WebSocket API

Purpose

In order to avoid polling for changes, we want to allow third party integrators to listen to a websocket to get notifications about changes.

Overview

The API consists of different topics, e.g. “invoices” and “supplier-invoices”. Each topic has an ordered list of events. Each event has a unique offset within the topic. You can listen to multiple topics and multiple tenants to get notified about changes. If you are unexpectedly disconnected you can supply an offset when reconnecting, to fetch the events that you might have missed.

Example event:

```
{
  “topic”:”invoices”, // The topic of the event
  “offset”: “xDy7J”, // The offset of this message in the topic
  "type":"invoicepayment-bookkeep-v1", // A payment of an invoice was bookkept
  "tenantId":29302, // Identifies the tenant
  "entityId":"3817", // Identifies the affected entity (payment in this event)
  "timestamp":"2017-12-28T14:59:16.500+01:00" // When the event was created
}
```

Events are json serialized and contain the tenantId affected and the topic and type of the event. Event types are versioned to allow for breaking changes to be migrated in a controlled manner. Non-breaking changes (adding new fields) will be done without increasing the version.

Events are minimal in payload and clients are encouraged to lookup the entity to find more info.

API design

The api consists of a single duplex websocket stream. When you connect with your credentials, the stream is silent.

Example:
`wss://ws.fortnox.se/topics-v1`
You can now send commands to add topics and tenants to your subscription. When you are satisfied, you can start the stream of events with a subscribe command. You can add new tenants or topics on the fly while the stream is active.

Replaying events

Upon reconnect after downtime you can supply offsets for each topic to get events replayed up to 14 days back. Each event received has an offset. Just supply the offset when reconnecting and events from that offset (not including it) and forward will be replayed. If an offset provided is older than 14 days, the offset will be set to the earliest possible offset instead.

Commands

Command	**Response**

<pre>{
  “command”: “add-tenants-v1”,
  “clientSecret: “6565df”,
  “accessTokens”: [
  “ouhohohhho89796976”,
  ”lijldifdf856587”
  ]
}</pre>
	Adds new tenants to your stream. The response contains only the tenants from the request, not all the currently subscribed tenants.
	
	Possible responses:
	<pre>{
	  “response”: “add-tenants-v1”,
	  “result”: “ok”
	  “tenantIds”: {
	  “ouhohohhho89796”:34231,
	  “lijldifdf856587”:29302
	  }
	}
	
	{
	  “response”: “add-tenants-v1”,
	  “result”: “error”,
	  “invalidTokens”: [
	  “lijldifdf856587”
	  ]
	}</pre>

<pre>{</pre>	Removes tenants from your stream. Possible

"command":"remove-tenants-v1",
  "tenants":
  [12345, 23456]
}
	responses:
	
	{
	  “response”:
	“remove-tenants-v1”,
	  “result”: “ok”,
	}
	
	{
	  “response”:
	“remove-tenants-v1”,
	  “result”: “error”
	}

{
  “command”: “add-topics-v1”,
  “topics”: [
  {
  “topic”: “invoices”,
  “offset”: “hd72U”
  }, {
  “topic”:
“supplier-invoices”
  }
]}
	Adds new topics to your stream.
	
	Possible responses:
	{
	  “response”: “add-topics-v1”,
	  “result”: “ok”,
	}
	
	{
	  “response”: “add-topics-v1”,
	  “result”: “error”,
	  “invalidTopics”: [
	  “invoices”
	  ]
	}

{
  “command”: “subscribe-v1”,
}
	Starts the subscription.
	
	Possible responses:
	{
	  “response”: “subscribe-v1”,
	  “result”: “ok”
	}
	<events>

Tenant ID

Integrators are not familiar with tenantId, which is the primary key used for a tenant in Fortnox. In order for an integrator to identify the correct customer for each event, there is a mapping from accessToken to tenantId returned in the response to add-tenants-v1

command. Fetching the tenantId can also be done in the API using “DatabaseNumber” on the endpoint https://api.fortnox.se/3/settings/company

Deserializer

When deserializing the received packets on the websocket, it might be a problem to determine what deserializer that should be used. The packet can be a response to a command or an event, and events will have different payload based on their type.

Events should have a “payload” field containing a nested json with the fields specific for the given event.

Topics and events

All events currently have similar structure and fields:

```
{
  “topic”: “invoices”,
  “type”: “invoice-created-v1",
  “entityId”: “23”,
  “timestamp”: “2018-01-08T11:47:54.236+01:00”
}
```

topic: invoices

- invoice-created-v1
- invoice-updated-v1
- invoice-cancelled-v1
- invoicepayment-bookkeep-v1
  - Additional fields: invoiceId
- invoicepayment-deleted-v1
  - Additional fields: invoiceId
- reminder-sent-v1
- reminder-sent-v2

topic: customers

- customer-created-v1
- customer-updated-v2
- customer-deleted-v1

topic: orders

- order-created-v1

- order-updated-v1
- order-cancelled-v1

topic: offers

- offer-created-v1
- offer-updated-v1

topic: articles

- article-created-v1
- article-updated-v1
- article-deleted-v1

topic: currencies

- currency-created-v1
- currency-updated-v1
- currency-deleted-v1

topic: termsofdeliveries

- termofdelivery-created-v1
- termofdelivery-updated-v1
- termofdelivery-deleted-v1

topic: waysofdeliveries

- wayofdelivery-created-v1
- wayofdelivery-updated-v1
- wayofdelivery-deleted-v1

topic: termsofpayments

- termsofpayments-created-v1
- termsofpayments-updated-v1
- termsofpayments-deleted-v1