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:

- **Data provisions** from online portals using the new EBP standard.
- **Maps creation** for the demo viewer.
- **Repository management** such as the administration zone, user creation and internal visualizations (maps and graphs).
- **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](chart.png)
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 its 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

Table of Contents

- Overview
- EBP workflow
  - Introduction
  - Data flow schema
  - Authentication
  - Connection scenarios
  - Modification scenarios
  - Data provision structure
  - Events and records
  - Schema validations
  - Validation phases
    - Global checks
    - Pre checks
    - Post checks
- Protocols
- Recommended system integration steps
- Metadata
  - Species list
  - Breeding codes
  - Partner sources

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 include 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 be 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 submitted 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 partners table and partner_sources where you will find the partner_source_code required for the data provision.

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 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`

```bash
curl -u client_id:client_secret -X POST 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.

```bash
```

## Connection scenarios

We will have three different connection scenarios:

<table>
<thead>
<tr>
<th>Mode</th>
<th>All data replaced</th>
<th>Delete events</th>
<th>Old events (outside provided date range)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bulk mode</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Standard mode</td>
<td>✗</td>
<td>✓</td>
<td>✓ (updates, removals)</td>
</tr>
<tr>
<td>Test mode</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

### 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 purposes.

## 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Changes</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>Update/insert</td>
<td>Send the complete event again (and modifications in case of event updates) with field state = 1</td>
</tr>
<tr>
<td>Event</td>
<td>Delete</td>
<td>Send the event again with field state = 0</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Changes</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record</td>
<td>Insert</td>
<td>Send new records with state=1</td>
</tr>
<tr>
<td>Record</td>
<td>Update</td>
<td>Send the updated records with state=1 again. They will be overwritten</td>
</tr>
<tr>
<td>Record</td>
<td>Delete</td>
<td>Send the removed records with field state=0</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Changes</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record</td>
<td>Update/insert</td>
<td>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</td>
</tr>
<tr>
<td>Record</td>
<td>Delete</td>
<td>No need to send the removed records</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partner_source</td>
<td>string</td>
<td>The unique of the EBP partner source</td>
</tr>
<tr>
<td>Property</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>start_date</td>
<td>date</td>
<td>Start date provided</td>
</tr>
<tr>
<td>end_date</td>
<td>date</td>
<td>Last date provided</td>
</tr>
<tr>
<td>events</td>
<td>array[Event]</td>
<td>List of events</td>
</tr>
<tr>
<td>records</td>
<td>array[Record]</td>
<td>List of records</td>
</tr>
<tr>
<td>mode</td>
<td>string</td>
<td>Provision mode B (Bulk: all data is replaced) / S (Standard: new data is</td>
</tr>
<tr>
<td></td>
<td>ennum[B,S,T]</td>
<td>provided) / T (Test: data validation porposes)</td>
</tr>
<tr>
<td>record_updates_mode</td>
<td>string</td>
<td>Update mode M (Only updated records will be provided) / Update mode A</td>
</tr>
<tr>
<td>(optional)</td>
<td>ennum[M,A]</td>
<td>(All records are provided). Default mode is M when record_updates_mode is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not provided</td>
</tr>
</tbody>
</table>

Data provision sample

The data provision structure

```json
{
  "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
    },
    {
      "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 provide an 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 list)
    'date': {'type': 'string'},
    'duration': {'type': 'number', 'description': 'Duration (in hours). Null if unknown or location_mode=A'},
    'event_id': {'type': 'string'},
    'location_mode': {'enum': ['E', 'D', 'A'], 'description': 'E (original exact location provided) / D (local
    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/covered during the observational event (e.g. 500m).'
    '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 relevant 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 omitted 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 standardization 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**

```json
{
    "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 completion"
            },
            "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 species"  
            },
            "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 qualifiers (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

<table>
<thead>
<tr>
<th>JSON schema validations</th>
<th>Error code</th>
<th>Implemented</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field should be an integer</td>
<td>integer_format</td>
<td>✓</td>
<td>records</td>
</tr>
<tr>
<td>Field should be a number</td>
<td>number_format</td>
<td>✓</td>
<td>duration, location_x, location_y, radius</td>
</tr>
<tr>
<td>Field should be a string</td>
<td>string_format</td>
<td>✓</td>
<td>event_id, flying_over, record_id, event_id, observer, protocol_id, partner_source, mode</td>
</tr>
</tbody>
</table>
### JSON schema validations

<table>
<thead>
<tr>
<th>Field name</th>
<th>Error code</th>
<th>Implemented</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required fields</td>
<td>required_field</td>
<td>✓</td>
<td>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</td>
</tr>
<tr>
<td>Field should be a date in ISO 8601 format (YYYY-MM-DD)</td>
<td>date_format</td>
<td>✓</td>
<td>start_date, end_date, date</td>
</tr>
<tr>
<td>Field should be a time in format HH:MM:SS</td>
<td>time_format</td>
<td>✓</td>
<td>time</td>
</tr>
<tr>
<td>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)</td>
<td>location_mode_format</td>
<td>✓</td>
<td>location</td>
</tr>
<tr>
<td>Mode should be B (bulk mode), S (standard mode) or T (test mode)</td>
<td>mode_format</td>
<td>✓</td>
<td>mode</td>
</tr>
</tbody>
</table>

### Global checks

<table>
<thead>
<tr>
<th>Global checks</th>
<th>Error code</th>
<th>Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partner id exists</td>
<td>partner_not_found</td>
<td>✓</td>
</tr>
<tr>
<td>Start date later than initial EBP date</td>
<td>partner_not_found</td>
<td>✓</td>
</tr>
<tr>
<td>End date not in the future</td>
<td>old_init_date</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Pre-checks

<table>
<thead>
<tr>
<th>Pre-checks</th>
<th>Error code</th>
<th>Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol code not found</td>
<td>protocol_not_found</td>
<td>✓</td>
</tr>
<tr>
<td>When location_mode is A (aggregated), field value has to be null (time,duration,radius,flayingover)</td>
<td>field_not_null_aggregated</td>
<td>✓</td>
</tr>
<tr>
<td>Location_mode E/D records_of_species &gt; 1</td>
<td>records_not_agg_gt_1</td>
<td>✓</td>
</tr>
<tr>
<td>Location_mode A observer different observers</td>
<td>observer_not_number</td>
<td>✓</td>
</tr>
<tr>
<td>Event_date outside provided range in bulk</td>
<td>outside_date_range</td>
<td>✓</td>
</tr>
<tr>
<td>Provided external_event_id is not unique, has been already provided</td>
<td>event_id_not_unique</td>
<td>✓</td>
</tr>
<tr>
<td>Provided external_record_id is not unique, has been already provided</td>
<td>record_id_not_unique</td>
<td>✓</td>
</tr>
<tr>
<td>Provided species_code is repeated in the same event</td>
<td>species_code_not_unique</td>
<td>✓</td>
</tr>
<tr>
<td>Pre-check</td>
<td>Error code</td>
<td>Implemented</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Duration should be smaller than 24 hours</td>
<td>duration_gt_24h</td>
<td>✔</td>
</tr>
<tr>
<td>Records must be greater than 0</td>
<td>zero_records</td>
<td>✔</td>
</tr>
</tbody>
</table>

### Post-check

<table>
<thead>
<tr>
<th>Post-check</th>
<th>Error code</th>
<th>Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location is outside partners area</td>
<td>outside_location</td>
<td>✔</td>
</tr>
<tr>
<td>Species_code not found in the EBP species list</td>
<td>species_code_not_found</td>
<td>✔</td>
</tr>
<tr>
<td>Provided external_event_id in record not found in provided events</td>
<td>event_id_not_found</td>
<td>✔</td>
</tr>
<tr>
<td>Provided species_code when protocol data is outside fixed list</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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 repository 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.

```json
{
  "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": "Identical 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;5"
}
```

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

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

- **method**

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

- **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/jjsjoe).
- **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.

<table>
<thead>
<tr>
<th>Species tags</th>
<th>Species tag description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFO</td>
<td>no fly-overs</td>
</tr>
<tr>
<td>ORB</td>
<td>only ringed/trapped birds</td>
</tr>
<tr>
<td>OBB</td>
<td>only breeding birds</td>
</tr>
<tr>
<td>OWB</td>
<td>only waterbirds</td>
</tr>
<tr>
<td>OSB</td>
<td>only seabirds</td>
</tr>
<tr>
<td>ORA</td>
<td>only raptors</td>
</tr>
<tr>
<td>ORS</td>
<td>only raptors and soaring birds</td>
</tr>
<tr>
<td>OAM</td>
<td>only active migrants</td>
</tr>
<tr>
<td>PLN</td>
<td>partial list no strict; other species can be reported</td>
</tr>
</tbody>
</table>

**API protocol methods**

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/protocols</td>
<td>GET</td>
<td>Get list of own protocols</td>
</tr>
<tr>
<td>/protocols</td>
<td>POST</td>
<td>Create new protocol from the provided JSON</td>
</tr>
<tr>
<td>/protocols/(protocol_code)</td>
<td>GET</td>
<td>Get a concrete protocol with (protocol_code)</td>
</tr>
<tr>
<td>/protocols/(protocol_code)</td>
<td>PUT</td>
<td>Modify the protocol (protocol_code) with the provided JSON</td>
</tr>
<tr>
<td>/protocols/(protocol_code)</td>
<td>DELETE</td>
<td>Delete the protocol (protocol_code) if it’s no related events</td>
</tr>
</tbody>
</table>

**Recommended 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 authentication (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 should be determined)

Metadata

Species list

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

Breeding codes

<table>
<thead>
<tr>
<th>Breeding code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Non breeding (species observed but suspected to be still on migration or to be summering non-breeder)</td>
</tr>
<tr>
<td>1</td>
<td>Species observed in breeding season in possible nesting habitat</td>
</tr>
<tr>
<td>2</td>
<td>Singing male(s) present (or breeding calls heard) in breeding season</td>
</tr>
<tr>
<td>3</td>
<td>Pair observed in suitable nesting habitat in breeding season</td>
</tr>
<tr>
<td>4</td>
<td>Permanent territory presumed through registration of territorial behaviour (song, etc.) on at least two different days a week or more apart at same place</td>
</tr>
<tr>
<td>5</td>
<td>Courtship and display</td>
</tr>
<tr>
<td>6</td>
<td>Visiting probable nest-site</td>
</tr>
<tr>
<td>7</td>
<td>Agitated behaviour or anxiety calls from adults</td>
</tr>
<tr>
<td>8</td>
<td>Brood patch on adult examined in the hand</td>
</tr>
<tr>
<td>Breeding code</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>9</td>
<td>Nest-building or excavating of nest-hole</td>
</tr>
<tr>
<td>10</td>
<td>Distraction-display or injury-feigning</td>
</tr>
<tr>
<td>11</td>
<td>Used nest or eggshells found (occupied or laid within period of survey)</td>
</tr>
<tr>
<td>12</td>
<td>Recently fledged young (nidicolous species) or downy young (nidifugous species)</td>
</tr>
<tr>
<td>13</td>
<td>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</td>
</tr>
<tr>
<td>14</td>
<td>Adult carrying a faecal sac or food for young</td>
</tr>
<tr>
<td>15</td>
<td>Nests containing eggs</td>
</tr>
<tr>
<td>16</td>
<td>Nests with young seen or heard</td>
</tr>
</tbody>
</table>

**Partner sources**

<table>
<thead>
<tr>
<th>Partner source codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWE_ART</td>
<td>artportalen.se</td>
</tr>
<tr>
<td>NOR_ART</td>
<td>artsobservasjoner.no</td>
</tr>
<tr>
<td>SLO_ASY</td>
<td>Aves-Symphony</td>
</tr>
<tr>
<td>CZE_BAZ</td>
<td>birds.cz</td>
</tr>
<tr>
<td>BUL_BTR</td>
<td>BirdTrack</td>
</tr>
<tr>
<td>CYP_BTR</td>
<td>BirdTrack</td>
</tr>
<tr>
<td>UKL_BTR</td>
<td>BirdTrack</td>
</tr>
<tr>
<td>SPA_BTR</td>
<td>BirdTrack</td>
</tr>
<tr>
<td>GRE_BTR</td>
<td>BirdTrack</td>
</tr>
<tr>
<td>LAT_DDA</td>
<td>Dabasdati (LV)</td>
</tr>
<tr>
<td>DEN_DBA</td>
<td>DOFbasen</td>
</tr>
<tr>
<td>SPA_EBI</td>
<td>eBird</td>
</tr>
<tr>
<td>ISR_EBI</td>
<td>eBird</td>
</tr>
<tr>
<td>GRE_EBI</td>
<td>eBird</td>
</tr>
<tr>
<td>TUR_EBI</td>
<td>eBird</td>
</tr>
<tr>
<td>POR_EBI</td>
<td>eBird</td>
</tr>
<tr>
<td>CRO_ORN</td>
<td>fauna.hr (Ornitho)</td>
</tr>
<tr>
<td>HUN_MAP</td>
<td>MAP</td>
</tr>
<tr>
<td>CAT_ODJ</td>
<td>ocellsdelsgardins.cat</td>
</tr>
<tr>
<td>RO1_OBM</td>
<td>OpenBirdMaps</td>
</tr>
<tr>
<td>AUS_ORN</td>
<td>ornitho.at</td>
</tr>
<tr>
<td>Partner source codes</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>CAT_ORN</td>
<td>ornitho.cat</td>
</tr>
<tr>
<td>SWI_ORN</td>
<td>ornitho.ch</td>
</tr>
<tr>
<td>DEU_ORN</td>
<td>ornitho.de</td>
</tr>
<tr>
<td>EUS_ORN</td>
<td>ornitho.eus</td>
</tr>
<tr>
<td>FRA_ORN</td>
<td>ornitho.fr</td>
</tr>
<tr>
<td>ITA_ORN</td>
<td>ornitho.it</td>
</tr>
<tr>
<td>POL_ORN</td>
<td>ornitho.pl</td>
</tr>
<tr>
<td>RO2_ODA</td>
<td>OrnitData</td>
</tr>
<tr>
<td>EST_PLU</td>
<td>Plutof</td>
</tr>
<tr>
<td>BUL_SBI</td>
<td>SmartBirds</td>
</tr>
<tr>
<td>NET_SOV</td>
<td>Sovon</td>
</tr>
<tr>
<td>FIN_TII</td>
<td>Tiira</td>
</tr>
<tr>
<td>NET_TRE</td>
<td>Trektellen</td>
</tr>
<tr>
<td>BEL_OBS</td>
<td>waarnemingen.be/observations.be</td>
</tr>
<tr>
<td>NET_OBS</td>
<td>waarneming.nl</td>
</tr>
</tbody>
</table>
# EBP API

**species**: Operations related to species

## GET /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

**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

**Response Class (Status 200)**

Success

**Model**: Example Value

```
[
    {
        "species_id": 0,
        "subspecies": "string",
        "latin": "string",
        "subspecies_id": 0,
        "english": "string"
    }
]
```
GET /species/{species_id}

Response Class (Status 200)
Success

Model
Example Value

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

Response Content Type: application/json

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
<th>Type</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>species_id</td>
<td>(required)</td>
<td>path</td>
<td></td>
<td>string</td>
</tr>
</tbody>
</table>

Response Messages

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Reason</th>
<th>Response Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>404</td>
<td>Species code not found.</td>
<td></td>
</tr>
</tbody>
</table>

Try it out!

protocols: Operations related to protocols

GET /protocols/

Response Class (Status 200)
Success

Model
Example Value

```json
[
    {
        "website": "string",
        "geographic_coverage": "string",
        "start_year": 0,
        "protocol_details": "string",
        "ebp_data_structure": "string",
        "protocol_code": "string",
        "title": "string",
        "end_year": 0,
        "publication": "string"
    }
]
```

Response Content Type: application/json

Try it out!

POST /protocols/

Creates a new Protocol

Parameters
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
<th>Parameter</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>payload</td>
<td>(required)</td>
<td></td>
<td>body</td>
<td>Example Value</td>
</tr>
</tbody>
</table>

**Parameter content type:** application/json

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Reason</th>
<th>Response Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>201</td>
<td>Protocol successfully created.</td>
<td></td>
</tr>
</tbody>
</table>

**Response Messages**

**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

**breeding_code**

Operations related to Breeding codes

**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

**breeding_code**

Operations related to Breeding codes

**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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
<th>Parameter Type</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>(required)</td>
<td></td>
<td>path</td>
<td>integer</td>
</tr>
</tbody>
</table>

Response Messages

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Reason</th>
<th>Response Model</th>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>404</td>
<td>Species code not found.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Try it out!

---

data : Operations related to data provision

POST /data/

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

Response Class (Status 200)
Success

Model: Example Value

```
[
    {
        "partner_source": "string",
        "status": "string",
        "end_date": "string",
        "data_log": [
            {
                "_type": "string",
                "audit_id_log": 0,
            }
        ]
    }
```
GET /data/audit/last

Response Class (Status 200)
Success

Model | Example Value

```json
[
{
  "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)

Response Class (Status 200)
Success

Model | Example Value

```json
[
{
  "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

Response Class (Status 200)
Success

Model | Example Value
oauth: Operations related to authorization

GET /oauth/me

The Authorization Server provides the user profile

Response Messages

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Reason</th>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Success</td>
<td></td>
</tr>
</tbody>
</table>

Try it out!

POST /oauth/revoke

Notify the authorization server that a previously obtained access token is no longer needed

Response Messages

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Reason</th>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Success</td>
<td></td>
</tr>
</tbody>
</table>

Try it out!

POST /oauth/token

The Authorization Server provides the access token

Response Messages

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Reason</th>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Success</td>
<td></td>
</tr>
</tbody>
</table>

Try it out!

admin: Operations related Administration or internal API tasks

GET /admin/

Administration console access

Response Messages

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Reason</th>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Success</td>
<td></td>
</tr>
</tbody>
</table>

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
### Response Messages

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Reason</th>
<th>Response Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Success</td>
<td></td>
</tr>
</tbody>
</table>

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

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

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
<th>Parameter Type</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td></td>
<td></td>
<td>path</td>
<td>string</td>
</tr>
<tr>
<td>species_code</td>
<td></td>
<td></td>
<td>path</td>
<td>string</td>
</tr>
<tr>
<td>partner_source</td>
<td></td>
<td></td>
<td>path</td>
<td>string</td>
</tr>
</tbody>
</table>

**Response Messages**

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Reason</th>
<th>Response Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Success</td>
<td></td>
</tr>
</tbody>
</table>

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

Generates a summary with data for concrete species and year.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
<th>Parameter Type</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td></td>
<td></td>
<td>path</td>
<td>string</td>
</tr>
<tr>
<td>species_code</td>
<td></td>
<td></td>
<td>path</td>
<td>string</td>
</tr>
<tr>
<td>partner_source</td>
<td></td>
<td></td>
<td>path</td>
<td>string</td>
</tr>
</tbody>
</table>

**Response Messages**

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Reason</th>
<th>Response Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Success</td>
<td></td>
</tr>
</tbody>
</table>

#### GET /admin/viewer/shp/{partner_source}

Get GeoJSON shapefile from partner's including buffer.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
<th>Parameter Type</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>partner_source</td>
<td></td>
<td></td>
<td>path</td>
<td>string</td>
</tr>
</tbody>
</table>

**Response Messages**

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Reason</th>
<th>Response Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Success</td>
<td></td>
</tr>
</tbody>
</table>

Try it out!
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
<th>Parameter Type</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>(required)</td>
<td></td>
<td>path</td>
<td>string</td>
</tr>
<tr>
<td>partner_source</td>
<td>(required)</td>
<td></td>
<td>path</td>
<td>string</td>
</tr>
</tbody>
</table>

## Response Messages

<table>
<thead>
<tr>
<th>HTTP Status Code</th>
<th>Reason</th>
<th>Response Model</th>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Success</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[ BASE URL: /, API VERSION: 1.0 ]