Building my first Espresso Logic Application

I have been developing enterprise web applications for many years. I have used a variety of languages, tools, technologies, platforms, COTS, open-source packages, and rules-engines. Each of these has a place in the enterprise architecture. I was surprised to see that I could take an idea to a completed and running HTML5 web application using a single vendor, Espresso Logic in a few hours. I started with the basic banking data model and built a very simple design (figure 1).

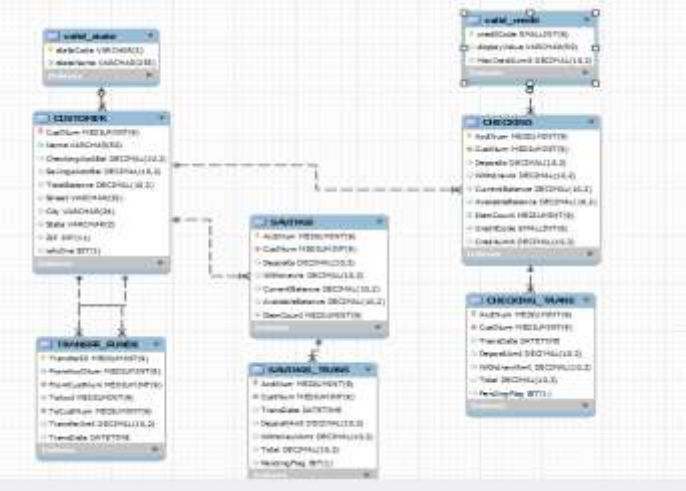

Figure 1 – Banking Schema

SQL Setup

With model in hand, I went to the Espresso Logic home page and created a new evaluation account (http://www.espressologic.com/pricing-plans/). In a minute or two I had a new cloud-server appliance. They also gave me a link to a PHP admin tool for MySQL to import my SQL Script. I found that I had to re-create my model several times to correct a few relationships, attribute names, and indexes.

My first question is where to begin. Using the email from the registration - I started by logging on the Logic Desgner (a web based studio) -this is the main admin tool for editing and maintaining the entire system. The left hand menu was organized by categories; Run, Structure, Live API, Live Logic, Runtime, and Security.

Database & Schema

So first things first, was to point the database to my new tables. Then I looked at the Structure->Schema and saw all my tables, columns, foreign keys, and indexes had been extracted into an object model. Nothing for me to do here. The Project allowed me to change the name and URL Fragment (we will see this later in our REST API). More importantly, there is a place to modify the Authentication provider (security system) as well as the ability to import JAR files and JavaScript to extend my application features. It turns out there is a GitHub account (http://github.com\EspressoLogicCafe) with many pre-built authentication services for LDAP, SQL, REST, Stormpath, and Windows AD.

Rules/Reactive Logic

Next came the Reactive /JS – the rules. I knew what I wanted to do so I began at the top with Customers. My spec had 3 spreadsheet rules and a formula. The logic code is not compiled or generated it is expressed with a series of dialogs and on-screen field helpers that made this easy. I then went down

the list of specifications to add sums, counts, formula to the other entities. Finally, I needed to put some validations in so that you could not overdraw the account or enter both a deposit and withdraw on the same transaction - it took a couple of tries to see that it has to return TRUE to succeed and return FALSE to see the error message. You can even insert the current attribute names as values into your message using {attrName}. Once I defined my rules – The Rules Editor displays them almost exactly as my specification – below is the logic I entered for the CHECKING account table.

- CHECKING
- Derive AvailableBalance as sum(CHECKING_TRANSList.Total where PendingFlag = 0)
- Derive CurrentBalance as sum(CHECKING_TRANSList.Total)
- Derive Deposits as sum(CHECKING_TRANSList.DepositAmt)
- Derive Withdrawls as sum(CHECKING_TRANSList.WithdrawlAmt)
- Derive ItemCount as count(CHECKING_TRANSList where WithdrawlAmt > 0)
- Derive CreditLimit as parentcopy(fk_checking_valid_credit1.MaxCreditLimit)
- Commit validation if(row.AvailableBalance < 0){ return false;;

The last piece of the puzzle was how to insert a checking transaction and savings transaction so I used an Event and wrote a few lines of JavaScript. The Espresso Logic folks have provided a nifty utility called **logicContext** that performs many different tasks. I found that calling **logicContext**.createPersistentBean('CHECKING_TRANS') and then populating with the from/to accounts/amounts etc I could simply do a logicContact.insert on the new bean. Easy – and the cool thing is that it fired all the validations, sums, counts, formula all the way up to the top of my tree Customer.

```
var fromAccount = logicContext.createPersistentBean(fromBean);
fromAccount.CustNum = row.FromCustNum;
fromAccount.AcctNum = row.FromAcct;
fromAccount.WithdrawlAmt = row.TransferAmt;
fromAccount.DepositAmt = 0;
fromAccount.PendingFlag = 0;
logicContext.insert(fromAccount);

var toAccount = logicContext.createPersistentBean(toBean);
toAccount.CustNum = row.ToCustNum
toAccount.AcctNum = row.ToAcct
toAccount.DepositAmt = row.TransferAmt;
toAccount.PendingFlag = 0;
toAccount.WithdrawlAmt = 0;
logicContext.insert(toAccount);

return true;
```

Figure 2 – insert into 2 different tables (from account, to account) using JavaScript

Live Browser

Time to test using a built-in HTML5 application called Live Browser. The default is to display a 3 pane window with a Grid/Scalar/Tab panels. So I entered a new Customer, a new Checking and Savings account, entered opening transaction deposits to both and then was ready to test. I tried my balance transfer and discovered my rules were not right. I simply jumped back to the LogicDesigner and fixed the rule and went back to the Browser to try my save again. Being able to see in real-time the balances change and the validation constraints fire is amazing. Also, being able to go back and forth between the data and the rules and realize that there is no middle-man, no generators, and only 8 lines of JavaScript is a real boost to enterprise development productivity.

LiveBrowser is more than just demo-ware – it has an author mode that allowed me to hide/show fields, change formatting, and field labels, as well as rename tab panels (based on foreign key roles). I can take this secure and integrated application to my development team and my stakeholders and get feedback on business logic, content, navigation, formatting, etc. In Figure 3, the automttically displayed page included search filters, insert mode, fetch more, drill down, tab panels for related joins, popup selections (picks), and more. Not bad for an out-of-the box HTML5 display.

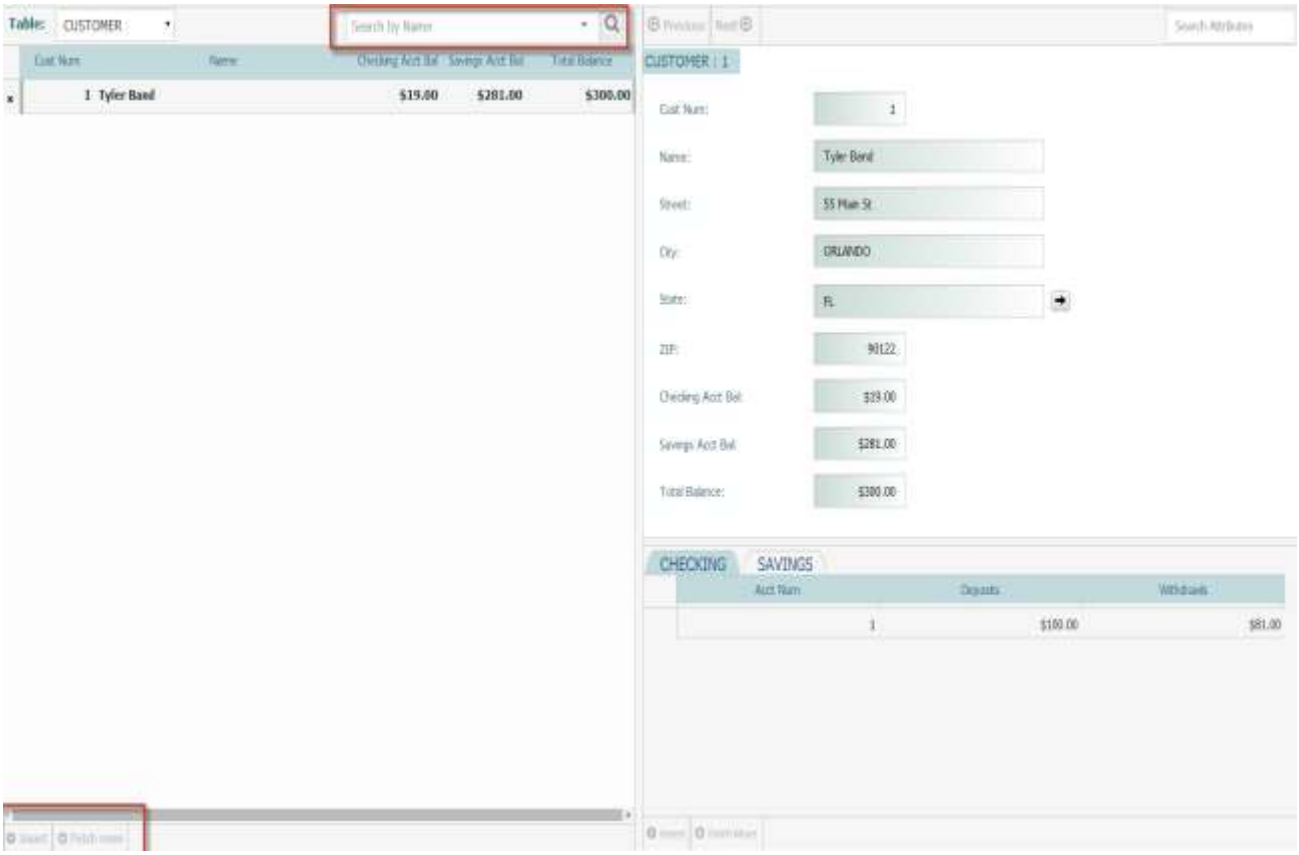

Figure 3- Live Browser

Live API

One of the features that is overlooked is the ability to create custom REST API **Resources**. These resources can be based on the existing schema (e.g. Customer with the last 30 days of Checking Account Transactions). I also have the ability to merge resources from other internal and external systems (e.g. reading a message queue, making a REST call based on existing parent data, calling another web service). Figure 4- shows the resource editor that provides a point and click way to create compound

resources with automatic joins. The Attribute tab let me select only the columns I wanted to send back to my mobile user and even alias the attribute name.

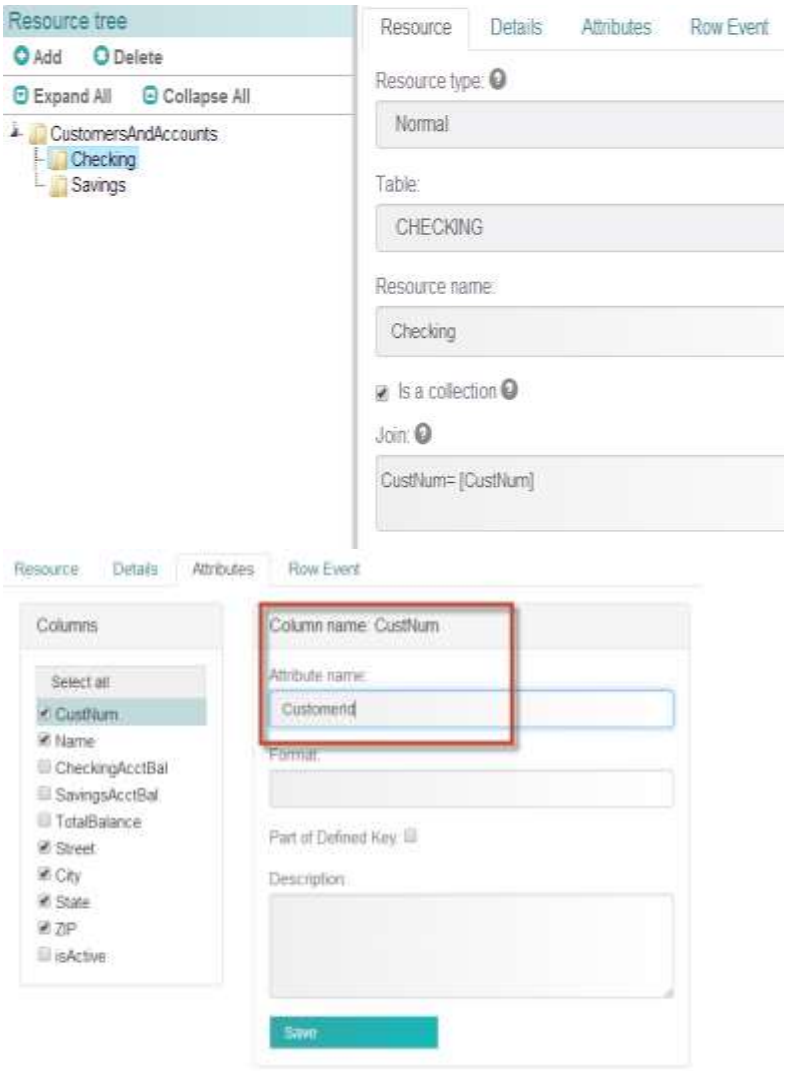

Figure 4- Custom Resource Definition and Column Selection and Attribute Alias

REST Lab

The REST Lab allows developers to test and see the REST API GET/PUT/POST/DELETE as well as debug rules. Since these rules are written in JavaScript – a built-in debugger is provided in the REST Lab to step through various rules state changes. I also noticed that the GET returned only 20 rows of my sample database (but provided me a link to fetch for the next batch – this is adjustable) and each JSON dataset had an optimistic locking hash that was used to determine if someone else modified my data before update. Figure 5 – shows the result of my new resource in the REST Lab. This is a lot of code I did not need to write.

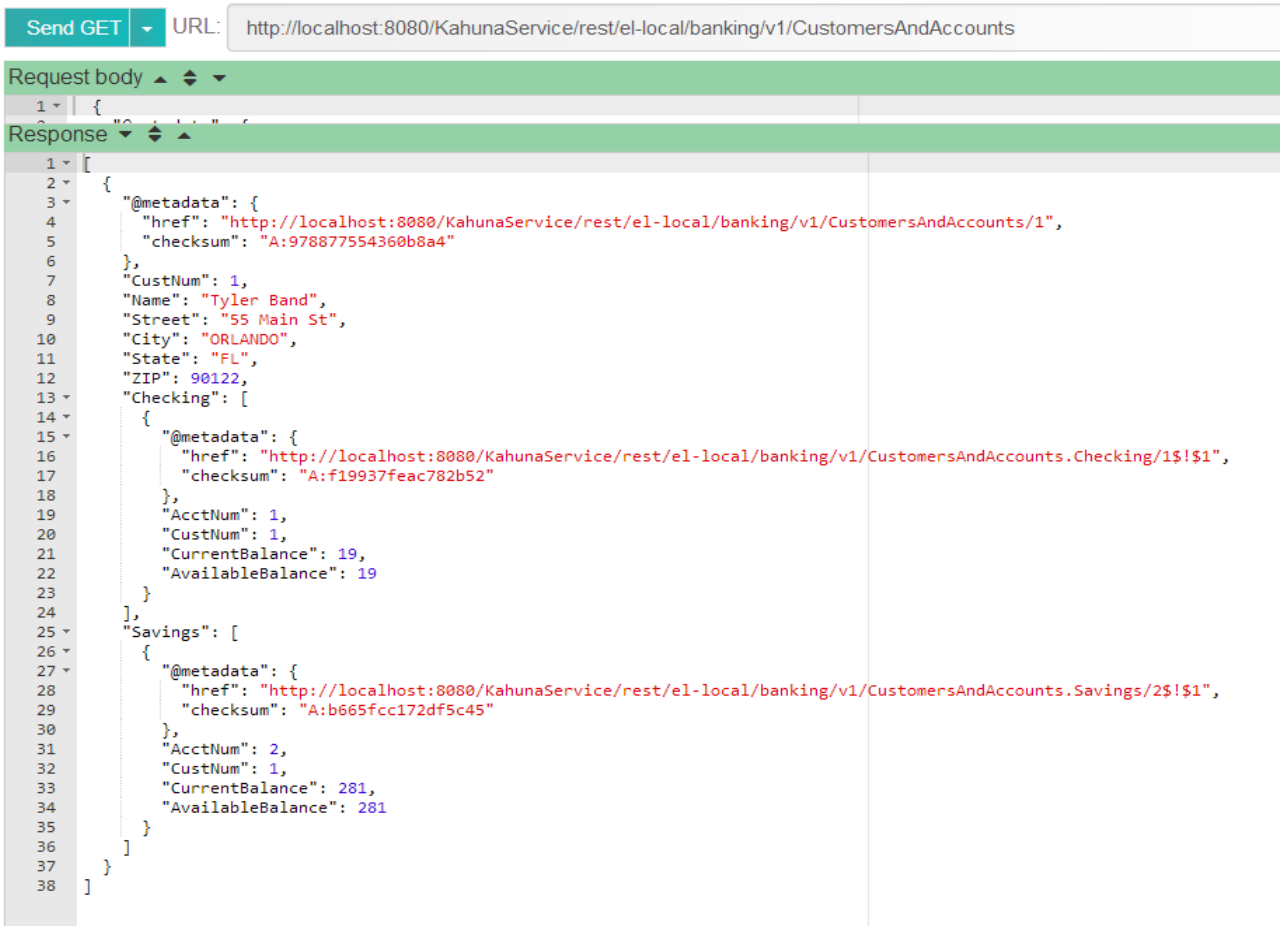

Figure 5 – REST Lab – test my new resource

Security

I need to talk a bit about security. I setup a new Role called Tellers that allowed them to insert new customers, accounts, and transfer funds. They were not allowed to delete funds. The user and role security that is built in is very fine grained and I could have restricted the tellers down to the row level (for example, they could not see private business accounts or customers). See figure 6. There is a lot of details that can be found in the documentation on security and access control.

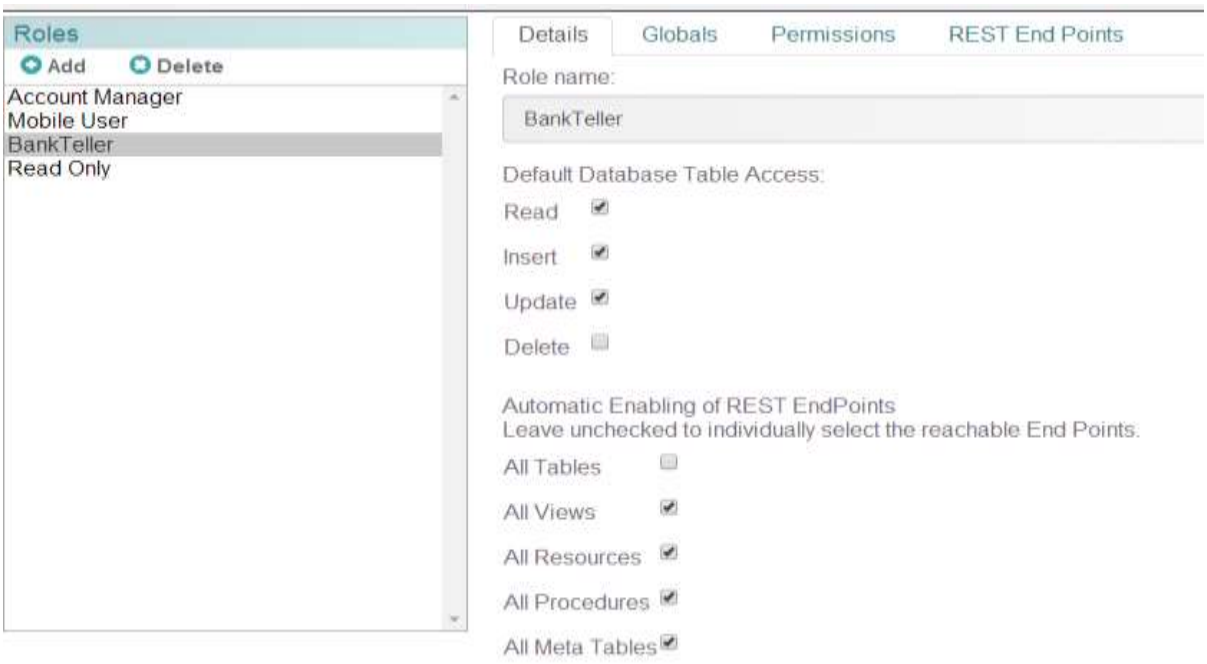

Figure 6- Role based Access Control

Logging

Finally – the logging feature allowed me to see the SQL that was called, the rules that were processed and the order of the firing (dependency) of each rule. I did not have to tell the system the order of my rules- it figured this out at runtime. The other thing I noticed is that it adjusted sums and counts - a very huge performance boost over doing an SQL sum. See Figure 7 to see how complex a simple funds transfer is and how optimized and dynamic the rules appear.

Figure 7- Logging and Tracing Rules

Behind the Scene – Architecture of Espresso Logic

I did a lit bit of investigation and found that Espresso Logic runs both as a cloud service on Amazon and Azure as well as on-premise as an appliance. The layered architecture made it easy to understand how complete a packaged solution they are providing. Extensible authentication services, event handlers, REST API, Business Rules and JavaScript Data Integration, complete persistence services to SQL. I did a little bit of experimentation with using MongoDB for mixing and matching SQL data and NoSQL using the JavaScript Resource editor (this will be in my next article).

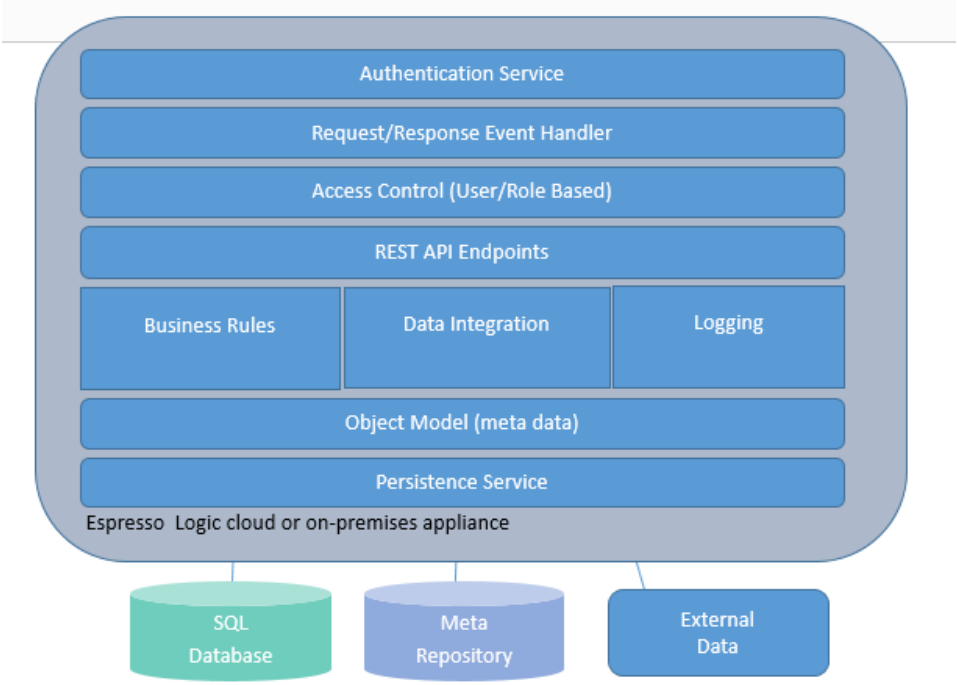

Figure 8 - Architecture

Summary

I have seen a lot of tools, appliances, open-source technology that claims to do similar things but this is the only one that actually delivers a complete solution for building enterprise mobile applications. The rules engine, the instant visualization and testing, and the simplicity of the JavaScript code makes this an excellent choice to add to my approved list of solutions.

Author: Tyler Band – Customer Advocate Engineer for Espresso Logic, Inc. mailto: tyler [at] espressologic.com