Arex APP improvements

By Joan J. Iglesias Blanch

Improvements

1. Developed two utilities to improve the Golang capabilities:
   a. A king of Exceptions for Go
   b. A Macro expansion system for Go
2. The use of an ORM (Gorm) in the project to minimize the code relative to some operations done with Postgresql
3. Used in the previous release, but I would like to mention:
   a. Domain and Type Postgresql objects to enforce data integrity in the backend.
   b. Postgresql Functions (add_bid) for complex SQL operations.
4. “Make” is used to help in the development process, especially for code generation

Files nomenclature

If a file needs Exceptions must end with “__.go” to be processed for the “error_track” utility.

If a file needs macro expansions must end with “_.go” to be processed for the “macro_expansion” utility.

The final file will be always a “.go” file with all the code added in the previous steps.

Exceptions example

```
130     var r *gorm.DB
131
132     ErrorTrack :
133     if r != nil {
134         panic (r)
135     }
136     ErrorTrack_err :
137     if err != nil {
138         panic (err)
139     }
140
141     if initTypes {
142         r = db.Exec  (string(typesDomains) )
143         r = db.Exec  (string(add_bid     ) )
144     }
145     // migrations:
146     err = db.AutoMigrate(&models.Client{}   )
147     err = db.AutoMigrate(&models.Invoice{}  )
148     err = db.AutoMigrate(&models.SellOrder{})
149     err = db.AutoMigrate(&models.Ledger{}   )
150
151     // END   - database objects initialization
```

Exceptions example explanation

At the top of the code there are a couple of levels. There levels and their corresponding “if” statements are only entered if one of the lines executed below generate any kind of error. The “error_track” utility automatically generate “if” statements below each executed line (if detects some kind of regular expressions) and the programmer **do not need to take care of these repetitive tasks**.

Macro expansion example

```
73 // macro expansion : the function is exapanded as another function in the .go files
74 //<<func (s *server) ListBids       (in *pb.Empty,stream pb.ArexServices_ListBidsServer     ) error {>>
75 //<<func (s *server) ListInvoices   (in *pb.Empty,stream pb.ArexServices_ListInvoicesServer ) error {>>
76 func (s *server) ListSellOrders (in *pb.Empty,stream pb.ArexServices_ListSellOrdersServer   ) error {
77 //<<var toRet []models.Ledger>>
78 //<<var toRet []models.Invoice>>
79     var toRet []models.SellOrder
80     res:= s.db.Order("id").Find(&toRet)
81     if res.Error != nil {
82         return res.Error
83     }
84     for _,e:=range toRet {
85         stream.Send(e.CastgRPC())
86     }
87     return nil
88 }
89 //<<end>>
```

Macro expansion example explanation

The macro_expansion utility helps to improve the amount of code to implement repetitive functionalities where the programmer cannot make efficient abstractions, because the code to make this abstraction will be very dully or inefficient.

In this case, it’s more efficient to do it with the macro expansions utility. In this case the utility detect reg expressions “//<<” and “>>” and duplicate the functionality with the expansion of the function or method code.

In the example the ListSellOrders method is used to generate two more methods: ListBids and ListInvoices, with just a **couple of lines of code** for each method.

Postgresql domains and types usage

```
 1 -- domains
 2 create domain amount_type_calc as
 3     numeric(11,2)   check (value >= 0)             ;   -- max amount : 999999999.99 and bigger than -1
 4 create domain amount_type as
 5     amount_type_calc not null                      ;   -- max amount : 999999999.99 and bigger than -1
 6 create domain discount_type as
 7     numeric(5,2)    check (value <=100.00 and value >=0.00);
 8
 9 -- types
10 create type sell_order_state        as enum ('ongoing','reversed','locked','committed');
11 create type invoice_state           as enum ('financing search','rejected','financed');
```

Postgresql domains and types usage

The usage of domain and types enforces the data integrity in Postgresql and allow developer to have a more fine grained control of the data stored in the database.

Postgresql functions

```
23 begin
24
25     -- sell order basic information retrive
26     select  sell_orders.id  ,state      ,size          ,amount         ,discount       ,finan_size     ,finan_amount
27     into    so_id           ,so_state   ,so_size        ,so_amount      ,so_discount    ,so_fin_size    ,so_fin_amount
28     from    sell_orders
29     where   sell_orders.id  = sell_order_id;
30
31     if (so_state != 'ongoing')
32     then
33         raise exception 'sell order state is not ongoing, is %',so_state;
34     end if;
35
36     -- discount control
37     if  (bid_discount < so_discount                    )       -- not an acceptable discount
38     then
39         raise exception 'discount of the bid not enought %', bid_discount;
40     end if;
41
42     -- sell order financed, but need to be recalculated the bid. Due to time limitations it's done here the adjustment.
43     if  ((so_fin_size  + investor_size) > so_size   )
44     then
45         -- if is adjusted we must insert an entry in the ledger too:
46         insert into ledgers(investor_id,sell_order_id,size         ,amount          ,balance    ,is_adjusted)
47                     values (investor_id,sell_order_id,investor_size,investor_amount,temp_balance,true       );
48
49         investor_size   = so_size - so_fin_size;
50         investor_amount = investor_size - (investor_size * (bid_discount/100));
51         is_adjusted     = true;
52     end if;
53
54     -- updating the investor balance
55     update  clients
56     set     balance = balance - investor_amount
57     where   clients.id = investor_id
58     returning balance into temp_balance;
```

Postgresql functions

For complex functionalities, like the “add bid”, it’s better to perform all the operations into a single Postgresql function, like the one shown in the previous page.

ORM usage in the project

```
74 type SellOrder struct {
75     gorm.Model
76     InvoiceID           uint
77     Size                string          `gorm:"type:amount_type"`
78     Amount              string          `gorm:"type:amount_type"`
79     Discount            string          `gorm:"->;type:discount_type generated always as (100 - (amount/size)*100) stored"`
80     FinanSize           string          `gorm:"type:amount_type_calc;default:0"`
81     FinanAmount         string          `gorm:"type:amount_type_calc;default:0"`
82     State               SellOrderState   `gorm:"type:sell_order_state;default:'ongoing'"`
83 }
84
85 func (i *SellOrder) CastgRPC() *pb.SellOrder {
86     return &pb.SellOrder{Id:uint64(i.ID),InvoiceId:uint64(i.InvoiceID),Size:i.Size,Amount:i.Amount,
87                          FinanSize:i.FinanSize,FinanAmount:i.FinanAmount,State:string(i.State)}
88 }
89
90 func  CastSellOrder(i *pb.SellOrder) *SellOrder {
91     return &SellOrder{InvoiceID:uint(i.InvoiceId),Size:i.Size,Amount:i.Amount,State:SellOrderState(i.State)}
92 }
```

ORM usage in the project

The use of the Gorm ORM reduces considerably the amount of code to manage and define the entities of Postgresql. At the same time, it allows a fine grained control of what is deployed in the database.

I also used CAST functions to easy cast gRPC objects to Gorm, and vice versa.

Testing with Testify

```
331     // invoice final state:
332
333     streamInv,err = c.ListInvoices(ctx,&pb.Empty{})
334     assert.Nil(t,err)
335     invoicesList = generateListInvoices(t,streamInv )
336
337     assert.Equal(t,1                    ,len(invoicesList)             )
338     assert.Equal(t,id.GetId()           ,invoicesList[0].GetClientId()  )
339     assert.Equal(t,"250.00"             ,invoicesList[0].GetAmount()    )
340     assert.Equal(t,"financed"           ,invoicesList[0].GetState()     )
341
342     // issuer checking:
343     stream, err = c.ListClients(ctx,&pb.IsInvestor{IsInvestor:false})
344     assert.Nil(t,err,"Error listint the two issuers inserted")
345
346     issuersList = generateListClients(t,stream)
347
348     assert.Equal(t,"2"          ,issuersList[0].GetFiscalIdentity(),    "Fiscal identity not equal to 2"    )
349     assert.Equal(t,"J2"         ,issuersList[0].GetName(),              "Name not equal to J2"              )
350     assert.Equal(t,"I2"         ,issuersList[0].GetSurname(),           "Surname not equal to I2"           )
351     assert.Equal(t,"2181.67"    ,issuersList[0].GetBalance(),           "Balance is not 2181.67"            )
352     assert.Equal(t,false        ,issuersList[0].GetIsInvestor(),        "Is investor is not false"          )
```

Testing with Testify

The client directory contains a golang client that make remote calls to the server and at the same time checks that all API works fine. This testing file could be refactored and more compact, but due to my limited time to do it, it was not done.