Design QR Code Generator for a Grocery Store

What's a QR Code System?

QR (quick response code) is a machine-readable 2-dimensional barcode consisting of an array of black and white squares, typically used for storing URLs or other information for reading by the camera on a reader e.g smartphone devices.

- QR code payment is one of the contactless payment methods to transfer funds from the buyer's wallet to the seller's wallet or account. here Payment is performed by QR code at POS from the mobile app.
- Shoppers only need a mobile phone with a camera and a mobile payment app to scan a QR code and pay. It is quick and easy to use and has increased convenience and security benefits.

System Requirements

Let's start with building a simpler version of the QR code payment transaction system. There are two types of users in our system: 1) Sellers 2) Buyers (Customers).

- Buyer and seller registers in the system.
- The seller downloads the QR code and displays it in front of his store.
- The buyer goes to the store and purchases items. (Stored in Buyer transaction table)
- Buyers scan the QR code displayed in the store through the phone app.
- The buyer enters the amount and clicks OK to proceed.
- Fund transferred from Buyer to Seller
- SMS notification sent to Seller and Buyer for confirmation

- Transaction completes.
- Extended: What changes do we need in our system to extend the system for dynamic QR code (Not in scope).

Use cases

The system should allow these basic use cases:

1. **Create QR code**: We need to create a unique QR code for each seller and provide it to the seller which can be displayed at the store. A customer would use this unique QR code to scan using his/her mobile camera through an app and make the payment to the seller account/wallet.
2. **Read QR code**: A customer at the grocery store would use his/her mobile phone camera to scan the QR code displayed at the grocery store. This scanned image needs to be decoded to get the seller information to which the customer's payment would go to.
3. **Register buyers and sellers**: Both buyer and seller would have to register themselves first in the system.
4. **Transfer fund from buyer to seller**: The system needs to transfer funds from the buyer's wallet to the seller's wallet.

Database schema

We need to store data about sellers of a grocery store, their wallets, and transactions, buyers, wallet, and buyers' transactions.

Ref: https://dbdiagram.io/d/5f8cc7e53a78976d7b7820fd

Seller Schema

- Seller Table: SellerID, SellerName, TaxId, Phone, Email, AC_Number, BankName, BankDetails
- Seller_wallet Table: SellerWalletID, SellerId, Balance
- Seller_transaction Table: ID, BuyerId, PurchasedItemID, Amount

Buyer Schema

- Buyer Table: BuyerId, BuyerName, Phone, Email, BankName, bank details
- Buyer_Wallet Table: BuyerWalletId, BuyerId, Balance
- Buyer_Transaction Table: ID, SellerId, BuyerId, PurchasedItem, Amount, Status

A straightforward approach for storing the above schema would be to use an RDBMS like MySQL since we require joins. But relational databases come with their challenges, especially when we need to scale them. For details, please take a look at SQL vs. NoSQL.

Capacity Estimation and Constraints

The number of sellers is significantly less than that of buyers. Let's assume:

- There is a 1:10 ratio for sellers to buyers.
- We have 500M total users(buyers), and 100M daily active users(buyers).
- On average each user makes 2 transactions/day using QR payment methods at different grocery stores. So a total of 200M transactions/day.
- The size of each entry in the transaction table is 1KB. This would require us 200M*1KB = 200GB of total storage per day.

Size of transaction table to store 10 year data = 200GB*365*10 = 730TB

Size of buyers table is 500M (total users)* 1KB (data per user)= 500GB total.

Key Service APIs

APIs for Seller Application

- RegisterSeller: Each seller needs to register itself before it's able to use the system. This API would create a database entry for sellers in the seller schema or return an error.

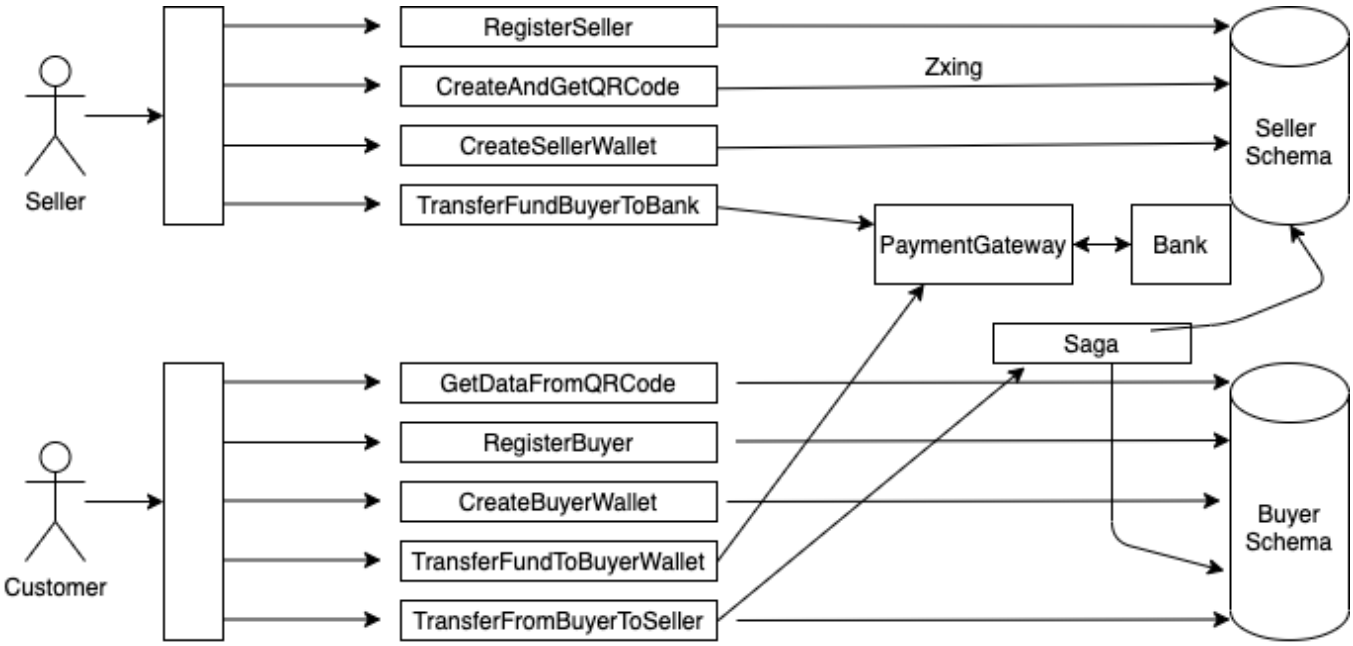
- CreateAndGetQRCode: The API will create and provide a unique QR code to the seller which contains the seller data and can be read by a QR code scanner.
- CreateSellerWallet: This API is responsible to create a wallet for the seller where money from the buyer would go.
- TransferFundBuyerToBank: This API will connect PaymentGateway to transfer money from buyer wallet to bank account.

APIs for the Buyer Application side

- GetDataFromQRCode: This will read and decrypt the QR code and get the data into the buyer's app.
- RegisterBuyer: This will be used to register a customer and update the buyer's schema table with the information provided in the API.
- CreateBuyer Wallet: This will create the buyer's wallet schema and store phone number, email, address, bank details.
- TransferFundToBuyerWallet: This API will connect PaymentGateway to transfer funds from the buyer's bank account to its wallet.
- TransferFundFromBuyerToSeller: This API will createSaga. It will also update the seller's and buyer's schema and complete or abort the transaction sending success or failure code to the buyer's application.

High-Level Design

Buyers and Sellers would use a relational database schema. We would use a payment gateway to transfer funds to or from buyers' bank accounts. We would also use a saga pattern for the distributed payment transaction.



Transfer Fund From Customer to Seller

We would use the saga pattern here. A saga is a sequence of local transactions where each transaction updates data within a single service.

- We would use an API TransferFundFromBuyerToSeller that first create a Pending status in the buyer_transaction table in buyerSchema.
- It then creates createSaga in parallel. This tries to go into the seller schema and adds money to the seller's wallet.
- CreateSaga sends the command to the seller_wallet table to update the wallet Amount +sales amount
- createSaga gets messages as updates*pass or update*fail.
- If update*pass then a) buyer*transaction table in buyer schema is updated b) wallet table is decremented and c) finally pending status is removed.
- If updated*fail then the Buyer*transaction table in buyer schema is updated with "canceled" status.

Scaling the System

CDN: system servers need to deliver many javascript assets, images, etc. Decoupling these assets delivery to CDN can offload work of servers improving server capacity.

Service on Docker and Docker on PODS: The number of requests to the system gradually increases over time. As the traffic increases even during peak hours like evening/morning, we need to handle additional traffic using auto-scaling by adding docker automatically based on CPU usage threshold or just traffic at each server's load balancer level.

Load balancers: we would use load balancers at each key service to distribute traffic using the least conns strategy.

Sharding the database: if we shard the data based on the buyer's id, then we can keep all transactions of a user in the same shard. But as the size of the transaction table is much larger than that of the user's data table, therefore, we can choose to shard the data based on transactionId which can be integer incremented value based on timestamp. This is more useful as we can retrieve transaction data from any user from a specific shard just by looking at the timestamp range of the request.

Thanks to Chiranjeev for his contribution in creating the first version of this content. If you have any queries/doubts/feedback, please write us at contact@enjoyalgorithms.com. Enjoy learning, Enjoy system design, Enjoy algorithms!