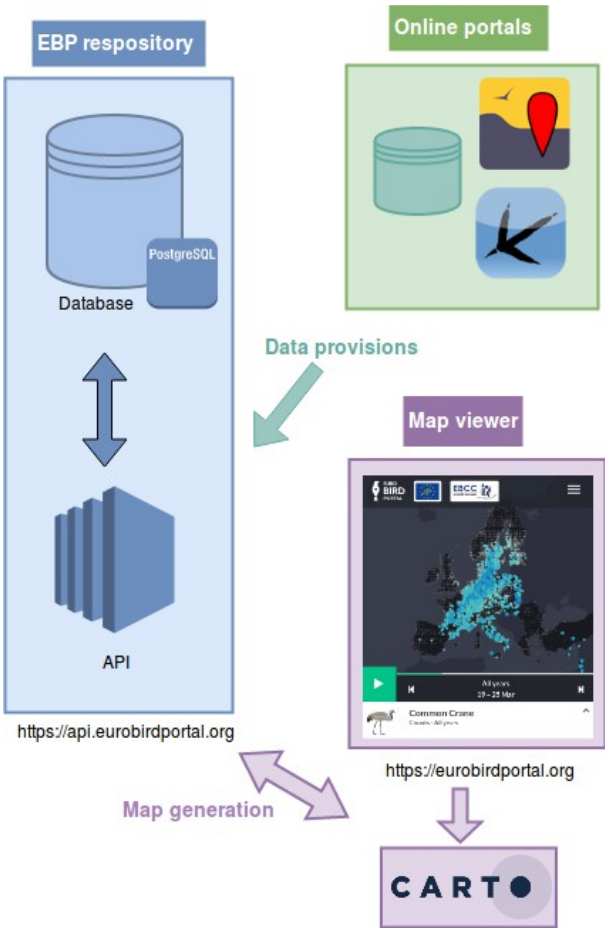
Euro Bird Portal (LIFE15 PRE/ES/000002)

Design of the new database repository and data-flow

Overview

This document describes the design of the new EBP database repository and associate data-flow system. The system is divided in two different parts: the database repository and the API/webservices.

Architecture overview



Database

The database has been developed using PostgreSQL on an Amazon instance. We have chosen PostgreSQL for several reasons:

- It is the most powerful openSource relational database.
- Can handle huge amounts data.
- Geographical objects and methods capabilities.
- Expertise among several online portals and main developers.

The Amazon services gives us the following advantages:

- Scalability. We can adapt the server capabilities to the requirements.
- Costs are adapted to the CPU, memory, bandwidth and storage requirements.

The PostgreSQL database structure with the final design of the main database tables and relations is shown in Annex 1.

Database optimizations

The amount of data in the database will be huge in the near future. Therefore, we had to analyse bottlenecks and try to find extra optimizations. We've created indices and primary keys/foreign keys in several database tables and optimized each field.

From the beginning, it was rather clear that it could be desirable to use partitioning on tables. Partitioning improves query performance but increases insert and update times. In our case, since most maps are done year by year, partitioning by year gave us the best results.

API/web services

Technology

All selected technology for the EBP repository is licensed as free software. Using free software solutions we reduce costs and we are not hardly linked to a privative solution. Moreover, all chosen technologies are competitive, actively maintained and powerful enough to fulfil our requirements.

The EBP repository API is deployed on a GNU/Linux server running on Apache webserver using WSGI libraries to run Python webapps.

The API has been developed using the Flask micro web framework, written mainly in Python programming language following the REST architectural style. Some other libraries have been used to facilitate main tasks: RESTPlus (REST APIs creation), SQLAlchemy (Database access as a SQL toolkit and Object Relational Mapper) and Authlib (Outh2 authentication).

As we identified some asynchronous tasks in the project, we had to create a queue and messaging system to handle these jobs. As a message broker we use a RabbitMQ software and for asynchronous task creation and scheduling we use the Celery library.

Security

We had to assure that only registered online portals can send data to the repository.

We have decided to use the OAuth 2.0 protocol for authentication and authorization with PasswordGrant credentials for the EBP repository uses. OAuth2 is an authorization framework that enables applications to obtain limited access to user accounts on a service. It provides authorization flows for web and desktop applications, and mobile devices.

Services

The API should offer several operations related to main data structures:

- Species lists.
- Protocols: creation, removal and updates.
- Breeding codes list.
- Data provision: online portals data provision handling.
- Oauth2 : operations related to authorization.

Modules

Four main modules have been designed and implemented to fulfil the functional requirements:

a) **Data provisions** from online portals using the new EBP standard.

b) **Maps creation** for the demo viewer.

c) **Repository management** such as the administration zone, user creation and internal visualizations (maps and graphs).

d) **Metadata handling** for species lists, breeding codes, audits.

a) Data provisions

We decided that the responsible of sending data will be the online portal and the exchange format will be the JSON file. Online portals will do the data aggregation, standard creation and updates/removals handling in their side. After that, they will automate the data-flow to send to an **API/web service** the list of events and it's composite records.

We had to support two kind of data provisions:

1. **Standard data provision:** consists on sending regularly (daily, weekly or monthly) data updates to be shown in the EBP viewer in a near real-time. Every data provision will send new data from a concrete period with old inserts, updates or deletes,
2. **Bulk data provision:** sending old data using data standard from 2010 until standard connection is established.

The online portal has to convert its own data to a JSON data provision following the standard and send it to the API. It gets authenticated by the API and data goes through three validations processes. The system creates an audit log where it's possible to access to errors to be fixed later. The first and second processes validate the format consistency, ids uniqueness, etc...

The third validation is done at database level as an asynchronous task. We send this tasks to the processing queues because those validations require time. It checks that data is inside the portal's country, species code existence, etc. When the validation process is finished the online portal can access to the audit with the validation errors.

Events and records without errors are inserted to the database linked to the partner source id and upload id. When providing removals or modifications, previous data in the database is modified or deleted depending on the state field added to the data provision event or record.

For the standard data provisions, the online portals have to create scheduled tasks to create the data aggregations and send the data provisions. Depending on technological capabilities it will be recommended to send data daily, weekly or monthly.

b) Maps creation

We've created a new module to aggregate data for the map viewer, adapting the previous code. The algorithm aggregates EBP repository data from different partners at each 30x30 km square and week and creates the different map types (occurrence, traces, counts and phenological maps). It also creates the inter-annual cycles maps (for example "2015/16"). The map generation process benefits from the GIS capabilities of the PostgreSQL database.

With the new near-real-time scenario, we also schedule, every week, the creation of the last 52 weeks maps. Those maps show data from the last week up to 51 weeks before.

Once data is aggregated, we upload and update the maps at the data visualization platform CARTO. The map viewer shows the maps in the browser using the CARTO technology.

c) Repository management

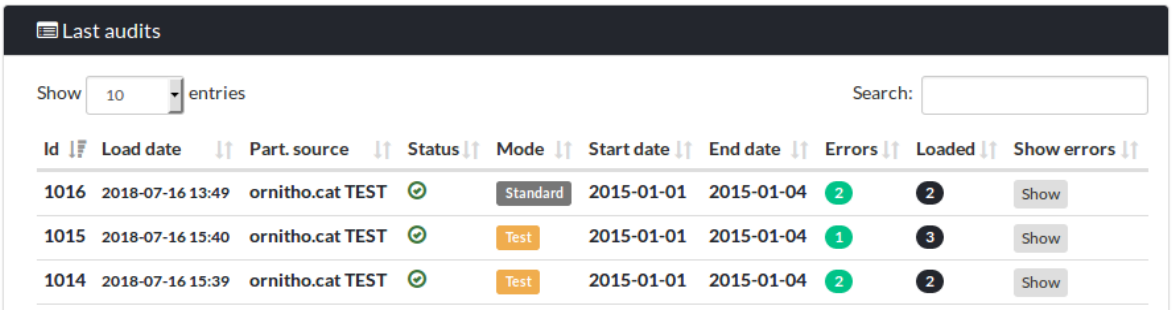
The EBP repository also contains an administration zone. There are two different access roles: online portal user and EBP repository admin.

The online portal user can access to the administration zone to:

- Get Oauth2 credentials.
- Access to its own partner sources and protocols.
- Get a the list of last audits from its own data provisions with information about: provision dates, events loaded, provision mode and validation errors.
- Maps showing provided data from portal partners and some graphs with summarised information.

The EBP repository admin can access to the administration zone to:

- Same operations as the online user but for all users and portals.
- Dashboard with information about the database state, the scheduled and asynchronous tasks.
- User management (create, delete, role assignment).
- Overall statistics of online portals data provisions.



Example of a list of data provision audits as shown in the administration zone

Map visualization of the submitted Ornitho data as shown in the administration zone

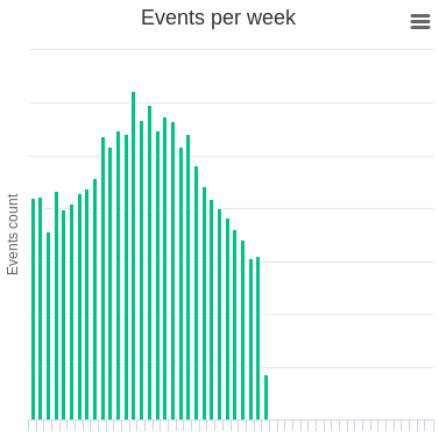


Chart with the number submitted events per week as shown in the administration zone

d) Metadata handling

The API offers several operations for the online portals to help them with their data-flow implementation.

In agreement with the the European Bird Census Council (EBCC), the EBP partnership agreed on using the Handbook of the Birds of the World (HBW) as standard taxonomy checklists.

Each online portal has to map it's own taxonomy to the HBW species codes. We added some queries to the API to access to the **species lists** to simplify this mapping process.

Online portals can also access to **breeding codes lists** and the **project type lists** for the protocol creation.

There are also some API methods to define, create and modify **protocols** into the EBP repository database. Extended information can be found in the protocols section of the documentation.

Scheduled and asynchronous task

As we have explained in the technology section, it was required to create a messaging system to handle scheduled and asynchronous tasks.

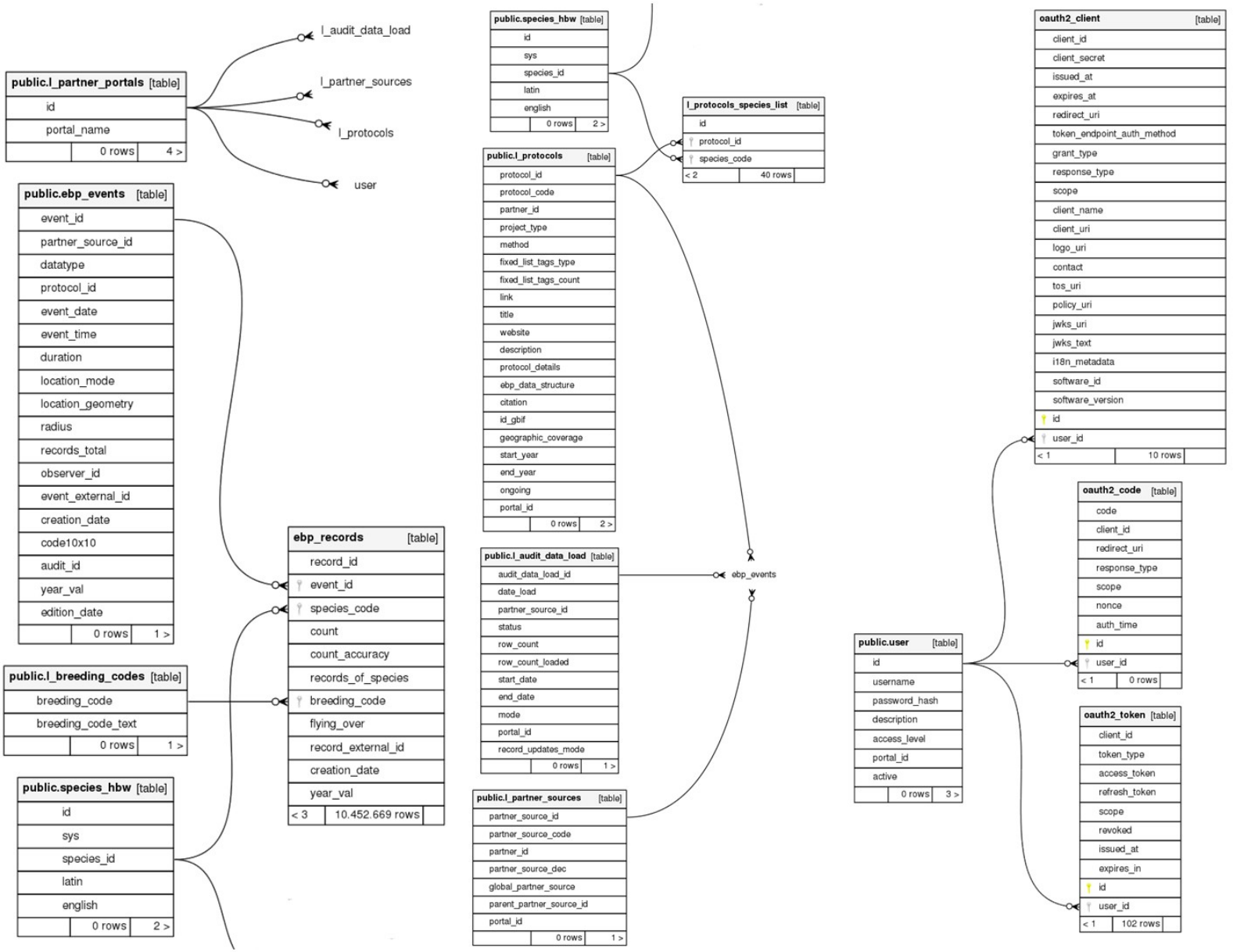
Asynchronous tasks: time consuming tasks are sent to a processing queue to be executed when the API is not busy. Complex data validations and bulk data processing are processed asynchronously.

Scheduled tasks: other tasks, like map generation, are also time consuming and they should be executed regularly. Last year and last 52 weeks maps are generated every week when weekly data provisions are finished.

Documentation

An extended documentation wiki has been created to help online portal developers to connect with the EBP repository API (Annex 2). We've also documented the API methods using the Swagger technology. Finally, developers can also access to a set of Postman examples.

Annex 1. Final database structure (main tables and relations)



Annex 2. API Documentation

Latest update: 30 May 2018

Table of Contents

Overview

The API is already on-line EBP API. ICO team has developed it in Python using a micro framework called Flask. You can start testing with your own data.

We have decided the JSON structure for the data provision EBP API, as you can see in the json schema. Data provisions should inclide always to a **particular date range** (i.e. 2016-01-04 / 2016-01-10) and **partner_data_source**. For example, if ICO wants to submit Bird Garden Survey and Ornitho.cat data it should sent data separately (since they originate from two different sources).

We have also implemented three provision modes: **Standard** (the one to be used for "standard" weekly updates), **Bulk** (all data submited is handled as "new" data that fully substitutes any data submitted previously; particularly useful for sending big chunks of data —e.g. data older than the one submitted once the weekly updates start—) and **Test** (used only to validate data provisions; no data is incorporated into the database). You will find more information in this document.

Events and records list follow the **New EBP data standard format** and the database structure proposal by BTO. JSON schema is also described in the document.

EBP workflow

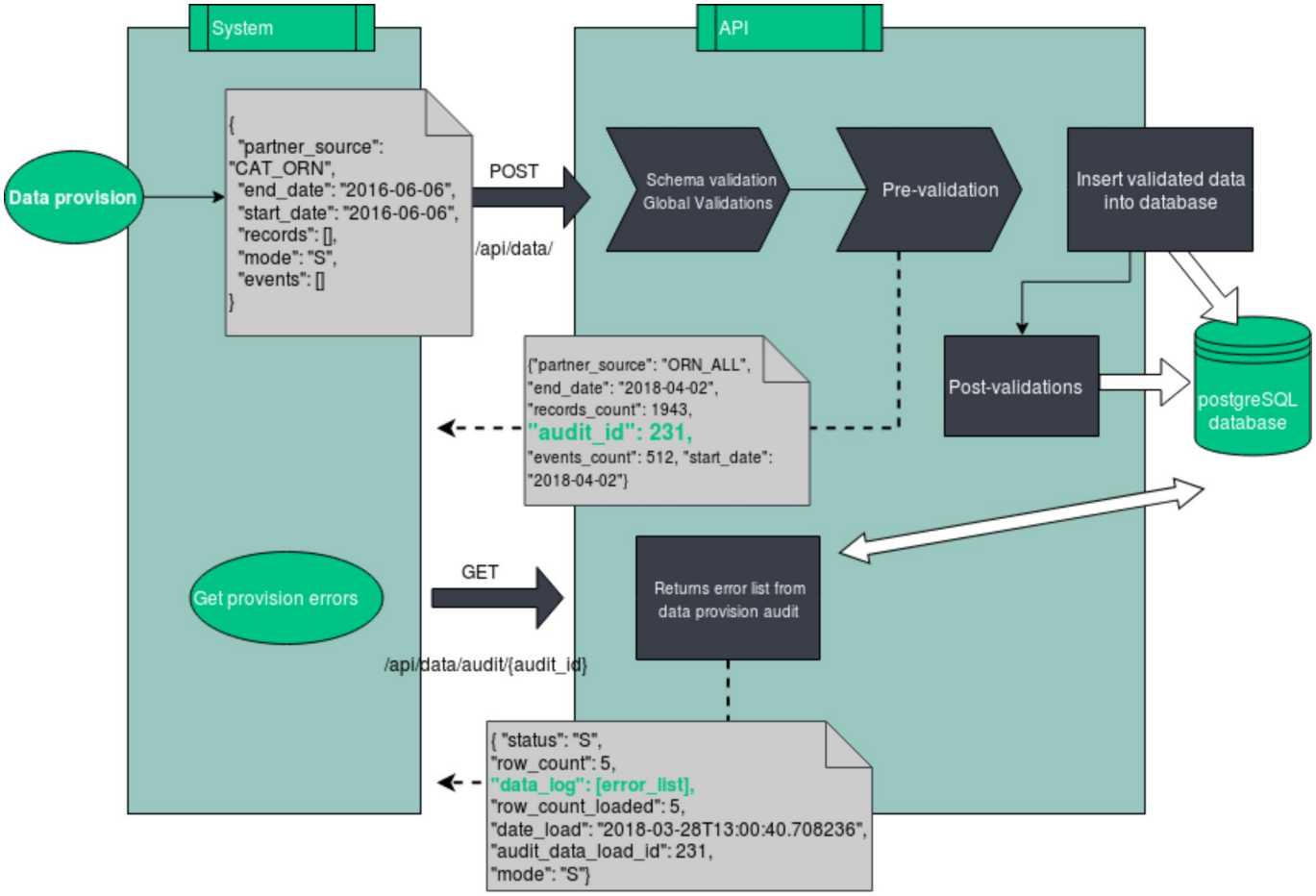
Introduction

Data provision will be done at partner_source level. All events and records must be from the same partner_source. We have defined a l_partners table and l_partner_sources where you will find the partner_source_code required for the data provison.

In agreement with the EBCC decision regarding species standard taxonomy checklists, we have used the HBW species_list and codes. Through the API you can have a list of these species and also to protocols, project_types for protocols and breeding_codes.

Data flow schema

1. System/partner sends periodically a data provision to the API
2. The API validates the format and checks consistency inside de provision (unique event ids, dates outside range, location format, etc)
3. The API gives a result to the system/partner with an audit id where the system can check validation errors
4. The API continues doing some background checkings to the data (location inside partner's area, unique event_id inside partner's data, etc)
5. System/partner can access to the provision errors through the API.



Authentication

The EBP repository uses the OAuth 2.0 protocol for authentication and authorization with PasswordGrant credentials.

a) Request a user. A user will be created for your provided username (user mail is desirable).

b) Once the user is created, you can obtain the OAuth 2.0 credentials (client_id and client_secret) from the EBP repository Console.

c) Your portal will get the token accessing via POST /oauth/token:

- **Request headers**
  - **Basic Auth header:** with client_id:client_secret (base64-encoded)
- **Request body**:
  - **grant_type:** password
  - **username:** YOUR_USERNAME
  - **password:** YOUR_PASSWORD
  - **scope:** api

```
curl -u client_id:client_secret -XPOST https://api.eurobirdportal.org/oauth/token -F grant_type=password -F username=YOUR_USERNAME -F password=YOUR_PASSWORD -F scope=api
```

d) Use the access token to access the secured routes from the API.

```
curl -X GET https://api.eurobirdportal.org/data/audit/last -H 'Authorization: Bearer ACCESS_TOKEN'
```

Connection scenarios

We will have three diferent connection scenarios:

Mode	All data replaced	Delete events	Old events (outside provided date range)
Bulk mode	✔️	❌	❌
Standard mode	❌	✔️	✔️ (updates, removals)
Test mode	❌	❌	✔️

Bulk mode (mode==B)

A bulk provision will be used when providing old data.

- All previous data inside the provided date interval will be removed
- No need to provide inserts or removals, all data will be overwritten.
- All provided events should be inside provided range.

Standard mode (mode==S)

An standard provision will be used during weekly or monthly updates.

- **New data**: All events inside the date range will be inserts
- **Previous updated data** are the events outside the provided date range (removed, modified or new past inserts). Past data will be overwritten:
- Insert/modification when event records > 0
- Removed when event records=0

Test mode (mode==T)

The data provision will not be uploaded to the database. It will be use for testing porpouses.

Modification scenarios

During standard updates, portals should send old modified and removed data. Events and records will have a state field to differentiate updates/new from deletes.

Event modifications

When an event is modified, it should be sent again with new values and state=1. All event values will be replaced.

Type	Changes	Action
Event	Update/insert	Send the complete event again (and modifications in case of event updates) with field state = 1
Event	Delete	Send the event again with field state = 0

Record modifications

In case of records modifications, the event should be sent anyway. In most cases event should be also modified. We will have two different modes to provide the records.

a) **Only modified records** (default update mode)

Send the complete event with the list modified records only.

- Advantages
  - Data provision size will be smaller.
- Disadvantages
  - You will need to track record removals or modifications.

Type	Changes	Action
Record	Insert	Send new records with state=1
Record	Update	Send the updated records with state=1 again. They will be overwritten
Record	Delete	Send the removed records with field state=0

b) **All records**

Send the complete event with the full list of records.

> To allow this mode **record_updates_mode = A** field in the data provision should be added.

- Advantages
  - You will always send the last state of the records. You don't need to track record removals or modifications.
- Disadvantages
  - Data provision size will be bigger.

Type	Changes	Action
Record	Update/insert	Send the complete list of records it doesn't matter if they are new or modified. All records will be overwritten with the new list of records
Record	Delete	No need to send the removed records

Data provision structure

To send data to the EBP repository you should send a JSON structure with the described mandatory fields. This is the API method to send the data

Property	Type	Description
partner_source	string	The unique of the EBP partner source

Property	Type	Description
start_date	date	Start date provided
end_date	date	Last date provided
events	array[Event]	List of events
records	array[Record]	List of records
mode	string ennun[B,S,T]	Provision mode B (Bulk: all data is replaced) / S (Standard: new data is provided/ T (Test: data validation porpouses)
record_updates_mode (optional)	string ennun[M,A]	Update mode M (Only updated records will be provided) / Update mode A (All records are provided). **Default mode is M when record_updates_mode is not provided**

Data provision sample

The data provision structure


```
{
    "mode": "S",
    "partner_source": "CAT_ORN",
    "start_date": "2016-01-04",
    "end_date": "2016-01-10",
    "events": [
        {
            "data_type": "L",
            "date": "2016-01-04",
            "event_id": "71456",
            "location": "POINT(3.056 41.813)",
            "location_mode": "E",
            "observer": "7840",
            "protocol_id": "",
            "records": 27,
            "time": "17:14:00",
            "state": 1

        }
    ],
    "records": [
        {
            "count": 2,
            "event_id": "71456",
            "flying_over": "N",
            "record_id": "3170459",
            "records_of_species": 1,
            "species_code": 52834,
            "state": 1
        },
        {
            "count": 2,
            "event_id": "71456",
            "flying_over": "N",
            "record_id": "3170448",
            "records_of_species": 1,
            "species_code": 58515,
            "state": 1
        }
    ]

}
```

Events and records

> Within the JSON definition for records and events we won't use nulls as the provided standard. When a field is null we can provided and empty string or remove the field from the json.

Events

```
{
    'properties': {
        'data_type': {'enum': [ 'C', 'L', 'F' ],'description': 'C (casual record) / L (complete list) / F (fixed l
        'date':  { 'type': 'string' },
        'duration': { 'type': 'number' ,'description':'Duration (in hours). Null if unknown or location_mode=A','r
        'event_id':  { 'type': 'string' },
        'location_mode': {'enum': [ 'E', 'D', 'A' ],'description': 'E (original exact location provided) / D (loca
        'location': { 'type': 'string', 'description':'Centroid of the location in Well Known Text (WKT) in WGS84'
        'observer': { 'type': 'string' },
        'protocol_id': { 'type': 'string' },
        'radius': { 'type': 'number','description':'Maximum distance (in m) to the location centroid travelled/cov
        'records': { 'type': 'integer', 'minimum': 1, 'description':'Total number of records.'},
        'time':  { 'type': 'string','required':False},
        'state':  { 'type': 'number'}
    },
    'type': 'object'
}
```

- **event_id** Identifier of the observational event (e.g. a given complete list).
  - For example: you can use julian date and 10km ETRS89-LAEA grid code as event_id for aggregated casual records (see locationMode = A below).
- **data_type** C (casual record) / L (complete list) / F (fixed list)
  - Note that in complete lists all species that are detected are recorded; in fixed lists only all species from a predefined list (e.g. Meadowbirds, Waterfowl) that are detected are recorded (this list should be provided in table Protocols (see below)).
  - ["In the rare cases where fixed lists cannot refer to a given predefined list of species their records must be provided as casual ones" has been deleted. Even if the list of species is not fully fixed, the tags in the protocol table allow to specify quite a lot of relavent information. If necessary, always such data can be used as casual.]
- **date** Date of the observational event.
- **time** Start time of the observational event in local time.
  - Empty or removed if unknown or locationMode=A.
- **location_mode**
  - E (original exact location provided)
  - D (location lowered to 10x10km level —ETRS89-LAEA grid—)
  - A (data aggregated at 10x10km level —ETRS89-LAEA grid—)
  - Note that complete and fixed lists can be provided either using locationMode E or D, while casual records must be provided always aggregated at 10x10 (i.e. using locationMode A).
- **location**
  - **location_mode=E** original exact location provided in Well Known Text (WKT) in WGS84. Example: POINT(3.056 41.813)
  - **location_mode = D/A** ETRS89-LAEA grid 10x10 code. Example: 10kmE353N212
- **protocol_id** Identifier of the protocol followed (e.g. a given Common Breeding Bird Survey). Leave blank in case of casual records and when complete lists do not proceed from standard monitoring projects.
- **radius** Maximum distance (in m) to the location centroid travelled/covered during the observational event (e.g. 500m).

    - Empty or removed when: unknown and when locationMode = A (note that when locationMode = D this info is still very useful —e.g. to identify complete lists where the observer travelled too far away—).
- **duration** Duration (in hours)
    - Empty or removed if unknown or locationMode=A.
- **records** (recordsTotal in the standard) Total number of records.
    - When **new or modified** data, must be always **>0**.
        - Note that when **locationMode = E/D**, the total number of records equals the number of species detected in the complete/fixed list (the total must include all species, not only those that are currently submitted to the EBP).
        - If records == 0 the complete/fixed list is empty. Event exists but records haven't been recorded or there are no records from EBP target species.
    - When locationMode = A, use as total number of records the number of different combinations of observer and species recorded in the given date and 10x10 square.
        - When **locationMode = A**, the total corresponds to the total number of aggregated records." has been changed for "When locationMode = A, use as total number of records the number of different combinations of observer and species recorded in the given date and 10x10 square.". Since this is the only way to ensure some standarization in the way casual records are counted.
- **observer**
    - If locationMode = E/D -> Identifier of the observer (observer ID). Observer must be unique at the level of the partnerID.
    - If locationMode = A -> Number of different observers submitting observations for the given 10x10 square and date.
- **protocol_id** Identifier of the protocol followed (e.g. a given Common Breeding Bird Survey).
    - Leave blank in case of casual records and when complete lists do not proceed from standard monitoring projects.
- **state** field will be provided during Standard data provisions.
    - **state = 0** provided event has been removed
    - **state = 1** provided event is new or has been modified

---

Removed from the standard

- partner_source_id ~~All event will come from the same partner_source_id provided in the data provision.~~

Records

```
{
    'properties': {
        'breeding_code': { 'type': 'integer', 'description':'Total number of records.','required':False},
        'count': { 'type': 'integer', 'minimum': 0, 'description':'Number of individuals counted (loc: E/D) or Max
        'event_id':  { 'type': 'string', 'description':'Identifier of the observational event (e.g. a given comple
        'flying_over':  { 'type': 'string' },
        'record_id': { 'type': 'string' , 'description': 'Identifier of the record'},
        'records_of_species':  { 'type': 'integer', 'minimum': 1, 'description':'Number of records of the given sp
        'state': {'type':'number'}
      }
    }
}
```

- **record_id** Identifier of the record.
    - For example, you can use eventID and speciesCode as recordID for aggregated casual records (i.e. when Events table locationMode = A).

- **event_id** Identifier of the observational event (e.g. a given complete list).
- **species_code** Species code (HBW codes)
- **count**
  - If locationMode = E/D -> Number of individuals counted.
  - If locationMode = A -> Maximum count of all records with counts.
  - Leave null if only presence is known.
  - Since some partners give option to use qualificators (e.g. >,=,aprox, etc), counts should be calculated on the raw numbers (e.g. using 200 for >200). Using only observations where numbers are qualified as exact numbers may reduce sample very much.
- **records_of_species** Number of records of the given species.
  - If locationMode = E/D then records_of_species must be always 1.
  - When locationMode = A, use as total number of records of the given species the number of different observers that have recorded it in the given date and 10x10 square. [to be homologous to Events table records]
- **breeding_code** Maximum breeding code. Codes based on EBBA2 standard.
- **flying_over** Y (yes) / N (no)
  - Empty or removed when: unknown/unclear or location_mode: A
- **state** field will be provided during Standard data provisions.
  - **state = 0** provided record has been removed
  - **state = 1** provided record is new or has been modified

Validation phases

We have splitted validation process in several phases. Phases 1 and 2 are done before giving the answer to the client. Phase 3 validation require more time and are sent to a queue and processed later.

1. **Global and schema validations**. It checks that the data provision fits the JSON schema. Also checks that partner_source and date_range are correct.
2. **Pre-validation**. It checks simple errors that can't be checked with the schema. For example: protocol_codes, event_code_repetitions inside the same provision or species_codes repetitions inside the same event,...
3. **Post-validations**. Those validations are done directly into the database. For example: points inside the partner area, correct species_codes, etc...

> We've created the audit tables for logging the validations during data provision process. You can check through the API the list of errors for each data provision. The reply from the server will give you the audit_id to access lately to the error list.

Schema validations

JSON schema validations	Error code	Implemented	Fields
Field should be an integer	integer_format	✅	records
Field should be a number	number_format	✅	duration, location_x, location_y, radius
Field should be a string	string_format	✅	event_id, flying_over, record_id, event_id, observer, protocol_id, partner_source, mode

JSON schema validations	Error code	Implemented	Fields
Required fields	required_field	✅	partner_source, start_date, end_date, mode, events, records, data_type, count, event_id, species_code, record_id, records_of_species, data_type,date, event_id, location_mode, location_x, location_y, location, observer, protocol_id, records
Field should be a date in ISO 8601 format (YYYY-MM-DD)	date_format	✅	start_date, end_date, date
Field should be a time in format HH:MM:SS	time_format	✅	time
Location_mode should be E (original exact location provided), D (location lowered to 10x10km level ETRS89-LAEA grid) or A (data aggregated at 10x10km level ETRS89-LAEA grid)	location_mode_format	✅	location
Mode should be B (bulk mode), S (standard mode) or T (test mode)	mode_format	✅	mode

Global checks

Global cheks	Error code	Implemented
Partner id exists	partner_not_found	✅
Start date later than initial EBP date	partner_not_found	✅
End date not in the future	old_init_date	✅

Pre-checks

Pre-checks	Error code	Implemented
Protocol code not found	protocol_not_found	✅
When location_mode is A (aggregated), field value has to be null (time,duration,radius,flyingover)	field_not_null_aggregated	✅
Location_mode E/D records_of_species > 1	records_not_agg_gt_1	✅
Location_mode A observer different observers	observer_not_number	✅
Event_date outside provided range in bulk	outside_date_range	✅
Provided extenal_event_id is not unique, has been already provided	event_id_not_unique	✅
Provided extenal_record_id is not unique, has been already provided	record_id_not_unique	✅
Provided species_code is repeated in the same event	species_code_not_unique	✅

Pre-checks	Error code	Implemented
Duration should be smaller than 24 hours	duration_gt_24h	✔️
Records must be greater than 0	zero_records	✔️

Post-check

Post-check	Error code	Implemented
Location is outside partners area	outside_location	✔️
Species_code not found in the EBP species list	species_code_not_found	✔️
Provided extenal_event_id in record not found in provided events	event_id_not_found	✔️
Provided species_code when protocol data is outside fixed list		

Protocols

Standard bird monitoring data is already collected by some local online portals and, therefore, we needed a standard that could handle it correctly. Properly storing this information will certainly increase the overall quality of the EBP data but also opens new possibilities in terms of data analysis and regarding the development of further synergies with other EBCC initiatives.

Note that to be able to deal with data coming from fixed lists and standard monitoring projects in general, we needed to add a third table to the ones already existing in the former standard: the tables events and records. This third table, named **protocols**, will collect the details of the protocol followed (e.g. a given Common Breeding Bird Survey) and, in case of fixed lists, the definition of the list.

We've created several API methods to define and create your own protocols into the EBP respository database. You can see the JSON protocol definition and the fields description. Once the protocol is created, you can use your created protocols code in the protocol_id field in the data provision events.


```
{
    "protocol_code": "ODJ",
    "title": "Ocells dels Jardins",
    "project_type": "GS",
    "method": "T",
    "website": "http://ocellsdelsjardins.cat/",
    "description": "Ocells dels Jardins is a citizen science project aimed to monitor the use of gardens and small
    "protocol_details": "Very simple protocol. Only birds detected in the defined sampling area (i.e. garden) and
    "ebp_data_structure": "Identital to original database",
    "citation": "2015. Ocells dels Jardins, Catalan Ornithological Institute",
    "id_gbif" : "",
    "geographic_coverage": "Catalonia, Spain",
    "start_year": 2014,
    "end_year": "",
    "ongoing": true,
    "link": "",
    "fixed_list_tags": "ESP(54105;54154;57821;58496;58952;58861;60925;61286;61290;53077;54565;55328;55871;57729;57
}
```


Protocol fields description

> Link, id_gbif, end_year and website are optional. If they are emtpy or null, it's not necessary to send the fields in the JSON file.

- **protocol_code** Identifier of the protocol followed (e.g. a given Common Breeding Bird Survey).
- **title** Procotol name/title
- **project_type**

project_type	Project title
CB	Common breeding bird survey
CW	Common winter bird survey
WW	Winter waterbird count
BA	Breeding bird atlas
MC	Migration count
WA	Winter bird atlas
GS	Garden bird survey
RB	Rare breeding bird survey
OT	other monitoring project
BR	Bird ringing/banding results
NF	Nocturnal flight calls survey

- **method**

method	Method description
P	point counts
M	mapping methods
L	line-transect
T	flexible surveys in which only time is controlled and there is no special requirement regarding the area/distance covered or speed

- **website** url of the project/protocol (if existing)
- **description** Brief description of the protocol.
- **protocol_details** Details about the protocol that complement the information given in fixedlistTags.
- **ebp_data_structure** Details about how the data has been "downgraded" to a complete/fixed list format.
- **citation** Reference to the protocol.
- **id_gbif** GBIF doi url to the metadata persistent (doi) of the metadata/dataset uploaded to gbif (i.e. http://doi.org/10.15468/jsjoae).
- **geographic_coverage** Area covered by the protocol/project.
- **start_year** Start year.
- **end_year** Finishing year. Leave empty if not finished.
- **ongoing** true or false

- **fixed_list_tags** (only for dataType = F)
    - If the protocol has a list of target species, you can explicitly provide it. Give a list of all these species, including non-target species, within the tag "ESP()" separated with a semicolon(;) (e.g. "ESP(54105;54154;57821)". Use species codes from HBW codes.
    - You can also add a predefined tag from tto include or exclude a group of species (i.e. only raptors or no fly-overs). Use a semicolon (;) to separate them; in many cases just one tag will be enough.

Species tags	Species tag description
NFO	no fly-overs
ORB	only ringed/trapped birds
OBB	only breeding birds
OWB	only waterbirds
OSB	only seabirds
ORA	only raptors
ORS	only raptors and soaring birds
OAM	only active migrants
PLN	partial list no strict: other species can be reported

API protocol methods

Endpoint	Method	Description
/protocols	GET	Get list of own protocols
/protocols	POST	Create new protocol from the provided JSON
/protocols/{procotol_code}	GET	Get a concrete protocol with {protocol_code}
/protocols/{procotol_code}	PUT	Modify the protocol {protocol_code} with the provided JSON
/protocols/{procotol_code}	DELETE	Delete the protocol {protocol_code} if it's no related events

Recomended system integration steps

1. DB/system preparation work

- Id's generation (uniques inside partner_source)
    - events (i.e. when aggregated: date + 10x10_code)
    - records (i.e. when aggregated: event_id+species_code)
- Create species table mappings
- Create breeding codes mappings
- Unique id's generation
- Handle/track updates and removals
- Data aggregation (10x10 for casual data -> ETRS89-LAEA grid)

2. Basic data provision testing

- Create the system username for authentiation (ask ICO-team)
- Get access to https://api.eurobirdportal.org/admin/ API admin
- Get through the API the Oauth2 token: **using username,password, client_id, client_secret, scope=api**
- Start with simple data provisions in Test Mode (T)
- Determinate protocol data and create protocols through the API

3. Standard data flow integration

- Create **Standard Mode (S)** data provisions with real data.
- Create cronjobs or equivalent to send data as periodic tasks.
- Test removals and past modifications.
- Create Standard Mode (S) data provisions with real data
  - Decide real-time connection time window **daily/weekly/monthly**
  - Decide records update mode
    - (A) all records
    - (M) only inserted, modified and removed
- Test removals and previous data modifications
- Create cronjobs or equivalent to send data as periodic tasks

4. Complete data flow with old data

- Send old data in **Bulk Mode (B)** in chunks. (optimal provision size shoud be determined)

Metadata

Species list

- Access to all species list from HBW
- Access to all ebp target species list

Breeding codes

Breeding code	Description
0	Non breeding (species observed but suspected to be still on migration or to be summering non-breeder)
1	Species observed in breeding season in possible nesting habitat
2	Singing male(s) present (or breeding calls heard) in breeding season
3	Pair observed in suitable nesting habitat in breeding season
4	Permanent territory presumed through registration of territorial behaviour (song, etc.) on at least two different days a week or more apart at same place
5	Courtship and display
6	Visiting probable nest-site
7	Agitated behaviour or anxiety calls from adults
8	Brood patch on adult examined in the hand

Breeding code	Description
9	Nest-building or excavating of nest-hole
10	Distraction-display or injury-feigning
11	Used nest or eggshells found (occupied or laid within period of survey)
12	Recently fledged young (nidicolous species) or downy young (nidifugous species)
13	Adults entering or leaving nest-site in circumstances indicating occupied nest (including high nests or nest holes, the contents of which cannot be seen) or adult seen incubating
14	Adult carrying a faecal sac or food for young
15	Nests containing eggs
16	Nests with young seen or heard

Partner sources

Partner source codes	Description
SWE_ART	artportalen.se
NOR_ART	artsobservasjoner.no
SLO_ASY	Aves-Symfony
CZE_BCZ	birds.cz
BUL_BTR	BirdTrack
CYP_BTR	BirdTrack
UKI_BTR	BirdTrack
SPA_BTR	BirdTrack
GRE_BTR	BirdTrack
LAT_DDA	Dabasdati (LV)
DEN_DBA	DOFbasen
SPA_EBI	eBird
ISR_EBI	eBird
GRE_EBI	eBird
TUR_EBI	eBird
POR_EBI	eBird
CRO_ORN	fauna.hr (Ornitho)
HUN_MAP	MAP
CAT_ODJ	ocellsdelsjardins.cat
RO1_OBM	OpenBirdMaps
AUS_ORN	ornitho.at

Partner source codes	Description
CAT_ORN	ornitho.cat
SWI_ORN	ornitho.ch
DEU_ORN	ornitho.de
EUS_ORN	ornitho.eus
FRA_ORN	ornitho.fr
ITA_ORN	ornitho.it
POL_ORN	ornitho.pl
RO2_ODA	OrnitoData
EST_PLU	Plutof
BUL_SBI	SmartBirds
NET_SOV	Sovon
FIN_TII	Tiira
NET_TRE	Trektellen
BEL_OBS	waarnemingen.be/observations.be
NET_OBS	waarneming.nl

EBP API

EBP API repository

species : Operations related to species

Show/Hide | List Operations | Expand Operations

GET /species/ — Returns list of all species

Response Class (Status 200)

Success

Model | Example Value


```
[
  {
    "species_id": 0,
    "latin": "string",
    "english": "string"
  }
]
```


Response Content Type application/json

Try it out!

GET /species/ebp — Returns list of all target EBP species

Response Class (Status 200)

Success

Model | Example Value


```
[
  {
    "species_id": 0,
    "latin": "string",
    "english": "string"
  }
]
```


Response Content Type application/json

Try it out!

GET /species/subspecies — Returns list of all species at subspecies level

Response Class (Status 200)

Success

Model | Example Value


```
[
  {
    "species_id": 0,
    "subspecies": "string",
    "latin": "string",
    "subspecies_id": 0,
    "english": "string"
```

```
  }
]
```


Response Content Type application/json ▾

Try it out!

GET /species/{species_id} Returns species for a concrete code

Response Class (Status 200)

Success

Model | Example Value


```
{
  "species_id": 0,
  "latin": "string",
  "english": "string"
}
```


Response Content Type application/json ▾

Parameters

Parameter	Value	Description	Parameter Type	Data Type
species_id	(required)		path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
404	Species code not found.		

Try it out!

protocols : Operations related to protocols

Show/Hide | List Operations | Expand Operations

GET /protocols/ [WIP] Returns list of all protocols

Response Class (Status 200)

Success

Model | Example Value


```
[
  {
    "website": "string",
    "geographic_coverage": "string",
    "start_year": 0,
    "protocol_details": "string",
    "ebp_data_structure": "string",
    "protocol_code": "string",
    "title": "string",
    "end_year": 0,
    "citation": "string"
```


Response Content Type application/json ▾

Try it out!

POST /protocols/ Creates a new Protocol

Parameters

Parameter	Value	Description	Parameter Type	Data Type
payload	(required)		body	Model Example Value

Parameter content type: application/json


```
{
  "website": "string",
  "geographic_coverage": "string",
  "start_year": 0,
  "protocol_details": "string",
  "ebp_data_structure": "string",
  "protocol_code": "string",
  "title": "string",
  "end_year": 0,
  "citation": "string",
  "link": "string",
  "project_type": "string",
```


Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Protocol successfully created.		

Try it out!

GET /protocols/project_types — Returns list of all project types in protocols

Response Class (Status 200)

Success

Model Example Value


```
[
  {
    "proj_code": "string",
    "description": "string"
  }
]
```


Response Content Type application/json

Try it out!

breeding_code : Operations related to Breeding codes

Show/Hide | List Operations | Expand Operations

GET /breeding_code/ — Returns list of all breeding codes

Response Class (Status 200)

Success

Model Example Value


```
[
  {
    "breeding_code_text": "string",
    "breeding_code": 0
  }
]
```


Response Content Type application/json

Try it out!

GET /breeding_code/{code} — Returns breeding description for a concrete code

Response Class (Status 200)

Success

Model | Example Value


```
{
  "breeding_code_text": "string",
  "breeding_code": 0
}
```


Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
code	(required)		path	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
404	Species code not found.		

Try it out!

data : Operations related to data provision

Show/Hide | List Operations | Expand Operations

POST /data/ — Send a new data provision

Response Class (Status 200)

Success

Model | Example Value


```
{
  "partner_source": "string",
  "end_date": "2018-06-04",
  "records_count": 0,
  "audit_id": 0,
  "events_count": 0,
  "start_date": "2018-06-04"
}
```


Response Content Type application/json

Try it out!

GET /data/audit — Returns all data provision audits send by partner json

Response Class (Status 200)

Success

Model | Example Value


```
[
  {
    "partner_source": "string",
    "status": "string",
    "end_date": "string",
    "data_log": [
      {
        "_type": "string",
        "audit_id_log": 0,
```

```
"error_message": "string",
```


Response Content Type application/json

Try it out!

GET /data/audit/last — Returns last data provision audits send by partner json

Response Class (Status 200)

Success

Model | Example Value


```
[
  {
    "partner_source": "string",
    "status": "string",
    "end_date": "string",
    "data_log": [
      {
        "_type": "string",
        "audit_id_log": 0,
        "error_message": "string",
        "field_source": "string"
```


Response Content Type application/json

Try it out!

GET /data/audit/{code} — Returns data audit from data provision with provided code

Response Class (Status 200)

Success

Model | Example Value


```
[
  {
    "partner_source": "string",
    "status": "string",
    "end_date": "string",
    "data_log": [
      {
        "_type": "string",
        "audit_id_log": 0,
        "error_message": "string",
        "field_source": "string"
```


Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
code	(required)		path	string

Try it out!

POST /data/bulk — Send a new data provision

Response Class (Status 200)

Success

Model | Example Value

```
{
  "partner_source": "string",
  "status": "string",
  "end_date": "string",
  "data_log": [
    {
      "_type": "string",
      "audit_id_log": 0,
      "error_message": "string",
      "field_source": "string",
      "field": "string"
```


Response Content Type application/json

Try it out!

oauth : Operations related to authorization

Show/Hide | List Operations | Expand Operations

GET /oauth/me — The Authorization Server provides the user profile

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

Try it out!

POST /oauth/revoke — Notify the authorization server that a previously obtained access token is no longer needed

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

Try it out!

POST /oauth/token — The Authorization Server provides the access token

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

Try it out!

admin : Operations related Administration or internal API tasks

Show/Hide | List Operations | Expand Operations

GET /admin/ — Administration console access

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

Try it out!

GET /admin/viewer/chart/last_weeks/{weeks} — Generates a summary with data week by week for a concrete species and year

Parameters

Parameter	Value	Description	Parameter Type	Data Type
weeks	(required)		path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

Try it out!

GET /admin/viewer/chart/{partner_source}/{species_code}/{year}

Generates a summary with data week by week for a concrete species and year

Parameters

Parameter	Value	Description	Parameter Type	Data Type
year	(required)		path	string
species_code	(required)		path	string
partner_source	(required)		path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

Try it out!

GET /admin/viewer/data/{partner_source}/{species_code}/{year}

Generates a summary with data for concrete species and year

Parameters

Parameter	Value	Description	Parameter Type	Data Type
year	(required)		path	string
species_code	(required)		path	string
partner_source	(required)		path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

Try it out!

GET /admin/viewer/shp/{partner_source}

Get GeoJSON shapefile from parnter's including buffer

Parameters

Parameter	Value	Description	Parameter Type	Data Type
partner_source	(required)		path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

Try it out!

GET /admin/viewer/{partner_source}/{year} Generates a summary with data for concrete week and year

Parameters

Parameter	Value	Description	Parameter Type	Data Type
year	(required)		path	string
partner_source	(required)		path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	Success		

Try it out!

[BASE URL: / , API VERSION: 1.0]