LightCorn API ver. 1.1 reference

1 Introduction to LightCorn

LightCorn is a Web application, accessible via REST API only, which enables external entities to use and take advantage of TrustStamp internal biometrics and data extraction micro-services. It's designed as a relatively thin wrapper for the exposed micro-services: it fixes some idiosyncrasies in the APIs and prunes unneeded or redundant data from returned results, and also logs all the communication with services to the database. It implements a few minor features on top of that, but in essence it's just a gateway to (or proxy for) micro-services, which do the actual processing.

Currently only 5 of **Trust Stamp** services are exposed via **LightCorn**. These services implement:

- **Proof of Liveness (PoL)** - given a photo of a face it checks if it depicts a live human or not. It rejects images where the face is printed on paper or displayed on a smartphone or computer monitor.
- **document OCR** - given a photo of a front of Driver's License or utility bill, it extracts and returns personal identification information of a person.
- **barcode reading** - given a photo of an `AAMVA`-compatible barcode[1] on the back side of **USA Driver's License**, it reads and returns personal identification information (so called PII) contained in it.
- **face matching** - given two photos of faces, it checks if they belong to the same person.
- **OCRed and barcode data comparison** - checks if the personal information extracted from front and back of a Driver's License match, while allowing for small differences due to OCR inaccuracy.

The services have assigned codenames, which you need to pass as a service parameter in all the **LightCorn** API calls. The names are as follows:

function	codename	section
PoL	Sunflower	4.1
PoL	Rye	4.2
Driver's License OCR	Wheat	4.3
barcode reading	Poppy	4.4
face matching	Wheat	4.3
face matching	Graylist	4.5
utility bills OCR	Farro	4.6
front and back DL compare	Poppy	4.4

See section 4 for complete reference of each service.

2 Quick start guide

In case of any problems or additional questions, please contact on Slack or via email one of the following people:

Jakub Zalewski	jzalewski@truststamp.net	developer (active)
Piotr Klibert	pklibert@truststamp.net	developer (active)
Alexander Kononenko	akononenko@truststamp.net	developer (support)
Marcin Stępniak	marcin.stepniak@10clouds.com	manager (active)
Jacek Suwalski	jacek@truststamp.net	manager (support)

2.1 Available LightCorn instances

This is the state of instances for **2018-10-08**.

type	instance url
production	N/A
staging	N/A
devel	https://lightcorn-dev.truststamp.net/

The staging instance should be used for development. We will update the list with a production instance once it's set up, which should then be used for integration with customer-facing systems.

2.2 Account credentials

To use **LightCorn**, you need to know a valid `username` and `password`. To make sure these credentials are not disclosed to unauthorised entities, they're being shared via a LastPass[2] application.

People responsible for providing the credentials are:

- **Marcin Stępniak**
- **Jacek Suwalski**

2.3 JWT-based Authentication

LightCorn uses JSON Web Token[3] for authentication. Each request to the API has to include the token in the `Authorization` HTTP header. See section 3.2

To obtain a token, issue a POST request to `$INSTANCE/api-token-auth/` (eg. https://lightcorn-dev.truststamp.net//api-token-auth/) endpoint, providing JSON-encoded credentials in the body of the request.

In response you will get token string, which will be valid for 24 hours.

2.3.1 Example request

```
curl -X POST 'https://<instance_url>/api-token-auth/' \
     -H 'Content-Type: application/json' \
     -d '{
          "username": "<username>",
          "password": "<password>"
     }'
```

2.3.2 Example response body

```
{
    "token": "<token>"
}
```

2.3.3 JWT refresh

It is possible to refresh the token - invalidate current and issue new one - by sending it to `$INSTANCE/api-token-refresh/` (eg. https://lightcorn-dev.truststamp.net//api-token-refresh/) endpoint.

2.4 Synchronous requests and receiving results

There are three different ways getting the results from **LightCorn**, see section 3.5 for details of the other options, but the most straightforward way to do this is by using the immediate (synchronous) mode, which means that the results of processing are returned as a response for the API request.

Currently this mode is the default one, but this may change in the future, so it's recommended to request it explicitly, by including the `return_now: true` attribute in the JSON sent in the request.

2.5 Simple request-response session - full example

Make sure you have `cURL`[4] and `jq`[5] installed locally, then issue the following commands on the command line (assuming `sh` compatible shell):

1. Set some handy variables. Remember to replace `USER` and `PASS` values with valid ones.

```
export INSTANCE="https://lightcorn-dev.truststamp.net/"
export USER="******"
export PASS="******"
```

2. Obtain the JWT token

```
export TOKEN=$(
  curl -s -X POST "$INSTANCE/api-token-auth/" \
       -H "Content-Type: application/json" \
       -d "{
           \"username\": \"$USER\",
           \"password\": \"$PASS\"
       }" | jq -r .token
  )
```

3. Prepare the image to be analyzed and the request body. Save the body in an environment variable called `$BODY`, like this:

```
export BODY='{
  "service": "Sunflower",
  "media_url": "https://s3-us-west-2.amazonaws.com/truststamp-serv-e2e-tests/sunflow
  "UUID": "3c171d18-bb2c-11e8-ab76-9cb6d0fc2569",
  "return_now": true
}'
```

   For this example we use one of the images we use for automated testing of our services, available here. As you can see, we select **Sunflower** as a service we want to use.

4. Send the request and receive the results:

```
curl -s -X POST "$INSTANCE/api/v1/single/" \
     -H "Content-Type: application/json" \
     -H "Authorization: JWT $TOKEN" \
     -d "$BODY" | jq .
```

5. This should print the response looking like this:

```
{
  "request_id": "fcab3d5a-4e9b-48bc-b1e4-cd44cced0fdf",
  "status_message": "success",
  "error_code": null,
  "data": {
    "fake_detected": false
  }
}
```

6. You can use the same command to call other services; for example, to call **Farro**, which is a service for parsing address data in images, you'd only need to redefine the `$BODY` variable, like this:

```
export BODY='{
  "service": "Farro",
  "media_url": "http://alexanderkononenko.com/farro/addr5.png",
  "UUID": "3c171d18-bb2c-11e8-ab76-9cb6d0fc2569",
  "return_now": true
}'
```

Then, issuing the same `curl` command as in step 4. would give you:


```
{
  "request_id": "c48d105c-bdee-4d4d-986b-c919b1f1388a",
  "status_message": "success",
  "error_code": null,
  "data": {
    "city": "bedford",
    "house_number": "69",
    "postcode": "mk42 9gh",
    "road": "croyland drive"
  }
}
```

3 LightCorn API details

Except for authentication and JWT-related requests, **all** API calls have to be sent to a single endpoint, located at:

`/api/v1/single/`

This endpoint path needs to be combined with host instance URL, to form a full URL. For example, API calls to devel instance would have the following URL:

https://lightcorn-dev.truststamp.net//api/v1/single/

3.1 Forward and backward API compatibility

The API described in this document is stable, which means that if a particular set of parameters sent to the endpoint is valid, it will stay that way forever. In other words, there won't be any backward incompatible changes to the API. In case a breaking change is required, it will be exposed under a new version number, leaving the current API unchanged.

The API may still get support for new parameters and features, as long as they are strictly backward compatible. In that case, this document will be revised.

In addition, if a parameter is marked as obsolete, it just means that it won't be included in the next version of the API. It will continue working with the current version.

3.2 Common API request structure

All requests must have `Authorization` and `Content-Type` headers set:

```
Authorization: JWT <token string>
Content-Type: application/json
```

JSON sent in the body of the request must have at least these fields:

- `UUID` – unique identifier, used for grouping requests to different services and identifying them as belonging to a single user (or single flow).
- `return_now` – see sections 2.4 and 3.5
- `media_url` – an URL to the image or video sent for processing. It has to be accessible via HTTP or HTTPS from the **LightCorn** instance. NOTE: `image_url` is a deprecated alias for `media_url`.
- `service` – name of the service to call. Can be one of:
  - `"Sunflower"`
  - `"Rye"`
  - `"Wheat"`
  - `"Poppy"`
  - `"Farro"`
  - `"Graylist"`

  See section 4 for details.

Some services may require more parameters, see for example Wheat in section 4.3

3.3 Common API result structure

Depending on the value of `return_now` parameter the result may be returned in the immediate response or sent to a callback URL, but it's structure is identical in both cases. The results are always JSON-encoded.

The exact fields in the returned JSON differ between the services, but they all include at least the following keys:

- `request_id` - a unique identifier (as a string) of the request. In asynchronous mode it is used for linking the results to the request. May be ignored in sync mode.
- `status_message` - either `"success"` or `"fail"`. See section 3.6
- `error_code` - either null or a number. See section 3.6.
- `data` - results of processing, as an object. The keys vary between services.

Depending on the service, the response may have one or two additional fields.

3.4 Calling services in the right order

It is important to call the services in the correct order, because some services need information from the previous API calls to function. The `UUID` parameter is used for finding that data, which is why it is important for clients to issue a new `UUID` for every customer and using the same `UUID` in all requests related to this customer.

For example, you can't call **Wheat** without first calling **Sunflower**, because **Wheat** service performs face matching between the selfie and Driver's License front. Without first supplying the selfie image to **Sunflower** service, **Wheat** call will fail.

Currently, the required order is as follows:

1. **Sunflower** or **Rye**
2. **Wheat**
3. **Poppy**

Farro and **Graylist** are services which don't depend on, and are not depended on by, any other service call, and so you can call them whenever you want. Technically, there is no need for the `UUID` in Farro request to be any specific value, but it's recommended to use the same `UUID` that was used in other service calls for the given user session.

3.5 Synchronous and asynchronous processing

3.5.1 Receiving results as HTTP response

This mode is selected by passing `return_now: true` parameter with the request. The HTTP connection is kept open in this case until the service processes the image and we want to wait for full response, if `false`, we want to get request ID immediatelly and use it later for retrieving full service response.

3.5.2 Receiving results via a callback URL

This mode is used when `return_now` is set to `false`, and there is a callback URL provided for the user that the request authenticates as. You can set the callback URL using Admin panel, accessible at `$INSTANCE/admin/`, by navigating to **API** → **Clients** → **username** → **Callback URL**, putting the URL there and clicking **Save**. In case of multi-tenant instances, please ask your Administrator to do this.

In this mode, the response to the POST request contains a `request_id`, which is then also included in the data sent to the given URL.

3.5.3 Polling an endpoint for the results

If neither of the above is true, the only way of getting the results is to send GET requests to the endpoint, passing it the `request_id`. In response you will get either information that the request is still processing (status code 202), or the result if it already finished.

3.6 Error handling

There are two fields in the returned results which say if the request was successful or not. These are:

- `status_message` - a string, one of two possible values: `"success"` or `"fail"`
- `error_code` - either `null` or an integer, which exact meaning is dependent on which service was called. When used with `return_now: true`, this error code is also returned as a HTTP response status code.

In general, if `status_message` is `"fail"`, it means that the processing of the given image failed - this makes the services following the current one inaccessible (see section 3.4). You can safely re-send the request to the service which failed with another image - if it succeeds, it will simply overwrite the failed request and will allow you to send requests to the following services normally, without changing the `UUID`.

If you want to reset the whole process, starting again from **Sunflower**, you can do this by generating a fresh `UUID` and issuing requests normally.

Other than errors specific for a given service, there are some general errors, which are used if the request cannot be processed at all, for example:

- **400 BAD REQUEST** - invalid or missing parameters in the request
- **403 FORBIDDEN** - no `Authorization` header provided or bad token provided
- **404 NOT FOUND** - invalid endpoint URL used or invalid `image_url` passed
- **500 INTERNAL SERVER ERROR** - the **LightCorn** itself encountered a problem and couldn't deal with it (shouldn't happen - please report as a bug if you see it)

These are returned as an immediate response HTTP status code, no matter if the `return_now` is set or not.

3.7 Account Management and Admin Panel

As described in section 2.3, LightCorn uses JSON Web Token[3] for authenticating requests to its APIs. To obtain a token, you have to pass a valid set of credentials to the `$INSTANCE/api-token-auth/` (eg. https://lightcorn-dev.truststamp.net//api-token-auth/) endpoint.

These credentials are also valid for logging into an Admin Panel in single-tenant deployments, which is accessible at `$INSTANCE/admin/` path (eg. https://lightcorn-dev.truststamp.net//admin/)

Once logged in, you can create additional accounts with their own credentials and permissions. To do this, you need to create a new User object, where you set username and password, and a related `Client` object, where you can set the `Callback URL` (as explained in 2.4 and 3.5). The `is_staff` attribute of the `User` objects controls whether that user is able to log into the Admin Panel.

Other than account management, the Admin Panel allows viewing the details of all requests to the API, responses from the micro-services, and **LightCorn** own responses.

LightCorn requests and responses are stored in the `CustomerResponse` objects. Navigate to **API** → **Customer** responses to see the list, click on any row to see the details.

Communication between **LightCorn** and micro-services is logged in `ServiceResponse` objects. As with `CustomerResponse` objects, you can view a list of **LightCorn** requests to, and responses it got from, micro-services by navigating to **API** → **Service** responses. Clicking on any row will display the details.

Don't rely on the shape of service responses!

Service responses are considered an **internal implementation detail** and as such it is not covered by compatibility guarantees. The format of the requests and responses between LightCorn and its backing services may change at any time without warning.

4 Services reference

4.1 Sunflower service

Sunflower is a service which implements Proof of Liveness check. Given a photo of a face (selfie) it checks if it depicts a real human. It rejects photos of other photos (whether printed or displayed on a smarphone or laptop), photos of masks and sculptures, and so on.

Getting `fake_detected: true` from Sunflower means that there's something suspicious in the given photo, and that it shouldn't be used to verify the user's identity. Getting `false` means that most likely it's a photo of a real person, and that it can be used for comparison with other photos to verify or find out the user's identity.

4.1.1 Request

```
{
  "service": "Sunflower",
  "media_url": string,          // url of a selfie image
  "UUID": string,
  "return_now": boolean,
}
```

4.1.2 Result

```
{
  "request_id": string,
  "status_message": "fail" | "success",
  "error_code": null | int,
  "data": {
    "fake_detected": boolean
  }
}
```

4.1.3 Error Codes

- **406 NOT ACCEPTABLE** - we were not able to detect a face in the picture

4.2 Rye service

Rye is a service which implements Proof of Liveness check. It differs from Sunflower in that it works on videos and not photos; the video should be about 30 seconds long and depict a user turning their head left and then right.

As with sunflower, `fake_detected: true` means that the face on the video is likely recaptured from a flat display (ie. printed or shown on a smartphone screen). `fake_detected: false` means that the video most likely depicts a real person.

4.2.1 Request

```
{
  "service": "Rye",
  "media_url": string,          // url of a selfie image
  "UUID": string,
  "return_now": boolean,
}
```

4.2.2 Result

```
{
  "request_id": string,
  "status_message": "fail" | "success",
  "error_code": null | int,
  "data": {
    "fake_detected": boolean
  }
}
```

4.2.3 Error Codes

- **406 NOT ACCEPTABLE** - we were not able to detect a face in the picture

4.3 Wheat service

Wheat is a service which primary function is to scan and extract personal identification data from photos of documents (mainly Driver's Licenses, but also passports and others). If successful, it returns a JSON object with fields it managed to read.

Wheat has a secondary function, which is to compare the face visible in the Driver's License with the one sent to Sunflower earlier. The result of the comparison is returned as a `face_match` field in the response.

Wheat returns a `twosided` attribute along with whatever it managed to read from the document. If its value is true, it means that the document is a Driver's License, and it's possible to run Poppy on the back side of it. If it's `false`, it means that the document is not a DL, and it cannot be used with Poppy.

4.3.1 Request

```
{
  "service": "Wheat",
  "media_url": string,              // url of a Driver's License front
  "country_code": string,           // eg. "US", "GB", "EU"
  "state_code": string,             // eg. "TN", "GA", "NL"
  "UUID": string,
  "return_now": boolean,
}
```

The `country_code` and `state_code` are both optional, but (if correct) they speed up the processing time greatly.

`country_code` is a 2 letter code, for example: `"US"`, `"GB"`. If not provided, Wheat assumes `"US"`.

`state_code` is a 2 letter code, for example: `"GA"`, `"TN"`. If not provided, Wheat tries to match all the kinds of Driver's Licenses it knows about until it finds a match. This makes the processing time much longer than it would be with the correct code provided.

European Union countries

EU countries are a special case: you need to pass `"EU"` as the `country_code` and a specific country code (eg. `"DE"`, `"NL"`, `"FR"`) in the `state_code`.

4.3.2 Result

```
{
  "request_id": string,
  "status_message": "fail" | "success",
  "error_code": null | int,
  "data": {
    "address": string,            // eg. "50 EXAMPLE RD"
    "apartment_nr": string,       // eg. "13"
    "city": string,               // eg. "Some City"
    "dob": string,                // eg. "1990-03-02"
    "exp": string,                // eg. "2019-03-02"
    "first_name": string,         // eg. "JANE"
    "iss": string,                // eg. "1999-04-02"
    "last_name": string,          // eg. "SMITH"
    "middle_initial": string,     // eg. "J"
    "state": string,              // eg. "TN"
    "zip_code": string            // eg. "30016-5217"
    "twosided": boolean           // see description above
  },
  "face_match": boolean
}
```

4.3.3 Error Codes

- **406 NOT ACCEPTABLE** - none of the Driver's Licenses templates matched
- **422 UNPROCESSABLE ENTITY** - no face found in the image

4.4 Poppy service

Poppy is a service for reading the barcodes[1] found on the back sides of USA Driver's Licenses. In many jurisdictions, the barcode encodes personal identification information of a DL holder. Poppy reads such barcodes and returns the results in a format similar to 4.3.

Poppy has a secondary function, which is to compare the data it read with the data OCRed in previous call. The result of the comparison, which is an int indicating how many fields are different, is returned in the `side_comparison` field in the response. The value of `0` means that there is a perfect match between data extracted from the front and read from the back of a Driver's License.

4.4.1 Request

```
{
  "service": "Poppy",
  "media_url": string,          // url of an image of Driver's License back
  "UUID": string,
  "return_now": boolean,
}
```

4.4.2 Result

```
{
  "request_id": string,
  "status_message": "fail" | "success",
  "error_code": null | int,
  "data": {                     // eg. "710 EXAMPLE WIND CT"
    "address": string,          // eg. "WOODSTOCK"
    "city": string,             // eg. "2022-09-03"
    "date_exp": string,         // eg. "1960-09-03"
    "dob": string,              // eg. "JOHN"
    "last_name": string,        // eg. "SMITH"
    "middle_initial": string,   // eg. "P"
    "state": string,            // eg. "GA"
    "zip_code": string          // eg. "401698757"
  },
  "side_comparison": int
}
```

4.4.3 Error Codes

- **406 NOT ACCEPTABLE** - barcode not found in the image

4.5 Graylist service

Graylist is a biometric hash (compact representation of face features) storage and search service. It accepts a photo of a face and answers the question if it's the first time the system seen this face or not. It automatically saves every encountered biometric hash into internal database.

The results are returned as a list of biohash identifiers along with the `distance` (how different is one biohash from another). An empty list means that there was no match and the given person is encountered for the first time.

4.5.1 Request

```
{
  "service": "Graylist",
  "media_url": string,          // url of a selfie image
  "UUID": string,
  "return_now": boolean,
}
```

4.5.2 Result

```
{
  "request_id": string,
  "status_message": "fail" | "success",
  "error_code": null | int,
  "data": [
    {"person_number": int, "distance": int},
    ...
  ]
}
```

4.5.3 Error Codes

- **406 NOT ACCEPTABLE** - we were not able to detect a face in the picture

4.6 Farro service

Farro is a service for extracting address information from various documents, mainly utility bills. It returns its results in a different format than Wheat and Poppy, because it uses completely different technology under the hood. This means that the results are not directly comparable with the data returned from other endpoints, except in the most general fields, like `country`, `state` or `city`.

Note

The input image needs to be cropped so that only the address is visible in it.

4.6.1 Request

```
{
  "service": "Farro",
  "media_url": string,          // url of an image of a utility bill
  "UUID": string,
  "return_now": boolean,
}
```

4.6.2 Result

```
{
  "request_id": string,
  "status_message": "fail" | "success",
  "error_code": null | int,
  "data": {
    "house_number": string,     // eg. "39"
    "road": string,             // eg. "Example Strasse"
    "postcode": string,         // eg. "30016-5217"
    "city": string,             // eg. "Berlin"
    "state": string,            // eg. "Berlin"
    "country": string,          // eg. "Germany"
  }
}
```

4.6.3 Error Codes

- **406 NOT ACCEPTABLE** - no adress data found in the image

Footnotes:

https://www.aamva.org/uploadedFiles/MainSite/Content/SolutionsBestPractices/BestPracticesModelLegislatic

1

2 https://www.lastpass.com/

3 https://jwt.io/

4 https://curl.haxx.se/

5 https://stedolan.github.io/jq/

Author: Piotr Klibert (pklibert@truststamp.net)
Last updated: 2019-02-04 pon 18:32