Twikit Custom Integration

1.1.2

Twikit Custom Integration

This document is the technical documentation for integration with Twikit Customization. It will give you access to our JavaScript Plugin (visualisation/customization engine) and our automated manufacturing process.

Integration Publisher

The Integration Publisher manages everything related to the integration. It provides:

- a testing environment for new products, as well as changes to existing products
- an API to our manufacturing process
- a configuration interface for webhooks and other settings
- a link to our JavaScript Plugin (which governs visualization and customization)
- a comprehensible guide on its usage

Customer -> Production Flow

1. Customer goes to the Client website
2. Customer navigates to a twikit customizable product
    - Client has the Twikit Product Key linked to this product (key can be found in the publisher)
3. Client loads the Twikit JavaScript Plugin
4. Client tells the JavaScript plugin which product to load
5. The JavaScript plugin creates a visualisation, and a set of parameters.
6. Client shows the visualisation and the parameters to the customer
7. The customer changes parameters until satisfied and clicks the “add to cart” button
8. Client tells the JavaScript API to save the customized product, and gets a Configuration ID
9. Client links the Configuration ID to the order item ien the cart
10. The customer proceeds to checkout and pays for their order
11. Client verifies the payment, all goes well
12. Client sends an automated signal to Twikit (Rest API) to start production on these paid order items
13. Twikit verifies the integrity of the order with security algorithms
    - This can include double-checking with Client’s servers to make sure the order is valid
14. Twikit creates a printable 3D mesh and alerts the manufacturer
15. The manufacturer downloads the mesh and prints it, then sends it to Client
16. The manufacturer logs into our system and marks the order as complete
17. Twikit uses a configured Webhook to send a signal to Client that the order has been printed
18. Client finishes the order and handles shipping and client communication

Product creation Flow

1. Client asks Twikit to create a product
    - Alternatively, Client may use Twikit’s Product Builder to create a product
2. Twikit creates the product for Client and adds it to the system
3. Client verifies the product is working as intended using the publisher
4. Twikit publishes the product
    - This will make it available for inclusion on the Client website
    - Twikit sends a signal to the Client system that a new version is available
5. Client includes the newly published product on their website

JavaScript Plugin

Including the script

To use the JavaScript API, which enables visualization, you must first include the Twikit Plugin Script. This script contains mostly logic, but also some data required for rendering. For this reason, every time we change something to any product, we will, after testing, publish a new script containing all required logic & data to a new URL, which will trigger a webhook to your system. This is done to guarantee maximum stability.

Initializing the script

First, it is recommended you check if the browser is compatible with the twikit software:

```
window.twikit.canVisualizeForBrowser(); // returns true or false
```

Once the script is loaded, you will be able to initialize the JavaScript plugin in the following way:

```
var twikitContext = window.twikit.createContext(productKey, locale, target,
options);
```

Parameters:

productKey	the key of the product (you can get the keys of the products from our publisher).
locale	The localization to use for translations.
target	The target element for the visualisation (html element, jQuery element or jQuery selector)

options	JSON object which may contain any of these callbacks:
	
	`loaded: function(): void`
	`parametersReady: function(parameters: JQuery): void`
	`statusChanged: function(progress: number, status: string): void`
	`variationChanged: function(variation: IVariation): void`
	`previewBackgroundColor: string //#ffffff by default`
	
	`IVariation: {`
	`  id: string,`
	`  sku: string,`
	`  price: number`
	`}`
	
	`loaded` is called when the plugin is fully loaded and drawn.
	
	`parametersReady` is called when the parameters are ready. `parameterElements` contains HTML elements containing linked & bound parameters.
	
	`statusChanged` is called when the status changes. `progress` is a percentile progress towards rendering (0-100). Status can be `ready`, `loading` or `busy`.
	
	`variationChanged` is called when a change in parameters causes the configuration to be of another variant. Variants are defined to group categories of customization into chunks that have the same price and SKU.
	
	`previewBackgroundColor` is the color to be used in the background when the preview is rendered (for cart etc.) If left blank, transparent is used.

Example use:

```
var visualisationTarget = $('#visualization');
window.twikit.initialize('diamond-letter-jewel', 'default',
visualisationTarget , {
    loaded: function() { console.log('twikit is done loading'); },
    parameterReady: function(parameters) {
        $('#parameters').append(parameters);
    },
    statusChanged: function(progress, status) {
        $('#progressBar').css('width', progress + '%');
        if(status == 'ready')
            $('#progressBar').hide();
        $('#spinner').toggle(status !== 'ready');
    },
    variationChanged: function(variation) {
        console.log('variation changed', variation);
    },
    previewBackgroundColor: '#eeeeee'
);
```

Adding a product to the cart

When the customer adds the product to their cart, we must be able to identify the customization parameters for their order. Twikit will store this data for you and generate a customized image to display in the cart overview or during checkout. We will provide a unique ID for this configuration, which will allow you to later (after payment), schedule this configuration for production.

This is done in the following way (assuming the variable `twikitContext` is initialized as examplified earlier):

```
twikitContext.save(savedCallback);
```

Parameters:

savedCallback	This function is called when the product is done saving. Its signature should be:
	
	`function(err: any, saveData: ISaveData): void`
	
	The “saveData” parameter contains these options:
	
	`ISaveData: {`
	`  configurationId: number,`
	`  previewUrl: string`
	`}`

Example use:

```
$('#orderButton').click(function() {
    twikitContext.save(function(err, saveData) {
        if(err) return console.warn(err); // maybe alert customer?
        $('#twikit-configuration-id').val(saveData.configurationId);
        $('#customized-preview-url').val(saveData.previewUrl);
        $('#orderForm').submit();
    });
});
```

Rest API

For transactions that require security, or are not directly dependent on user input, we provide a rest API that can be consumed by Client.

The endpoint (base url) for this api is:

```
https://custom-integration-publisher.twikit.com/api/v1
```

Create Order

In order to start production of a client's configuration, an API call must be made to create an order. This order will then automatically be executed by Twikit. An order is a group of configurations earlier stored by Twikit.

To create an order:

endpoint	/orders/{integration-id}
method	POST

request	{
	  orderPayload: string, // stringified IOrderPayload
	  hash: string
	}
	
	IOrderPayload:{
	  items: IOrderItem[],
	  identifier: string,
	  billingAddress: IOrderAddress,
	  deliveryAddress: IOrderAddress,
	  data: any,
	}
	
	IOrderItem: {
	  configurationId: number,
	  amount: number,
	  identifier: string
	}
	
	IOrderAddress: {
	  firstName: string,
	  lastName: string,
	  email: string,
	  addressLine1: string,
	  addressLine2: string, // optional
	  zip: string,
	  city: string,
	  countryIso: string,
	  companyName: string, // optional
	  companyVat: string, // optional
	  phone: string // optional

	} `Integration-id` is the identifier as shown in your site's configuration. `payload` is a string containing the json of the `IOrderPayload` type. This is sent intentionally as a string to ensure consistent hash generation on both sides. `items` is an array of order items belonging to a single order. Only items with a twikit configuration ID should be sent to our API. These configuration IDs should be those returned by the `save` function on the `twikitContext` in JavaScript. `identifier` should contain your unique identifier for the order/orderItem. We will use this later (see webhooks) to verify the integrity of the order to prevent fraud. Twikit will only process an order once. If you send two orders with the same identifier, they will be considered the same orders and the second one will bounce back. `amount` is the amount to be produced. `data` can contain any data. It will be sent back for validation, we will ignore its contents. `hash` is the security hash. This must contain `md5(orderPayload + payload-secret)` to ensure no alterations to the payload occurred in transit. Our application will provide you with this payload-secret, it will be valid until invalidated and can be found in your site's configuration. `billingAddress` & `deliveryAddress` should contain the addresses to which the order should be billed and delivered. If the order should be delivered to you for post-processing, you should set the deliveryAddress to your address. Similarly, in most cases the item will be billed to you after which you generate the customer's bill, so you can set the billing address to your address too.

response	{
	`resultMessage: string,`
	`resultCode: number,`
	`resultCodeName: string`
	}
	
	If all goes well, you will receive an response with http status 200 and the message that the order has been created and either set to paid (resultCode 1) or unhealthy (resultCode 2).
	For all other failure paths the http status code will not be 200 (usually 400 or 500 depending on the error) and the resultcode and resultmessage will give you more insight in what went wrong.
	
	At the moment the defined resultcodes are:
	0: **UnexpectedError**, the catch all for not yet finer defined errors
	1: **OrderCreatedSuccesfully**, the order was created successfully (Succes!)
	2: **OrderCreatedUnhealty**, the order was created but could not be verified (see verification webhook)
	3: **VerificationFailed**, webhook verification declared the order invalid
	4: **OrderPayloadParsingFailed**, we could not parse the orderPayload object
	5: **HashParameterEmpty**, the hash parameter was empty
	6: **Md5HashMismatch**, the md5 hash did not match the orderPayload string
	7: **OrderPayloadParameterMissing**, something went wrong creating the order from the order payload. Check the message to see what went wrong.
	8: **OrderPayloadValidationFailed**, the order payload is missing a required parameter. Check the message to see which one.

	9: **IdentifierAlreadyExists**, the orderPayload or orderItem identifier is already in use in our system, indicating and attempt to create an order that was already created. 10: **CouldNotFindConfig**, One of the configurations in the order could not be found. As the system evolves, more resultcodes might be added but once defined they should never change. The only exception is errors that currently return 0: UnexpectedError might in the future be updated to return more specific errors if they occur often enough.

Order Production File

Instead of creating a Twikit-fulfilled order, producible files can be directly requested by configuration ID. Twikit will automatically create an internal order when this happens.

To directly order a production file:

endpoint	/configuration/{integration-id}/export
method	POST

request	`{`
	`  payload: string, // stringified IConfigExportPayload`
	`  hash: string`
	`}`
	
	`IConfigExportPayload:{`
	`  configurationId: number,`
	`  amount: number,`
	`  data: string | json serializable object`
	`}`
	
	`Integration-id` is the identifier as shown in your site's configuration.
	
	`payload` is a string containing the json of the `IConfigExportPayload` type. This is sent intentionally as a string to ensure consistent hash generation on both sides.
	
	`hash` is the security hash. This must contain `md5(payload + payload-secret)` to ensure no alterations to the payload occurred in transit.
	Our application will provide you with this payload-secret, it will be valid until invalidated and can be found in your site's configuration.
	
	`configurationId` is an of order items belonging to a single order. Only items with a twikit configuration ID should be sent to our API. These configuration IDs should be those returned by the `save` function on the `twikitContext` in JavaScript.
	
	`amount` is the amount of items that will be produced based on the file.
	
	`data` is an optional attribute that may be used to send extra (third-party) data to our systems. This data must a serializable json object or a string.

response	If all goes well, you will receive an response with http status 200. The body of the response will contain a ZIP file containing the produced files. For all other failure paths the http status code will not be 200 (usually 400 or 500 depending on the error) and the `resultCode` and `resultMessage` will give you more insight in what went wrong. At the moment the defined resultcodes are: 0: **UnexpectedError**, the catch all for not yet finer defined errors 1: **OrderCreatedSuccesfully**, the order was created successfully (Succes!)

	2: **OrderCreatedUnhealty**, the order was created but could not be verified (see verification webhook) As the system evolves, more resultcodes might be added but once defined they should never change. The only exception is errors that currently return 0: UnexpectedError might in the future be updated to return more specific errors if they occur often enough.

Webhooks

When Twikit wants to verify a payment or let you know that production of a certain item is finished, we will attempt to use a webhook to verify this with you. The endpoint for these webhooks can be configured in our user interface. We will assist you with the set-up of these webhooks.

Payment Verification

After you have sent us a payment, we will use a web-hook to verify the validity of the transaction. You must check that the order is indeed in a status of being confirmed.

action	If no webhook is configured, we will not verify payments. (not advised!) If payment verification is faulty (error), we will create the order as “unhealthy”. It will not start automatically and we will contact you. If payment verification is unsuccessful (invalid order), we will ignore the order. If payment verification is successful, we will create the order and automatically process it.
endpoint	*as configured by you in our system*
method	POST

request	`{`
	`  configurationIds: number[],`
	`  identifier: string`
	`}`
	
	These are copied over from the original request as sent to us by you.

response	`{`
	`  error: string,`
	`  success: boolean`
	`}`
	
	If `error` is filled in, or the status code is not 200, we will assume a faulty verification.
	If `success` is `false`, we will assume an unsuccessful verification.

Production Completion

action	If no webhook is configured, we will not send you this message If the webhook is faulty (error), we will retry once after 5 minutes.
endpoint	*as configured by you in our system*
method	POST

request	`{`
	`  identifier: string;`
	`  url: string;`
	`  success: boolean;`
	`  error: string;`
	`}`
	
	`identifier` will contain the identifier for the order originally passed to us.
	`url` will contain the tracking data provided by the producer.

response	`{`
	`  error: string`
	`}`
	
	If `error` is filled in, or the status code is not 200, we will assume a faulty verification & retry once.

Order Completion

Our back-office system allows the producer to mark an order as “ready”. At this point, we will automatically send you an update within a few minutes.

action	If no webhook is configured, we will not send you this message If the webhook is faulty (error), we will retry once after 5 minutes.
endpoint	*as configured by you in our system*
method	POST

request	`{`
	`  identifier: string;`
	`  trackingUrl: string;`
	`  trackingId: string;`
	`  lineItems: ILineItem[];`
	`}`
	`ILineItem: {`
	`  identifier: string;`
	`  configurationId: number;`
	`  previewUrl: string;`
	`  productKey: string;`
	`  amount: number;`
	`}`
	
	`identifier` will contain the identifiers for the order & line item originally passed to us.
	`trackingUrl` & `trackingId` will contain the tracking data provided by the producer.

response	`{`
	`  error: string`
	`}`
	
	If `error` is filled in, or the status code is not 200, we will assume a faulty verification & retry once.

Website Publication

When changes are made in our system, they are always contained within a sandbox environment until the “publish” button is pressed. When this happens, we will publish a new version of the JavaScript plugin. This might include code changes, as well as product changes and additions. This new version will be hosted on a new URL (to avoid caching issues). This webhook will notify you of this, and send you the new URL.

action	If no webhook is configured, we will not send you this message. You will need to manually get the URL from our website when you are ready for the new version. If the webhook is faulty (error), we will trigger an internal alarm and contact you.
endpoint	*as configured by you in our system*
method	POST

request	`{`
	`  scriptUrl: string,`
	`  products: IProduct[]`
	`}`
	
	`IProduct: {`
	`  key: string`
	`}`
	
	`scriptUrl` will contain the new, fully qualified, https URL to our JavaScript file.

response	`{`
	`  error: string`
	`}`
	
	If `error` is filled in, or the status code is not 200, we will assume a faulty verification.