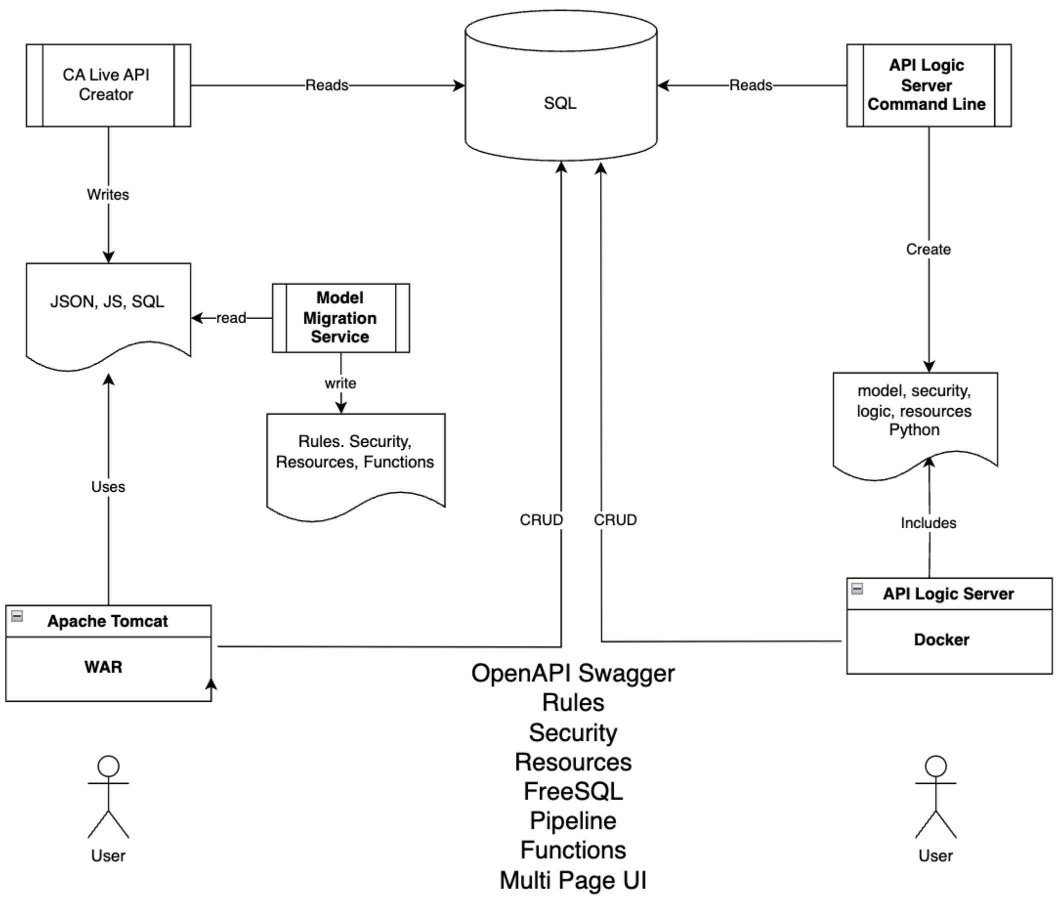
CA Live API Creator (LAC) migration to API Logic Server (ALS)

Overview

CA Live API Creator (**LAC**) from Broadcom was an API creation tool with declarative business logic. LAC was a mature product based on the acquisition of Espresso Logic in 2015. LAC is no longer offered by Broadcom (end-of-life fall 2022) and will end support in the fall of 2023. Customers are now looking for a migration path to move off the LAC platform. An open source solution named API Logic Server (**ALS**) can provide an excellent migration path for existing LAC customers. This comparison can be used to help LAC customers decide on a proper scope of work and migration strategy.

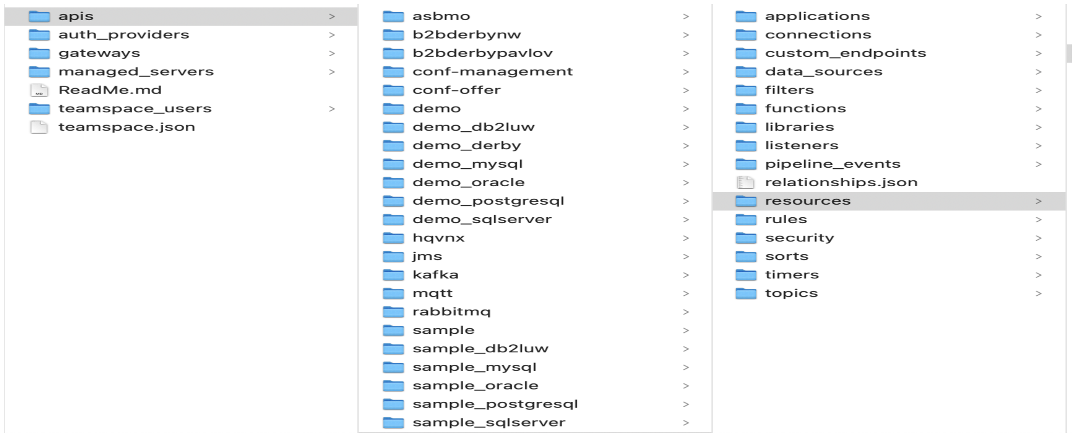
Migration from LAC to ALS



CA Live API Creator Studio

Live API Creator (LAC) studio is a self contained Java/Derby based client application that runs on a local desktop. Using an Angular client UI requires each developer to learn the workflow to connect to an SQL database and then use the UI to create/edit various components like rules, functions, query objects, security, etc. LAC uses a file based repository which contains all the definitions in JSON/JavaScript format which is packaged and linked to a runtime server (e.g.Derby or Apache Tomcat WAR). The LAC studio is a mixture of dialogs and JavaScript/SQL editors which is executed at runtime to implement the API calling declarative rules and resources.

LAC file based JSON repository

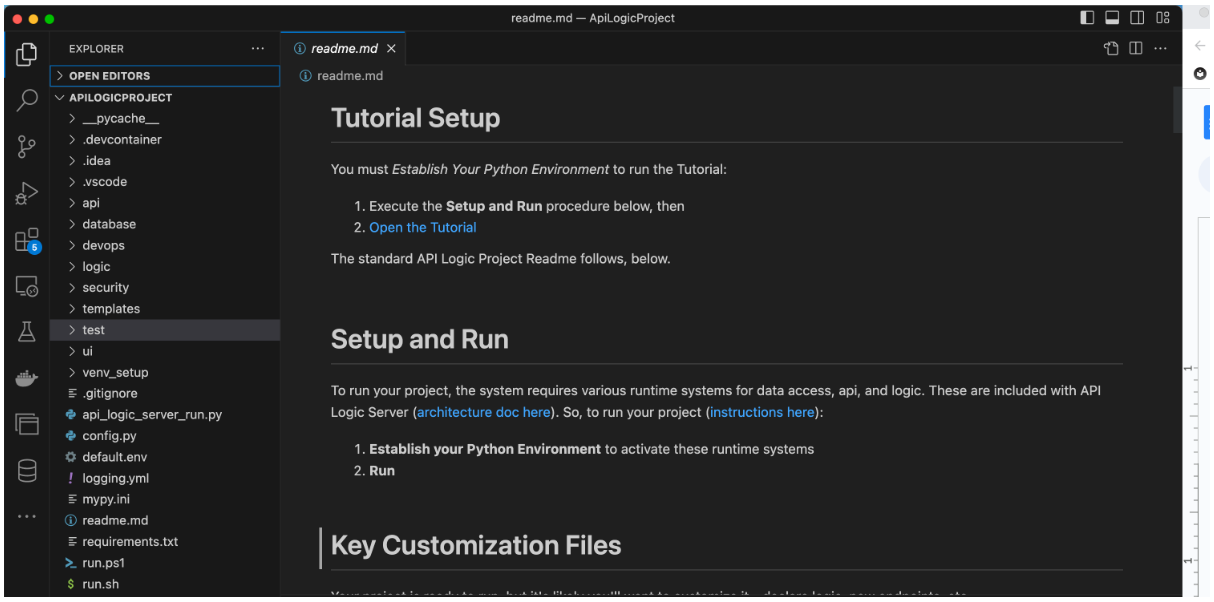


```
apis                    asbmo                   applications
auth_providers          b2bderbynw              connections
gateways                b2bderbypavlov          custom_endpoints
managed_servers         conf-management         data_sources
ReadMe.md               conf-offer              filters
teamspace_users         demo                    functions
teamspace.json          demo_db2luw             libraries
                        demo_derby              listeners
                        demo_mysql              pipeline_events
                        demo_oracle             relationships.json
                        demo_postgresql         resources
                        demo_sqlserver          rules
                        hqvnx                   security
                        jms                     sorts
                        kafka                   timers
                        mqtt                    topics
                        rabbitmq
                        sample
                        sample_db2luw
                        sample_mysql
                        sample_oracle
                        sample_postgresql
                        sample_sqlserver
```

ApiLogicServer (ALS) - BYO Editor

API Logic Server (ALS) is written in Python and uses a command line (CLI) interface to define a new project and connect to an existing database. The CLI will create a logical model, expose API objects in OpenAI/Swagger format and build a back office react-admin application. The CLI will generate a hierarchy of well organized Python code that can be extended using the editor of your choice (VSCode, IntelliJ, PyCharm, VIM, etc.). The ALS command line connects to the specified database (ability to include/exclude table filters is available) and creates a complete and running API server with a react-admin back office user interface (similar to LAC Data Explorer) and an Open API (aka Swagger) executable document. For detailed information on the architecture see here.

ALS Generated Python Hierarchy in VSCode



readme.md — ApiLogicProject

EXPLORER

OPEN EDITORS

APILOGICPROJECT

```
__pycache__
.devcontainer
.idea
.vscode
api
database
devops
logic
security
templates
test
ui
venv_setup
.gitignore
api_logic_server_run.py
config.py
default.env
logging.yml
mypy.ini
readme.md
requirements.txt
run.ps1
run.sh
```

readme.md

Tutorial Setup

You must *Establish Your Python Environment* to run the Tutorial:

1. Execute the **Setup and Run** procedure below, then
2. Open the Tutorial

The standard API Logic Project Readme follows, below.

Setup and Run

To run your project, the system requires various runtime systems for data access, api, and logic. These are included with API Logic Server (architecture doc here). So, to run your project (instructions here):

1. **Establish your Python Environment** to activate these runtime systems
2. **Run**

Key Customization Files

ALS Command Line tool

```
$ApiLogicServer --help

Welcome to API Logic Server 08.02.00
Usage: ApiLogicServer [OPTIONS] COMMAND [ARGS]...
     Creates [and runs] logic-enabled Python database API Logic Projects.
     Doc: https://apilogicserver.github.io/Docs
     Examples:
         ApiLogicServer tutorial
         ApiLogicServer create-and-run --db_url={db_url} –project_name={name}
         ApiLogicServer create
     Then, customize created API Logic Project in your IDE
Options:
  --help  Show this message and exit.
Commands:
  about                     Recent Changes, system information.
  add-auth                  Add authorization/authentication to current project.
  add-cust                  Add customizations to northwind project.
  add-db                    Add db (model, binds, api, app) to current project.
  create                    Creates new customizable project (overwrites).
  create-and-run            Creates new project and runs it (overwrites).
  create-ui                 Creates models.yaml from models.py (internal).
  examples                  Example of commands, including SQLAlchemy URIs.
  rebuild-from-database     Updates database, api, and ui from changed db.
  rebuild-from-model        Updates database, api, and ui from changed models.
  run                       Runs existing project.
  tutorial                  Creates (updates) Tutorial.
  welcome                   Just print version and exit.

example:
$ApiLogicServer create-and-run --project_name=myApiProject
--db_url=postgresql://admin:password@127.0.0.1:5432/postgres  –include_tables=incl.yaml
```

In this example, ALS connects to the Postgresql database and will create a complete running API server, an OpenAI (Swagger) documentation, and a fully functional react-admin UI application to navigate the data using the API's. This open source code can be viewed, edited, and deployed to a Docker container (see devops) or run as a stand-alone server.

Declarative Rules

Both ALS and LAC have a logic layer that executes declarative rules listed below. The syntax is slightly different but the end result is identical. LAC uses a combination of dialogs and JavaScript editor to implement rules. ALS is a pure Python based editor with code completion and examples. LAC stores the definitions in a file based repository in JSON and JavaScript. ALS stores the rules in a Python file as part of the server execution of REST calls.

Rule Features

LAC Rule	ALS Rule	ALS Sample
Sum	Rule.sum	Rule.sum(derive=models.Customer.Balance,as_sum_of=models.Order.AmountTotal, where=Lambda row: row.ShippedDate is None)
Count	Rule.count	Rule.count(derive=models.Customer.UnPaidOrders, as_count_of=models.Order, where=Lambda row: row.ShippedDate is None)
Parent Copy	Rule.copy	Rule.copy(derive: <class.attribute> from_parent:<parent-class.attribute > source of copy
Formula	Rule.formula	Rule.formula(derive=models.OrderDetail.Amount as_expression=lambda row: row.UnitPrice * row.Quantity)
Constraint	Rule.constraint	Rule.constraint(validate=models.Customer, as_condition=lambda row: row.Balance <= row.CreditLimit, error_msg="balance ({row.Balance}) exceeds credit ({row.CreditLimit})")
Early event	early_row_event	Rule.early_row_envet(on_class: <class> for event calling: function, passed row, old_row, logic_row
Event	Rule.row_event	Rule.row_enent(on_class: <class> for event calling: function, passed row, old_row, logic_row

Commit Event	commit_event	Rule.commit_row_envent(on_class: <class> for event calling: function, passed row, old_row, logic_row
Insert Parent	logic_row.insert	logic_row.insert
logicContext	logic_row	variable
oldRow	old_row	variable
row	row	variable
	early_row_event_all_classes	Rule.early_row_event_all_classes(early_row_event_all_classes=handle_all) used by optimistic locking and grant security
	parent_check	Rule.parent_check(...)

Model Migration Server - Rule Migration

An open source tool Model Migration Service (MMS) can parse an existing LAC repository and generate output which can be used as a starting point for migration.

```
$python3 reposreader.py --help
Generate a report of an existing CA Live API Creator Repository

options:
  -h, --help            show this help message and exit
  --repos REPOS         Full path to /User/guest/caliveapicreator.repository
  --project PROJECT     The name of the LAC project (teamspaces/api) default: demo
  --section SECTION     The api directory name to process [rules, resources, functions, etc.] default: all
  --version             print the version number and exit


python3 reposreader.py  --project demo --repos /Users/guest/caliveapicreator.repository --section all
```

Rule Output

Each rule appears in the output which has been transformed from the LAC JSON to ALS Python format. These rules can be pasted into ALS Rules (folder logic/declare_logic.py).

```
=========================
    RULES
=========================
# ENTITY: Customer

def fn_customer_validation_72(row: models.Customer, old_row: models.Customer, logic_row: LogicRow):
       return (row.areaCode>=0 and row.areaCode<10)

'''
```

```
    RuleType: validation
    Title: single digit area code
    Name: validation_72
    Entity: Customer
    Comments: None
'''
Rule.constraint(validate=models.Customer,
        calling=fn_customer_validation_72,
        error_msg="The area code must be a single digit 0-9")
```

Issues:

- The entity names may not match 100% - ALS will standardize the names and this may require manual intervention.
- The calling function will attempt to 'fixup' JavaScript to Python syntax - this WILL require manual intervention for each function to make sure the indentation and style are correct.
- Built-in functions like SysUtility are not available
- JavaScript libraries may need to be imported and modified (e.g. math is Python, Math is JavaScript)
- JS Functions may be replaced by 'as_expression=' to make it more readable
- LAC Libraries are global to rules and may need to be migrated as functions

DataSources and Models

LAC provides an editor to define one or more datasource connections and table filter properties. The LAC Schema editor provides dialogs to add virtual relationships between parent/child tables. The definition is stored in a JSON file that lists all the tables, columns, keys, and relationships.

ALS supports multiple datasources in a project - see link here. LAC uses a prefix to help separate conflicting table names (e.g. nw:customers, demo:customers). ALS uses __bind_key__ to identify different schemas.

ALS stores similar information in the model.py class definition. Virtual relationships can be added using any text editor or IDE. ALS offers a CLI to rebuild UI from model or rebuild model from database. Note: The model.py can be edited (add or rename relationships, define foreign keys, etc).

LAC (JSON)	**ALS (SQLAlchemy Model)**

Column:[{
"name": "CustomerID",
"remarks": null,
"jdbcSqlType": 0,
"size": 0,
	class Customer(SAFRSBase, Base):
	__tablename__ = 'Customer'
	_s_collection_name = 'Customer' # type: ignore
	__bind_key__ = 'None'

```
            "nullable": true,
            "autoIncrement": false,
            "isVersion": false,
            "isForeignKey": false,
            "defaulted": false,
            "dbColumnType": null,
            "baseTypeName": "VARCHAR(5)",
            "isUserDefinedType": false,
            "baseType": {
              "className":”metadata.udt.StringType",
              "dbTypeName": "VARCHAR(5)",
              "jdbcSqlType": 12,
              "genericType": "string",
              "isDistinctType": false,
              "isUserDefinedType": false,
              "maxLength": 5,
              "isFixedLength": false
            },

Relationship:[ {
   "parentEntity": "demo:Order",
   "childEntity": "demo:Item",
   "roleToParent": "lineitem_order",
   "roleToChild": "LineItemList",
   "deleteRule": "Cascade",
   "updateRule": "No Action",
   "parentColumns": [
     "order_number"
   ],
   "childColumns": [
     "order_number"
   ]
 },
```

```
        Id = Column(String(8000), primary_key=True)
        CompanyName = Column(String(8000))
        ContactName = Column(String(8000))
        ContactTitle = Column(String(8000))
        Address = Column(String(8000))
        City = Column(String(8000))
        Region = Column(String(8000))
        PostalCode = Column(String(8000))
        Country = Column(String(8000))
        Phone = Column(String(8000))
        Fax = Column(String(8000))
        Balance : DECIMAL = Column(DECIMAL)
        CreditLimit : DECIMAL = Column(DECIMAL)
        OrderCount = Column(Integer,
        server_default=text("0"))
        UnpaidOrderCount = Column(Integer,
        server_default=text("0"))
        Client_id = Column(Integer)
        allow_client_generated_ids = True

# parent relationships (access parent)

# child relationships (access children)
OrderList : Mapped[List["Order"]] =
            relationship(back_populates="Customer")
```

The LAC has an extensive vocabulary for dealing with different databases and column attribute types. This includes dealing with doubles, floats, intervals, blobs and clobs. Any POC must validate the database specific attribute types and values stored and retrieved.

Supported Databases

LAC supports many of the major databases and provides extensibility to map others. ALS using SQLAlchemy supports databases like MySQL, Postgresql, MS SQL Server, Oracle, SQLLite, and many others (see link here).

Tables, Views, Functions, Procedures

When LAC scans the metadata of an SQL database, it creates API’s for tables, views, functions, and procedures. While rules only apply to base tables, many clients use the views to shape a more complex SQL request as a REST API. The functions and procedures expose SQL based expressions as REST endpoints.

ALS currently exposes base tables by default. If the client application is using views, functions or procedures, this will require ALS to do additional work to expose these features.

SQLAlchemy classes can be created for views. Extensible Builders have been used to expose APIs for some T/SQL function.

LAC Functions

The LAC allows developers to write JavaScript functions which can be called by rules and resources. ALS can also define Python functions which can also be called by rules or security. JavaScript is very similar to Python and can be transformed into defined functions (def fn_{name}). See Model Migration fixup in Appendix A.

LAC Libraries

LAC allows developers to write or import JavaScript libraries which can be used by rules or resources. ALS is a full featured Python environment with access to a large collection of libraries that can easily be added to the virtual environment and called by rules or resources.

LAC Resources

LAC editor allows the developer to shape an API endpoint as a named resource based on an existing table, adding or removing attributes, adding child objects (nested and filter queries) (e.g. Customer/Orders/Items/Product). Each level in LAC allows a join condition and optional filter (e.g. paid orders) in SQL syntax. This metadata is stored in the file based repository using JSON objects. ALS can provide a similar multi-table document using safrs-JSON client include.

LAC TableBase API

```
{
  "name": "AccountSummary",
  "description": null,
  "siblingRank": 100,
  "resourceType": "TableBased",
  "prefix": "nw",
  "entity": "Customers",
  "isCollection": true,
  "isCombined": false,
  "filter": null,
  "order": null,
  "useSchemaAttributes": false,
  "attributes": [
    {
      "attribute": "CustomerID",
      "alias": "CustomerID",
      "description": null,
      "isKey": false
    },
    {
      "attribute": "CompanyName",
      "alias": "CompanyName",
      "description": null,
      "isKey": false
    },
    {
      "attribute": "Balance",
      "alias": "Balance",
      "description": null,
      "isKey": false
    },
    {
      "attribute": "CreditLimit",
      "alias": "CreditLimit",
      "description": null,
      "isKey": false
    }
  ]}
```

ALS CustomEndpoint

Model Migration Service will generate a user defined resource using the class api/system/CustomEndpoint. This uses the SQLAlchemy model for code completion (e.g. models.Customer.CompanyName). A CustomEndpoint is defined by a model_class and an optional alias name. The fields are optional tuples (fieldName, and optional alias name) but allow the shaping of the response including aliasing of columns. Finally, the ‘calling=’ allows a function to be introduced for each row (allowing creation of virtual attributes).

```
  @app.route("/rest/default/demo/v1//CustomerResource", methods=[‘GET’,’OPTIONS’])
  @admin_required()
  @jwt_required()
  @cross_origin(supports_credentials=True)
  def CustomerResource():
      root = CustomEndpoint(model_class=models.Customer, alias="Customer"
          , fields = [(models.Customer.CompanyName, "name"),(models.Customer.Balance)]
          , calling=fn_testCalling
      )

      result = root.execute(request)
      return root.transform(‘LAC’,’customer’,result)

def fn_testCalling(row: dict, tableRow: dict, parentRow: dict):
      """
      Example called by Resource for each row
      Args:
          :row (dict): current row to modify
          :tableRow (dict): the full row before modification
          :parentRow: (dict): the linked parent row

      """
      row.virtual = "Virtual"

Example:
$curl -X 'GET' "http://localhost:5656/CustomerResource?Id=ALFKI"
{"Customer": [{"name": "Alfreds Futterkiste", "Balance": 2102.0, "virtual": "Virtual"}]}
```

ALS Nested Resource

The CustomEndpoint can also handle nested calls. For child resources, the introduction of the attribute **join_on**=model.Table.Attribute which will link the child resource back to the primaryKey of the parent. If the relationship is ManyToOne - use the keyword isParent=True.

```
customer = CustomEndpoint(model_class = models.Customer, alias="Customer"
      , fields = [(models.Customer.CompanyName, "name"),(models.Customer.Balance)]
      , include = [
          CustomEndpoint(model_class = models.Order, alias = "orders", join_on=models.Order.CustomerId
              , fields = [models.Order.AmountTotal, models.Order.ShippedDate, models.Order.OrderDetailCount]
              , include = CustomEndpoint(model_class = models.OrderDetail, alias = "details"
                     ,join_on=models.OrderDetail.OrderId
                   , fields = [models.OrderDetail.Quantity, models.OrderDetail.UnitPrice,  models.OrderDetail.Amount]
                   )
              )
          ]
      )

result = customer.execute(request)
return customer.transform('LAC','',result)
```

CustomEndpoint Issues:

- Model Migration currently deals with TableBased defined resources. (JavaScript will require a different approach)
- FreeSQL is available as a call to the SQL expression - this will require some fixup of the passed parameters.
- The get_event.js becomes a calling function - the JavaScript may not translate and will require additional intervention
- These requests are not exposed in OpenAI by default (but can be) - but this only for LAC transformations, the existing LAC Swagger should be the same.

ALS SQLAlchemy Example

ALS has an example on how to also hand craft a nested API t shown below. These shaped API's can reduce client-server interaction and LAC can handle PUT/POST state change on each of these API. ALS would require base API (table) updates. ALS uses safrs-JSON which allows some flexibility on shaping the request (attributes and filters). This area of ALS can be refactored into a function passing in the tables, relationships, filters, and joins.

ALS Server Side Customized Nested API

```
    """
    ALS Illustrates:
     * Returning a nested result set response
     * Using SQLAlchemy to obtain data
     * Restructuring row results to desired json (e.g., for tool such as Sencha)

    """
     order_id = request.args.get('Id')
     db = safrs.DB         # Use the safrs.DB, not db!
     session = db.session  # sqlalchemy.orm.scoping.scoped_session
     order = session.query(models.Order).filter(models.Order.Id == order_id).one()

     result_std_dict = util.row_to_dict(order
                                   , replace_attribute_tag='data'
                                   , remove_links_relationships=True)
     result_std_dict['data']['Customer_Name'] = order.Customer.CompanyName # eager fetch
     result_std_dict['data']['OrderDetailListAsDicts'] = []
     for each_order_detail in order.OrderDetailList:       # lazy fetch
         each_order_detail_dict = util.row_to_dict(row=each_order_detail
                                             , replace_attribute_tag='data'
                                             , remove_links_relationships=True)
         each_order_detail_dict['data']['ProductName'] = each_order_detail.Product.ProductName
         result_std_dict['data']['OrderDetailListAsDicts'].append(each_order_detail_dict)
     return result_std_dict
```

LAC N+1 optimization on nested queries (passing in the parent keys to filter the selection of children) is a key performance improvement. It will be important to measure the performance of ALS when creating nested resources.

ALS Client Side Nested API using SAFRS-JSONAPI

Safrs-JSON can include nested document requests including relationship model names- this allows the client to do a transform of the result. (see link).

```
curl -X GET "http://localhost:5656/api/Customer/ALFKI/?\
children=OrderList%2COrderList.OrderDetailList%2COrderList.OrderDetailList.Product\
  -H 'accept: application/vnd.api+json' \
  -H 'Content-Type: application/vnd.api+json'
```

SAFRS CustomEndpoint Shaped Response

CustomEndpoint definition can be used to shape a SAFRS-JSON query. The “get” function will take the result of the SAFRS-JSON call and return a shaped nested document with all the same functionality (virtual attributes and alias names).

```
@app.route("/safrsrequest" , methods=[‘GET’,’OPTIONS’])
@admin_required()
@jwt_required()
@cross_origin(supports_credentials=True)
def safrsrequest():
   # user defined from OpenAPI/Swagger
   #query = "http://localhost:5656/api/Customer/ALFKI/"
    #query += "?include=OrderList%2COrderList.OrderDetailList%2COrderList.OrderDetailList.Product"
    #requests.get(query)

    customer = CustomEndpoint(model_class = models.Customer, alias="Customer"
    , fields = [(models.Customer.CompanyName, "name"),(models.Customer.Balance)]
    , children= [
       CustomEndpoint(model_class = models.Order, alias = "Orders", join_on=models.Order.CustomerId
          , fields = [models.Order.AmountTotal, models.Order.ShippedDate]
          , children= CustomEndpoint(model_class = models.OrderDetail, alias = "Details"
             ,join_on=models.OrderDetail.OrderId
             , fields = [models.OrderDetail.Quantity, models.OrderDetail.Amount]
             , children= CustomEndpoint(model_class = models.Product, alias = "Product"
                 ,join_on=models.OrderDetail.ProductId
                , fields=[ models.Product.ProductName, models.Product.UnitPrice, models.Product.UnitsInStock]
                , isParent=True # this tells the system to use the foreign key from parent
                , isCombined=True # merge with parent row
                )
             )
          )
       ]
    )
    result = customer.get(request,"OrderList%2COrderList.OrderDetailList%2COrderList.OrderDetailList.Product","ALFKI")
   return customer.transform('LAC','customer',result)
```

Virtual Attributes

LAC Resources support the ability to add virtual attributes to a defined endpoint. Logic rules (get_event.js) will populate the values on a GET request. The calling function will pass in the row, the tableRow, and the parentRow to use in the user defined function.

```
def fn_recastTarget_data_mappedfirstlevel_event(row: dict, tableRow: dict, parentRow:  dict):
   row.expanded = row.leaf
   row.subAreaCode = parentRow.subAreaCode
   row.virtual = "Virtual"
```

LAC JS Resource

This is just another named API endpoint that executes JavaScript code to return a JSON response. This will require custom Python coding in ALS. LAC can pass in args used by sorts, where, offset, and page count. The "requests" library can be used to retrieve from other API servers using request.get("http://{server}:{port}/{api_endpoint}").

LAC FreeSQL Resource

This is just another named API endpoint that executes SQL code to return a JSON response. This will use custom Python coding in ALS FreeSQL class to execute and return JSON.

```
  @app.route('/rest/default/ucf/v1/freesqltest , methods=['GET','OPTIONS'])
   def freesqltest():
        sql = get_freesqltest(request)
        return FreeSQL(sqlExpression=sql).execute(request)

def get_freesqltest(request):
    Id = request.args.get(“CustomerId”) or “-1”
    return f“select * from Customer where CustomerId = {id}”
```

API Pagination

LAC provides an API to define how many rows are returned in an API result set (start row, offset). ALS supports a similar concept. (e.g. ?page[limit]=20&page[offset]=0)

Optimistic Locking

LAC generates a locking key based on the hash of the object attributes. This helps with optimistic locking with CRUD operations. Many applications bypass the optimistic locking feature based on the use case of how they access data. ALS has added a similar concept (see here). The CustomEndpoint transform will move the S_CheckSum to the LAC style for GET requests and reverse the JSON for PUT/PATCH.

```
[{"@metadata":{"checksum":"-2726117700064711271"}, … truncated…)]
```

LAC Security

Both LAC and ALS use a role based access control to define authentication and authorization to API endpoints. This information from LAC can be used to seed ALS security,

LAC ACL Security JSON

```
{
  "name": "Supplier",
  "description": "Login as pavlov. The Custom Auth Provider filters supplier rows, using userData. Rest Endpoints
are limited (click tab above).",
  "defaultTablePermission": "N",
  "defaultViewPermission": "N",
  "defaultFunctionPermission": "N",
  "globals": {
  },
  "apiVisibility": {
    "table": {
      "isRestricted": true,
      "restrictedTo": [
        "nw:Suppliers",
        "promos:promotions"
      ]
    },
    "resource": {
      "isRestricted": true,
      "restrictedTo": [
        "v1/SupplierAlert",
        "v1/SupplierInfo"
      ]
    }, …truncated…
```

ALS Security Model

API Logic Server has a grant based access control (ACL) model that maps users to roles. The default out-of-the-box uses a SQLite database to store users and roles (database/authentication_db.sqlite). Mapping CA Live API Creator (LAC) security model is described below.

User and UserRole

The login to ALS uses the User table to validate the username and password. ALS uses a SQLite3 table for username and password (plaintext). The LAC JSON model stores each user and password (hashed and salted) in a JSON file. In addition, LAC stores the roles for each user.

LAC User JSON

```
{
  "name": "sjb",
  "fullname": "Sam Bachman",
  "email": "sbachman@clarity-iq.com",
  "isActive": true,
  "comments": null,
  "keyLifetimeSeconds": 172800,
  "passwordHash": "5oQEU/OIBWZBzhbyQ==",
  "passwordSalt": "45gUiXydPshXB",
  "roles": [
    "backoffice",
    "masterPlanner",
    "recaster",
    "recastUploader"
  ],
  "globals": {
  }
}
```

ALS Translated all Roles

This is generated from all the roles and users in the LAC security/roles folder. These roles are only used for code completion and must match the Roles in the database.

```
class Roles():
        backoffice = 'backoffice'
        ClientReportDetailer = 'ClientReportDetailer'
        masterPlanner = 'masterPlanner'
        clientUser = 'clientUser'
        recaster = 'recaster'
        recastUploader = 'recastUploader'
        winter17 = 'winter17'
        manager = 'manager'
```

LAC Roles JSON

Each role is stored in a JSON file and describes both the roles' default table permissions (N-None, R-Read Only, F-Full, X-Execute) and each entity permissions and access level.

```
{
  "name": "masterPlanner",
  "description": null,
  "defaultTablePermission": "N",
  "defaultViewPermission": "N",
  "defaultFunctionPermission": "X",
   "entityPermission": {
    "actionPlanScenarioCRUD": {
      "entity": "main:actionPlanScenario",
      "description": null,
      "accessLevels": [
        "ALL"
      ]
    },.. truncated…
```

ALS Security Definition

The security/declare_security.py contains the generated LAC security mapped from the LAC JSON file. The DefaultRolePermission defines the roles overall permission. The Grant setting maps for each entity and optional filter (this inserts a row level where clause on each select for the entity). The comment line reflects the source from LAC. The filter needs to be modified from LAC style to ALS style using code completion and lambda signature..

```
#Role: backoffice TablePermission: R
DefaultRolePermission(can_read=True, can_insert=False , can_update=False , can_delete=False,
to_role='backoffice')

#Role: ClientReportDetailer TablePermission: R
DefaultRolePermission(can_read=True, can_insert=False , can_update=False , can_delete=False,
to_role='ClientReportDetailer')

#Role: masterPlanner TablePermission: N
DefaultRolePermission(can_read=False, can_insert=False , can_update=False , can_delete=False,
to_role='masterPlanner')

#Role: clientUser TablePermission: N
DefaultRolePermission(can_read=False, can_insert=False , can_update=False , can_delete=False,
to_role='clientUser')

#Access Levels: ['READ'] TablePermissions: N description: clientSafeReports
Grant(on_entity=models.Report ,
        can_read = True,
        can_update = False,
        can_insert = False,
         can_delete = False,
         to_role=Roles.clientUser
        ,filter= lambda: models.Report.isClientUserSafe == True)
```

Logic Rules for security

In the logic/declare_logic.py - a global rule is used to trigger role based ACL for insert, update, and delete. [`Grant.process_updates(logic_row=logic_row)`] This will trigger an early_row_event (before logic) and raise a GrantSecurityException if the user and their union of roles and grant entity determines if it needs to raise an error.. [`raise GrantSecurityException(user=user,entity_name=entity_name,access="insert")`]. Select does the same thing but will add an optional filter on the get to apply row level access.

ALS SQLite (database/authentication_db.sqlite) was used as a default to store User, Role, and UserRole and allow the default login to work out-of-the-box. For production, a more robust data model will be added to MySQL. (security/authorization.py)

Note 1: auth tokens are not used by ALS - these are predefined values that can be sent in the Authorization header. This is not recommended for security going forward.

Note 2: Currently we are using http scheme - the plan is to use https (NGINX/gunicorn) to secure the communications between Tomcat and ALS. Working on a docker compose file.

Tenant Filter

The filter must be in SQLAlchemy format (e.g. filter= lambda: models.Customer.client_id == Security.current_user().client_id). This style will limit all access to the Customer object to the client_id of the logged in user (from the modified User table).

Timers

LAC timers were used to execute JavaScript events. This can easily be implemented in ALS using native CRON tools to call an API running Python code.

REST Lab

The LAC Rest lab is a built-in version of Postman that allows quick execution of API endpoints. ALS generates an Open API (aka **Swagger**) like LAC which can also be used to test api endpoints. The big difference is that ALS creates a safrs-JSON which allows more shaping and filtering of the REST response.

Pipelines/Transforms

LAC pipelines are post-processors for different event lifecycles (GET, PUT, POST, DELETE, PATCH) to add or remove JSON results based on the state. This can be done in ALS using custom code on exposed endpoints. This can be called by the resource endpoint transform function. For example - LAC uses @metadata: {“checksum”: value} - while ALS uses S_CheckSum as an additional attribute. So a custom transform is needed to move this value to/from @metadata.

```
{
  "name": "somePipeLineExample",
  "eventType": "response",
  "comments": null,
  "isActive": false,
  "appliesTo": {
    "get": true,
    "put": false,
    "patch": false,
    "post": false,
    "delete": false
  },
  "isRestricted": true,
  "restrictedTo": [
    "v1/recastAssembled"
  ]
}
```

Messaging

LAC provides examples and pre-installed libraries for messaging, including use of Query Objects for payload creation. ALS enables access to all Python libraries (e.g., messaging), with payload creation as described for Resources.

Debugger

LAC requires starting 2 Derby instances (so that the debugger can interrupt the running Java VM). ALS supports debugging inside VSCode out-of-the-box with no special instructions. This is a big plus for developers learning ALS and Python.

Data Explorer vs ALS Admin App

LAC provides an AngularJS (v1.0) back office user interface named Data Explorer. The application provides some editing and control of layout and field selection. ALS also builds a similar react-admin user interface and provides a yaml file for editing content. Both provide multi-table (parent/child) navigation, parent lookups, and the ability to edit layout.

Test Framework

ALS provides examples to build and test API and rules using the Behave framework. LAC does not offer any test framework out of the box. A command line tool written in NodeJS can be used to connect (authentication) to store the JWT token and perform GET/PATCH/POST/DELETE on various ALS endpoints (found here).

Open AI (Swagger)

Both LAC and ALS generate an Open AI for all defined REST API endpoints. However, ALS uses safrs-JSON which allows clients to shape the request (columns) and filters. SQLAlchemy supports [limit][offset] like LAC from the client request along with nested relationships.

```
http://localhost:5000/api/People/?include=books_read.author&page[books_read][limit]=2&page[books_read][offset]=2&page[limit]=1
```

Runtime Server

LAC is a client based editor that creates the file based metadata which is required for a runtime server (e.g. Apache Tomcat) to function. ALS is a completely self-contained Python project which can be packaged and deployed as a stand-alone running service or packaged as a Docker container (see the devops folder).

ALS Architecture

There are really two parts to ALS architecture. The command line tool uses a combination of other components (e.g SQLAlchemy 2.0 ORM, LogicBank for Rules, Flask, SAFRS-JSON, react-admin, and OpenAPI (Swagger)) to generate an ORM model, a runtime REST API, a react-admin UI, and documentation in an executable server. Since this is an open source Python project, all of the components and code can be maintained as part of the ongoing development process.

ALS Learning Curve

The ALS tutorial(s) produce an extensive set of examples that can help a new developer understand the flow of execution. Since this is Python based development, some basic understanding of the selected editor (e.g. VSCode) and the language syntax as well the rules, model, and user interface.

Summary

LAC and ALS both provide filtering, sorting, pagination, and multi-table retrieval. The styles are different:

- LAC APIs are **server-defined**, with custom Resource definition
- ALS APIs are **client-defined** using include and other safrs-JSON syntax

ALS has an active open source standards-based dev experience - logic definition, source control, rule definition, debugging, test management.

ALS has an open source Python code base and is not vendor dependent.

Declarative logic rules and Data Explorer are functionally similar.

Prepare a POC

In preparation for analysis, the LAC client repository is needed to evaluate the size and scope of work using Model Migration Service:

- Datasources
- Tables. Views, Functions
- Column DataTypes
- Relationships
- Functions
- Pipelines
- Libraries
- Resources
- Security (ACL) Users, Roles, Grants

- Rules

The parser tool will navigate the LAC repository project and generate a report (and JavaScript transforms) for some of the repository objects that will provide a starting point to migrate LAC to ALS.

Rule Transform to ALS declare_logic.py

```
LAC formula_amount.json
'''
RuleType: formula
Title: Discounted price*qty
Name: formula_amount
Entity: LineItem
Comments: Reactive Logic is expressed in JavaScript, so you use...
- conditional logic (as above),
- standard JavaScript services (e.g., moment date functions - enable in Project > Libraries),
- SQL / external services.... whatever is required.
'''
Rule.formula(derive=models.LineItem.amount calling=fn_formula_amount)

LAC formula_amount.js
def fn_formula_amount(row: models.ENTITY, old_row: models.ENTITY, logic_row: LogicRow):
        if (row.qty_ordered <= 6)
           return row.product_price * row.qty_ordered
        else
           return row.product_price * row.qty_ordered * 0.8
```

Database(s)

Access to the database(s), connection information (user/password), connection string and sample data set (not live data). This will be used by ALS to create the first set of table based API's. [ApiLogicServer create –db_url={serverURL}]

Proof of Concept

A limited POC should be performed to demonstrate the base concepts of creating a rule backed API running on the ALS server. The specific criteria will be based on the analysis of the existing LAC repository to determine which features are required and which features need to be added to ALS. In addition, performance measurement of the API's will be performed to compare the current LAC implementation with the ALS rule backed service.

- Running ALS (API Logic Server) in a Docker container
- API Endpoints for each SQL table
- OpenAPI (Swagger) documentation for created API endpoints
- Executable react-admin User Interface (UI) for all base tables (read, insert, update, and delete)
- Conversion of LAC rules to ALS logic (LAC uses JavaScript and ALS is Python) goal is the migration for POC (formula, sum, count, event, validation, copy)

- Create custom nested API based on LAC Resource model
- A simple role based ACL (access control list) and login authentication will be implemented
- Performance timing of all API endpoints
- Package and run a Docker server

Appendix A

Model Migration Service

The Model Migration Service project can parse the LAC (v5.4) file repository and convert some of the information (rules) to ALS. Conversion of JS to Python. While this does not address every issue it will give a good start to edit each JS wrapped in a Python function.

FIXUP to convert JavaScript to Python

```
def fixup(str):
   newStr =  str.replace("oldRow","old_row",20)
   newStr = newStr.replace("logicContext","logic_row",20)
   newStr = newStr.replace("log.","logic_row.log.",20)
   newStr = newStr.replace("var ","",20)
   newStr = newStr.replace("// ","#",200)
   newStr = newStr.replace("createPersistentBean","logic_row.new_logic_row")
   newStr = newStr.replace(";","",200)
   newStr = newStr.replace("?"," if ",400)
   newStr = newStr.replace(":"," else ",400)
   newStr = newStr.replace("} else {","else:", 100)
   newStr = newStr.replace("}else {","else:", 100)
   newStr = newStr.replace(") {","):",40)
   newStr = newStr.replace("){","):",40)
   newStr = newStr.replace("function ","def ",40)
   newStr = newStr.replace("} else if","elif ")
   newStr = newStr.replace("}else if","elif ",20)
   newStr = newStr.replace("||","|",20)
   newStr = newStr.replace("&&","&",20)
   newStr = newStr.replace("}else{","else:", 20)
   newStr = newStr.replace("null","None",40)
   newStr = newStr.replace("===","==",40)
   newStr = newStr.replace("}","",40)
   newStr = newStr.replace("else  if ","elif", 20)
   newStr = newStr.replace("true","True", 30)
   newStr = newStr.replace("false","False", 30)
   newStr = newStr.replace("if (","if ", 30)
   newStr = newStr.replace("):",":", 30)
   newStr = newStr.replace("logic_row.verb == \"INSERT\"","logic_row.is_inserted() ")
   newStr = newStr.replace("logic_row.verb == \"UPDATE\"","logic_row.is_updated()")
   newStr = newStr.replace("logic_row.verb == \"DELETE\"","logic_row.is_deleted()")
   # SysUtility ???
   return newStr.replace("log.debug(","log(",20)
```

Rule Transform

Each rule has a specific definition. For rules that use JavaScript - the conversion to Python as a defined expression is shown below.

Example generated rule

```
def fn_validation_74(row: models.targetFirstLevel, old_row: models.targetFirstLevel, logic_row: LogicRow):
     if row.isAccrualAdjustable:
        return (row.targetForAccrualAdj == 4 |  row.targetForAccrualAdj == 5)
    else:
        return True

'''
     Title: Validation: if (row.isAccrualAdjustable)
     Name: validation_74
     Entity: targetFirstLevel
     Comments: None
     RuleType: validation
'''
     Rule.constraint(validate=models.targetFirstLevel, calling=fn_validation_74, error_msg="You can only target Revenue or Expense accounts.  Enter 4 for Revenue and 5 for Expense.")
```

SAFRS Client Nested Documents (Resources)

SAFRS JSON should handle a nested call (see curl example below)

```
$curl - x GET \
'http://localhost:5656/api/Customer/ALFKI/?include=OrderList%2COrderList.OrderDetailList%2COrderList.OrderDetailList.Product&fields%5BCustomer%5D=Id%2CCompanyName%2CContactName%2CContactTitle%2CAddress%2CCity%2CRegion%2CPostalCode%2CCountry%2CPhone%2CFax%2CBalance%2CCreditLimit%2COrderCount%2CUnpaidOrderCount%2CClient_id' \
  -H 'accept: application/vnd.api+json' \
  -H 'Content-Type: application/vnd.api+json'
```

LAC Resource - Server Side CustomEndpoints (D - directory F - file)

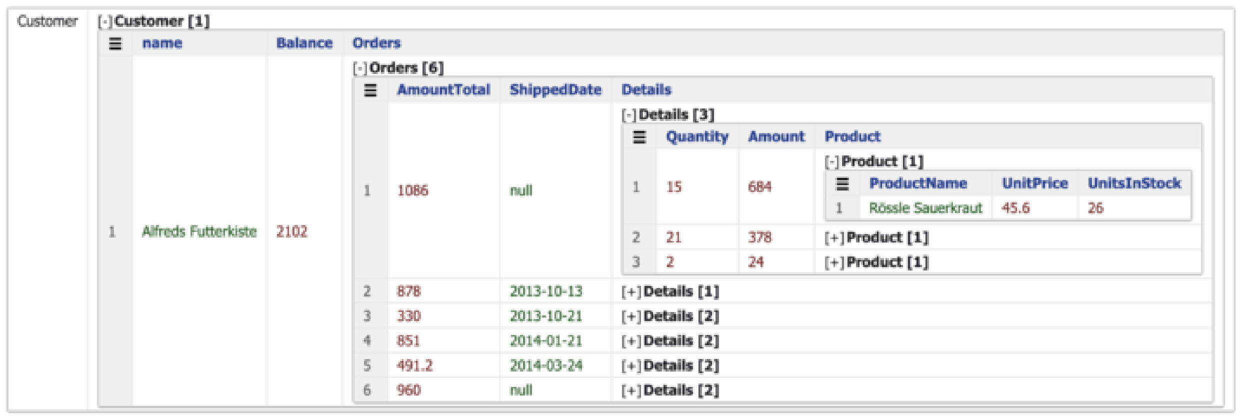
```
-- D v1
--- D Customers
--- F Customers.json Entity: customer  Attrs: (name,balance,credit_limit))
------ D Orders
------ F Orders.json Entity: PurchaseOrder Join: ("customer_name" = [name])
       Attrs: (order_number,amount_total,paid,notes))
--------- D LineItems
--------- F LineItems.json Entity: LineItem Join: ("order_number" = [order_number])
             Attrs: (product_number,order_number,qty_ordered,product_price,amount))
------------ D Product
------------ F Product.json Entity: product Join: ("product_number" = [product_number])
              Attrs: (name,price,product_number))
```

CustomEndpoint Nested Document Style

The ability to execute a custom nested document has been completed using ALS.

```
    @app.route('/customers/<pkey>', methods=['GET''])
    def customers(pkey):
        root = CustomEndpoint(models.Customer,"Customers"
         ,fields=[ (models.Customer.Name, "Name"), (models.Customer.Balance, "Balance"),
                  (models.Customer.CreditLimit, "CreditLimit")]
         ,children=CustomEndpoint(model_class=models.PurchaseOrder,alias="Orders"
         ,join_on=models.PurchaseOrder.name
         ,fields=[ (models.PurchaseOrder.OrderNumber, "OrderNumber")
                , (models.PurchaseOrder.TotalAmount, "TotalAmount"), (models.PurchaseOrder.Paid, "Paid"),
                 (models.PurchaseOrder.Notes, "Notes")]
             ,children=CustomEndpoint(model_class=models.LineItem,alias="LineItems"
             ,join_on=models.LineItem.order_number
             ,fields=[ (models.LineItem.ProductNumber, "ProductNumber"),
                    (models.LineItem.OrderNumber, "OrderNumber"), (models.LineItem.Quantity, "Quantity"),
                     (models.LineItem.Price, "Price"),
                    (models.LineItem.Amount, "Amount")]
                 ,children=CustomEndpoint(model_class=models.Product,alias="Product"
                 ,join_on=models.Product.product_number
                 ,fields=[ (models.Product.Name, "Name"), (models.Product.Price, "Price"),
                        (models.Product.ProductId, "ProductId")]
                 ,isParent=True
                 )
             )
         )
        )

       result = root.execute(request, pkey)
       # or use the safrs jsonapi
       #return root.get(request,"OrderList%2COrderList.OrderDetailList%2COrderList.OrderDetailList.Product",
 pkey)
     return root.transform('LAC','',result)
```


Customer [-]Customer [1]

	name	Balance	Orders
1	Alfreds Futterkiste	2102	[-]Orders [6]

[-]Orders [6]

	AmountTotal	ShippedDate	Details
1	1086	null	[-]Details [3]
2	878	2013-10-13	[+]Details [1]
3	330	2013-10-21	[+]Details [2]
4	851	2014-01-21	[+]Details [2]
5	491.2	2014-03-24	[+]Details [2]
6	960	null	[+]Details [2]

[-]Details [3]

	Quantity	Amount	Product
1	15	684	[-]Product [1]
2	21	378	[+]Product [1]
3	2	24	[+]Product [1]

[-]Product [1]

	ProductName	UnitPrice	UnitsInStock
1	Rössle Sauerkraut	45.6	26

NOTE: join_on is either a field (models.Customer.Id) linked to the parent primary key or a tuple of fields or explicit parent/child pairs [(models.Customer.Id, models.Orders.CustomerId),...]

Security

This is a simple listing of users and rules (A - all, R - read only, N - none) - each user can have 1 or more Roles, the UserRole defines specific access rights. The Grant object is on each entity for a given role and can include specific rights (can_insert, can_update, can_delete) as well as a filter (additional where clause called on select).

```
=========================
     security
=========================
class Roles():
        backoffice = 'backoffice'
        Readonly = 'Read only'
        Fullaccess = 'Full access'

#TablePermissions: R description: read only on role
Grant(on_entity=”R”, can_read=True, to_role=Roles.backoffice)

#Access Levels: ['ALL'] description: ActorTempates CRUD
Grant(on_entity=models.ActorTemplate ,can_update=True, can_insert=True, can_delete=True
to_role=Roles.backoffice)
```

CLI Testing

There is a new command line tool for API Logic Server testing found here. This will allow the developer the ability to login to a secure server (save the JWT token) and perform GET, PATCH, POST, and DELETE via command line. The Model Migration Service ‘tests’ section will create test scripts for both LAC resources and LAC tables.

```
$als login http://localhost:5656 -u u1 -p 1 -a northwind
Logging in...
Login successful, JWT key will expire on: 2023-11-18T15:03:37.342Z


$als get "apl/Employee" -k 1 -m json
```