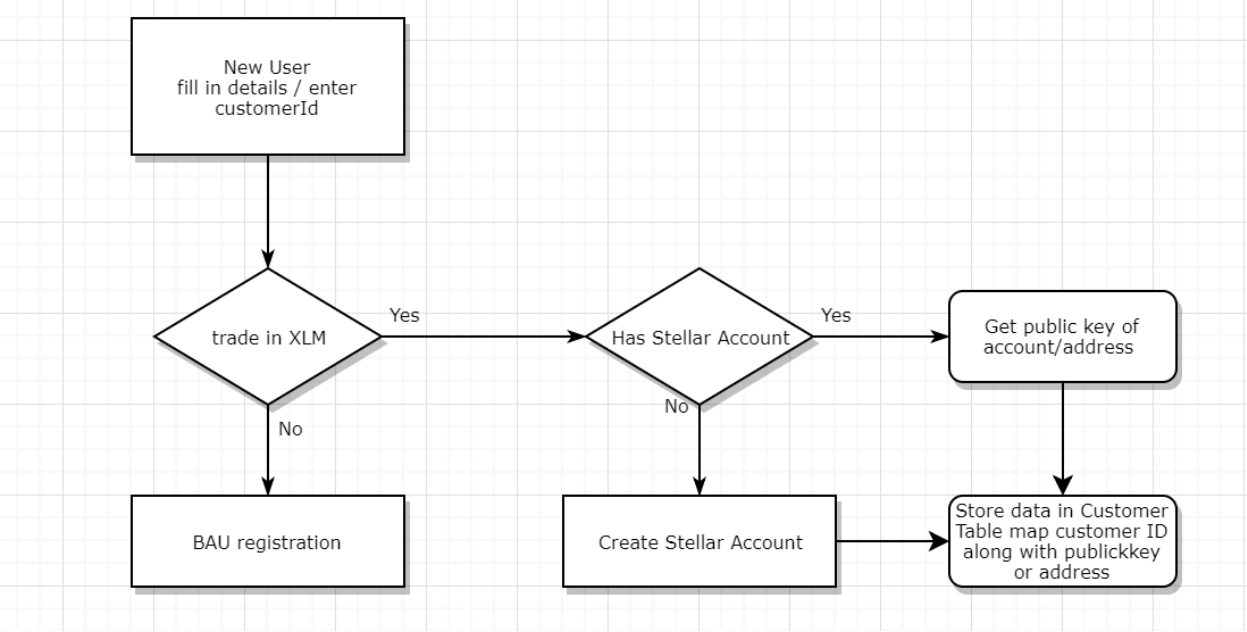
**Introduction**: This article is a sample guide through to add stellar lumens to the existing exchange.

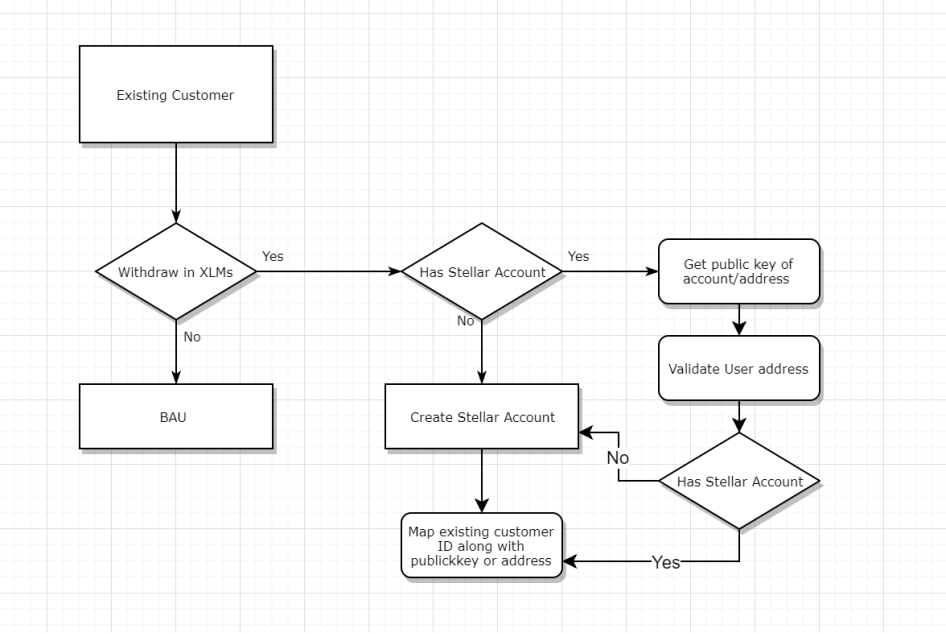
**Assumption**: Existing exchange is a web app and has a built in set of customers and a mechanism to add new customer.

## A. Customer Registration:

**Scenario1:** Customer interested to trade in XLM.(Deposit/Trade in XLMs).



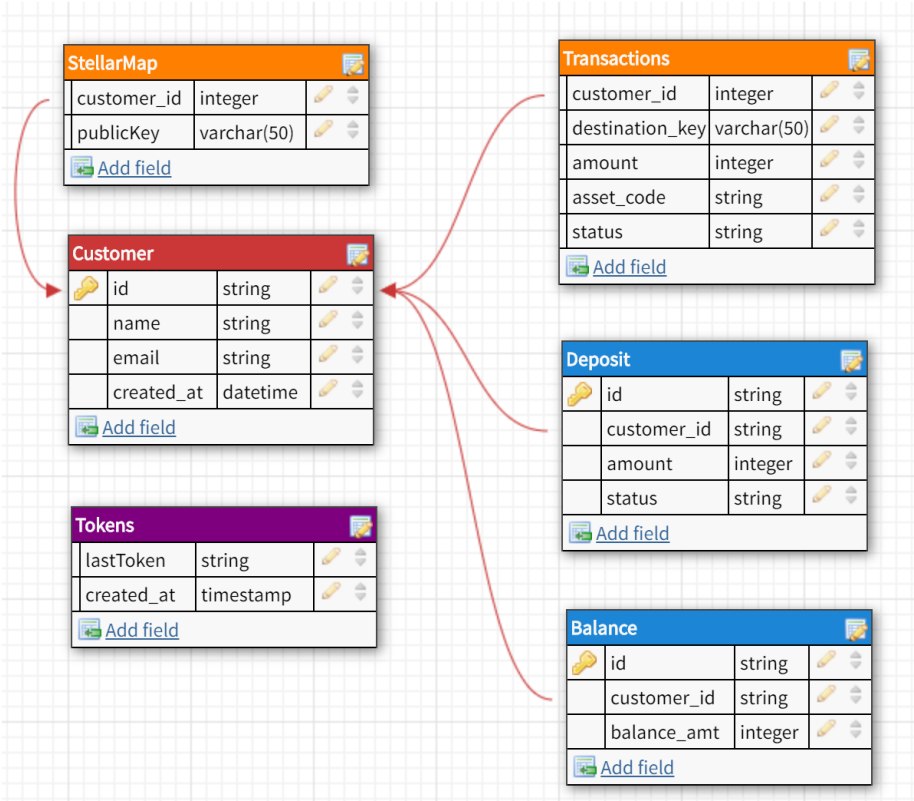

**Scenario2:** Existing customer would like to withdraw in XLMs



**B. Database Design :** We need below tables on high level , right now the approach is with relational DBs to keep it simple but I would like to mix up the NoSQL and SQL , NoSqls especially to maintain the transactions queue and SQLs to aggregate for statistics , filtering etc.

1. **StellarMap** : Each Customer has a unique customerId which is central to our exchange , this table maps the customers stellar account addresses or public keys for easy access when they withdraw, validate or create complex transactions.
2. **Customer**: I assume this would be an existing database in exchange with all customer centric details along with a customerId centric to our exchange.
3. **Deposits**: Customer Deposit table when they like to trade in XLMs , this table stores all the customer deposits credited to exchange stellar account for trade as a wallet data, this can be scaled up to other assets in future (keep it simple for now).
4. **Transactions**: All Customer withdrawals requests when they like to be paid in XLMs. This table also acts a queue for immediate transactions to sit here until the processing has been done to stellar networks.
5. **Balance**: Quick reference to check Customer balance in lumens (can be scaled to other assets)
6. **Tokens**: This table holds the cursor position of exchange stellar account to remember the last deposit tokens. (look further details in detailed architecture).

Here is the high level DB diagram for better understanding:



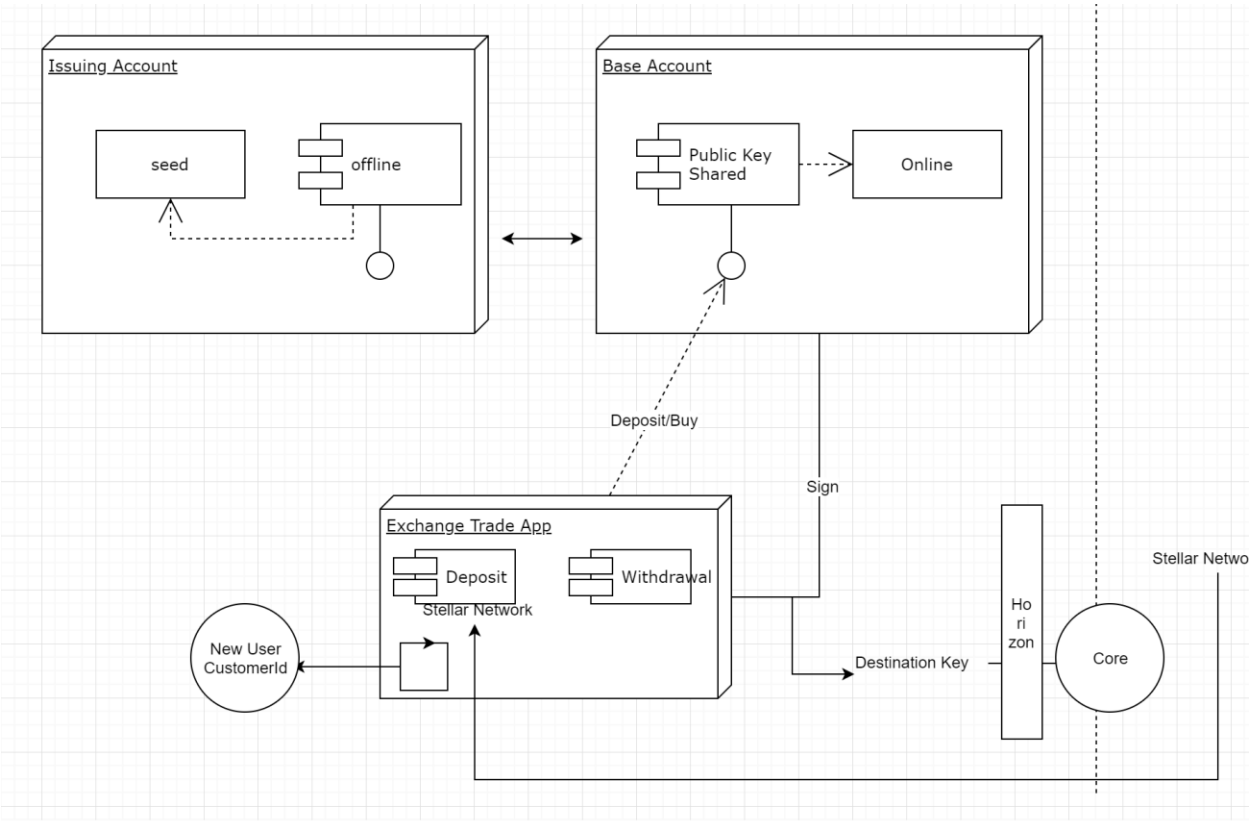
## C. Exchange Architecture:

In order to integrate stellar lumens with our exchange , we need some stellar accounts so that customers would use this to deposit lumens in our exchange , these accounts serves as wallets for the XLMs which customers can use to trade or withdraw.

Section 1: We need at the basic level 2 accounts :
one serves as **Cold Wallet** & one **Hot Wallet** but there can be more than one accounts to manage high transaction rate (this concept ***is Channels*** which uses different source account for operations vs source account used in transactions).

**Account 1 – Source/Issuing Account** – Offline exchange account(stellar) which has maximum deposits of all customers and maintained manually to be freed from hack.
**Account 2 – Base Account** – Immediate account(stellar) used for transactions which has limited customer funds to process withdraws deposits etc.

**Diagram below explains the basic interaction , we could see Exchange Trade App which has some basic functionalities like Deposit, Withdraw and Registration functionalities :**




**Integrations :** Exchange integrates with Stellar network in following cases

1. When Deposit payments in lumens or XLM made by customers to exchange base account.
2. When Customer requests withdrawal for Lumens from exchange.

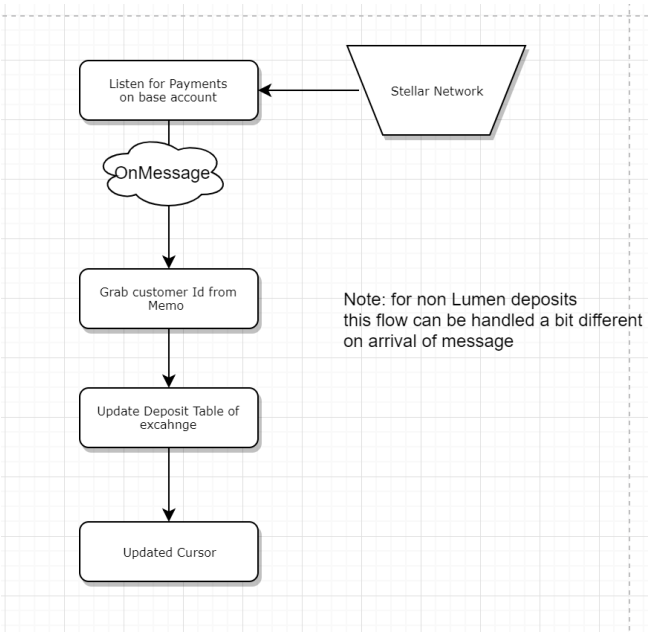
**Setup:**

1. Exchange connects to Stellar Core through Horizon API .
2. Stellar Core can be a public core provided by stellar.org or can be our one own node which follows SCM protocol which follows consensus of stellar protocol.
3. Stellar Network is collection of Stellar Cores worldwide which is distributed and works on transactions and has a Ledger.

**Deposits :**

1. Our Exchange provides the functionality to users to deposit lumens -> When user wants to deposit XLMs into exchange so they can trade lumens.
2. User deposit lumens where they need to enter public key of base account of exchange and add a memo with their unique exchange customer Id.
3. Memo is the transaction remarks to identify any remarks which in our case we use to identify customers through Id.
4. Exchange listens for all deposit payments through stream which is event based where we can execute functionality to handle deposits **onmessage**

JS -> stellar-SDK provides API to handle this as per Stellar documentation:




**Sample Code:**

```
var StellarSdk = require('stellar-sdk');

// Start listening for payments from where you last stopped

var lastToken = get token from Tokens Database.
```

// GET https://**{{horizon-api}}**/accounts/{baseAccountAddress}/payments?cursor={lastToken}

let deposits = server.payments().forAccount(config.baseAccount);

if (lastToken) {

deposits.cursor(lastToken);

}

deposits.stream({onmessage: function(record){

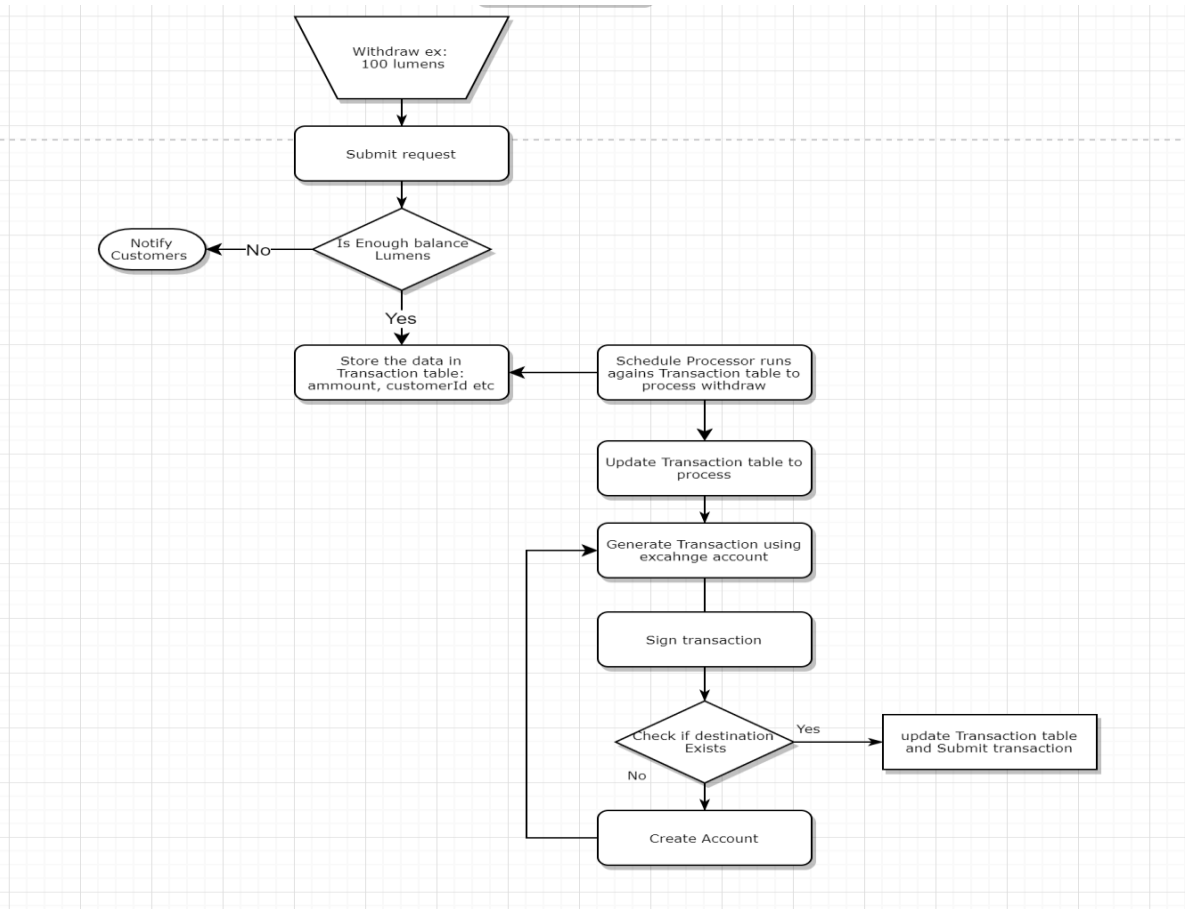
//get transaction from record recored.transaction() -> this returns promise with a record data

Record data if of type {}


}});

**Withdraw:**

1. Generate stellar transactions when user initiates to withdraw XLMs from exchange.
2. We can use the Queues to store all withdrawals in **Transaction** Table.
3. Process all transactions from **Transaction table** periodically or on arrival to stellar network to release XLMs to customer accounts.
4. Each request can be validated by checking if customer has enough balance of XLMs from **Balance Table**




**MultiSignature:**

**At any point of time the transaction can be authorized by more than one seed if customers would like to.**

**We have to invite or let customer add other users with public key who can be added to trust line to add multisig.**

**Few Points to consider:**

1. Channels can be scaled up on high transaction as described in section 1 by having a different store account for operation and transactions.
2. Fault tolerance can be handled by having the transactions Queue hosted in Azure Cloud Queues which provides high level of scale for high rate transactions.
3. We can have unique customerIds for each trader or user which can be scaled to unlimited by having a map to the respective Stellar accounts rather than random creation of Stellar accounts.