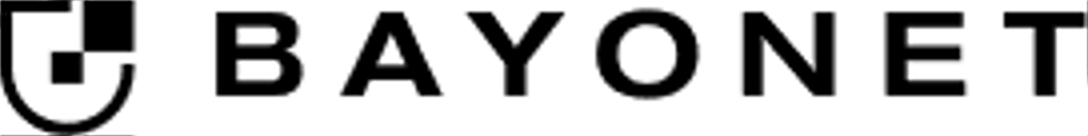
BAYONET PRESTASHOP MODULE



**Internal Manual**

Bayonet’s PrestaShop module is the tool that we developed in order to bring our fraud prevention tools to businesses without the required resources to perform a full implementation of these tools that want to reduce the fraud they are experiencing. All this in an easy-to-install package, making the tasks of installing, configuring and managing, as simple as possible, in this way, it will not matter if the client does not have any technical knowledge.

\*Note: full implementation refers to the implementation without using a platform like in this case, PrestaShop; where that implementation is done by the software department of the business instead of us.

Throughout this manual, we will cover each feature of the module, explaining in detail the way they are implemented.

This manual will help to explain the reason some features, given the limitations of the PrestaShop platform, were implemented in a different way than in a full implementation.

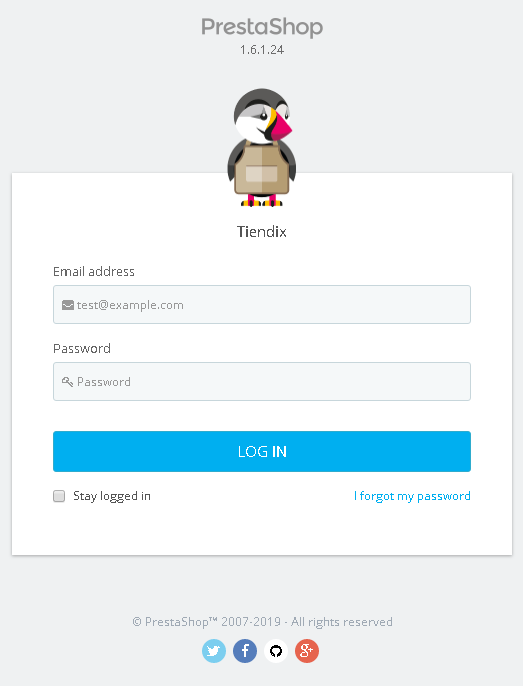
* This module was developed using XAMPP for Windows 64-bits which includes PHP in its 5.6.40 version.
* The module was developed for PrestaShop 1.6.
* The module was tested with the following payment modules:
  + Bankwire (default by PrestaShop)
  + Cheque (default by PrestaShop)
  + Conekta
  + Openpay
  + MercadoPago
  + Paypal

The code of the module follows the standards used in PrestaShop modules:

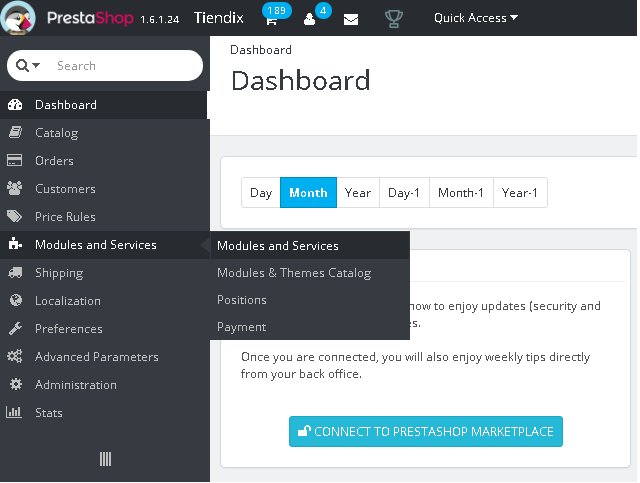
* PHP: PSR-1, PSR-2 and Symfony.
* JavaScript: Airbnb JavaScript Style Guide
* HTML and CSS: Mark Otto’s coding standards
* Smarty: Mark Otto’s coding standards
* SQL: SQL guidelines by PrestaShop

**BAYONET’S MODULE INSTALLATION**

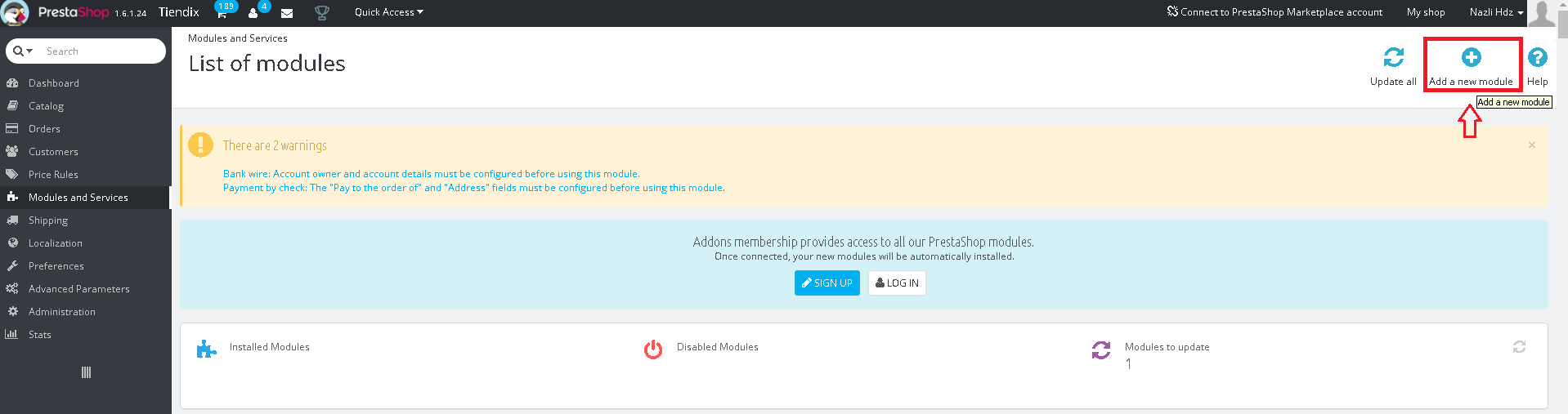
The installation of a module in PrestaShop is pretty straightforward, the first step is to log into the store.



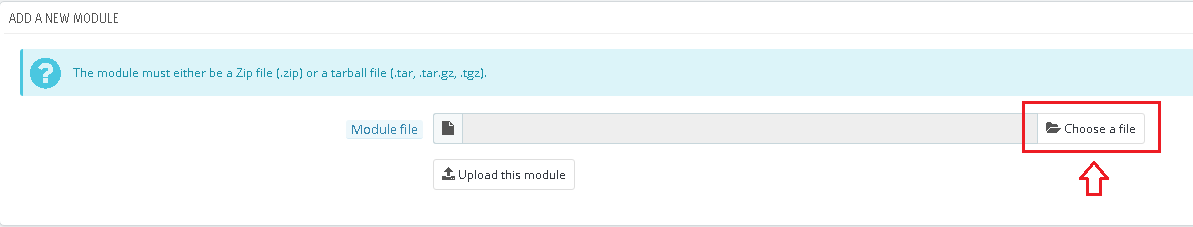
The next step is opening the “Modules and Services” page, this is done by choosing it in the sidebar.

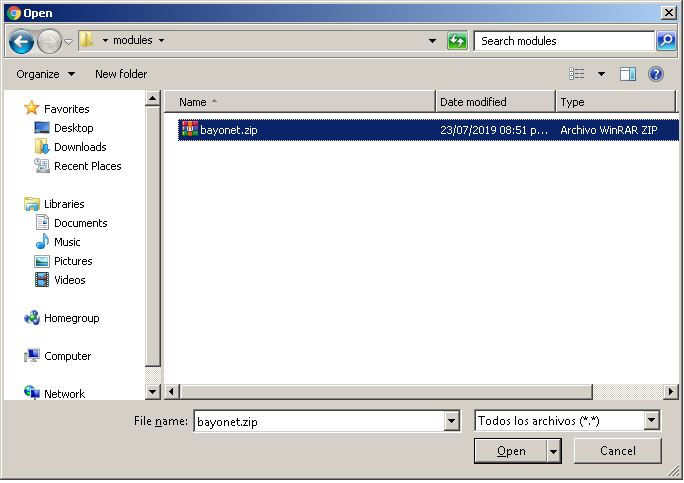


In the “Modules and Services” page there will be 3 buttons at the upper right, “Update all”, “Add a new module” and “Help”. The second one has to be pressed to show a panel to select the file to add.

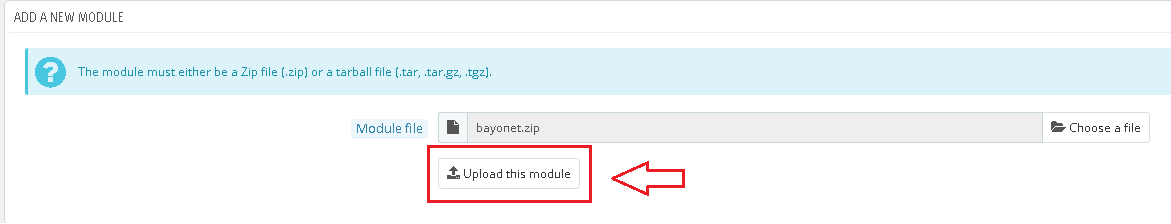


Then, pressing the “Choose a file” button will open a dialog box to select the compressed file to upload, in this case, “bayonet.zip”, which contains all the required files for the module to work.





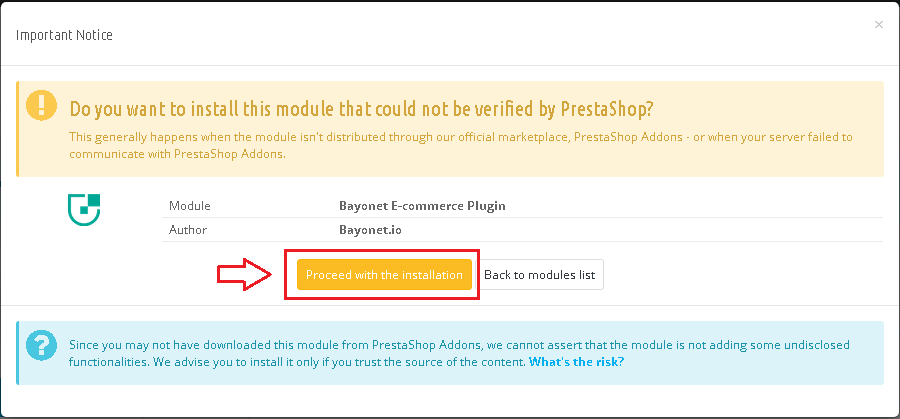
Once the file is selected, the next step is to press the “Upload this module” button, this will add the compressed files and folders to the modules directory of PrestaShop. A confirmation message will be shown and the option to install the module will become available.





Once the module is available in the modules list, the next step is to install it pressing the “Install” button. A dialog box will appear showing the module’s basic information and will ask to confirm the installation, pressing “Proceed with the installation” will do it. After the installation, PrestaShop will display the configuration page of the module.





Installing the module will do several things in PrestaShop’s database. First, it will add the configuration fields of the module to the “ps\_configuration” table, these fields are:

* **BAYONET\_API\_MODE**: sets the module behavior, choosing between the values 0 and 1, indicating sandbox and live modes, respectively; its default value is 0.
* **BAYONET\_API\_TEST\_KEY**: the Bayonet API key to necessary to use the module in sandbox mode; its default value is null.
* **BAYONET\_API\_LIVE\_KEY**: the Bayonet API key to necessary to use the module in live mode; its default value is null.
* **BAYONET\_JS\_TEST\_KEY**: the device fingerprinting API key to necessary to use it in sandbox mode; its default value is null.
* **BAYONET\_JS\_LIVE\_KEY**: the device fingerprinting API key to necessary to use it in live mode; its default value is null.
* **BAYONET\_BACKFILL\_MODE**: sets the backfill mode status, choosing between the values 1 and 0, indicating if the history backfill is executing or not, respectively; its default value is 0.

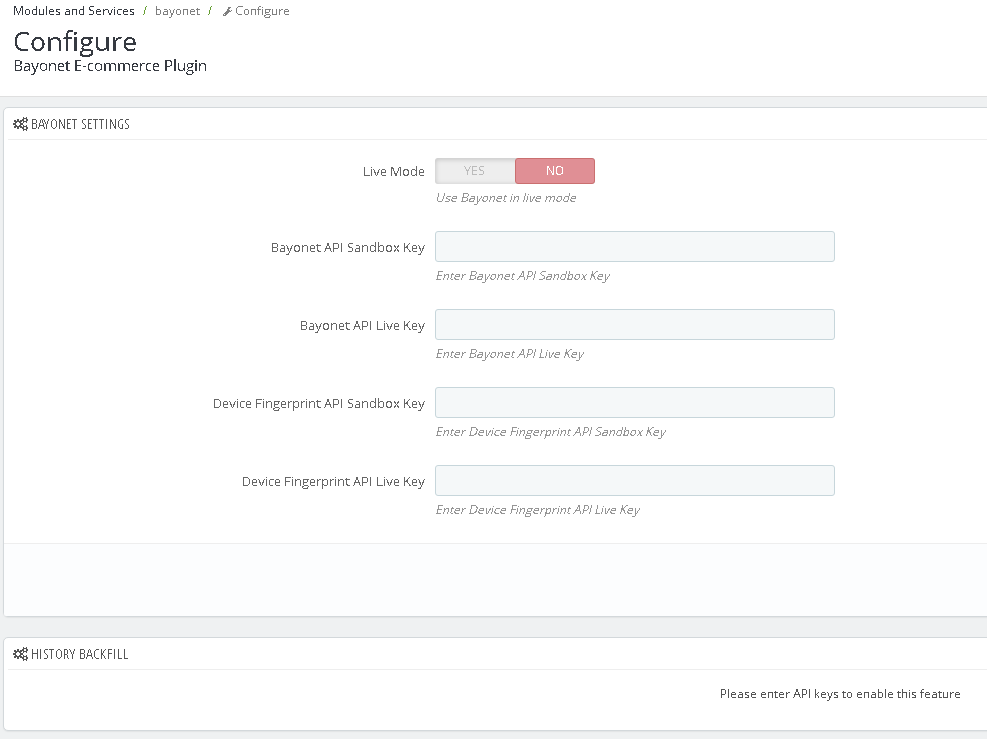
Secondly, it will create Bayonet’s own table “ps\_bayonet”, in which all the orders processed by Bayonet’s APIs will be stored. This table will have the next columns:

* **id\_bayonet**: the identifier of each record on the table.
* **id\_cart**: the cart ID for that PrestaShop order.
* **order\_no**: the order ID of that PrestaShop order.
* **bayonet\_tracking\_id**: the unique identifier generated by the consulting API for that specific order.
* **consulting\_api**: indicates if the consulting process was successful or not on that order (if executed).
* **consulting\_api\_response**: the response retrieved from executing the consulting process on that order.
* **feedback\_api**: indicates if the update transaction (feedback) process was successful or not on that order (if executed).
* **feedback\_api\_response**: the response retrieved from executing the feedback process on that order.
* **historical\_api**: indicates if the history backfill process was successful or not on that order (if executed).
* **historical\_api\_response**: the response retrieved from executing the history backfill process on that order.
* **decision**: the decision given by the consulting API when analyzing that order’s details.
* **rules\_triggered**: the rules triggered by the consulting API when analyzing that order (if triggered).
* **date\_add**: the date when this order was inserted on this table.

Finally, it will add Bayonet’s tab to the back office, in which the user will be able to check the status of the orders processed by the module; also, the hooks to be used throughout the module’s execution are added here.

**BAYONET’S MODULE CONFIGURATION**

After installing the module, PrestaShop will display the configuration window with two sections to configure the parameters to make it work properly.



In the first section, there are the Bayonet settings that are needed to define before utilizing any module’s feature. Each one of these settings are configuration values stored in PrestaShop’s table “ps\_configuration”.

**Live Mode (BAYONET\_API\_MODE)**: this option will set Bayonet’s API mode.

When “No” is selected, every Bayonet function (consulting, feedback, history feedback and device fingerprinting) will be performed in sandbox mode, this is represented by the value of 0 in the configuration table. Selecting “Yes”, will change the value from 0 to 1 in the configuration table, switching from sandbox mode to live mode.

Its default value when installing the module is 0.

**Bayonet API Sandbox Key (BAYONET\_API\_TEST\_KEY)**: this key is needed to use Bayonet’s module in sandbox mode. Provided to the client in Bayonet’s dashboard.

Its default value when installing is null.

**Bayonet API Live Key (BAYONET\_API\_LIVE\_KEY)**: this key is needed to use Bayonet’s module in live mode. Provided to the client in Bayonet’s dashboard.

Its default value when installing is null.

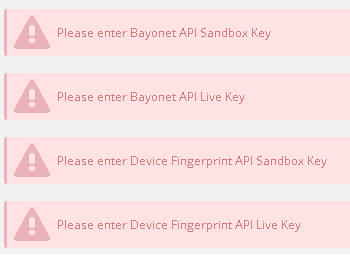
**Device Fingerprint API Sandbox Key (BAYONET\_JS\_TEST\_KEY)**: this key is needed to use the Device Fingerprint API in sandbox mode. Provided to the client in Bayonet’s dashboard.

Its default value when installing is null.

**Device Fingerprint API Live Key (BAYONET\_JS\_LIVE\_KEY)**: this key is needed to use the Device Fingerprint API in live mode. Provided to the client in Bayonet’s dashboard.

Its default value when installing is null.

If any of the fields in the configuration panel is incorrect while trying to save the settings, no value will be saved to the database and an error message will be displayed indicating which setting needs to be corrected before trying to save again, otherwise, a confirmation message will be shown.

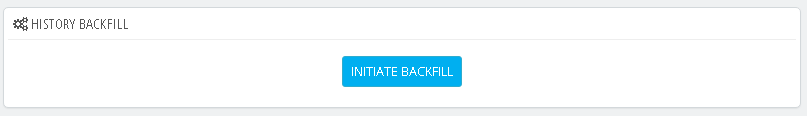




**BAYONET’S MODULE FEATURES**

**History Backfill**

This section is where it is executed the process to analyze the existing orders of the store with Bayonet. At first, this section will appear disabled, this is because the value of the smarty variable controlling what this panel shows is 0 (backfill\_enable), therefore, it is necessary to first add the API keys and save them in order to change the “backfill\_enable” variable to 1, making this section available. Once the API keys are successfully saved and the “backfill\_enable” variable has been changed to 1, it will evaluate the “BAYONET\_BACKFILL\_MODE” configuration value, if it equals to 0, this is how the backfill section will look.



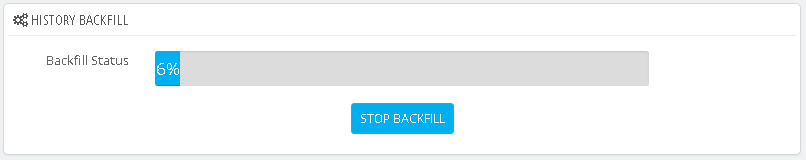
The history backfill will process all the successful orders in the store. The reason of this is because PrestaShop does not have a standard way to know if an order was chargedback, since the statuses of the orders depend on which payment modules a store has installed. A PrestaShop store has two default payment modules/methods when is installed, these are bankwire and cheque, and each one of them has its own statuses. These statuses will work together with the other predefined statuses, which cover the complete flow of an order.

When installing a new payment module, it will add its own custom statuses, for example, openpay will only add “Awaiting payment”, and conekta will add “En espera de pago” for both of its offline payment methods, which are bank transfer and store payment. Another example is MercadoPago, which adds a total of 8 statuses, it uses its own statuses throughout the flow of an order, it does not use the default statuses by PrestaShop, and actually one of these statuses is called “Transaction Chargedback”, but it does not show a reason for the chargeback, which is required in the request body of the API call, besides, it would only work with MercadoPago, it would not be possible to have this information for every payment module.

The situation is similar with the bank decline case. Every payment module has its own payment methods, some have only card payment, while others can also accept bank transfers, store payment and bitcoins; in the specific case of card payment, each module has an iframe in the payment section to validate and process the payment, this means that if a payment is declined by the bank, the module will display an error and it will not proceed to create an order, and no hook will be triggered, therefore, there’s no actual way to know when a payment is declined by the bank.

This is different when it comes to know if a payment was successful. Since every time a payment is successfully validated it is added to the table “ps\_order\_payment”, there is only need to select all the orders from table “ps\_orders” that are on the former, after having those orders, the next step is to select from that set, all the orders that are not on Bayonet’s module table “ps\_bayonet”.

Pressing the “INITIATE BACKFILL” button will start the backfill process. This will change the “BAYONET\_BACKFILL\_MODE” value to 1, indicating that is being executed, it will also reload the page now with this value, which will make the “INITIATE BUTTON” change to “STOP BACKFILL” and show a progress bar above it.



Once all the orders that meet the condition mentioned before have been selected, for each one of them, it will check if the “BAYONET\_BACKFILL\_MODE” value keeps being 1, if this is true, it will proceed to insert a new record on the “ps\_bayonet” table with the order ID and cart ID regarding to that specific order, it will also add a 0 to the “is\_executed” field, meaning that the order has not been processed yet.

After the order is inserted on the table, the process will proceed to execute a call to the feedback historical API, it will use a BayonetClient object, which includes the test or live Bayonet API key, depending on the “BAYONET\_API\_MODE” configuration value, and the callback functions for success and failure. The content of the request body is as it follows:

* channel
* consumer\_name
* consumer\_internal\_id
* transaction\_amount
* currency\_code
* email
* payment\_gateway
* payment\_method
* shipping\_address
  + line\_1
  + line\_2
  + city
  + state
  + country
  + zip\_code
* billing\_address
  + line\_1
  + line\_2
  + city
  + state
  + country
  + zip\_code
* products
* order\_id
* transaction\_status
* transaction\_time
* telephone

Once the request body is generated, the BayonetClient object will execute the feedbackHistorical function, which will make a call to the feedback-historical API (https://api.bayonet.io/v2/sigma/feedback-historical) using the API key and request body defined before.

Depending on the API response, it will set its content to either the success or the failure callback function. In both of these callback functions, the record just inserted on “ps\_bayonet” will be updated setting values for the “historical\_api”, “historical\_api\_response” and “is\_executed”.

In the success function, the value for “historical\_api” will be 1, indicating that on this order, the process was successfully executed, meanwhile, in the failure function, its value will be 0, which means the process was not successful on this order.

After all orders are processed, the configuration value for “BAYONET\_BACKFILL\_MODE” will change to 0, and the execution will finish.

Once the module is properly installed and configured, and the backfill process has been performed successfully, the module is ready to process each one of the new orders of the store.

**Consulting and Device Fingerprinting**

A customer in a PrestaShop store will follow these steps to place a new order:

* Add products to the cart
* Check the cart resume
* Log in (skipped if is already logged in)
* Choose shipping and billing addresses
* Choose delivery carrier/method
* Choose a payment method
* Confirm order

At the moment an order is confirmed, the payment module of the payment method selected will validate (depending on the case) the payment and the order. A successful validation will create a new order in PrestaShop’s database, adding a new record to the “ps\_order” table.

Given the way PrestaShop’s order’s flow works, which is triggering hooks throughout its course, it was necessary to choose one of these to execute Bayonet’s validation process. The problem though, was that no action hook executes right before a payment if submitted.

As stated in the backfill process, each payment module, has its own iframe in the payment section, and that’s where their own validation takes place, which means that pressing the confirmation button to confirm an order will not trigger any hooks until the payment and order details are validated by the payment module, this happens when selecting credit/debit card as payment method since other methods like bank transfer or store payment require the confirmation of the PrestaShop store’s manager, in these methods, only the order details are validated when confirming the order.

Having that situation, it was necessary to find the proper way to implement the behavior of Bayonet’s module in order to do what it was intended to do. At first, considering how PrestaShop works, using a hook was the most viable option, but since it was needed to do Bayonet’s validation before actually placing, none of the hooks met the requirements to do that.

In order to choose the best solution, a deep research was performed, which included:

* Checking all the available hooks on PrestaShop 1.6 and 1.7
* Checking if it was possible to create a custom action hook
* Posting on PrestaShop forums and stackoverflow to see if someone has dealt with this situation before and could provide with a solution
* Performing tests with javascript, trying to intercept the order’s flow before trying to validate the payment
* Checking the way other fraud detection modules dealt with this situation

After performing this research and tests, the first option was still being the best option, which was using a hook, but taking into account PrestaShop limitations and considering how other modules handled this situation, the approach had to be changed, instead of validating before the order’s placement and alerting the customer of being rejected, the order was going to be validated right after placing it and the alert would appear in the back office in the order’s details, in this way, the store’s manager would be able to take actions on the orders depending on Bayonet’s decision and the customer experience would not be altered.

The hook to use for this feature’s implementation was “actionValidateOrder”, formerly known as “newOrder”, which is triggered right after a new order has been created. To use a hook, you define a new function with the name of the hook on it, in this case “hookActionValidateOrder”, this function will receive an array, this array and the current context object, both contain the data available at that moment of the cart, order, customer, currency and cookies. Using these variables, it is possible to construct the request body for Bayonet’s consulting call.

The request body for the consulting call includes the same data fields as in the historical feedback call, plus the Bayonet Fingerprint Token:

* channel
* consumer\_name
* consumer\_internal\_id
* transaction\_amount
* currency\_code
* email
* payment\_gateway
* payment\_method
* shipping\_address
  + line\_1
  + line\_2
  + city
  + state
  + country
  + zip\_code
* billing\_address
  + line\_1
  + line\_2
  + city
  + state
  + country
  + zip\_code
* products
* order\_id
* transaction\_status
* transaction\_time
* telephone
* bayonet\_fingerprint\_token

It is important to explain the way the bayonet fingerprint token, and both the payment gateway and payment method are obtained.

The fingerprint token, needs to be generated before confirming the order to be available at the moment the request body is constructed. This is done by executing the fingerprinting javascript snippet at the moment the customer chooses which payment module to make their payment. Once it is executed and generated, it sends that token to a php controller via an ajax call to add it as a cookie value to the context, in this way, when the “actionValidateOrder” hook code is executed, the token will be available to add it to the request body; its value in the context will be removed after adding it.

Regarding the payment gateway and method, PrestaShop does not have a standard way to name a payment method, a payment module handles the name of its payment methods in different ways.

The order object in the array received in the hook function, contains two variables, which are “module” and “payment”, the former refers to the module used to make the payment, while the latter refers to the name assigned to that payment method by the payment module.

Two examples of this are, first, the conekta payment module, this module has 3 payment methods (card, bank transfer, store payment) and regardless of which method is chosen, the value of the “payment” variable will be “Conekta Prestashop”.

The second example is OpenPay, in this case, the “payment” variable contains a description of the actual payment method used to make the payment, this description is helpful in knowing with certainty, what value assign to the “payment\_method” element in the request body.

Having that in mind, it was decided that to get the payment method of an order, two approaches were going to be applied. The first one, in the case of payment modules like OpenPay, it’s easy to check if the payment description contains words like “tarjeta”, “card”, “store”, “tienda”, etc., then assign the corresponding value.

The second one, while dealing with payment modules like conekta, it was decided to handle it as “tokenized\_card”, since there’s no way to detect at that moment which method was chosen by the customer.

Handle it as “tokenized\_card” goes hand in hand with the first approach, given the fact that if in a payment module like OpenPay, card is detected as payment method, the card information is not available, which makes sense since that would mean the module is not securing the customer’s card information; this is why even though the method can be detected in those modules, defining it as “credit\_card” is not possible because the required data is not accessible, so it had to be defined as “tokenized\_card”.

Once the request body is completed, a BayonetClient object is created, this object, like in the backfill process includes the Bayonet API key, the request body and the success and failure callback functions.

This object executes the consulting function (https://api.bayonet.io/v2/sigma/consult), if the execution’s response is successful it will insert a new record on “ps\_bayonet” table, this will contain:

* Cart ID (id\_cart)
* Order ID (order\_no)
* Decision (decision)
* Rules Triggered (rules\_triggered), if triggered
* Bayonet Tracking ID (bayonet\_tracking\_id)
* Consulting API (consulting\_api)
* Consulting API Response (consulting\_api\_response)
  + Reason Code (reason\_code)
  + Reason Message (reason\_message)
* Is Executed (is\_executed)

If the execution’s response is not successful, the data to insert will be the same, except it will not contain the rules triggered element.

Every time a consulting call is executed, the “consulting\_api” value will either be 0 or 1, which will indicate if the response was unsuccessful or successful, respectively.

**Update Transaction**

The update process takes the same approach as in the feedback historical process. Since there is no a standard way to know if a payment was rejected or charged back, it was decided to update only the successful orders, this is, when a payment is accepted.

There are various hooks that are triggered when a payment is accepted, the most important are “actionObjectOrderPaymentAddBefore”, “actionPaymentConfirmation” and “actionOrderStatusUpdate” which makes it possible to make the update call right when the payment is validated, regardless of what payment method was chosen.

At first, “actionPaymentConfirmation” was the chosen hook, but after performing tests with various payment modules, MercadoPago was not triggering it after the payment was made.

After analyzing the behavior of a payment being accepted, it was detected that the status of an order is changed after validating the payment; the statuses are stored on the “ps\_order\_state” table, and one of its columns is named “paid”, which can have a value of either 0 or 1, indicating not paid and paid, respectively.

With this, when a payment is confirmed, it’s only necessary to check if the new status’ paid value equals to 1, if that’s true, then the update transaction call is performed.

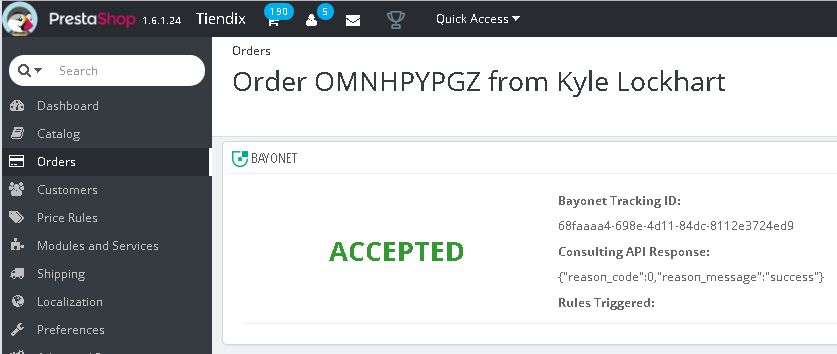
When the hook is executed, it will run the code inside its function implementation. The first step is to get the Bayonet order to update using the order ID contained in the parameters received by the function. If no order is selected, then no update process will be performed, otherwise, it will proceed to check if the selected order has a valid Bayonet Tracking ID value, if this is true it means that order was processed by the consulting API, therefore, it will create the request body using the Bayonet Tracking ID and the transaction status, which since it is processing an order with a payment just validated, will be “success”.

After generating the request body, the next step is the same as in the consulting and feedback features, which is creating a BayonetClient object with the API key, request body and callback functions. The success callback function will update the Bayonet order in the database, adding a 1 to the “feedback\_api” column and the code and message retrieved from the response to the “feedback\_api\_response” column. In the failure callback function, the value for the “feedback\_api” will be 0.

**Bayonet Panel in Order Details in the Back Office**

At the back office of a PrestaShop store, it is possible to check all the information regarding all the orders, this is done by selecting the “Orders” category in the “Orders” tab. This will show a table with every order until that moment, clicking on one of these orders, will open a new page with several panels in it with information about the order itself, the customer, the products, the shipping method, etc.

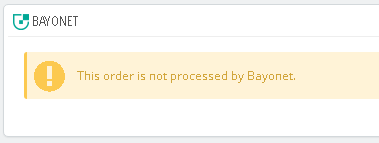
A panel was added to display the Bayonet information regarding that specific order, which looks like this:

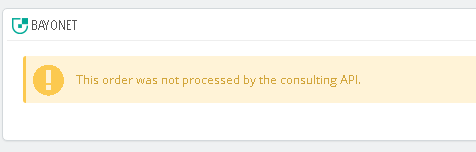


This panel, will include the resulting decision of the Bayonet analysis displayed in a bigger font size and using a color illustrating the decision. Next to the decision, 3 more elements will be shown, these are the Bayonet Tracking ID, the Consulting API response, and (if triggered) the rules triggered.

At the moment an order is selected from the table shown in the back office, a hook is triggered, “displayAdminOrder”, using this hook’s implementation, it is possible to add the Bayonet panel to the details. Using the order ID in the array received as parameter, the select query is executed on the “ps\_bayonet” table, if no record is retrieved, it will assign a true value to the smart variable “unprocessed\_order” in the “admin\_order” template, which is where the html code to generate and show the panel is contained. If an order is indeed retrieved but has no Bayonet Tracking ID, it will set “not\_consulting\_order” to true, which will indicate that the selected order was not processed by the consulting API, these two cases are illustrated in the images below.

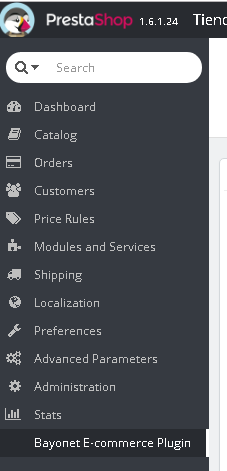
Finally, is an order is retrieved and has a Bayonet Tracking ID, it will display the information as shown in the image above; the decision will be evaluated to change the font color depending on its value, “Accepted” will be green, “Review” will be orange and “Declined” will be red.



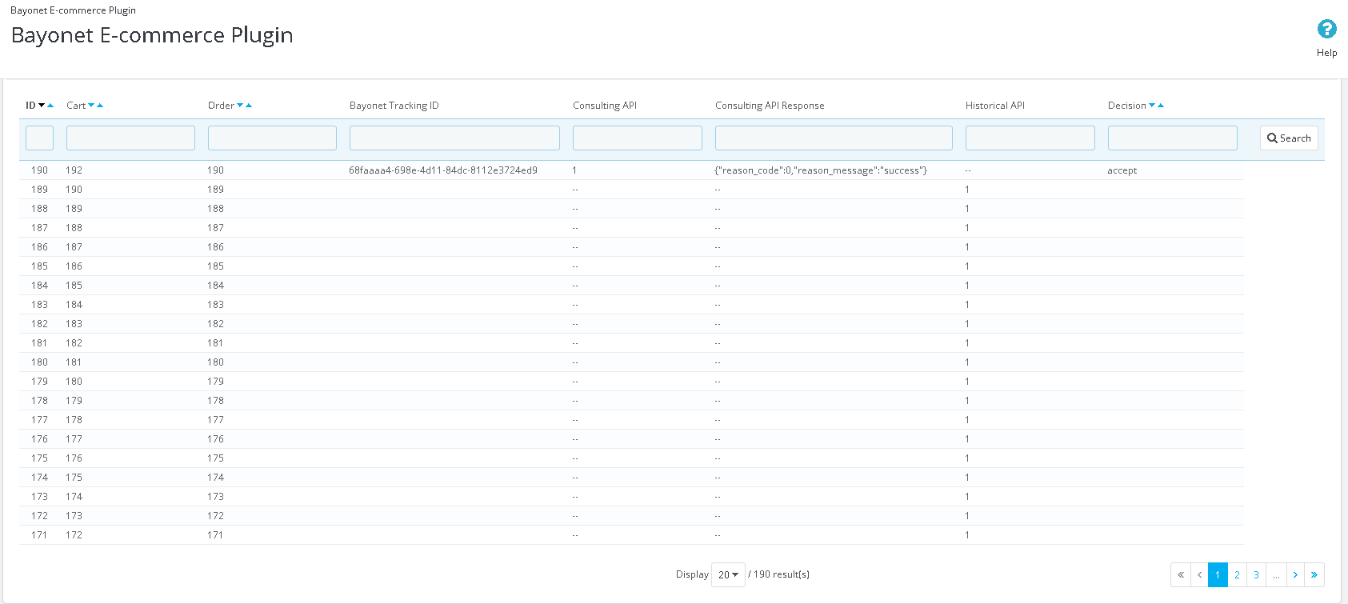


**Bayonet Tab in Back Office**

The module’s installation adds a new tab in the back office, this is located at the bottom of the sidebar, with the legend “Bayonet E-commerce Plugin”.



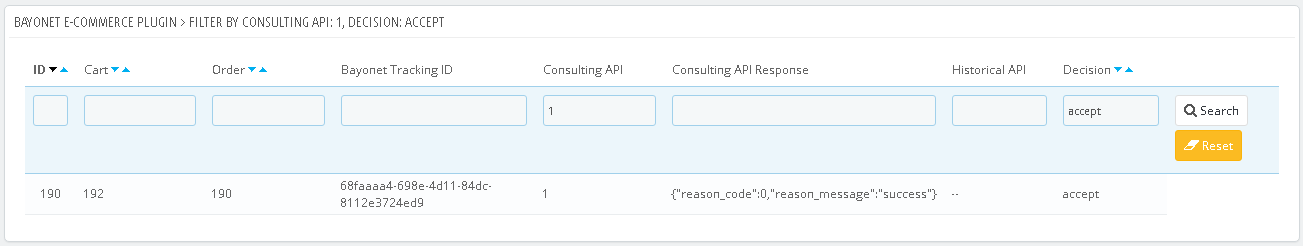
This tab contains a controller, similar to the “Orders” tab, the controller shows a table with all the records on Bayonet’s table in the database.



The columns of this table are:

* **ID**: the ID of the table.
* **Cart**: the cart ID of that specific order.
* **Order**: the ID of the order.
* **Bayonet Tracking ID**: the tracking ID generated by Bayonet at the consulting process.
* **Consulting API**: indicates the execution result of the consulting process for that order. This is useful to know if an order was processed using the consulting or the history backfill process.
  + Empty: not processed by consulting API
  + 0: unsuccessful
  + 1: successful
* **Consulting API Response**: the response retrieved after attempting to perform the consulting process, it will include the response code and its message.
* **Historical API**: indicates the execution result of the backfill process for that order.
  + Empty: not processed by history backfill
  + 0: unsuccessful
  + 1: successful
* **Decision**: the decision obtained in the consulting process (if the order was processed by the consulting API).
  + Accept
  + Review
  + Decline

The table also has filtering, ordering and pagination. At the top, below the column names a text box is located for each one of them, in these text boxes, it is possible to enter values for the filtering criteria, also, the filter accepts more than one column which can get more specific results. For the ordering feature, only four out of the eight columns can be used for this, these are (ID, Cart, Order and Decision); to perform an ordering, it is necessary to press one of the two chevron arrows next to the name of these four column, the downward arrow will do a descending order, while the upwards arrow, will arrange the rows in an ascending order.



Finally, in the lower part of the table, a pagination will appear if the table has more than 20 rows, to define how the pagination will work, it is necessary to set the number of rows to display in a page, it is possible to choose between 20, 50, 100, 300, 500 and 1000 rows per page, once this value is set, the table will be refreshed showing the selected number of rows, and if not all the orders are being displayed in the current page, the pagination feature will become available.

