Transactional messages in Custobar

Transactional messages are triggered and populated by custom built events, where the type and the attributes will define which campaign should fire and what content should be displayed in the campaign content.

The trigger events can be used to fire an email, an SMS or a push, so in order to better sort out the different channels and triggers, it is good practice to include the campaign name, language version and channel type in the event type. This can make debugging, pairing triggers and automation, finding out why a customer received a specific message and testing much easier.

For example, the trigger for a transactional email could be event.type “ORDER_CONFIRMATION_COM” and the trigger for the matching SMS could be “ORDER_COMFIRMATION_SMS_COM".

The Custobar data schema uses a default field for transactional events labeled “transactional_message_data”. This field contains the content variables to be displayed in the html rendering as one or multiple nested objects (see example below). Those variables will define the content rendering, with the possibility of conditional rules to hide or show content based on the value (for example depending on the shipping method, optional purchasing options etc.).

In the html code, the activation snippet {% load tags %} will enable the use of this tag {% with e=events.0 %} which in turn extends the data from the trigger event.

It will then be possible to use {{ e.xxxx }} placeholders to fill in the html with the strings or values in the event data. For example, {{ e.paymentMethod }} could be used to show the payment method in bolder text.

In advanced cases, the attributes can also be used to specify the method used to contact the customer, for example the presence or not of a phone number in the data can be used to redirect the message through an SMS channel or default to an email.

All the content to be displayed should be included in the object, including url for images, product titles and descriptions, prices and the link. This is necessary since in the context of transactional messages, fetching that data from the database would potentially take minutes which is too long for an order confirmation email, a pick-up ready SMS or any other time-critical messages.

Please note: While the default Event based trigger and module is recommended, it is also possible to use a Sale based transactional trigger and dynamic module.

The requirements are for the Sale to contain the field "sale_id" on top of the usual Sales fields.
The dynamic module properties will be set to the "Email sale module" type, and the tag `{%`

`with sales.0 as s %}` will allow extending the Sales data, for example `{{ s.external_id }}` or `{{ s.date|parse_date|date:"d.m.Y" }}`

When using the Sale based trigger, you will need a way to extend product data since the product image url, link and description might not be included in the Sales data directly. To do so, use the tag `{% for p in s.products|extend_sale_products_data:"xxx" %}` where xxx must be replaced with your Custobar environment's id.

The data can then be fetched with tags such as {% format_price p.unit_price "." 2 "EUR" %} or {{ p.product_id }}

Example of json object

```
     [
        {
            "transactional_message_data": {
                "lines": [
                    {
                        "OrderedQuantity": "2",
                        "itemDescription": "Article 1",
                        "itemProductcode": "121212",
                        "lineAmountInclVAT": "15,00",
                        "unitPriceInclVAT": "30,00"
                    },
                    {
                        "OrderedQuantity": "1",
                        "itemDescription": "Article 2 , flower
bouquet BB",
                        "itemProductcode": "ZBDBD",
                        "lineAmountInclVAT": "7,80",
                        "unitPriceInclVAT": "7,80"
                    },
                    {
                        "OrderedQuantity": "1",
                        "itemDescription": "Shipping fee",
                        "itemProductcode": "SHP",
                        "lineAmountInclVAT": "5,00",
                        "unitPriceInclVAT": "5,00"
                    },
                    {
                        "OrderedQuantity": "***",
                        "itemDescription": "***",
                        "itemProductcode": "***",
                        "lineAmountInclVAT": "***",
                        "unitPriceInclVAT": "***"
                    }
                ],
                "orderAmountInclVat": 5,
                "orderNo": "3028827",
                "orderShippingCost": 5,
                "orderVatAmount": 0.97,
                "paymentMethod": "Postal_pickup",
                "sellTo_address": "Acmekatu 1",
                "sellTo_city": "Acmeville",
                "sellTo_email": "test @acme.com",
                "sellTo_name": "Tester",
                "shipTo_address": "Acmekatu 1",
                "shipTo_city": "Acmeville",
                "shipTo_name": "Tester",
                "shipTo_phone": "525225",
                "shipTo_postcode": "00100"
            },
            "customer_id": "200002",
            "date": "2021-11-24T10:39:08+01:00",
            "email": "test@acme.com",
            "type": "ORDER_CONFIRMATION"
        }
    ]
```

Testing the automation

The automation can be tested through the regular production channel, ie. placing an order in the appropriate system and letting the whole setup run its course up until the order confirmation reaches you or placing items in your wishlist and waiting for the back in stock notification when they become available again.

To speed things up, the automation can also be tested by pushing a dummy event in Custobar through the manual import tool or the events API. This will allow you to push any data combination to test the content rendering without having to deal with all the steps preceding this stage.

Please note that in order to test the proper channel, the matching trigger type or attribute combination must be used. Ie. if using the existence of an email address or phone number in the data object to decide whether an email or SMS will be fired, it is important to send multiple tests, one with email, one with phone number, one with both and one with neither to make sure the right channel is used in each case, and that all combinations are anticipated in the strategy used.

The same can be said for any variable that will modify the content displayed, language versions, or amount of data rows displayed.

Whether using production or dummy events to trigger the message, please remember that test messages will not carry the html rendering and will only show empty content.

A production message is required to fire with the final render. To that end, it is important to make sure only the test recipients will be using the trigger event types, and until the testing phase is successfully completed, regular customers' purchases and other actions should not be connected so they do not trigger any transactional events or automation firing incomplete or empty messages in large amounts.

In the case of dummy events, it is also important to make sure the date of the trigger event is somewhat close to “right now” so the automation fires right away and not several days in the future.

Templates

Custobar event modules are created to properly pick up the data from the transactional trigger and render it in a pleasant display for the recipient.

In addition to regular HTML rules, the Email event modules use Django tags as well as these Custobar specific additional tags using this activation snippet : `{% load tags %}`

add_company_id(customer, company_id=None)

add_control_class(field)

add_domain(url, company)

automated_email_link(context, key)

campaign_link(context, campaign_action_id)

cap_value(value, limit)

customer_link(context, customer_id)

customer_name(data, default='')

extend_products_data(events, company_and_language)

extend_sale_products_data(sale_products, company_and_language)

format_json_if_dict(data)

format_number(number, decimals=2)

format_price_tag(context, price, separator=None, decimals=2, currency=None, full_name_of(context, user)

get_should_render_main_nav(context)

hash_md5(txt)

image_exists(company, file_path)

img_url(url)

is_checkbox(field)

is_file(field)

is_multiple_checkbox(field)

is_radio(field)

is_subfield(field)

join_with_new_line(data)

json(data, indent=None)

load_snippet(url)

make_secure(url)

mask_email(s)

normalize_event_type_name(context, name)

parse_date(date_string)

product_image(data, company)

product_link(context, product_id)

product_title(data, language=None)

raw_json(data, indent=None)

remove_filter(filters, kv)

reverse(txt)

role_name_of(context, user)

safe_list(str_or_list)

sentry_js_dsn()

split(string, separator)

sub(value, arg)

transjs(key)

translate(key, company, lang)

translate_ctx(context, key)

translatejs(key, company, lang)

update_filters(filters, key, operator, val)

username_by_id(id)

username_or_id(d)

These tags are subject to change as new Django tags filling the same needs will replace the Custobar specific tags.