukan *indexing* pada tabel log, membuat fitur *search* untuk kolom *log content*.



Gambar 3.1. Menu Bulk Status
Sumber: *Internal* ruparupa *Website*

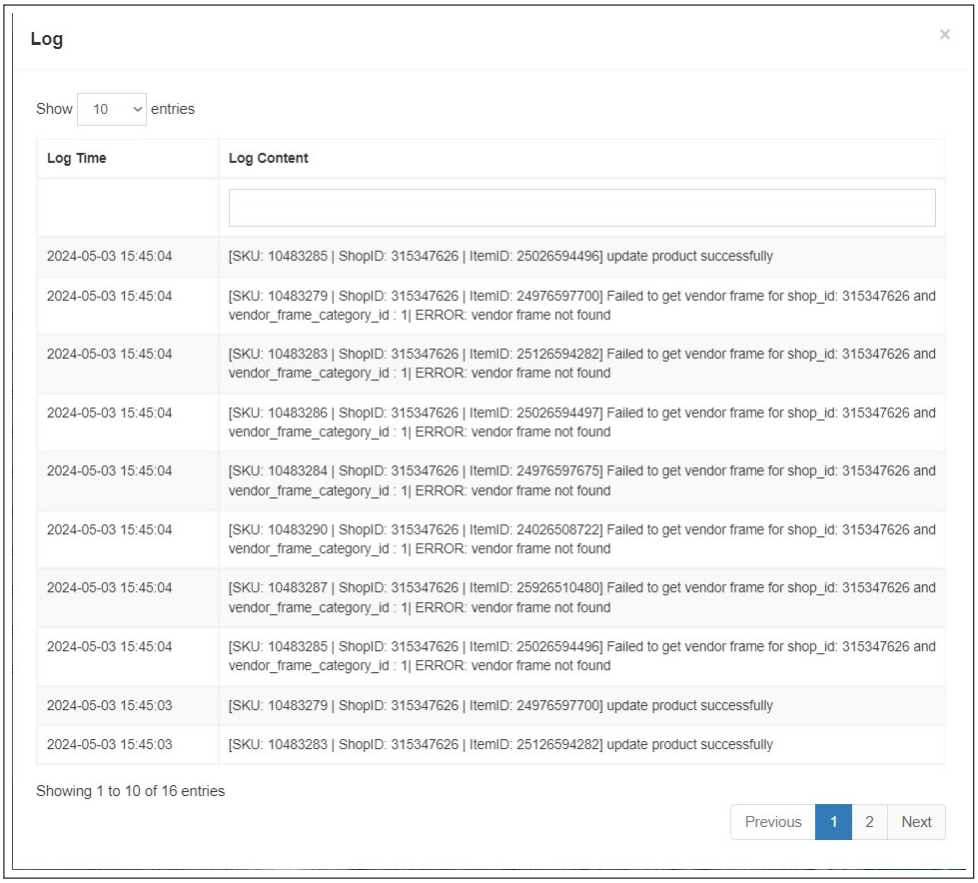


Gambar 3.2. Menu Show Log Bulk Status
Sumber: *Internal* ruparupa *Website*



Gambar 3.3. *Internal Analytics*
Sumber: *Internal* ruparupa *Website*

Pagination dilakukan dengan mengubah *format table*, menambahkan beberapa atribut, *parameter* pada repositori OMS, lalu menyesuaikan sistem *back-end* pada repositori order-v1 dengan menambahkan beberapa *parameter*, variabel dan menyesuaikan *query* MySQL untuk mengambil data di *database*

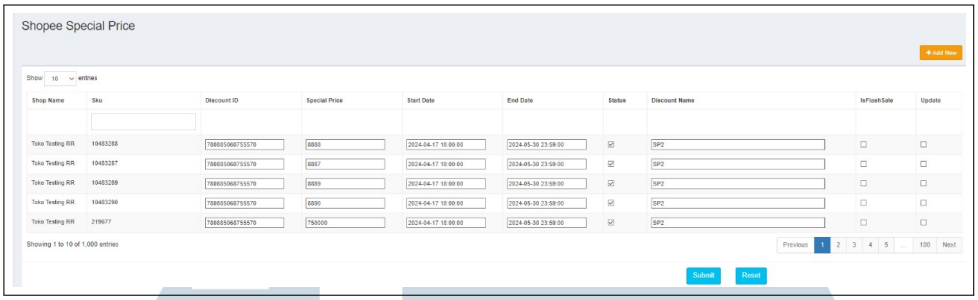


Gambar 3.4. Menu Show Log Bulk Status
Sumber: *Internal* ruparupa *Website*

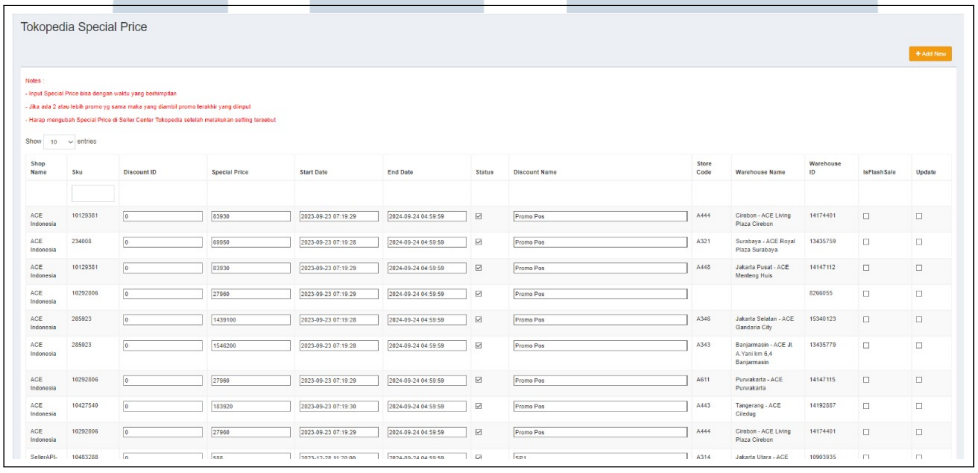
Setelah dilakukan optimisasi, pengambilan *log* sudah dilimitasi sebanyak 10 data jika ada lebih dari 10 data maka pengguna harus menekan *page* selanjutnya. Setiap *page* akan melakukan *request* API untuk mengambil data dengan *limit* dan *offset* yang sudah ditentukan.

3.3.4 Meningkatkan Efisiensi Waktu dari *Service dashboard special price*

Pada *website internal* ruparupa terdapat 2 *menu dashboard special price* untuk *vendor* Shopee dan Tokopedia seperti Gambar 3.5 dan Gambar 3.6 yang berisikan *special price* yang akan diterapkan di *vendor*. Mulanya metode pengambilan data tidak dilakukan *pagination* yang menyebabkan *response time* yang lama diakibatkan banyakanya *special price* yang diterapkan. Banyaknya toko juga menyebabkan sulitnya mengatur *special price*. Oleh karena itu dilakukan *pagination* dan fitur *filter by shop*.

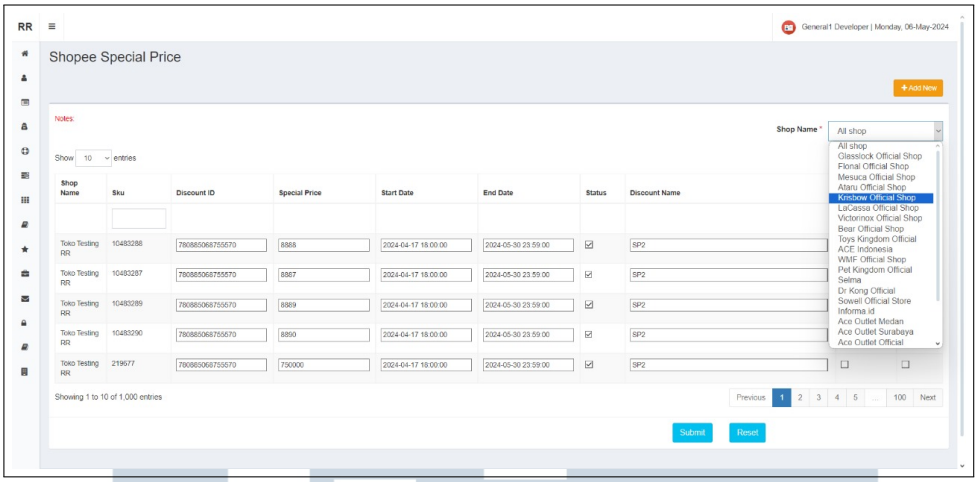


Gambar 3.5. Menu Dashboard Special Price Shopee sebelum dilakukan optimisasi
Sumber: *Internal* ruparupa *Website*

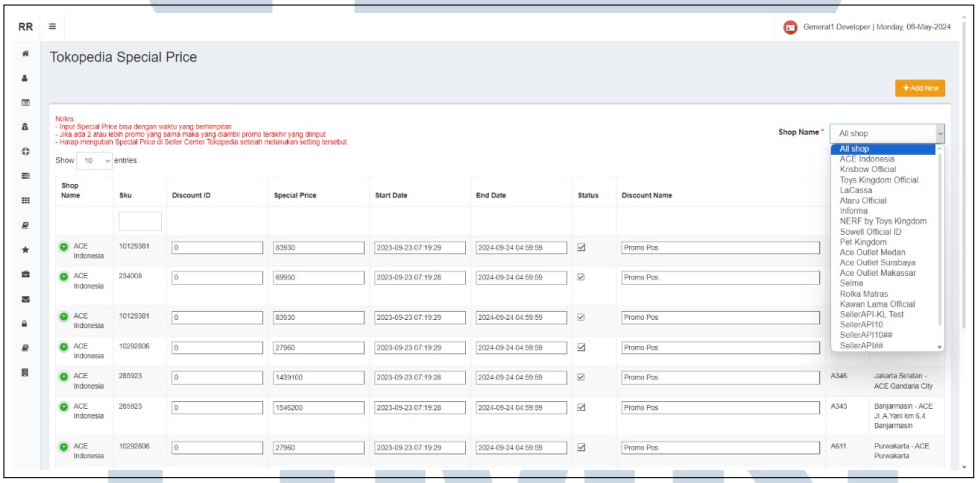


Gambar 3.6. Menu Dashboard Special Price Tokopedia sebelum dilakukan optimisasi
Sumber: *Internal* ruparupa *Website*

Pagination dilakukan dengan menambahkan beberapa atribut, *parameter*, membuat pada repositori OMS, lalu menyesuaikan sistem *back-end* pada repositori order-v1 dengan menambahkan beberapa *parameter*, variabel dan menyesuaikan *query* MySQL untuk mengambil data di *database*.



Gambar 3.7. Menu Dashboard Special Price Shopee sesudah dilakukan optimisasi
Sumber: *Internal* ruparupa *Website*



Gambar 3.8. Menu Dashboard Special Price Tokopedia sesudah dilakukan optimisasi
Sumber: *Internal* ruparupa *Website*

Setelah dilakukan *pagination*, setiap *page* dibatasi pengambilan data sebanyak 10 buah dan dengan fitur *sort by shop* dengan *default all store*, hal ini memungkin user untuk melihat *special price* yang hanya ada di *store* yg diinginkan, hasil dapat dilihat pada Gambar 3.7 dan Gambar 3.8.

3.3.5 Meningkatan Cakupan Kode dalam *Unit Test*

Unit test adalah praktik dimana *developer* harus melakukan *test* pada kode yang ditulis sampai pada hal terkecil (*unit*). Hal ini diharapkan *software* yang dibangun berkualitas dengan meminimalisir adanya *bug* atau *error* dan menjadi

dokumentasi bagi *developer* yang ingin melakukan modifikasi atau refaktorisasi. Refaktorisasi adalah proses merubah kode dari suatu fungsi atau layanan tanpa mengubah tujuan dari fungsi atau layanan tersebut. Untuk melakukan *unit test* maka dibutuhkan *framework*. *Framework* GoMock dipakai untuk *unit test* sistem-sistem yang menggunakan bahasa Go dan *library* doenv yang dibuat oleh Brandon Keepers dan John Barton. *Library* doenv digunakan untuk melihat *total coverage* dari *unit test* yang telah dibuat. Sistem yang dilakukan *unit test* meliputi repositori *wrapper, product dan order*. Proses *unit test* dimulai dengan melakukan *mocking* semua *package* repositori, *service* dan *adapter* dengan *tool mockgen* dari *framework* GoMock. Lalu mulai menguji semua skenario yang dapat terjadi di fungsi atau *unit* pada kode.

```
t.Run("success resolve hold price", func(t *testing.T) {
  mockProductVendorAdapter := mocks.NewMockGoProductVendorAdapter(
    ctrl)
  mockNSQadapter := mocks.NewMockNSQAdapter(ctrl)
  mockGoAuthVendorAdapter := mocks.NewMockGoAuthVendorAdapter(ctrl
    )
  mockGoWrapperVendorAdapter := mocks.
    NewMockShopeeGoWrapperVendorV2Adapter(ctrl)

  mockSalesOrderVendorRepository := mocks.
    NewMockSalesOrderVendorRepository(ctrl)
  mockCartVendorRepository := mocks.NewMockCartVendorRepository(
    ctrl)
  mockSalesOrderVendorItemRepository := mocks.
    NewMockSalesOrderVendorItemRepository(ctrl)

  mockProductPriceService := mocks.NewMockProductPriceService(ctrl
    )
  mockCartService := mocks.NewMockCartVendorService(ctrl)

  mockSalesOrderVendorRepository.EXPECT().GetHoldOrder(
    gomock.Any(),
    constants.SHOPEE,
  ).Return(
    []models.SalesOrderVendor{
      {
        OrderNoVendor: "ORDERSN",
        ShopID:        1,
      },
    },
```

```
    nil,
  )

  mockCartVendorRepository.EXPECT().GetCartVendorByOrderVendorNo(
    gomock.Any(),
    gomock.Any(),
    "ORDERSN",
  ).Return(
    nil,
  ).SetArg(
    1,
    models.Cart{
      OrderVendorNo: "ORDERSN",
      Items: []models.Item{
        {
          IsRestrictedEvent: true,
          Sku:               "SKU",
          Price: models.Prices{
            NormalPrice:  10000,
            SellingPrice: 5000,
          },
        },
      },
    },
  )

  mockSalesOrderVendorItemRepository.EXPECT().
   GetSalesOrderItemPromotionType(
    gomock.Any(),
    "ORDERSN",
    "SKU",
  ).Return(
    "promotion",
    nil,
  )

  mockSalesOrderVendorItemRepository.EXPECT().Get(
    gomock.Any(),
  ).Return([]models.SalesOrderVendorItem{}, nil)

  mockProductVendorAdapter.EXPECT().GetProductShopee(
    gomock.Any(),
    gomock.Any(),
```

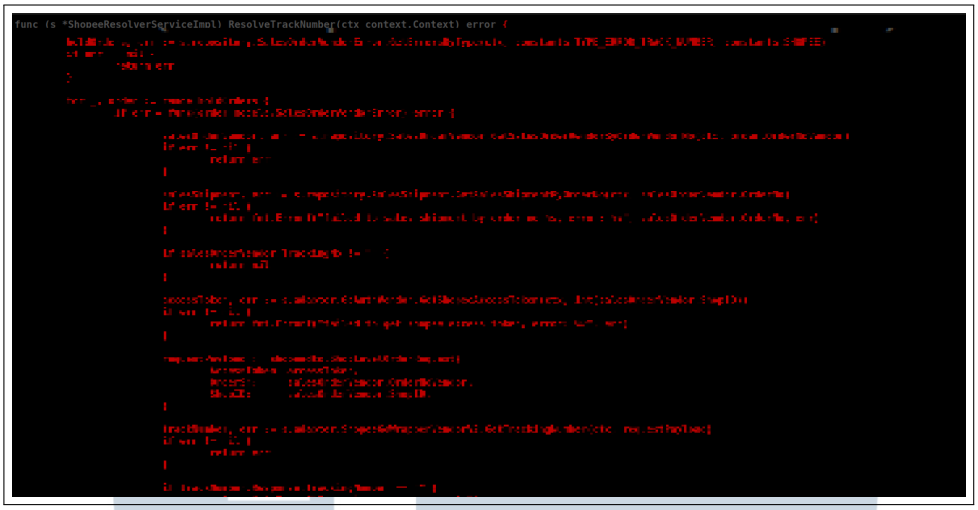
```
    gomock.Any(),
    gomock.Any(),
).Return(models.MongoProductShopee{}, nil)

mockProductPriceService.EXPECT().GetAppliedPriceShopee(
    gomock.Any(),
    gomock.Any(),
    gomock.Any(),
    gomock.Any(),
    gomock.Any(),
    gomock.Any(),
).Return(
    models.Prices{
        NormalPrice:  10000,
        SellingPrice: 4000,
    },
    "",
    nil,
)

mockCartService.EXPECT().AssignSubtotalItemAndCalculateSubTotal(
    gomock.Any(),
).Return(10000.0)

mockCartVendorRepository.EXPECT().UpdateCartVendor(
    gomock.Any(),
    gomock.Any(),
).Return(nil)

mockNSQadapter.EXPECT().SendCreateOrder(
    gomock.Any(),
    gomock.Any(),
).Return(nil)

mockGoAuthVendorAdapter.EXPECT().GetShopeeAccessToken(gomock.Any
  (), gomock.Any()).Return("access_token", nil)

mockGoWrapperVendorAdapter.EXPECT().GetOrderDetail(gomock.Any(),
   gomock.Any(), gomock.Any(), gomock.Any()).Return(&shopeedto.
  Order{}, nil)

mockCartService.EXPECT().AssignPaymentShopee(gomock.Any(),
  gomock.Any())
```

```
105
106 mockSalesOrderVendorRepository.EXPECT().UpdateSalesOrderVendor(
107   gomock.Any(),
108   "ORDERSN",
109   gomock.Any(),
110 ).Return(nil)
111
112 builder, _ := di.NewBuilder()
113 builder.Add(di.Def{
114   Name: constants.Service,
115   Build: func(ctn di.Container) (interface{}, error) {
116     return &Service{
117       CartVendor:    mockCartService,
118       ProductPrice: mockProductPriceService,
119     }, nil
120   },
121 })
122 mockIoc := builder.Build()
123
124 type fields struct {
125   repository *repositories.Repository
126   adapter    *adapters.Adapter
127   service    *BaseService
128 }
129 type args struct {
130   ctx context.Context
131 }
132 test := struct {
133   fields  fields
134   args    args
135   wantErr bool
136 }{
137   fields: fields{
138     repository: &repositories.Repository{
139       SalesOrderVendor:     mockSalesOrderVendorRepository,
140       SalesOrderVendorItem: mockSalesOrderVendorItemRepository,
141       CartVendor:           mockCartVendorRepository,
142     },
143     adapter: &adapters.Adapter{
144       NSQ:                      mockNSQadapter,
145       GoProductVendor:          mockProductVendorAdapter,
146       GoAuthVendor:             mockGoAuthVendorAdapter,
147       ShopeeGoWrapperVendorV2: mockGoWrapperVendorAdapter,
```

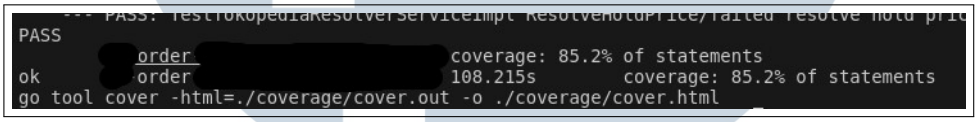
```
148         },
149         service: NewBaseService(mockIoc),
150       },
151       args: args{
152         ctx: ctx,
153       },
154       wantErr: false,
155     }
156     s := &ShopeeResolverServiceImpl{
157       repository: test.fields.repository,
158       adapter:    test.fields.adapter,
159       service:    test.fields.service,
160     }
161     err := s.ResolveHoldPrice(test.args.ctx)
162     time.Sleep(time.Second * 1)
163     if (err != nil) != test.wantErr {
164       t.Errorf("ShopeeResolverServiceImpl.ResolveHoldPrice() error =
        %v, wantErr %v", err, test.wantErr)
165     }
166 })
```

Kode 3.4: Potongan kode unit test

Gambar Kode 3.4 merupakan salah satu *unit test* yang dilakukan pada fungsi ResolveHoldPrice. Skenario yang dilakukan pada *unit test* tersebut adalah sukses melakukan *resolve order* yang terkena *hold order*. Untuk membuat *unit test* pertama deklarasi semua *mock adapter*, repositori dan *service* yang akan dipakai di fungsi ResolveHoldPrice. Lalu buatlah ekspektasi paramater dan response dari semua fungsi yang ada di fungsi ResolveHoldPrice. Jika terdapat *service*, lakukan *depedency injection* untuk semua *service* yang akan dipakai di fungsi ResolveHoldPrice. Fungsi *depedency injection* adalah untuk memasukkan *dependency* atau *class ke suatu fungsi*. Lalu deklarasikan semua parameter yang akan dimasukkan ke fungsi ResolveHoldPrice serta semua *adapter*, repositori dan *service*. Lalu lakukan pengecekan, apakah dengan parameter yang dikirim menghasilkan hasil dengan ekspektasi yang sama.

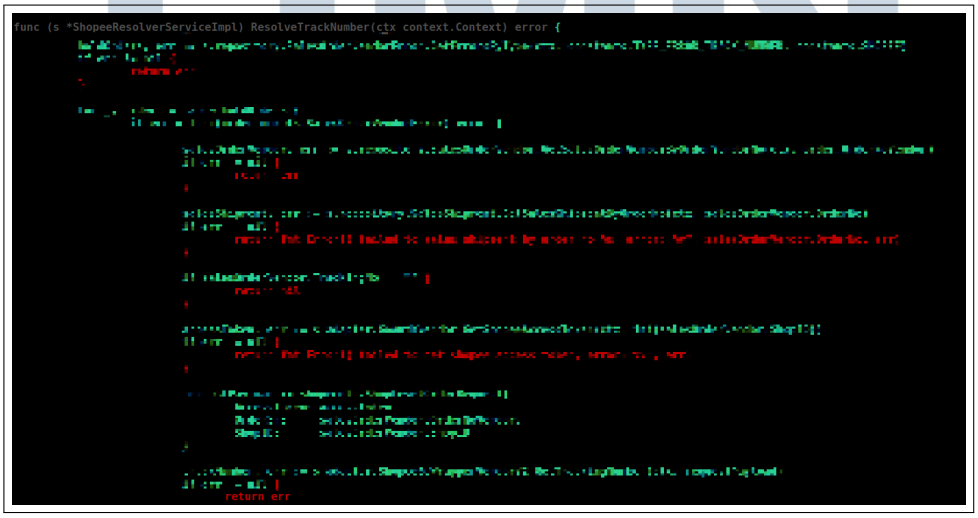


Gambar 3.9. Coverage unit test fungsi ResolveHoldPrice sebelum *unit test* dibuat
Sumber: Dokumentasi pribadi



Gambar 3.10. Presentase coverage di repositori order sebelum *unit test* dibuat
Sumber: Dokumentasi pribadi

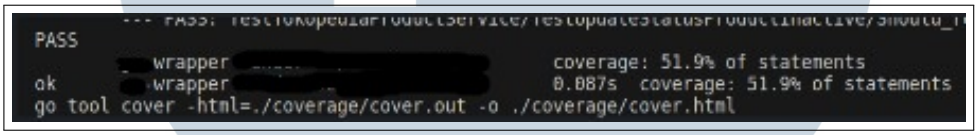
Gambar 3.9 adalah *coverage* dari fungsi ResolveHoldPrice. Berdasarkan gambar tersebut terdapat barisan kode yang ditandai merah, tanda merah berarti barisan kode tersebut belum di *test*. Gambar 3.10 adalah presentase cakupan kode sebelum dilakukan *unit test* pada fungsi ResolveHoldPrice.



Gambar 3.11. Coverage unit test fungsi ResolveHoldPrice sesudah *unit test* dibuat
Sumber: Dokumentasi pribadi

Gambar 3.12. Presentase coverage di repositori order sesudah *unit test* dibuat
Sumber: Dokumentasi pribadi

Gambar 3.11 adalah *coverage* dari fungsi ResolveHoldPrice setelah *unit test* diimplementasikan. Berdasarkan gambar tersebut terdapat barisan kode yang ditandai warna hijau, tanda hijau berarti barisan kode tersebut telah dilakukan *testing*. Gambar 3.10 adalah presentase cakupan kode yang telah dilakukan setelah *unit test* pada fungsi ResolveHoldPrice. Dapat dilihat bahwa terdapat kenaikan presentase cakupan kode dari 85.2% menjadi 85.9%.



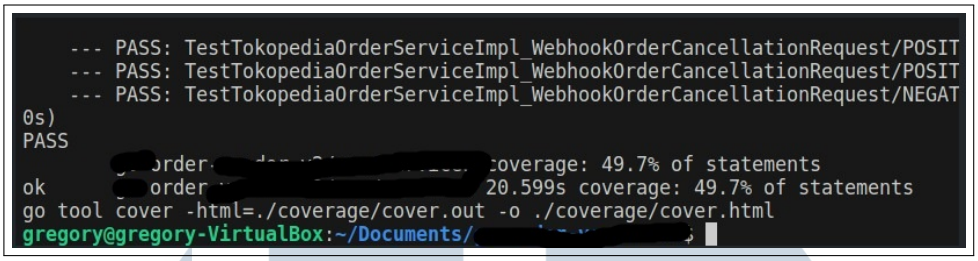
Gambar 3.13. Presentase coverage di repositori wrapper sebelum *unit test* dibuat
Sumber: Dokumentasi pribadi

Gambar 3.14. Presentase coverage di repositori wrapper sebelum sesudah *unit test* dibuat
Sumber: Dokumentasi pribadi

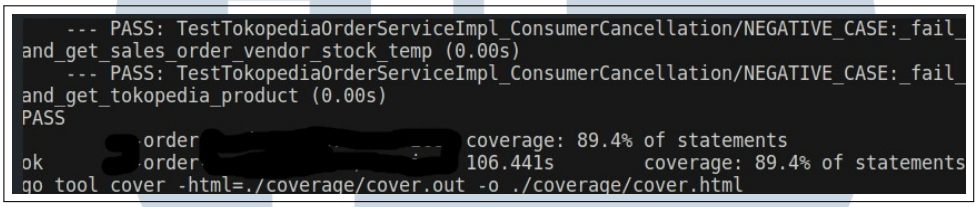
Gambar 3.13 dan Gambar 3.14 menunjukkan presentase coverage dari repositori wrapper sebelum dan sesudah penugasan diselesaikan. Presentase coverage repositori wrapper meningkat sebanyak 25,5%, dari 51,9% menjadi 77,4%.

Gambar 3.15. Presentase coverage di repositori product sebelum sesudah *unit test* dibuat
Sumber: Dokumentasi pribadi

Gambar 3.15 menunjukkan presentase coverage dari repositori product sesudah penugasan diselesaikan. Presentase coverage repositori product meningkat menjadi 87,4%.



Gambar 3.16. Presentase coverage di repositori order sebelum *unit test* dibuat
Sumber: Dokumentasi pribadi



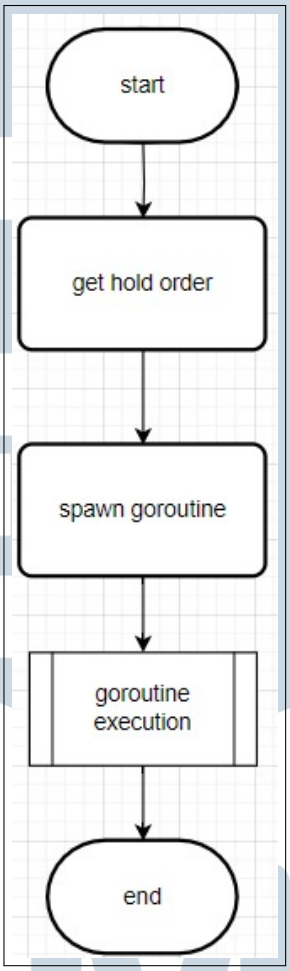
Gambar 3.17. Presentase coverage di repositori order sebelum sesudah *unit test* dibuat
Sumber: Dokumentasi pribadi

Gambar 3.16 dan Gambar 3.17 menunjukkan presentase coverage dari repositori order sebelum dan sesudah penugasan diselesaikan. Presentase coverage repositori order meningkat sebanyak 39,7%, dari 49,7% menjadi 89,4%.

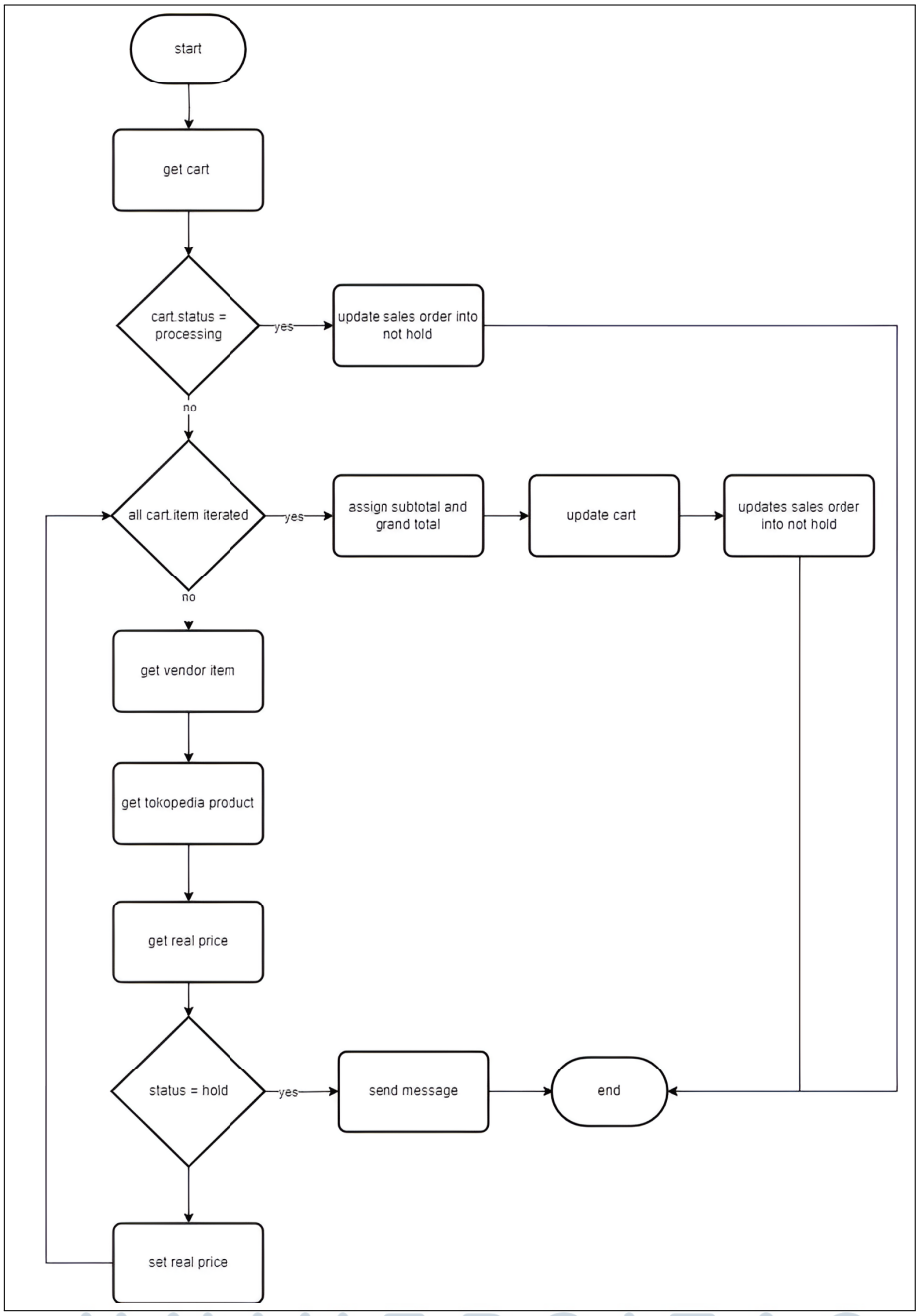
3.3.6 Memindahkan Fitur *Resolve Hold Price Tokopedia* dari Repositori Order V1 ke V2

Fitur *resolve hold price* Tokopedia adalah fitur yang berguna untuk melepaskan *order* pada *vendor* tokopedia yang sedang terkena *hold*. *Hold order* dapat terjadi diakibatkan oleh perbedaan harga yang sedang dijual dan harga yang seharusnya dijual. Perbedaan harga ini dapat diatasi dengan melakukan penyesuaian harga secara manual pada *website internal*. Setelah dilakukan penyesuaian akan dilakukan *request* API yang menjalankan fungsi ResolveHoldPrice. Tugas pemindahan fungsi ResolveHoldPrice dari repositori order v1 ke v2 dilakukan karena ada beberapa alasan. Pertama, repositori order V1 sudah tidak dipelihara sehingga pemindahan fungsi tersebut ke repositori order V2 diharapkan dapat mempermudah *developer* untuk melakukan pemeliharaan kode. Kedua, repositori order V1 telah menerapkan *design pattern layered architecture* yang lebih bersih, pembagian *package* yang lebih terstruktur dan penggunaan *library* yang lebih *modern* dan tidak rumit. Ketiga, terdapat banyak *error* yang terjadi di fungsi

tersebut sehingga pemindahan diharapkan dapat mengurangi *error* yang akan terjadi.



Gambar 3.18. Flowchart fungsi



Gambar 3.19. Flowchart goroutine execution

Gambar 3.18 dan Gambar 3.19 adalah flowchart dari fungsi ResolveHoldPrice. Fungsi dimulai dengan mengambil semua order yang terkena *hold* lalu program memulai beberapa goroutine dan *hold order* akan dikonsumsi oleh goroutine yang telah terbentuk. Pada goroutine akan dilakukan pengambilan *cart* lalu *status cart* akan dicek, jika *status cart* berada di keadaan "*processing*" maka *order* akan dilakukan *update* bahwa *order* tidak terkena *hold* dan proses akan berhenti, jika tidak maka akan lanjut ke proses berikutnya. Proses dilanjutkan ke tahap pengiterasian semua *item* dalam *cart* yang bertujuan untuk mengecek apakah harga yang dijual sesuai dengan harga yang seharusnya. Jika ada harga *item* yang tidak sesuai/seharusnya maka akan dikirim pesan *error* dan proses akan berhenti, jika tidak ada harga *item* yang tidak wajar maka akan dilakukan penghitungan *sub total* dan *grand total*. Setelah itu *order* akan dilakukan *update* menjadi tidak terkena *hold*.

3.4 Kendala dan Solusi yang Ditemukan

Kendala-kendala yang ditemukan selama pelaksanaan magang di PT Omni Digitama Internusa, antara lain:

1. Minimnya dokumentasi pada repositori sebuah layanan dan penggunaan docker untuk mengeksekusi layanan di komputer lokal menyebabkan dibutuhkannya waktu ekstra untuk memahami maksud dari kode dan menjalankan layanan di komputer lokal.
2. Sebagian besar repositori sebuah layanan hanya bisa berjalan di sistem operasi linux khususnya Ubuntu yang menyebabkan kesulitan pada saat pertama kali ingin mempelajari sebuah layanan.

Upaya-upaya yang telah dilakukan dalam mengatasi kendala-kendala yang ditemukan adalah:

1. Mempelajari kode dan cara menjalankan layanan di komputer lokal secara mandiri dan dengan meminta bantu rekan kerja yang sudah berpengalaman.
2. Mengunduh VirtualBox untuk menjalankan *virtual machine* dengan sistem operasi Ubuntu