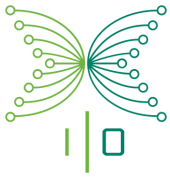
Backend - Api Doc



Introduction

Backend provides a number of blockchain services to the Sales Application, in this document you'll find the documentation of the different endpoints that are needed to interact with it.

Every particular API endpoint has a description section, a flow diagram with involved actors, technical definitions of the endpoint itself and a sequence explanation of the different workflows it might follow.

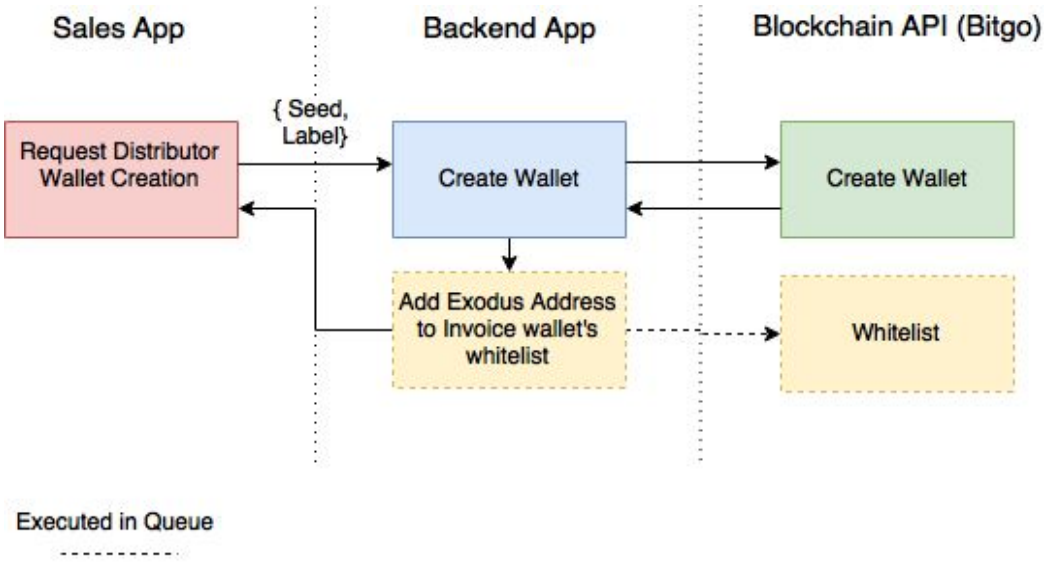
Distributor Named Invoice Wallet

Description

Each distributor in the network will have a wallet seeded with account specific information and named after the user or user's email.

Making the seed (Sales-App):
sha256("UUID + 1st or original Email address")

Diagram



Endpoint

[**Security**: Requires RW Authorization Token]

```
POST /api/v1/invoiceWallets body
{
	"seed": [String],
	"walletName": [String](Optional)
}
→ { walletId : [String] }
```

Workflow

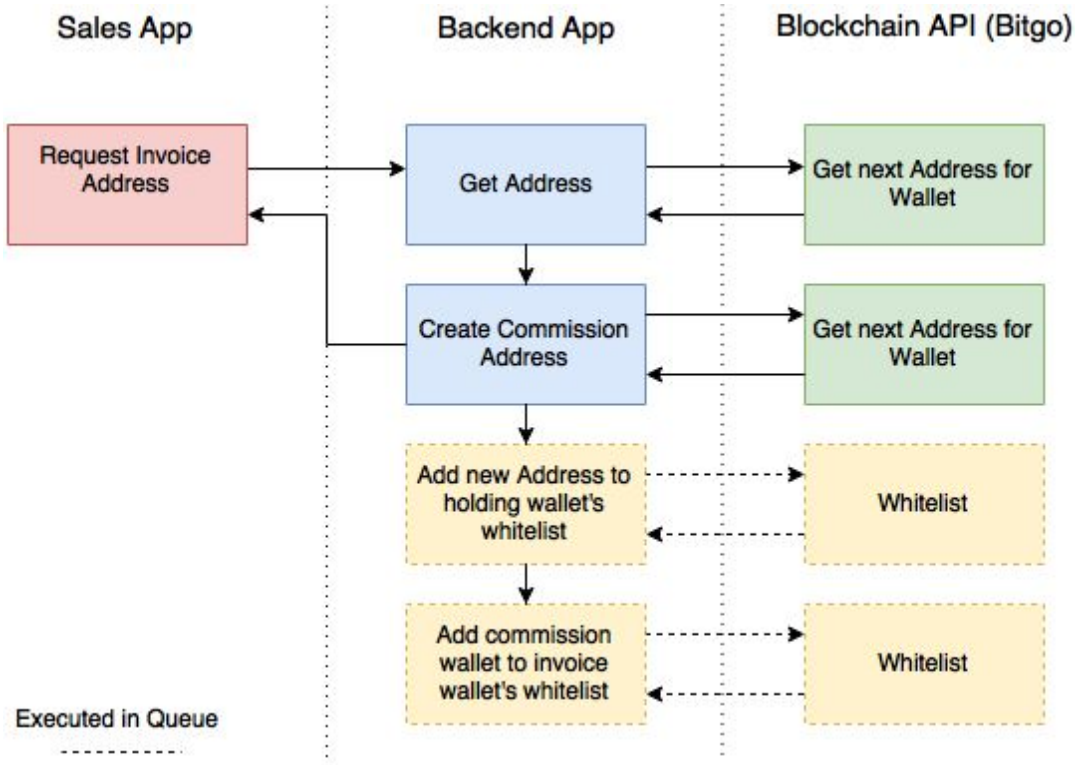
1. Sales App requests for a new Invoice Wallet
2. Backend App calls bitgo to create a wallet
3. Backend App queues a whitelist operation in order to add exodus wallet into new wallet's whitelist
4. Backend App returns:
    a. 200 and the wallet id if everything is ok
    b. 400 and the corresponding error message if there was a problem

Get New Invoice Addresses

Description

Each order will request an invoice address from the buyer's distributor's wallet
The address will be attached to the order. Each address should be watched by the backend.

Diagram



Endpoint

[**Security**: Requires RW Authorization Token]

```
PUT /api/v1/invoiceWallets/WALLET_ID/newAddress → { address : [String] }
```

Workflow

1. Sales app requests for a new address given an invoice wallet Id
2. Backend App request for a new address

3. Backend App creates a new commission address and associates it with the new invoice address
4. Backend app queues an operation to add the new address to the holding wallet's whitelist, in order to be able to receive funds if this address is used in a bundle
5. Backend app queues an operation to add the commission address to the invoice wallet's whitelist, in order to be able to do commission payout

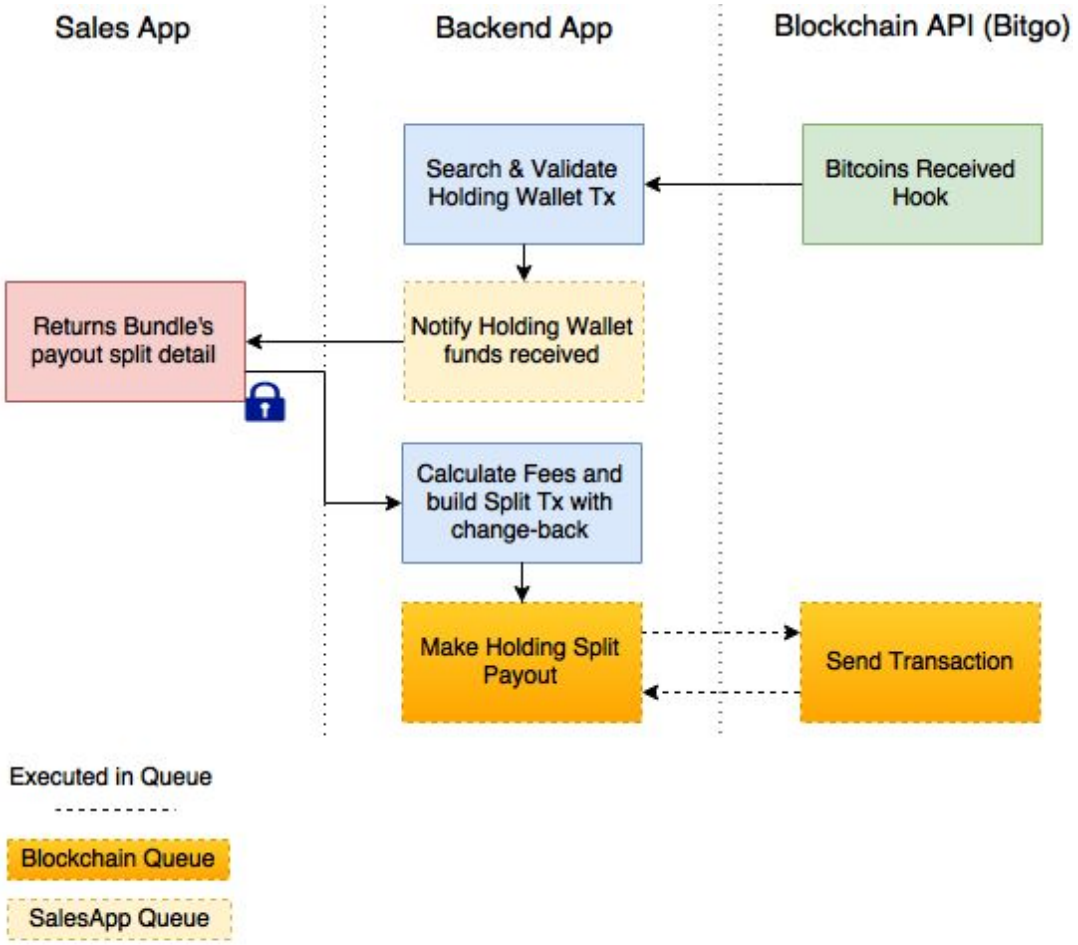
Holding Wallet Funds Received

Description

Bank transactions are all bundled together in one single bitcoin Transaction that then gets split into individual invoices. This single transaction is handled in a particular Wallet, named Holding Wallet.
When a transaction is received for this Wallet, the backend will notify the sales app and expect a response containing all the invoices with each amount the funds should be split in. The sum should be the total of the amount received in the transaction.
The Fees for this transaction are taken from a specific *fee address* which is also used as *change-back* to keep funds available for following transactions. This design requirement restricts the amount of parallel transactions that can be done for this Wallet. **It's very risky and should be avoided at all terms** to start a new Holding Transaction if previous one has not yet been confirmed.

***Note**: All Invoice addresses are previously white-listed (on creating) for the Holding Wallet, this prevents the Split Tx to be complete if not every payout address is an invoice address.*

Diagram



Endpoint

[**Security**: Public, doesn't need Authorization Token]

POST `/api/v1/bitgoCallback body { "hash": [String], "type": “transaction”, walletId: [String] }  → 200`

Condition: `walletId == HOLDING_WALLET_ID`

Sales App Notifications

1) Holding Wallet Received Bitcoins

[**Security**: Requires Payload Signature]

```
POST /api/holdingWalletReceivedBtc body
       {
              “holdingWalletAddress”:”address”,
              “satoshisReceived”:”#ofsatoshisreceived”,
              “transactionId”: “transactionId”
       }
→ 200 signed(*) body
   { “payout” : [ { invoiceAddress”: “invoiceAddress”, “amount”:”amount”}, ... ] }
→ < 555 => (any error < 555 triggers retry)
→ > 555 => Stop Call
```

***(*)** Note: The payload body must be signed with the backend given key.*

Workflow

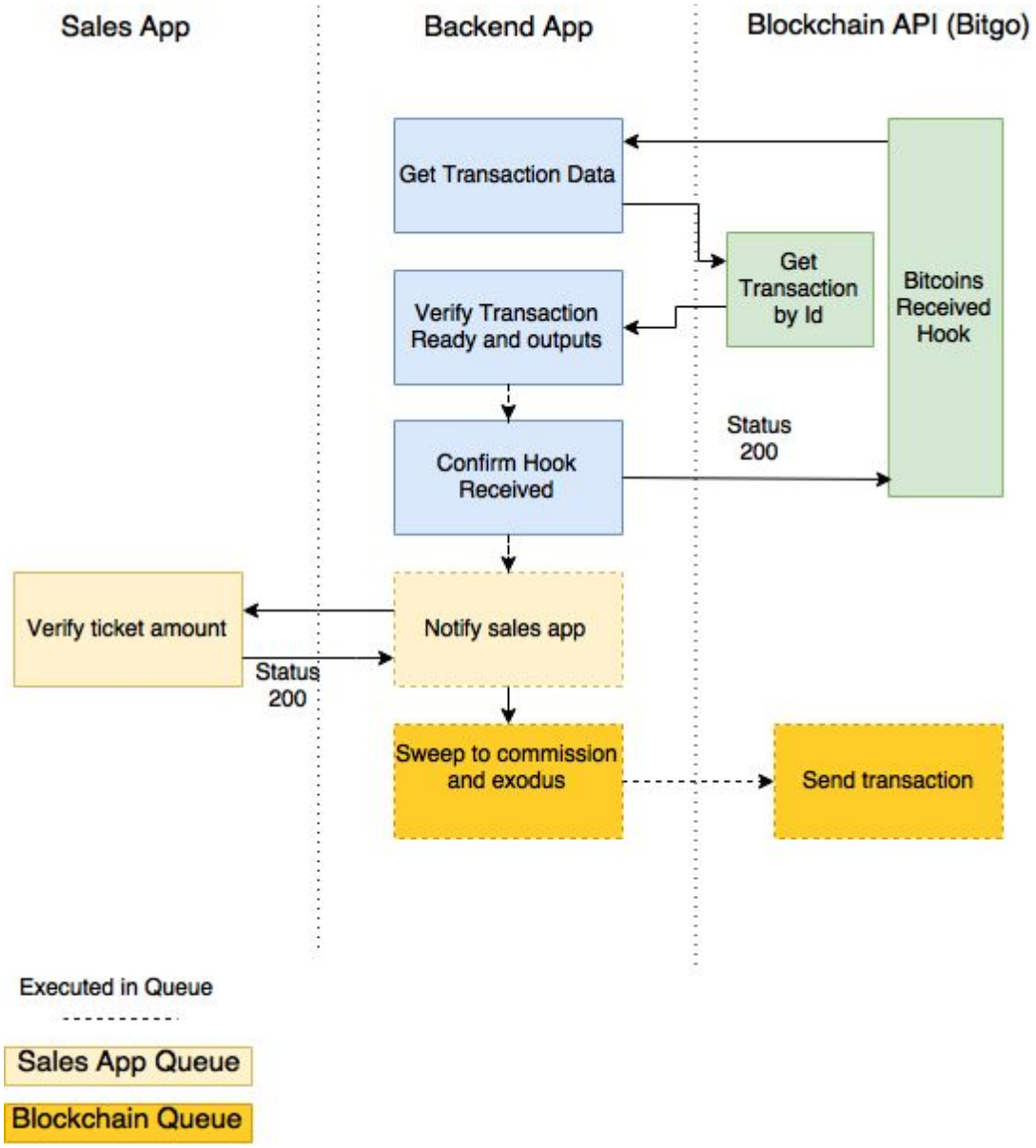
1. Bitgo notifies backend app that a transaction has been detected for the Holding Wallet
2. Backend app checks if the transaction is confirmed, if not, finishes the execution
3. Backend queues the call to the Sales-App `*holdingWalletReceivedBtc*` with the transaction info.
4. Once the Notification gets executed, Sales-App responds 200 with a signed payload containing the invoices split payout, address and amounts for each one.
5. The Backend builds the transactions, calculates the fees and assigns the holding fee address as secondary input, and that same address as charge-back. (See fees document for details)
6. The Transaction is queued for it to be processed
7. Once the Worker processes the Tx and the network confirms it, each invoice notification will be received and it will start the flow described in Bitcoin Received on Invoice Address

Bitcoin Received on Invoice Address

Description

When one of the requested invoice addresses receives funds, the sales app needs to be notified in order to split funds between exodus and commission wallets.

Diagram



Endpoint

[**Security**: Public, doesn't need Authorization Token]

```
POST /api/v1/bitgoCallback body
{
       "hash": [String],
       "type": “transaction”,
       walletId: [String]
}  → 200
```

Sales App Notifications

[**Security**: Only response code needed]

```
POST /api/invoiceAddressReceivedBtc body
{
       "invoiceAddress": [String],
       "satoshisReceived": [Integer],
       “transactionId”: [String],
       “date” : [DateTime],
       “confirmations” :  [Integer],
}  → { 200 | 4xx | 5xx}
```

Workflow

1. Bitgo notifies backend app that a transaction has been detected
2. Backend app checks the transaction output and queues a job for each output corresponding to an invoice address
3. Backend App (Sales app queue) queries sales app in order to ask permission to split funds between exodus and commission
4. If Sales app returns:
    a. Code 200 → go to step (6)
    b. Code < 555 → funds are not split but this job will be retried (go back to step 4)
        i. 400: Malformed request body, Invalid bitcoin address, invoiceWalletAddress, satoshisReceived must be a positive value
        ii. 549: Amount received is invalid for ticket #### having address XXXX
    c. Code > 555 → funds are not split and this job is marked as failed and won't be executed again
        i. 556: Ticket not found for address
        ii. 586: Ticket has been canceled
        iii. 587: Ticket has already received funds
        iv. 589: Bitstamp prices not valid for ticket

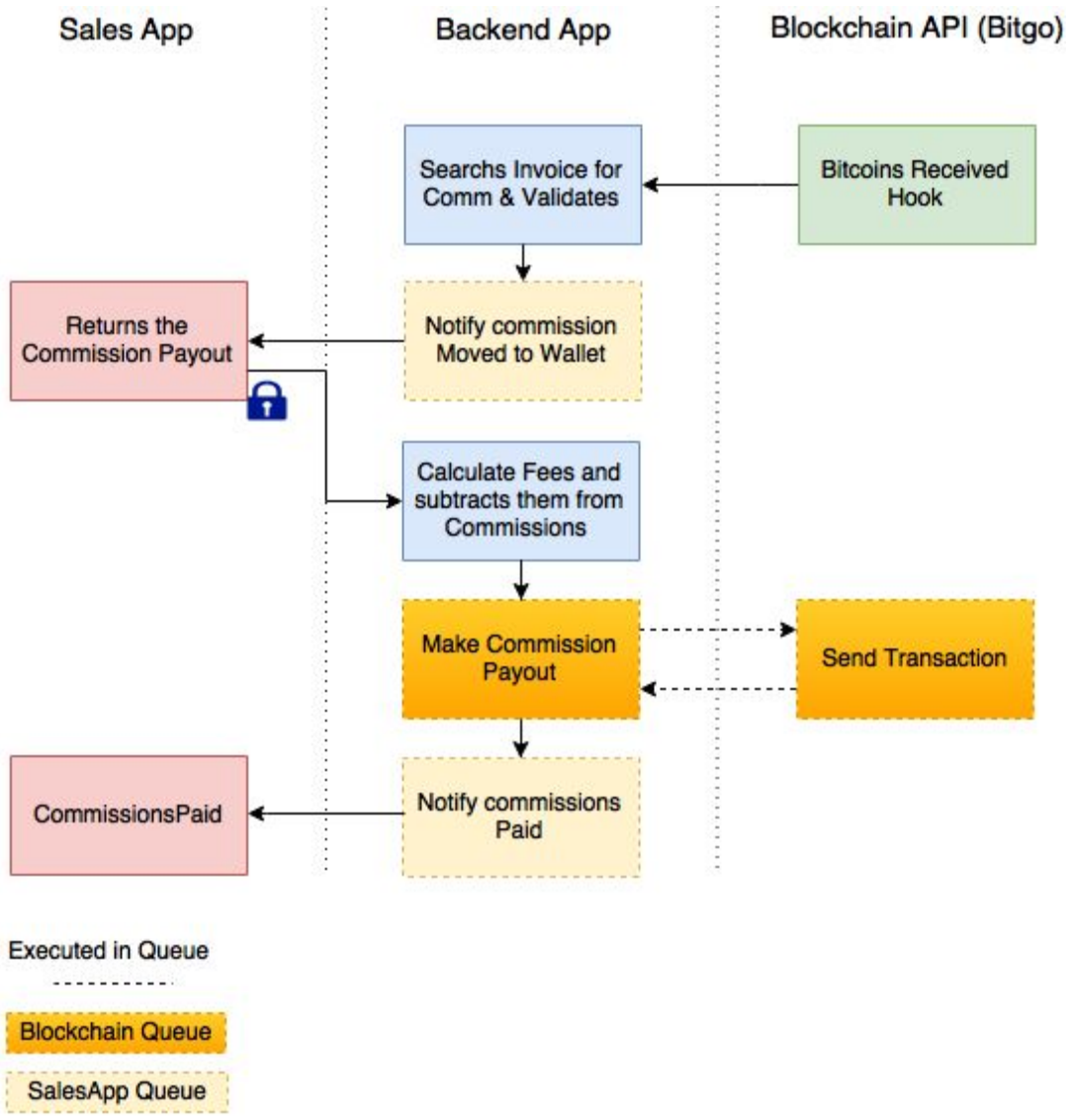
5. Backend App (Sales App queue) queues an Invoice Split Job in Blockchain Queue
6. Backend App (Blockchain queue) splits 80-20% to exodus and commission

Commission Wallet Funds Received

Description

Once the exodus/commission split transaction is confirmed, the back-end app receives the hook and broadcasts it to the sales-app, It adds the invoice and commission address for this transaction info in the payload. The sales-app responds with the payout detail.

Diagram



Endpoint

[**Security**: Public, doesn't need Authorization Token]

```
POST /api/v1/bitgoCallback body { "hash": [String], "type": "transaction", walletId:
[String] }  → 200
```

Sales App Notifications

2) Commissions Moved To Wallet

[**Security**: Requires Payload Signature]

```
POST /api/commissionsMovedToWallet body
      {"invoiceAddress": "invoice-address",
      "commissionWalletAddress":"commission-address",
      "satoshisAmount": "satoshis-amount",
      "transactionId": "transactionId",
      "date": "tx-date"}
→ 200 signed(*) body
      {"commissions": [{"commissionAddresss": "address", "amount": "#ofsatoshis"}]
→ < 555 => (any error < 555 triggers retry)
→ > 555 => Stop Call
```

***(*)** Note: The payload body must be signed with the backend given key.*

3) Commissions Paid

```
POST /api/commissionsPaid body
      {"invoiceAddress": "invoice-address", "transactionId": "transactionId"}
→ 200 => Success/Confirmation
→ < 555 => (any error < 555 triggers retry)
→ > 555 => Stop Call
```

Workflow

1. Bitgo notifies backend app that a transaction has been detected for the Commission Wallet
2. Backend app checks if the transaction is confirmed, if not, finishes the execution

3. Backend app checks the transaction output, as it's a Commision event, the address should be stored in the DB associated with the corresponding Invoice Address.
4. It queues the calls the Sales-App `*commissionMovedToWallet*` with the transaction info and the invoice address.
5. Once the Notification gets executed, the Sales-App responds 200 with a signed payload containing the commissions payouts, address and amounts to pay.
6. The Backend builds the transactions, calculates the fees and subtracts those (and the split ones) to make the amounts close-up to "0". (See fees document to details)
7. The Transaction is queued to be processed
8. Once the Worker processed it, the Sales-App Notification is queued to notify the `*commissionsPaid*` for this Transaction and Invoice Address.

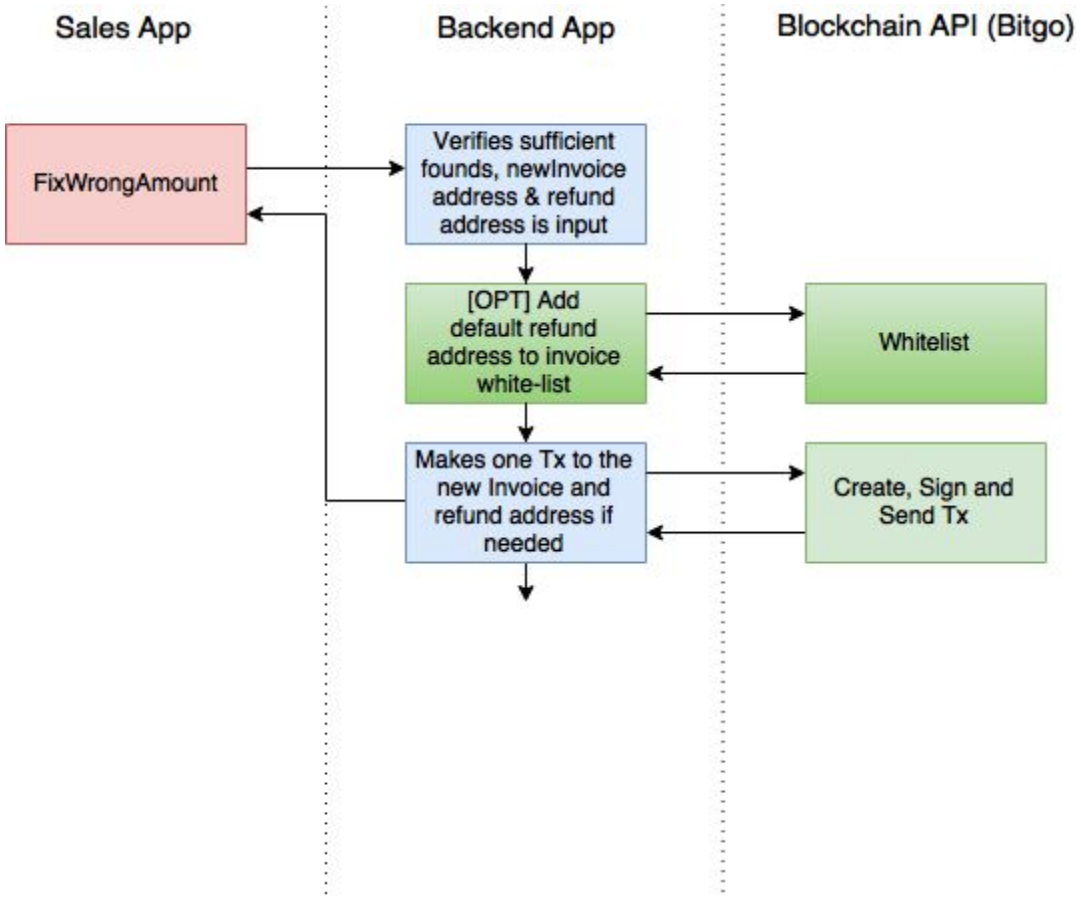
Fix Wrong Amount and Invoice Refunds

Description

This endpoint is intended to amend the case where the buyer sends a wrong Amount of bitcoins to one Invoice Address, this applies to:

1. Buyer sends less than expected and another Tx is made to complete the amount
2. Buyer sends more than expected and, if wanted, some funds needs to be refunded. This case includes the double sent scenario.
3. Buyer sends either less or more and the funds needs to be completely refunded.

Diagram



Endpoint

[**Security**: Requires RW Authorization Token]

```
POST /api/v1/wallets/WALLET_ID/fixWrongAmount body
{
       "invoiceAddress": [String] required,
       "expectedAmount": [Integer] required,
       "newInvoiceAddress": [String] optional,
       "fee": [Integer] optional,
       "refundAddress": [String] optional
}  → 200
```

Payload fields description and restrictions:

- Invoice Address: The invoices where the wrong amount was sent to.
- Expected Amount: The total amount of satoshis that was originally expected for this order. Can be zero if all funds would be refunded.
- New Invoice Address: A new Invoice address generated for the same Wallet, this address should also be updated in the Sales-app as the new Order Invoice Address. In the case ‘expectedAmount’ is zero, meaning that everything will be refunded, this parameter can not be included. But is required if expected amount is bigger than zero.
- Fee: The fee can be optionally supplied in the payload to be payed in the Tx, this is intended to allow a zero balance for this wallet once the funds are moved. If not included, the fees will be assigned by the backend, but there are no guarantees that current funds will be enough to cover them or in the other hand some coins be left in the wallet.
- Refund Address: If included, the extra funds (Balance - ExpectedAmount - Fee) will be returned to this address. This address should be the same as the one configured in the backend app secret configs. This restriction blocks the possibility of anyone trying to send funds to a different place.

Workflow 1: Buyer Sends less than expected

1. Buyer sends less satoshis than expected (example 10000 expected, 8000 received) to the invoice A.
2. The transaction is confirmed, the back-end receives the hook and forwards it to the sales-app, which rejects the funds because of an incorrect amount (8000 < 10000).
3. The buyer is asked to send the extra missing funds plus an extra to cover the fees (2000 + 10)

4. The second transaction is completed, the hook is received and once again the sales-app rejects it because of an invalid amount (2010 < 10000)
5. A sales-app operator creates a new Invoice Address B for this same Distributor's Wallet (using backend create address functionality) and then updates the sales app ticket Invoice wallet.
6. The *fixWrongAmount* is called with the payload
   { invoiceAddress: A, newInvoiceAddres: B, expectedAmount: 10000, fee: 10 }
7. The Backend will validate this data and if everything is ok, it will make the Transaction from A to B for 10000 satoshis paying 10 fee, and leaving A with zero balance.

Workflow 2: Buyer Sends more than expected, needs refund

1. Buyer sends more satoshis than expected (example 10000 expected, 11000 received) to the invoice A, from the address C.
2. The transaction is confirmed, the back-end receives the hook and forwards it to the sales-app, which rejects the funds because of an incorrect amount (11000 > 10000).
3. A Sales-app operator creates a new Invoice Address B for this same Distributor's Wallet.
4. The *fixWrongAmount* is called with the payload:
   { invoiceAddress: A, newInvoiceAddres: B, expectedAmount: 10000, refundAddress: SPECIAL_ADDRESS }
5. The Backend will validate this data and if everything is ok, calculate the fees (let's say 8) and it will make the Transaction from A to B for 10000, 992 to *SPECIAL_ADDRESS* paying 8 fee, leaving A with zero balance.

Workflow 3: All funds needs to be refunded

1. Buyer sends 10000 to the invoice A from the address C; but for any reason, the funds needs to be refunded.
2. The transaction is confirmed, the back-end receives the hook and forwards it to the sales-app, which rejects the funds because of an incorrect amount or other reason.
3. The *fixWrongAmount* is called with the payload:
   { invoiceAddress: A, expectedAmount: 0, refundAddress: SPECIAL_ADDRESS }
4. The Backend will validate this data and if everything is ok, calculate the fees (let's say 7) and it will make the Transaction from A to *SPECIAL_ADDRESS* for 9993 paying 7 fee, and again leaving A with zero balance.

Notes:

- If the refund address is not provided, the funds will be left in the InvoiceAddress
- If funds are not enough to pay fees, the Tx will be rejected