Gravity REST API

This API is not public anymore. Existing endpoints are maintained, but no support will be provided for new integrations.

Quick reference

Latest version	Documentation
1.0 (alpha)	REST API endpoints

The reference of public Gravity REST API endpoints is available at the link above.

Every endpoint is described and the required parameters are presented. A graphical interface allows to test every endpoint and consult the Models used by the interface.

Usage notes

Currently, REST APIs are still coupled with internet points of sales. This means, that calls done on the REST API, must be performed starting from the internet URL, and are accessible through the following URL pattern:

https://<point of sales url>/tnwr/v1/

This will be subject to change in future versions, which will be hosted in a separated and centralized environment. The context parameter, set in the header **X-Secutix-Host** (see Context handling section below) is required but redundant right now, but will allow in the future for a seamless migration to the new API hub.

Introduction

This document presents the architectural principles of the Gravity REST API and provides information about its usage. This API is incomplete and will progressively be enhanced in parallel to the Gravity project.

This REST interface is an addition to the already existing interfaces provided by SecuTix. It works at a higher level than the interfaces provided by the back-end services. Its goal is to provide a simple set of entry points that aggregate all the information required for a front-end implementation, and that allow the user interactions involved in a ticketing purchase process. As an example, the catalog entry points return products that are pre-processed: they contain the right availabilities, propose prices that are accessible to the specific user and offer all information that is required for user interaction. User interactions, like adding a ticket to the shopping cart, are simplified and executed through calls to the interface.

This interface follows the industry standards applied to REST interfaces.

Basic concepts

The key principles of REST involve separating APIs into logical resources. These resources are manipulated using HTTP requests, where the method has specific meaning.

A resource is defined by a noun that makes sense from the perspective of the API consumer. In SecuTix, a resource can for instance be a ***Product***, a ***Ticket*** or a ***User***. Resources are not directly mapping with the underlaying models of SecuTix.

Every resource has a set of actions that applies to it. Actions allow to retrieve the information relative to resources and manipulate them. Every action is independent from the other, and is mapped to a unique endpoint and HTTP method. The HTTP method defines which kind of action is being done (based on CRUD model). For instance, GET will be used for a read-only access to retrieve information, PUT to update the information, etc.

As an example scenario, a Ticket could allow for two actions: one to retrieve its content and one to set the beneficiary. In this case, the endpoints could be defined as follows:

Endpoint	Description
GET /tickets	retrieves the information of all tickets associated to the connected user
GET /tickets/123	retrieves the information for the ticket with id 123
PATCH /tickets /123	(with additional form data {first name, last name} passed in the request body as JSON, see below for request format): sets the beneficiary of ticket 123

Notice that the same endpoint can allow for different granularities (in the example above, the complete list of tickets and a specific ticket).

REST API specifications

Disclaimer

The specifications presented in the following chapters are not completely implemented in the current version of the Gravity REST API. They are meant to be used as general guidelines and provide an overview of the future implementation of this API.

The API must be considered in an alpha stage: it is considered to be fully usable in the documented endpoints, but incomplete.

Endpoint definition: convention and supported HTTP methods

As described in the previous section, the definition of an endpoint must be linked to a resource name. As a convention, the name of a resource is defined with a plural lowercase noun. Moreover, the correct mapping method must be used to define every action that applies to the selected resource (CRUD mapping to HTTP methods).

The following table shows the supported HTTP methods and for which kind of action they are used:

Operation	HTTP method
Create	POST
Read	GET
Update	PUT (complete data) - PATCH (partial data)
Delete	DELETE

As a convention, parameters required for actions on particular resources (for instance actions applied to a specific ticket in the previous example), are defined as path variables. Query parameters are only used to refine the request (like filtering or sorting). When a frequently used action requires the setting of specific parameters, then it is aliased (i.e. a new action with abstraction of parameters).

Warning: actions are configured with the adequate security settings, based on the requested user privileges.

Format of requests

For POST/PATCH/PUT requests, the content of the modifications is sent in the request body in JSON format, with the ***Content-Type*** header set to ***application/json*** (if the header is not set a 415 Unsupported Media Type response is returned).

Format of responses

The Gravity REST API supports a single response format: JSON. All response objects are serialized in this standard format. As a guideline, objects returned by the API are as flat as possible and easy to access by a front-end. Moreover, when handling requests that potentially return many objects, a strategy based on paging is applied.

Some datatypes require specific format that must be respected in all REST actions. For instance, dates are always automatically serialized in an ISO8601 timestamp.

Every response object of the API contains a *success* field, indicating whether the request was processed successfully (see next section for details of when this is not the case).

As an example, a response for a list of products available in a specific point of sales could look like the following:

GET /products?limit=10
{success:true, data: {“123”:{code:”product1”, startDate:”2016-12-20T17:30Z”, …}}

Notice the limit parameter, which specifies a maximum number of records that should be returned (see the section on pagination below for details). Also notice the fact that every product is contained in a map structure, to ease the reading by a front-end (which can selectively retrieve data without looping on a list).

In general, a request that is modifying some resource (PUT, PATCH, POST) should return the complete updated resource in the response.

Error handling

As already mentioned, every response object of the API contains a success field. Whenever an error occurs, the success field is set to false, and an error code and message are returned (these are always set).The error message is destined for the user of the API (not the end user of the web application). The error code can be looked up in the API documentation for details of the error and can be used by the frontend implementation to determine the error message to display to the user. For these reasons, the error code is as specific as possible.

In addition, the HTTP status code of any unsuccessful call is taken from among the following:

HTTP error code	Meaning & use cases
400 – Bad request	Request parameters passed with the request are missing or incorrect
401 – Unauthorized	The action requires authentication and sufficient permission levels for the contact
403 – Forbidden	The action is not allowed for the current contact (not enough privileges)
404 – Not found	The resource targeted with the action does not exist
415 – Unsupported Media Type	The request is not in the correct format, or the Content-Type header is not set accordingly.
500 – Internal server error	An unforeseen server error occurred

In the case an internal server error occurs, the error is logged to ease the follow-up of the issues. All other errors are not logged. In addition a 500 error response will always include fields to indicate whether the user was logged out and/or if the order was cancelled, allowing the client to correctly handle the following steps.

Filtering, sorting, searching and pagination

Filtering requests can be done by adding supported fields as request parameters, for example to request all tickets in status printed:

GET /tickets?status=printed

The API documentation will specify precisely for which fields filtering is supported. Initially no more advanced search functionalities will be provided (such as “or” operators or more advanced text searches).

Similarly, a first version of the API will not provide the option to specify the fields in the response (for future reference, this should follow the pattern GET /products?fields=name,code)

Initially, no sorting functionality is to be provided (to be done by the caller).

Pagination is done following established standards for handling this case. Any endpoint providing pagination supports the offset and limit parameters, specifying at which record to start and the maximum number of records to return (if these are not provided, it defaults to 0 and 20). The total amount of records is always returned in the X-Total-Amount header. 4 links are also returned in the Link header, pointing the client to other useful subset of the collection. For example, the following request for products of type “show”:

GET /products?type=show&offset=20&limit=10

Assuming we have say 254 show products available, the response body then contains 10 products starting at position 20 (the internal list may be ordered by product name for instance):

{success:true, data: {“123”:{code:”product1”, startDate:”2016-12-20T17:30Z”, …}}

Whilst the response headers are set as follows:

Link: <https://... /products?type=show&offset=30&limit=10>; rel="next",

< https://... /products?type=show&offset=250&limit=10>; rel="last",

< https://... /products?type=show&offset=0&limit=10>; rel="first",

< https://... /products?type=show&offset=10&limit=10>; rel="previous"

X-Total-Amount: 254

Cross-origin support

For security reasons, browsers prohibit AJAX calls to resources residing outside the current origin. Cross-origin resource sharing (CORS) is a W3C specification implemented by most browsers that allows to overcome these limitations by authorizing access to different domains. CORS is automatically enabled, and allows an unrestricted access from all domains to all resources. The supported HTTP methods are the following: GET, POST, PUT, PATCH, DELETE and HEAD. A restriction can be configured in SecuTix, to authorize only specific domains.

To enable the cross-origin support, the Origin header must be set in the request, and indicate the current host that is accessing the resource. The response headers will then automatically be enhanced to contain the information required to authorize cross-origin in a browser. For more information, please refer to the following guide: https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS.

SSL encryption

Only requests made on https will be handled. Requests sent to http will not be redirected, but result in error status 404.

Session handling

Sessions continue to exist on the server side as in the current online sales back-end and are created when required. When a session is created, an X-auth-token header is returned, containing the unique session token string. To perform actions for this session, the client must in turn include this header in any request.

Context handling

The Secutix point of sale on which the purchase is made is specified by the custom header ***X-Secutix-Host***, which takes the form of a POS-specific token that is provided to the client. All requests to the API must contain a valid context header, if not a 400 Bad Request response is returned.

Secutix contains a number of contact-specific catalog data, such as advantages and prices. When a user is logged in, none of the API endpoints will make implicit use of this context, except to check the user is authorized to perform any requested actions (such as viewing an order). Endpoints returning contact-specific values will require the contact number to be explicitly passed as parameter (of course, with checks that the logged-in user has indeed the rights to access this data).

Internationalization

The RESTful interface supports internationalization. The external languages defined in the Secutix configuration screens determine which language can be used with the interface for every point of sales.

The language used to respond an API call, is either determined automatically or can be forced for every call. When a query parameter lang=xx (xx being the ISO-639 code of the language) is supplied, the response of the call will be translated in the given language code (if accepted by th e point of sales). When this parameter is not supplied, or it is not accepted, then the content of the Accept-Language header is checked against the accepted languages. If the language is not accepted by the point of sales, then the default language is used (i.e. the first language set in the configuration).

Versioning

The API is versioned, with the version appearing in the URL, such as /v1/products or /v2/products. A published version does not change, meaning the exposed DTO formats are unchanged and only backwards-compatible bug fixes are allowed (corresponding to patch versions in the terminology of the guidelines at http://semver.org/). As well as numbered versions, an “experimental” version containing the latest features is available under /vexp/products. Backwards-compatible changes of the latest versioned API will be published on the experimental version in production. The new frontend servers will use the experimental version.