TABLE OF CONTENTS

AVATAR API

Avatar offers three HTTP api’s that let POS devices communicate with all the applications that comprise the avatar environment: avatar-api, avatar-vsdc and avatar books. Avatar Books is an entirely different application, but avatar-api, avatar-vsdc are referred generically as aTax server.

These api’s are available at https;//[avatar_domain_name]/avatar-api, https://[avatar_domain_name]/avatar-vsdc, https://[avatar_domain_name]/avatar-books

The first api is used by devices with a SAM card to sign the invoices themselves, and has two versions with the same functionality: one that sends the information as JSON objects and another that encodes the information in binary structures to save bandwidth. It is up to the manufacturer of the devices to choose one or the other. An audit mechanism (POA) must be performed on all SAM cards regularly or they will lock and stop signing invoices. The maximum number of invoices and maximum value of sales is a parameter set in every SAM card by the tax authority and is not accessible by the device. Once a SAM card is locked, only a successful POA will unlock it.

Those devices that don’t have a SAM card use avatar-vsdc to send the invoices to be signed at the server. Devices are authenticated by avatar-vsdc before signing (thus making it valid). In order to do so, avatar-vsdc makes sure the device’s HTTPS request is authenticated by its registered certificate, issued by Avatar’s CA (each registered device has its own certificate).

In addition to the infrastructure required to process invoices online, and only for the convenience of merchants, avatar includes avatar-books, an optional accounting package. This web application can help to maintain and distribute lists of products available for sale as well as the current tax rates to all the devices. This functionality is also available to certified devices, which must support another HTTPS api to make use of this functionality.

Finally, although technically not part of the API offered by aTax to devices, it is expected that the fiscal authority will make compulsory to provide a URL (stored as a QR code) in all printed tickets so that end customers can validate invoices independently from merchants. This document will explain how to produce that URL.

AVATAR API - INVOICE ENDPOINT

POST /invoice – JSON content-type: application/json

Invoices are usually uploaded to aTax as a JSON object with the following properties. See Avatar VSDC endpoint for an explanation of how invoices are parsed by aTax, providing the name of the fields match those of the following table.

Invoice fields:

Field	Type	Description
tin	String	Taxpayer id Size range: ..15
date	Date	Date and Time of Issue (ISO 8601)
place	String	Place of issue Size range: ..15
bid	String	Buyer Id
ccid	String	Buyer Cost Center Id Size range: ..15
itype	String	Invoice Type Allowed values: "Normal", "Pro Forma", "Training"
ttype	String	Transaction Type Allowed values: "Sale", "Refund"
mcr	String	Device Id Size range: ..15
ino	String	Invoice Number Size range: ..15
hr	BigDecimal	High VAT Rate
hb	BigDecimal	High Taxable Amount
ht	BigDecimal	High VAT Amount
mr	BigDecimal	Mid VAT Rate
mb	BigDecimal	Mid Taxable Amount
mt	BigDecimal	Mid VAT Amount
lr	BigDecimal	Low VAT Rate
lb	BigDecimal	Low Taxable Amount

lt	BigDecimal	Low VAT Amount
zr	BigDecimal	Zero VAT Rate
zb	BigDecimal	Zero Taxable Amount
zt	BigDecimal	Zero VAT Amount
cr	BigDecimal	Consumption rate
ct	BigDecimal	Consumption tax amount
salesTax	BigDecimal	Total Sales Tax Amount for all Normal Sale invoices Default value: BigDecimal.ZERO
taxReturn	BigDecimal	Total Return Tax Amount for all Normal Refund invoices Default value: BigDecimal.ZERO
sales	BigDecimal	Total Sales Amount Default value: BigDecimal.ZERO
refunds	BigDecimal	Total Refund Amount Default value: BigDecimal.ZERO
icount	int	Total Invoice Counter
scount	int	Sales Invoices Counter
rcount	int	Refunds Invoices Counter
journal	String	Representation of the ticket printed after a successful transaction. It may contain some placeholders to be replaced with the actual information by the device. It's up to the fiscal authority to enforce their presence.
lines	Object[]	Array with the invoice lines included in the transaction. Each of the invoice lines has the following 7 fields.
ean	String	The EAN code for this item Size range: 8..18
name	String	The item description Size range: ..60
quantity	BigDecimal	Number of units included in this line
base	BigDecimal	Taxable amount per item
discount	BigDecimal	Applied discount per item
code	String	Tax code used for this line
total	BigDecimal	Total amount of the line
internalDataHash	byte[]	Hash of SAM inner counters
encryptedCounters	byte[]	Encrypted SAM inner counters
signature	String	See detailed description of signed information below

Signature Validation fields

The last three fields let aTax validate invoices issued by devices with a SAM card. Every time an invoice is produced,the device will execute the APDU command SIGN_INVOICE_T, with the following fields,concatenated, to be signed: tin, invoiceDate, bid, itype, ttype, mcr, ino, hr, hb, ht, mr, mb, mt, lr, lb, lt, zr, zb, ztz.

The SAM card doesn't simply sign this piece of information, but concatenates it with a hash of the inner counters and signs everything. The returned value of SIGN_INVOICE_T contains the last three fields sent to aTax.

The encrypted counters fields is not used to validate the invoice's signature, it will simply decrypt and store the value of the counters after validating the invoice.

POST /invoice - binary invoice format

The data from the invoice can be sent directly in binary format as the payload of a POST request. The information required is the same information defined for the json format, but in order to use the binary form, it must be encoded in an array of bytes following these rules:

- **Dates**, must be converted to an 8-bytes array containing the number of milliseconds since January 1, 1970, 00:00:00 GMT. The byte array will be in big-endian byte-order.
- **Floating point numbers**, must be converted to a byte array containing its two's-complement representation. The byte array will be in big-endian byte-order. The array will contain the minimum number of bytes required to represent the number, including at least one sign bit. When included in the signature any trailing zeros must be removed from the unscaled value before converting it to bytes.
- **Integer numbers**, must be converted to a fixed-length byte array. The byte array will be in big-endian byte-order. The number of bytes will depend on the type of the value (4 or 8 bytes).
- **Strings**, must be encoded into a byte array using the UTF-8 charset.

Unlike the JSON request, the fields of the invoice must be serialized in a specific order following the above rules so that aTax can decode the information. The following java code snippet shows how serialization is performed, so that its binary output can be replicated in any other programming language. The code is available here:

https://github.com/avatarTechnologies/javaPos/blob/master/src/main/java/com/systemonenoc/avatar/client/Invoice.java

```
public byte[] serialize() throws IOException {
   ByteArrayOutputStream byteStream;
   if (lines != null) {
       byteStream = new ByteArrayOutputStream(defaultLength +
             lines.size() * defaultLineLength);
       } else
          byteStream = new ByteArrayOutputStream(defaultLength);
       DataOutputStream dataStream = new DataOutputStream(byteStream);
       dataStream.writeUTF(tin);
       dataStream.writeUTF(date);
       dataStream.writeLong(invoiceDate.getTime());
       writeUTF(dataStream, place);
       writeUTF(dataStream, bid);
       writeUTF(dataStream, ccid);
       writeUTF(dataStream, phone);
       dataStream.writeUTF(itype);
       dataStream.writeUTF(ttype);
       dataStream.writeUTF(mcr);
       writeUTF(dataStream, clientMcr);
       dataStream.writeUTF(ino);
       writeBigDecimal(dataStream, hr);
       writeBigDecimal(dataStream, hb);
       writeBigDecimal(dataStream, ht);
       writeBigDecimal(dataStream, mr);
       writeBigDecimal(dataStream, mb);
       writeBigDecimal(dataStream, mt);
       writeBigDecimal(dataStream, lr);
       writeBigDecimal(dataStream, lb);
       writeBigDecimal(dataStream, lt);
       writeBigDecimal(dataStream, zr);
       writeBigDecimal(dataStream, zb);
       writeBigDecimal(dataStream, zt);
       writeBigDecimal(dataStream, cr);
       writeBigDecimal(dataStream, ct);
       dataStream.writeInt(invoiceCounter);
       dataStream.writeInt(partialCounter);
       dataStream.writeUTF(encodedString(internalDataHash));
       dataStream.writeUTF(encodedString(signature));
       dataStream.writeUTF(encodedString(encryptedCounters));
       writeUTF(dataStream, journal);
       if (lines != null) {
          for (InvoiceLine line : lines) {
             if (line != null) {
                IOService.writeUTF(dataStream, line.getEan());
                IOService.writeUTF(dataStream, line.getName());
                writeBigDecimal(dataStream, line.getQuantity());
                writeBigDecimal(dataStream, line.getBase());
                writeBigDecimal(dataStream, line.getDiscount());
                writeUTF(dataStream, line.getCode());
                writeBigDecimal(dataStream, line.getTotal());
                writeUTF(dataStream, line.getCurrentHash());
             }
          }
       }
       return byteStream.toByteArray();
}
```

POST /encrypted

The data from the invoice can also be sent encrypted, although this is not normally used by third-party integrators, since the use of HTTPS already provides encryption.

The information required is the same information defined for the json format, but in order to send it encrypted, first it must be encoded in binary format following the same rules defined for the unencrypted binary format and then encrypted using the following algorithm:

- **Symmetric encryption** The first step is to create a random key and use it to encrypt the data. In this step we will use 256-bit AES encryption. A single-use random key with the required length must be generated and used to encrypt the data with the AES/CBC/PKCS5Padding algorithm. Furthermore, a random 16 byte initialization vector must be applied to ensure each encrypted message will be unique.
- **Asymmetric encryption** The second step is to use the public key provided to encrypt the previously generated random key. Once everything is properly encrypted it can be sent safely to aTax. In this step we will use 2048-bit RSA encryption. The public key provided by aTax must be used to encrypt the previously generated random key. Once the symmetric key is encrypted it must be concatenated with the initialization vector and the encrypted data from the previous step and sent to aTax.

Invoice – Common problems and error codes

When a transaction is received, aTax will process it in several steps. In any of these steps there is a number of different errors that can occur. The application will save as much information as possible when this happens to help identify the problem and fix the issue and/or re-issue the transaction if possible.
Here is a list of the most common issues, what information is saved about them and where that information is saved. *Unless otherwise noted all errors are saved in the server logs and a failed_transaction record is created with the details of the data being processed.* We provide as well some tips on what to do after an error has occurred.

Unknown Error (Code: 0)	These errors are always possible but should not happen very often. Here we include several ***unexpected*** errors that may appear during development but should never happen in a production environment. Errors starting with "could not execute query" or "could not execute statement or "could not extract ResultSet": These are the result of a change in the format of the invoice, the requirements of the fields or some internal modification of the database layer. Should never happen in a production environment. When this happens the error will be saved in the server logs, but it's possible that neither a transaction nor a failed_transaction record has been created. Errors starting

	with "I/O error on GET request for http://": aTax needs to connect to a certificate service in order to validate certificates, verify signatures and decrypt data. If this connection fails an error like this is thrown. When this happens the transaction is not created. Errors mentioning "bad arguments", "malformed input", "input length" and similar: These are probably the most common errors of this kind. These happen when the data received is wrong in a way we cannot handle. It could be that the content-type specified in the request is wrong. Or the data is encrypted when it shouldn't or vice versa. Empty errors: These are definitely unexpected. When these happen is not possible to know what happened nor whether a failed_transaction record has been created or not. At least, the error will be saved in the server logs, but these should never happen in a production environment.
Signature Error (Code: 1)	For instance, the error "Received signature is not verifiable". When this happens, it means that the signature we received is not valid. Either the key used to sign is invalid or it's not the right one. The best option is to double check that the key is correct and contact the support department to verify that the right one is being used.
Format Error (Code: 2)	These errors should have a very descriptive message and the solution should be very obvious. Something like 'TIN cannot be longer than 15 characters', for instance. This kind of errors would be the result of a typo or a copy and paste mistake when sending the information.
Merchant Error (Code: 3)	These errors occur when the status of the taxpayer is not valid and, hence, they are not allowed to issue invoices. It could be that it don't exists, or is not enabled, or is not active. In any case, the only way to fix it is to contact the support department and make sure that the taxpayer is created, enabled and active in the system. Example: "TIN <12345678> is not active"
Device Error (Code: 4)	These errors occur when the status of the device is not valid and, hence, they are not allowed to issue invoices. It could be that it don't exists, or is not enabled, or is not assigned to the right taxpayer. In any case, the only way to fix it is contact the support department and make sure that the device is allowed to issue invoices. Example: "MCR <XXXXXXXXXXXXXX> is not assigned to TIN <1234567>""
Data Error (Code: 5)	These errors happen when the data received did not pass one of the validations before the transaction is created. For instance "The current invoice counter does not match the received value." . These errors should give a hint about what to do. The user should verify that the data is correct and contact the support department for further advice.

AVATAR-API: POA ENDPOINT

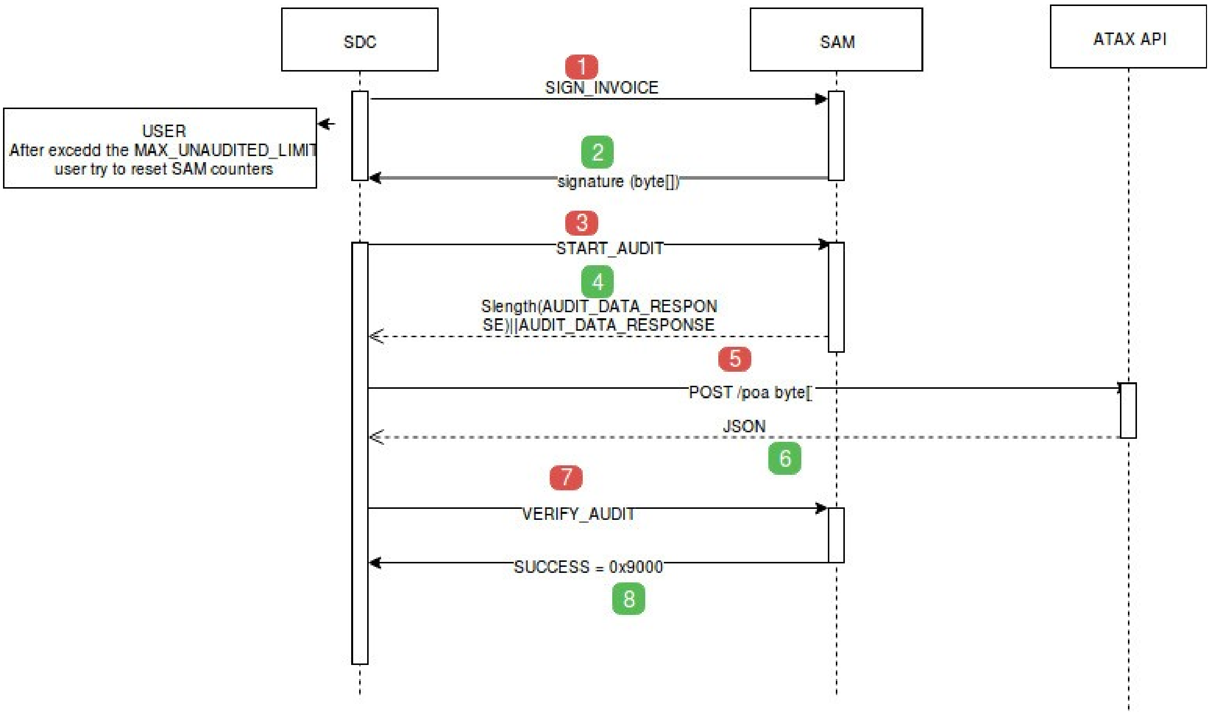
POA workflow

The basic POA workflow contains several steps:

1. SDC needs to sign a transaction then tries to execute SIGN_INVOICE command.
2. SAM card returns a byte array containing signature. If no more signatures are allowed and audit procedure is needed a

SW_MAX_UNAUDITED_TRANSACTIONS response will be thrown (it is possible to perform a POA before the maximum number of transactions allowed is reached).

3. At this point SDC has to start POA sending a START_AUDIT command.
4. Once the SAM receives the command, and after making some initial checks, it starts building an object (auditData). This piece of information is returned to the card, although not directly. For privacy and integrity reasons it is encrypted and signed. auditData is composed by the following elements:
    - The SerialID of the certificate present in the card.
    - TRANSACTIONS_COUNTER.
    - LAST_AUDITED_TRANSACTION_COUNTER.
    - The COUNTERS object.
    - A random token of 32 Bytes.
5. ATAX API contains an endpoint (/poa) to receive auditData. SDC is responsible to send the request. Server behaviour is explained in the next section.
6. ATAX API answers a HTTP code with status and a JSON payload containing information about the flow.
7. At this point If the POA is complete SDC will be able to send VERIFY_AUDIT command to SAM card
    7.1. If POA is not complete SDC must send all the pending invoices and perform POA again.
8. If the VERIFY_AUDIT is complete SAM responses SUCCESS. Otherwise and error is thrown.



POA workflow at aTax server

1. Read request [totalLength, encryptLength, encApiPub, SignLength, signatureData] and create GenAuditTokenResponse
2. Decrypt GenAuditTokenResponse.encApiPub to get AuditData
3. Read AuditData [TotalLen, serialNumber, TID, LastAuditedTID, counters, TokenIDArr]
4. Verify signature with serialNumber, auditData, signatureData
5. Verify Device associated with serialNumber
6. Verify POAs in progress for that Device. If there are any pending POAs, cancel them, and start a new one
7. Get transactions between POA.lastAuditedTID and POA.tid
    - Verify that:
        - All transactions are collected
        - Transaction.journal is filled
        - Transaction.counters is filled
    - If everything is complete:
        - update POA.complete = true and Poa.history = ProofOfAuditStep.WAITING
        - Otherwise POA.history = ProofOfAuditStep.WAITING
8. Return POA data to client

Request

/poa	POST	byte[]	Contains the SAM card START_AUDIT response

Responses

If POA is started a JSON will be returned to client. This payload contains several fields:

- cancel, boolean, TRUE if POA is cancelled
- complete, boolean, TRUE is POA is completed
- mrc, device assigned to POA
- token, decrypted token generated by SAM during startAudit
- signature, Base64 containing token signed by ATAX private key

202 Accepted	No pending Proof Of Audit
400, Bad Request	Exception field
200 OK	{ "cancel":false,

	"complete":false, "mrc":"test", "token":"rV0VqGO6UPDBkVVCSX1rkffaC9pqZJqh2POFIpvyTX4=" }

200 OK	{
	"cancel":false,
	"complete":**true**,
	"mrc":"test",
	“token":"rV0VqGO6UPDBkVVCSX1rkffaC9pqZJqh2POFIpvyTX4=",
	"**signature**": "O6UPDBk..."
	}

AVATAR VSDC - API

Those devices that don’t have a SAM card send the invoices to a VSDC, which signs them on behalf of the device and keep the value of its inner counters. Once an invoice has been accepted by the VSDC, it is a valid transaction and aTax guarantees it will be processed eventually. Devices are identified by a certificate obtained from the fiscal authority.

Configuration & Requirements

- Is mandatory to use valid certificates during communications, server application will validate client certificate against Certificate Authority(CA) during SSL handshake.
- You will need your merchant/taxpayer account TIN number, please contact system admin to obtain a valid certificate. You should use it as param when it's necessary (issuing invoices).
- There is only one method relevant to device manufacturers, invoice. This endpoint is used to issue tickets to aTax, which in turn will validate it against aTAX, sign it and send to aTAX server. This endpoint accepts JSON/XML format as payload. In 'JSON Receipt example' you will find how fields are configured.

POST /invoice

Regular invoice format defines minimum data that has to be signed and submitted to the Tax Authority. Invoice consist of following data.

Depending on the content-type of the request aTax will try to parse it and obtain all the information contained in the invoice. Valid content-types are application/xml and application/json

SAMPLE URL
https://[avatar_domain_name]/avatar-vsdc/merchant/invoice

SAMPLE XML Invoice

```
<Invoice>
  <tin>test</tin>
  <date>2017-11-22 14:53:31+0000</date>
  <place>zgz</place>
  <bid/>
  <ccid/>
  <phone/>
  <itype>Normal</itype>
  <ttype>Sale</ttype>
  <mcr>test</mcr>
  <clientMcr/>
  <ino>xxxyyy001</ino>
  <journal>$$date$$ $$nev$$ $$numberRecu$$ $$idata$$ $$signature$$</journal>
  <hr>0</hr>
  <hb>0</hb>
  <ht>0</ht>
  <mr>18</mr>
  <mb>0</mb>
  <mt>0</mt>
  <lr>16</lr>
  <lb>0</lb>
  <lt>0</lt>
  <zr>32</zr>
  <zb>0</zb>
  <zt>0</zt>
  <cr>1</cr>
  <ct>0.00000</ct>
  <lines/>
  <invoiceDate>1511362411070</invoiceDate>
  <invoiceCounter>0</invoiceCounter>
  <partialCounter>0</partialCounter>
</Invoice>
```

SAMPLE JSON Invoice

```
{"tin":"test","date":"2017-11-22 15:14:20 +0000",
"place":"zgz","bid":null,"ccid":null,"phone":null,
"itype":"Normal","ttype":"Sale","mcr":"test",
"clientMcr":null,"ino":"xxxyyy001",
"journal":"$$date$$ $$nev$$ $$numberRecu$$ $$idata$$ $$signature$$",
"hr":0,"hb":0,"ht":0,"mr":18,"mb":0,
"mt":0,"lr":16,"lb":0,"lt":0,"zr":32,"zb":0,"zt":0,"cr":1,"ct":0.00000,
"lines":[],"invoiceDate":1511363660947,"invoiceCounter":0,"partialCounter":0}
```

Fields

Field	Type	Description
tin	String	Tax payer id Size range: ..15
date	Date	Date and Time of Issue (ISO 8601)
place optional	String	Place of issue Size range: ..15
bid optional	String	Buyer Id
ccid optional	String	Provided by buyer for automatic expense/budget tracking Size range: ..15
itype	String	Invoice Type Allowed values: "Normal", "Pro Forma", "Training"
ttype	String	Transaction Type

		Allowed values: "Sale", "Refund"
mcr	String	Device Id Size range: ..15
ino	String	Invoice Number Size range: ..15
journal	String	The text to be printed after a successful transaction. It may contain some placeholders to be replaced with the relevant information by the device or the VSDC. These placeholder are set by default, but the tax authority could define more. $$date$$: V-SDC Date And Time $$nev$$: V-SDC Number $$numberRecu$$: Number and Type of Invoice $$idata$$: Internal Data $$signature$$: Digital Signature
phone **optional**	String	Use this filed to send conformation SMS to buyer
md5 **optional**	String	Hash generated by client to identify transactions

Tax

Field	Type	Description
hr	BigDecimal	High VAT Rate
hb	BigDecimal	High Taxable Amount
ht	BigDecimal	High VAT Amount
mr	BigDecimal	Mid VAT Rate
mb	BigDecimal	Mid Taxable Amount
mt	BigDecimal	Mid VAT Amount
lr	BigDecimal	Low VAT Rate
lb	BigDecimal	Low Taxable Amount
lt	BigDecimal	Low VAT Amount
zr	BigDecimal	Zero VAT Rate
zb	BigDecimal	Zero Taxable Amount
zt	BigDecimal	Zero VAT Amount
ct **optional**	BigDecimal	Consumption tax
cr **optional**	BigDecimal	Consumption rate

Extended

Field	Type	Description
lines optional	Object[]	an array with all the invoice lines included in the

		transaction
ean	String	The EAN code for this item Size range: 8..18
name	String	The item description Size range: ..60
quantity	BigDecimal	Number of units included in this line
base	BigDecimal	Taxable amount per item
discount	BigDecimal	Applied discount per item
code	String	Tax code used for this line
total	BigDecimal	Total amount of the line

Success 200

Field	Type	Description
success	Boolean	Whether the request was successfully processed or not.
status	String	
message	String	Any relevant information regarding the received invoice.
errors	String	Array of errors found processing ticket.
hash	String	MD5 string representing ticket data (Cash Register ID, Invoice No, Date & hour)
journal	String	The journal text updated with newly genezr data.
vsdcNumber	Long	This virtual device's serial number for reference.
signingDate	String	V-SDC date and time of signing.
internalData	String	Hash of internal data (aka counters).
signature	String	Signature of the invoice.
invoiceNumber	String	Number of Invoice
invoiceType	String	Type of Invoice.
verificationUrl	String	URL used to validate ticket against aTax. This field should be used to print QR code.
totalInvoices	Integer	Total number of invoices signed by the V-SDC tenant.

400

Field	Type	Description
success	Boolean	Whether the request was successfully processed or not.
message	String	Any relevant information regarding the received invoice.

errors	String[]	A list of the errors found while processing the received invoice.

AVATAR BOOKS API

As explained in the introduction, Avatar Books (or simply, aBooks) is a tool intended to help small merchants manage their POS devices and stock of products on sale, but its use is entirely optional. All the devices supported by Avatar Technologies are able to communicate with aBooks, but third party manufactures can also choose to support it.

It is beyond the scope of this document to describe the functionalities of aBooks in detail but the following concepts to help developers understand the description of the methods.

- Every device and the POS included with the web application /webPOS) have their own catalog of products and prices identified with a hash. All the catalogs are stored in the central database so all of them can be recovered at any time providing their hash is known
- The hash value is generated by the POS devices as described in aBooks documentation before uploading a new catalog or product
- The catalog of the webPOS is considered the master and can be distributed to all the devices. A device's catalog, however, cannot be distributed. It has to be uploaded, set as master at aBooks and then distributed the rest of devices
- aBooks keeps track of the current tax rates and, should they change, will be returned with the next product request. They are also included the first time the catalog is downloaded.
- Besides products, aBooks api can be used to edit a merchant's cashiers

POST api/products/{current_hash}

Param	Required	Value	Description
merchant	Yes	String	Taxpayer ID
mcr	Yes	String	Device ID
initial_hash	No	String	root hash
name	Yes	String	product's name
barcode	Yes	String	product's EAN code
price	Yes	String	unit price
rate	Yes	String	Tax rate
desc	No	String	Description
deleted	No	Int	boolean value to set as deleted
client_date	Yes	Long	Timestamp

Response: { "success": true, "msg": "201" }

SUCCESS	MSG	Description
True	201	
False	418	Merchant not found
False	419	Tax value not found
False	420	Device not found
False	500	Exception found

POST api/products/

Param	Required	Value	Description
merchant	Yes	String	merchant ID
mcr	Yes	String	Device ID
products	Yes	[]	product’s list

Param	Required	Value	Description
current_hash	Yes	String	modification Hash
initial_hash	No	String	root Hash
name	Yes	String	product’s name
barcode	Yes	String	product’s EAN code
price	Yes	String	unit price
rate	Yes	String	Tax rate (A,B,C,D)
desc	No	String	Description
deleted	No	Int	boolean, 1 if it’s deleted
client_date	Yes	Long	Timestamp

Response: { "success": false, "msg": "400", "result": [{"currentHash": "####1","error": "401","Message": "Unable to"}....{}]}

SUCCESS	MSG	RESULT	Description
True	201	[]	
False	400	[{...},....{}]	There is an issue in product’s list
False	418	[]	Merchant not found
False	420	[]	Device not found
False	500	[]	Exception found

CURRENT HASH	ERROR	Description
###value###	419	Tax value not found
###value###	500	Exception found

POST api/products/{initial_hash}/image

This method associates an image to a product.

Param	Value	Description
image	binary	bin file with product’s image

GET api/products

Param	Required	Value	Description
timestamp	Yes	String	
merchant	Yes	String	Merchant ID
offset	No	Int	
limit	No	Int	

Response: 200 OK

```
{
   "success": true,
   "msg": "200",
 "timestamp": "1231235664",
   "total": "200",
 "products": [
    {
 "name": "CocaCola",
 "barcode": "85670000054",
 "current_hash": "4356782f",
            "initial_hash": "4356782f"
            ...
    },
    {
 "name": "Nestea",
 "ean": "852634000012"
            ...
    },
    {}T
 ],
 “taxes”: [
“A”:”0”,
“B”:”0”,
“C”:”5”,
“D”:”0”,
“Consumption tax”:”5”
]
}
```

Param	Value	Description
initial_hash	String	root Hash
current_hash	String	modification Hash
name	String	product’s name
barcode	String	product’s EAN code
price	String	unit price
rate	String	Tax rate (A,B,C,D)
desc	String	Description
deleted	Int	boolean, 1 if it’s deleted

POST api/cashiers/{userId}

Param	Required	Value	Description
name	Yes	String	cashier’s name
pass	Yes	String	password (SHA-1)
mcr	Yes	String	device’s mcr
merchant	Yes	String	Merchant ID
fullname	Yes	String	cashier’s full name
phone	No	String	phone number
email	Yes	String	
deleted	No	Int	boolean, 1 if it’s deleted

Create
Response: 201 OK

```
{
   "success": true,
   "msg": "201",
   "userId": 122663,
}
```

Modify
Response: 200 OK

```
{
 "success": true,
  "msg": "200",
}
```

SUCCESS	MSG	Description
True	201	

True	200	
False	418	merchant not found
False	419	Email exists in data base
False	404	cashier not found
False	420	device not found
False	500	Exception found

GET api/cashiers

Param	Required	Value	Description
timestamp	Yes	String	
merchant	Yes	String	merchant ID
offset	No	Int	
limit	No	Int	

Response: 200 OK

```
{
 "timestamp": "1236687986",
"total": "200",
 "cashiers": [
   {
 "name": "Cajero1",
 "pass": "35434kj5iwaj0w",
 "userId": "4356782",
  ...
   },
   {
 "name": "Cajero2",
 "pass": "36534kjks53587s"
          ...
   },
   {}
 ]
}
```

GET api/merchants/{email}

SUCCESS	MSG	DESCRIPTION
True	200	
False	404	Merchant not found

```
{
   "success": true,
   "msg": "200",
    "userId": 122663,
}
```

Relationship between devices (parameters and longitudes)

G5	Length(G5)	ANDROID	Length (Android)	ABOOKS	
Name	60	name	60	sku	
DescEn	20	description	250	description	
Price	10(total) 2(decimales)	price	12 (2 decimales)	price	
Tax	2	tax	1	tax_rate_id	
Ean	18	barcode	18	barcode	
X		image_url		image_id	
X		type		X	
X		X		inventory	
X		X		sold	
X		X		sold_amount	
exist	integer	X		enabled	
X		X		category_id	
HashInit	9		9		
HashCur	9		9		
TStamp	8 bytes integer		8		

Param	Required	Value	Description	Longitude G5	Server
name	Yes	String	cashier’s name	100	100
pass	Yes	String	password SHA-1	100	100
mcr	Yes	String	device’s MCR	15	100
merchant	Yes	String	merchant ID	15	100
fullname	Yes	String	cashier’s full name	100	100
phone	No	String	phone number	100	100
email	No	String		100	100
deleted	No	Int			

AVATAR PUBLIC INVOICE VALIDATION API

The following method shows how the validation URL is built by aTax. Please note that byte[] values (internalDataHash and Signature) are Base64 encoded before adding them to the url. API_PUBLI_HOST is [avatar_domain_name] and API_PUBLIC_VERIFY is /ticket/verify

```
private String buildVerificationUrl(String tin, String date, String bid, String
itype, String ttype, String mcr,
                    String ino, BigDecimal hr, BigDecimal hb, BigDecimal ht,
BigDecimal mr, BigDecimal mb, BigDecimal mt,
                    BigDecimal lr, BigDecimal lb, BigDecimal lt, BigDecimal zr,
BigDecimal zb, BigDecimal zt,
                    String internalData, String signature, BigDecimal cr,
BigDecimal ct) {
             try {
                    StringBuilder sb = new StringBuilder();
                    sb.append(environment.getRequiredProperty(API_PUBLIC_HOST));
                    sb.append(environment.getRequiredProperty(API_PUBLIC_VERIFY));
                    append(sb, '?', TIN, tin);
                    append(sb, '&', DATE, date);
                    append(sb, '&', BID, bid);
                    append(sb, '&', ITYPE, itype.substring(0, 1));
                    append(sb, '&', TTYPE, ttype.substring(0, 1));
                    append(sb, '&', MCR, mcr);
                    append(sb, '&', INO, ino);
                    append(sb, '&', DATA, internalData);
                    append(sb, '&', SIGN, signature);
                    append(sb, '&', HR, hr);
                    append(sb, '&', HB, hb);
                    append(sb, '&', HT, ht);
                    append(sb, '&', MR, mr);
                    append(sb, '&', MB, mb);
                    append(sb, '&', MT, mt);
                    append(sb, '&', LR, lr);
                    append(sb, '&', LB, lb);
                    append(sb, '&', LT, lt);
                    append(sb, '&', ZR, zr);
                    append(sb, '&', ZB, zb);
                    append(sb, '&', ZT, zt);
                    append(sb, '&', CR, cr);
                    append(sb, '&', CT, ct);
                    return sb.toString();
             } catch (IllegalStateException | IOException e) {
                    throw exception("Unable to generate verification url", e);
             }
      }
```