Doshii Docs and Process

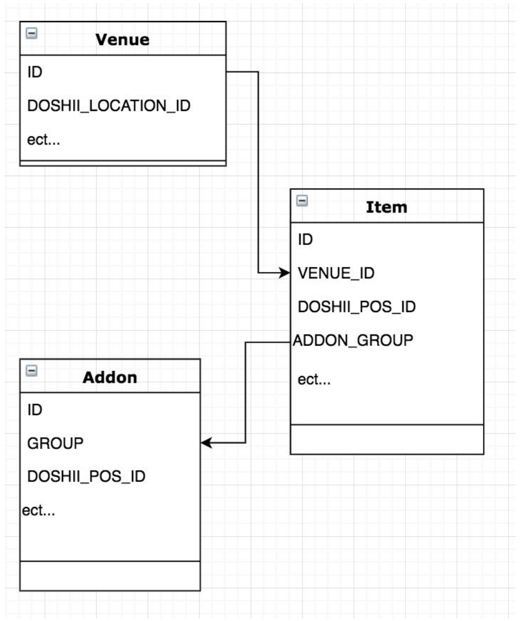
Database Server

AWS Info:
mr-yum-db-prod
ec2-13-54-97-198.ap-southeast-2.compute.amazonaws.com

The Database server is an EC2 instance that runs an express server application that uses the sequelize library to manage a postgres server that sits on the instance. This database is a relational database that processes the Airtable data set into a more robust schema to ensure data integrity and to provide cleaner payloads for the front end application.

On the server there are two databases: **yum_visual_menu** which holds the schema of the airtable data and **yum_ordering** which stores the orders.

- yum_visual_menu (attribute list can be found under src/models)



- yum_ordering

  yum_ordering only has one entity: ORDER. The attributes can be found in src/models/order.js

Routes:

The routes were crafted to reflect basic sql queries; you can get all venues, venues by category, items by venue, all orders, orders by id. You can also insert new orders and update order status. There are other routes listed in app.js, for example to refresh the menu database with the latest airtable data, you can use the /reload route.

Considerations:

Its important to note that there are **two** databases used on this server and as such it requires two standalone sequelize instances. Changes to one instance will only affect that instance. i.e. there are separate {force: true} statements that cause a reload for each. In express.js file, if you want to wipe the database and repopulate set force to true for the one you'd like to wipe. Once the ordering database is in prod, you should not wipe that database ever or you will lose orders. That said you will need to implement a back up of the orders to Airtable for the times you need to update the code for that database or if you push a new version of ordering_bff repo to prod.

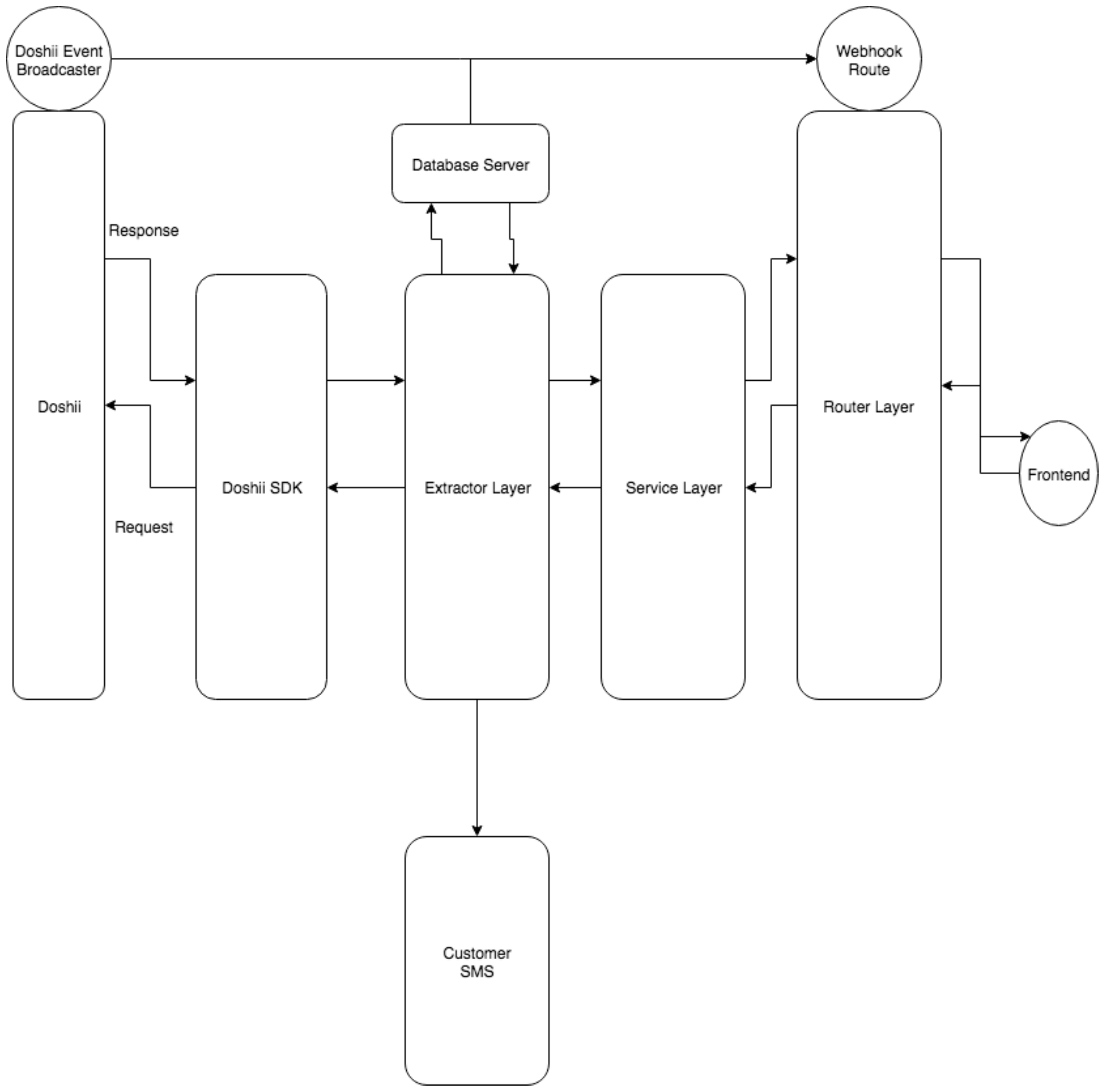
One of the most important tasks of the database is to set a timer after order creation and call the bff webhook when the timer is completed. This is used to set an upper bound on the amount of time an order can be allowed to exist without being accepted by the POS. More on this in the integration portion of the doc.

BFF QA Server

AWS Info:
mr-yum-bff-qa
ec2-54-252-185-155.ap-southeast-2.compute.amazonaws.com

The bff qa server is an EC2 instance that runs an express server application. This application retrieves data from the database server and transforms it for the front end, on request. This application also integrates with doshii and maintains the two-way connection via webhook. This server is responsible for sending new orders to the database, maintaining the status of those orders, and updating the customer about those statuses.

Overall Flow Schematic



Integration

Doshii calls are abstracted away into our own npm module called mryum/doshii-sdk. The purpose of this module is to handle doshii auth, abstract away the repetitive code that is used for every call, and make the bff code cleaner. The sdk does not do any data transformation, but expects payloads to be in a format that doshii can accept. That said, there maybe times where you will need to edit the sdk if doshii updates their api or if anything was missed when DJ created the sdk.

The qa bff has the same structure as the production bff. The router, service, extractor, and transformation layers work, for the most part, in the same way. The calls to doshii primarily sit at the end of the extraction layer, and can be accessed through a route call. Order placement, however, does follow a bit more of a complicated path.

Doshii Dashboard and Documentation

The doshii documentation is, for the most part, thorough. You can find the docs here:
https://support.doshii.io/hc/en-us/categories/115000413493-Partner-App-API

There are two test sandboxes that have full menues with which to work from. You can also view orders that you have placed for testing.

- Fake Test Venue Sandbox
  LocationId: 8KXM0OD4
  Menu Link: https://sandbox-dashboard.doshii.io/venues/8KXM0OD4/menu
  Dashboard Link: https://sandbox-dashboard.doshii.io/venues/8KXM0OD4

- AVC Sandbox
  LocationId: 4rnOjGN4
  Menu Link: https://sandbox-dashboard.doshii.io/venues/4rnOjGN4/menu
  Dashboard Link: https://sandbox-dashboard.doshii.io/venues/4rnOjGN4

Subscribing and Unsubscribing to Locations

To be able to send and recieve data from a particular venue, you need to subscribe to that location. According to Doshii, once you subscribe once, you never need to subscribe again. Both of the locations above have been subscribed to. If you need to subscribe to them again, there is a route to the bff which can do this which will be provided in the postman collection.

Order Creation

- Step 1 – Call from the front end

  /ordering/:locationId/createOrder is called, a payload with the order object is passed from the front end via the body.

- Step 2 – Pass through every layer until the extractor.

  Pass through the service layer and extract the function associated with the CREATE_ORDER intent.

- Step 3 – Attempt to create an order in doshii

  The payload for the doshii order creation is formatted via the createOrderPreprocess function in the doshiiPreprocessors.js file. This is where transactions, addons, and the base order payload are formatted for doshii. Addons will be discussed later on.

  Every order is required to have an externalOrderRef field. We supply the stripe Id as the unique reference. We store orders in our database using the stripeId as the primary key as well.

  A transaction is supplied, method is always credit, and prepaid is always set to true. The reference is the stripeId. Only one transaction is created for the payment and a second transaction will be added in the case of a refund with a negative value. But no order will ever exceed 2 transactions since all orders are prepaid through stripe.

  There are four possible outcomes for order creation:

  1. **The request to doshii times out.**

  On line 75 of orders.js, a timeout is set before any of the order creation code is ran. On timeout, the order failure sms function is called. This timeout means that the network transaction between doshii and the app took too long. Under this condition the order does not need to be canceled because, from Doshii's perspective, it was never actually created.

  2. **The request to doshii is rejected.**

  If the request to doshii does not time out, but the request returns an error (which typically means there was something wrong with the payload). If this happens, the error is caught on line 90, the error is logged to the console, and the order failure sms function is called.

3. **The POS does not accept the order in the specified time.**

If the order creation has thus far proceeded successfully, doshii will return a success payload (lines 79-88), this payload will have the details from the order you've just created and the field 'status' will be listed as "pending". If an order is marked "pending", it means doshii has accepted it as valid but it still has not been accepted by the POS system.

Acceptance by the POS must occur within a certain timeframe, because we can't have the order hanging in limbo for too long from a customer experience perspective. The BFF first stores the new pending order in the database by calling the database server and passing order payload formatted by the function on line 65.

In the database, on line 43 in queryMapping.js, the function associated with the INSERT_NEW_ORDER intent can be found. The order gets inserted into the database (with a STATUS of "pending") and after insertion a time out (recommended at 60 sec) begins. After that time, the database will call the webhook route on the bff to alert it that the order specified in body had been created a minute ago and the bff should check to make sure that order has been updated to "accepted" in that time. The database will pass in the body a field called event. Event will be set to "pending_timeout" to differentiate it from the doshii events.

When the webhook route is called, it will map to the catchWebhook function in webhook.js. The event condition will evaluate to the code starting from line 49. It will call the database for the order and if it is still listed as "pending", the order will be canceled on doshii via the CANCEL_ORDER intent, the status of the order will be updated in the database to "canceled" and the order failure sms function will be called.

In the CANCEL_ORDER intent function, a second transaction will be added to denote a refund.

4. **The order creation is successful.**

If before the time out, doshii had called the webhook route with the event "order_updated" and changed the order status to "accepted" then the database was updated to reflect this and the order success sms function was called.

More Webhook Detail

Webhooks are used for two-way communication between the bff and doshii. It is obvious that when we need to do something with doshii we can call one of their routes. But it is not as obvious what to do when we need to receive up-to-date information from doshii that has will occur at some unknown point in the future. For example, it is unknown when an order will be completed, but we need to know when it happens.

To solve this, doshii allows to us to pass it a route of our own that doshii can call when a particular event is triggered. You cannot pass a webhook for every event type at once. You have to pass a webhook for whichever events you are interested in. The list of events can be found at https://support.doshii.io/hc/en-us/sections/360001531314.

We have created a route to create and cancel webhooks, simply pass the event type as the parameter. (see lines 204 & 219 of orderingRouter.js) You can also get all webhooks you currently have for a specified location(line 235).

NOTE: Doshii requires that create a webhook for each event you want to listen to, for EACH location.

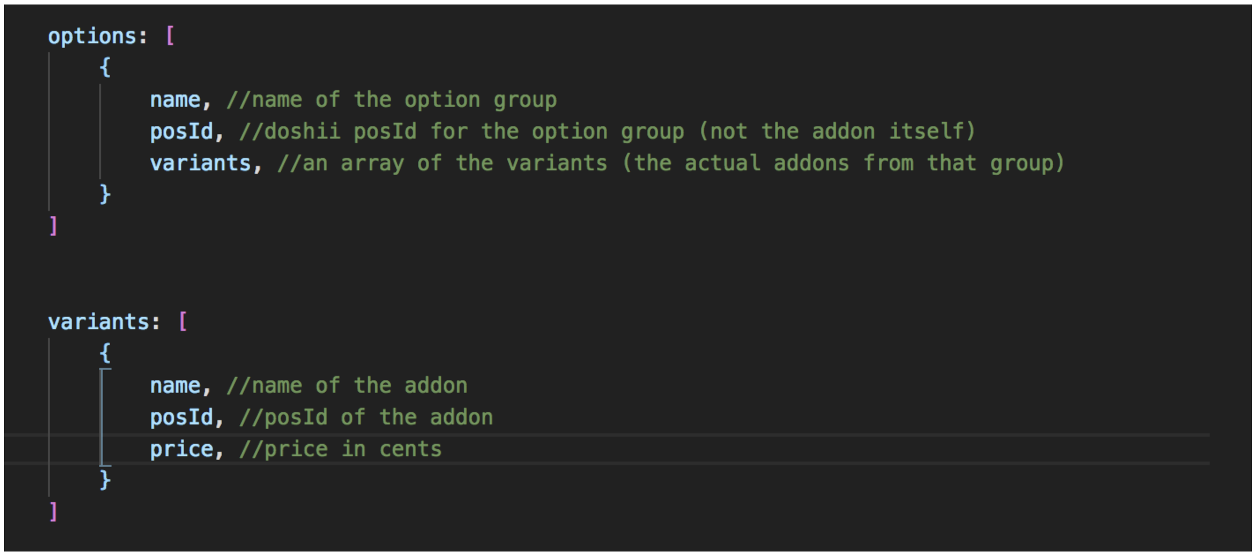
Currently, the bff is only listening for order updates. When an order update event is passed to the webhook route from doshii, it is caught and processed in the webhook.js file, line 21.

- If the order status is “accepted” it updates that order in the database and sends an order success SMS.
- If an order status is “rejected” it updates that order in the database and sends an order failure SMS.
- If an order status is “completed” it updates that order in the database and sends an ready for pickup SMS.

Add-Ons

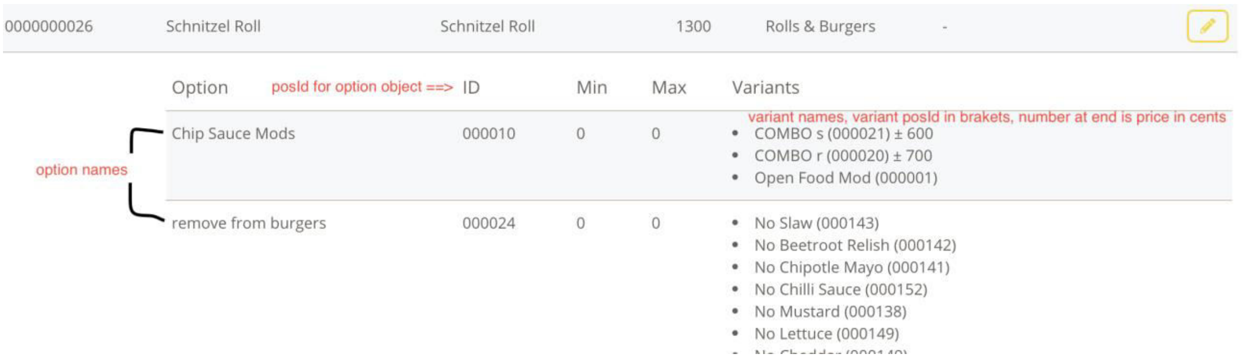
Add-ons are the last portion of the integration that needs to be verified before Alex will give live access. I’ll start by describe the structure of add-on payloads.

In the doshii order payload, add-ons are called *options* which is an array of option objects.



```
options: [
    {
        name, //name of the option group
        posId, //doshii posId for the option group (not the addon itself)
        variants, //an array of the variants (the actual addons from that group)
    }
]


variants: [
    {
        name, //name of the addon
        posId, //posId of the addon
        price, //price in cents
    }
]
```


posId for option object ==>
variant names, variant posId in brakets, number at end is price in cents
option names

0000000026	Schnitzel Roll	Schnitzel Roll	1300	Rolls & Burgers	-

Option	ID	Min	Max	Variants
Chip Sauce Mods	000010	0	0	• COMBO s (000021) ± 600 • COMBO r (000020) ± 700 • Open Food Mod (000001)
remove from burgers	000024	0	0	• No Slaw (000143) • No Beetroot Relish (000142) • No Chipotle Mayo (000141) • No Chilli Sauce (000152) • No Mustard (000138) • No Lettuce (000149)

To finish implementing fully, you should have the front end format orders in the way that doshii expects. It will be simpler in the long run to have one structure instead of two for orders. The following is an example of a order payload that the front end would send. Note that the price fields need to be calculated for every item taking into account the cost of addons. Alex will be checking that these amounts are correct. He will test that you can do different permutations of items/options with different quanities/cost/ect.

The following is what the order payload should be

```
{
    "items":[
       {
         "name":"Long Black",
         "options":[

         ],
         "unitPrice":400,
         "posId":"8502-19005",
         "quantity":2,
         "totalBeforeSurcounts": 800,
         "totalAfterSurcounts": 800,
         "surcounts": []
      },
      {
         "name":"Long Black",
         "options":[
             {
                 "name": "Coffee & Tea Mods",
                 "posId": "191",
                 "variants":
                     [
                         {
                             "name": "Add Soy",
                             "posId": "8502-19015",
                             "price": 50
                         },
                         {
                             "name": "Decaf",
                             "posId": "8502-19130",
                             "price": 0
                         }
                     ]
             }
         ],
         "unitPrice":400,
         "posId":"8502-19005",
         "quantity":1,
         "totalBeforeSurcounts": 450,
         "totalAfterSurcounts": 450,
         "surcounts": []
      },
      {
         "name":"Pizza Classic",
        "options":[],
        "unitPrice":1800,
        "posId":"8502-17308",
        "quantity":2,
        "totalBeforeSurcounts": 1800,
        "totalAfterSurcounts": 1800,
        "surcounts": []
     }],
   "stripeId":"12576834",
   "name":"Bichael",
   "phone":"+61413206203",
   "doshiiLocationId":"4rnOjGN4",
   "orderTotal":3050,
   "clientType":"pickup",
   "code":"1790",
   "venueName":"wv"
}
```