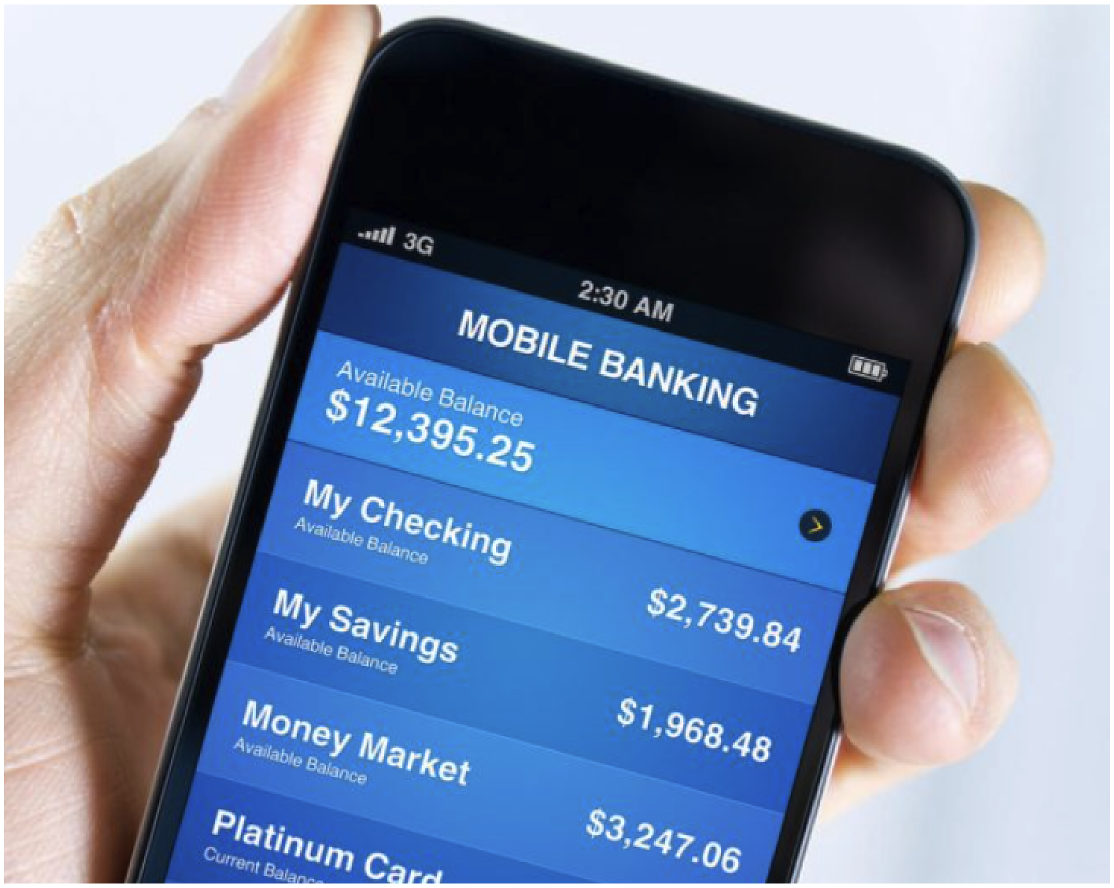
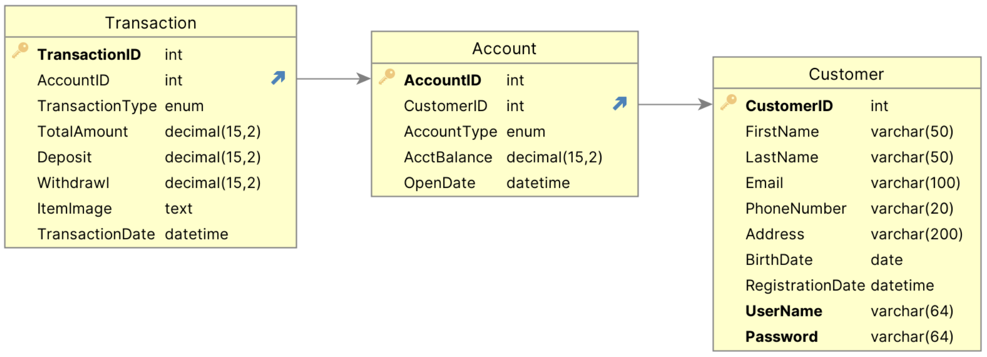
API Mobile Banking with logic & security

I have been using a new open source platform, API Logic Server (here) to deliver API microservices for a client. I wanted to build a complete mobile banking API from start to finish based on the TPC benchmark. This includes declarative business logic (aka spreadsheet like rules), security, react-admin UI, and an Open API (Swagger) documentation. API Logic Server (ALS) is an open source Python platform based on SQLAlchemy 2.0, Flask, safrs-JSON API, react-admin, and LogicBank (a declarative spreadsheet-like rules engine).



ChatGPT - SQL Model

I started by asking ChatGPT to generate a banking DDL based on the old TPC benchmark for MySQL. (Customer, Account, Transaction)



While ChatGPT gave me a usable DDL, I ask ChatGPT to add a few columns to allow declarative rules to do the heavy lifting.

Transaction

- Deposit DECIMAL(15,2)
- Withdrawal DECIMAL(15,2)
- Image (checks and withdrawal slips) TEXT

Authentication

- Customer UseName VARCHAR(64)
- Customer Password (hash) VARCHAR(64)

API Logic Server (ALS)

This is a full featured open source Python platform (like Django) to create a complete runtime API and UI solution. The command line feature of ALS made the creation of a running server with a multi-page react-admin U and an Open API a snap. The command will read the SQL 'banking' schema and create all the wiring needed for a REST API.

**ApiLogicServer create-and-run \
  –project_name=banking \
 –db_url=mysql+pymysql://root:password@127.0.0.1:3308/banking**

```
Welcome to API Logic Server 09.01.11
Customizable project banking created from database mysql+pymysql://root:password@127.0.0.1:3308/banking.  Next
steps:
Run API Logic Server:
  cd banking;  python api_logic_server_run.py
Customize using your IDE:
  code banking  # e.g., open VSCode on created project
  Establish your Python environment - see
https://apilogicserver.github.io/Docs/IDE-Execute/#execute-prebuilt-launch-configurations
API Logic Project (banking) Starting with CLI args:
.. /Users/tylerband/dev/ApiLogicServer/banking/api_logic_server_run.py, --create_and_run=True
Created August 17, 2023 09:44:38 at /Users/tylerband/dev/ApiLogicServer/banking
Data Model Loaded, customizing...
```

```
Logic Bank 01.08.04 - 1 rules loaded
Declare   Logic complete - logic/declare_logic.py (rules + code) -- 3 tables loaded
Declare   API - api/expose_api_models, endpoint for each table on localhost:5656, customizing...
API Logic Project loaded (not WSGI), version 09.01.11
 (running locally at flask_host: 0.0.0.0)
==> Customizable API Logic Project created and running:
..Open it with your IDE at /Users/tylerband/dev/ApiLogicServer/banking
API Logic Project (name: banking) starting:
..Explore data and API at http_scheme://swagger_host:port http://localhost:5656
.... with flask_host: 0.0.0.0
.... and  swagger_port: 5656
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI
server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5656
 * Running on http://192.168.1.10:5656
Press CTRL+C to quit
```

Rules

Declarative rules replace the manual task of writing code for the various use cases. Handling deposits and withdrawals, maintaining account balances, preventing overdrafts, and processing balance transfers. We start the process by writing the logic in a business user friendly way.

1. Derive Account balance is the sum of Transaction TotalAmount
2. Constraint: Account balance cannot overdraft (less than zero)
3. Constraint: Deposits and withdrawals must be greater than zero
4. Formula - transaction TotalAmount is Deposit less withdrawal
5. Customers can only transfer between their own accounts (ver 1)

Adding Rules

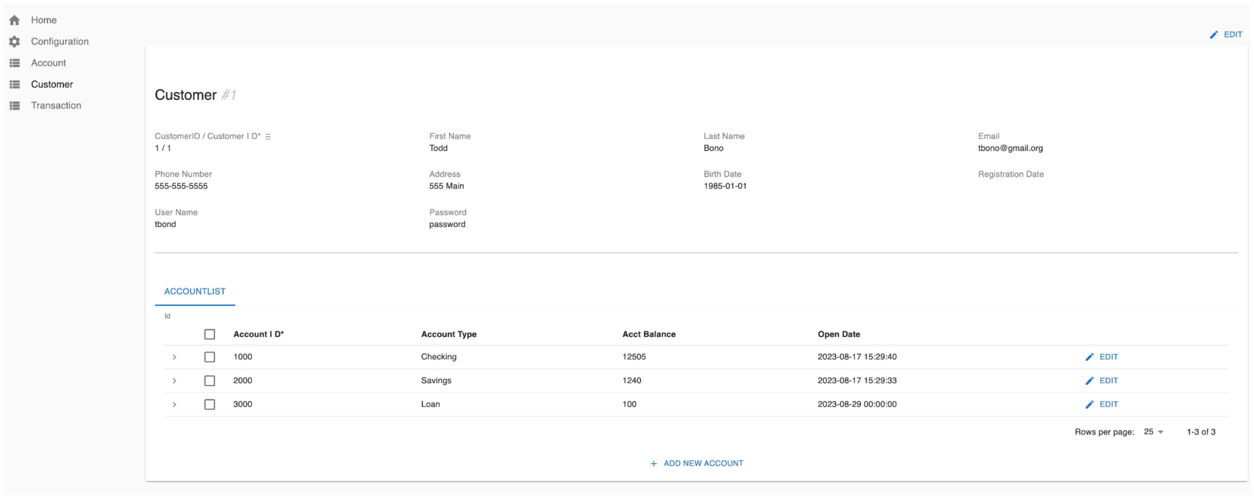
Using VSCode, the generated code is broken up into folders like database, api, logic, security, devops, etc. Under logic/declare_logic.py we convert our design rules into actual declarative rules, The code completion feature of Python makes this a breeze. Declarative rules handle all the use cases of adding transactions, deriving balances, and making sure the account does not overdraft.

logic/declare_logic.py

```
Rule.sum(derive=models.Account.AcctBalance,
    as_sum_of=models.Transaction.TotalAmount)

Rule.constraint(validate=models.Account,
    as_condition=lambda row: row.AcctBalance >= 0,
    error_msg="Account balance {row.AcctBalance} cannot be less than zero")

Rule.formula(derive=models.Transaction.TotalAmount,
    as_expression=lambda row: row.Deposit - row.Withdrawal)

Rule.commit_row_event(on_class=models.Transfer, calling=fn_transfer_funds)
```

Automated UI

ALS created a react-admin back office multi-table application for all the tables in the model. This allowed me to add a customer, accounts, and transactions.



Customer #1

CustomerID / Customer I D*	First Name	Last Name	Email
1 / 1	Todd	Bono	tbono@gmail.org
Phone Number	Address	Birth Date	Registration Date
555-555-5555	555 Main	1985-01-01	
User Name	Password		
tbond	password		

ACCOUNTLIST

Id

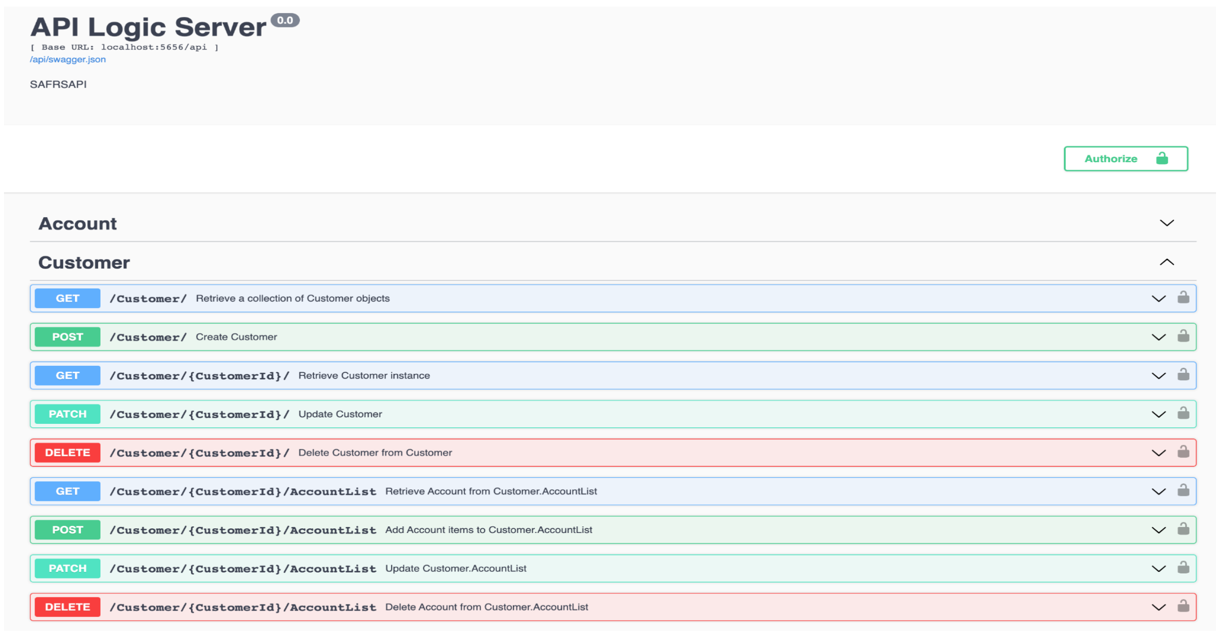
Account I D*	Account Type	Acct Balance	Open Date	
1000	Checking	12505	2023-08-17 15:29:40	EDIT
2000	Savings	1240	2023-08-17 15:29:33	EDIT
3000	Loan	100	2023-08-29 00:00:00	EDIT

Rows per page: 25 1-3 of 3

+ ADD NEW ACCOUNT

OpenAPI (Swagger)

ALS will also generate OpenAPI (Swagger) documentation. This is using the safrs Json-API which allows the ability to include child tables and specify which columns appear (much like GraphQL). This API will allow the front-end developer the ability to show the customer information, all their accounts, and a list of transactions (deposits and withdrawals) in a single API request. Another nice feature is each row returns a checksum which is used to support optimistic locking.



API Logic Server 0.0
[Base URL: localhost:5656/api]
/api/swagger.json

SAFRSAPI

Authorize

Account

Customer

GET /Customer/ Retrieve a collection of Customer objects
POST /Customer/ Create Customer
GET /Customer/{CustomerId}/ Retrieve Customer instance
PATCH /Customer/{CustomerId}/ Update Customer
DELETE /Customer/{CustomerId}/ Delete Customer from Customer
GET /Customer/{CustomerId}/AccountList Retrieve Account from Customer.AccountList
POST /Customer/{CustomerId}/AccountList Add Account items to Customer.AccountList
PATCH /Customer/{CustomerId}/AccountList Update Customer.AccountList
DELETE /Customer/{CustomerId}/AccountList Delete Account from Customer.AccountList

JSON-API can include fields and child relationships

```
curl -X 'GET' \
'http://localhost:5656/api/Customer/?include=AccountList&fields%5BCustomer%5D=CustomerID%2CFirstName%2CLastName%2CEmail%2CPhoneNumber%2CAddress%2CBirthDate%2CRegistrationDate%5BS_CheckSum&page%5Boffset%5D=0&page%5Blimit%5D=10&sort=id' \
  -H 'accept: application/vnd.api+json' \
  -H 'Content-Type: application/vnd.api+json'
```

Transfer Funds

The heart of the TPC benchmark was moving funds between 2 accounts in a single transaction. In this example, we let the rules do the formulas, derivations and validations but we need an API to POST the JSON. This is using the api/custom_api.py to build a Python class to transfer funds from one account to another. I then asked ChatGPT to generate the transfer funds code - so I added a new SQL Transfer table and a commit event rule to implement the change to do the same work.

```
curl -X 'POST' \
  'http://localhost:5656/api/Transfer' \
  -H 'accept: application/vnd.api+json' \
  -H 'Content-Type: application/json' \
  -d '{
  "meta": {
    "method": "transfer_funds",
    "args": {
      "FromAcctId": 2000, #Savings
      "YoAcctId": 1000,   #Checking
      "Amount": 1000
    }
  }
}'
```

Commit Row Event calling function

```
def fn_transfer_funds(row=models.Transfer, old_row=models.Transfer, logic_row=LogicRow):
        if logic_row.isInsert():
            fromAcctId = row.FromAccountID
            toAcctId = row.ToAccountID
            amount = row.Amount

            from_trans = models.Transaction()
            from_trans.TransactionID = nextTransId()
            from_trans.AccountID = fromAcctId
            from_trans.Withdrawl = amount
            from_trans.TransactionType = "Transfer"
            from_trans.TransactionDate = date.today()
            session.add(from_trans)

            to_trans = models.Transaction()
            to_trans.TransactionID = nextTransId()
            to_trans.AccountID = toAcctId
            to_trans.Deposit = amount
            to_trans.TransactionType = "Transfer"
            to_trans.TransactionDate = date.today()
            session.add(to_trans)

            print("Funds transferred successfully!")
```

Rules Fired

If an attempt to move too much money from the source account - the constraint rule fires on commit to prevent the transfer. The rule engine orders the firing of rules.

```
Rules Fired:            ## Account
1. Constraint Function: None
```

```
2. Derive Account.AcctBalance as Sum(Transaction.TotalAmount) Transaction
3. Derive Transaction.TotalAmount as Formula (1): as_expression=lambda row: row.Deposit -
row.Withdrawl

 "errors": [
    {
      "title": "Account balance -950.00 cannot be less than zero",
      "detail": {
        "model": "Account"
      },
      "code": 2001
    }
  ]
}
```

Security (Authentication/Authorization)

SInce this is a multi-tenant model, the declarative security model needs roles and filters that restrict different role players to specific CRUD tasks. The role based access control requires a separate data model for login, roles and user roles. We also will need an authentication process to validate users to login to the mobile banking system. ALS asks that you initialize the security model using the command line tool (ApiLogicServer add-auth db_url=${authdb}) which creates all the necessary components.

ROLES

- Customer - full access (limited to logon CustomerID)
- Teller - full access (any customer)
- Manager - read only (any customer)
- Admin - full access (any customer)

security/declare_security.py

```
DefaultRolePermission(to_role=Roles.customer,
    can_read=True, can_update=True, can_insert=True, can_dellete=False)

Grant( on_entity = models.Customer,
   to_role = Roles.customer,
   can_delete=False,
   filter = lambda : models.Customer.CustomerID == Security.current_user().CustomerID)
```

Handle Overdrafts

The real power here is the ability to introduce changes to the model, the rules, and the UI and let the ALS command-line handle these modifications. So now we can add a new Loan account

(UI) and add a new rule to transfer money to cover an overdraft amount before the constraint fires. The rules engine uses a DAG (directed acyclic graph) approach to order the firing of rules. This means the developer does not have to know in advance the impact of rule changes on the execution of declarative rules.

```
if from_account.AcctBalance > amount:
    # #Not Enough Funds - if Loan acct exists move to cover Overdraft (transfer Loan to from_acct)
    transfer = models.Transfer()
    transfer.FromAccountID = 3000 # add link account to checking and savings [from_acct.LoanAcctId]
    transfer.ToAccountID = fromAcctId
    transfer.Amount = amount - from_account.AcctBalance
     transfer.TransactionDate = date.today()
     session.add(transfer)
```

Docker Container

The devops/docker folder has shell scripts to build and deploy a running Docker image that can be deployed to the cloud in a few clicks. Just modify the docker compose properties for your database and security settings.

Summary

I was impressed with API Logic Server ability to create all the running API components from a single command line request. Using ChatGPT to get started and even iterate over the dev lifecycle was seamless. The front-end developers can begin writing the login (auth) and use the API calls from the Open API (Swagger) report while the final logic and security is being instrumented. While I am new to the Python language, this felt more like a DSL (domain specific language) with code completion and a well organized code space. The ALS documentation provides plenty of help and tutorials to help understand how to deliver your own logic and API.

#ChatGRT, #AI, #ApiILogicServer, #SQLAlchemy, #Flask, #MobileBanking, #ReactAdmin, #Python, #OpenAPI, #Swagger, #Docker, #TPC, #JSON, #MySQL