Data as resources: how to enhance data sharing capabilities between the Secretariat and the scientific community

PREPARED BY: IOTC SECRETARIAT[1], 18TH NOVEMBER 2016

Introduction

External access to non-confidential data managed by the Secretariat has always been historically mediated by the availability of such information through the IOTC web pages (online query panel, datasets prepared prior to each working party or available under data-specific sections of the website) and by direct communication between scientists and the Data Section of the Secretariat.

Typical examples of requests issued by scientists or data analysts are:

- Current nominal catches for a given fleet / fishery
- Total catches or efforts within a specific area (mostly, a country EEZ or the high seas)
- Size-frequency data converted to standard lengths for a given species or a subset of species (e.g. billfish)
- Nominal catches by fishery group or species group in a specific timeframe

While it's true that most of this information could be obtained by processing the standard datasets available for download from the corresponding sections of the website, the possibility of having direct access to the *live* data (i.e. the current content of the database, that due to revisions or scheduled updates might differ from the status of the data used for the preparation of the datasets uploaded on the website) was so far missing.

Additionally, some of the required processing tasks – while trivial for most experienced users – could turn out to be unnecessarily complex for a relevant part of the audience interested to this type of information (e.g. transformation of geospatial information into grids of a given level of resolution, or intersection of that same geospatial information with specific areas of the Indian Ocean).

Fulfilling the different types of requests directed at the Data Section of the Secretariat is a time-consuming task, and the effort required by the team to respond to such requests reiterated over time has an impact on the overall productivity.

Beside these efficiency reasons – which are in any case worth considering – the asynchronous nature of a workflow in which data is available to scientists either when it has been prepared by the Secretariat (according to specific users' requests) or when it's finally published on the website as part of the usual update cycle, prevents a convenient and streamlined access to the information.

Furthermore, the format in which data is provided to end users might not always be the best choice for scientists' needs: downloading and unzipping an XLS file for its content to be available as a CSV dataset as input to an R script, is something that could be improved and rationalized.

The recent evolution and restructuring of the data management processes in place at the Secretariat ([IOTC-2016-WPDCS12-25_Rev1]) resulted in the development of an integrated information system that has the purpose of ingesting and managing all the information provided by CPCs in accordance with mandatory requirements as per Resolution 15/02 (among others).

When redesigning the tools and processes to be used internally for the management of the overall information cycle, it was a natural choice to implement this set of functionalities as a remotely accessible system, that users from the data section of the Secretariat could access through their web browsers, without having to install any additional software or third-party component and without the need of physically be connected to the IOTC local area network.

[1] Fabio Fiorellato (fabio.fiorellato@iotc.org) , James Geehan (james.geehan@iotc.org) and Lucia Pierre (lucia.pierre@iotc.org)

This design choice, which is in line with recent trends in the development of collaborative and remote environments for data processing and manipulation, had also the benefit to provide out of the box a broad set of remote business services that could potentially be made available to users outside the Secretariat.

Business operations as REST services

The REST specification ([Thomas]) was the solution of choice to enable communication and data exchange to and from the remote business services, in order to provide a lightweight, text-based, robust and well understood protocol for information sharing across multiple formats.

REST-compliant web services, such as those developed for this purpose, enable clients to access and manipulate textual representations of web resources using a uniform and predefined set of stateless operations. In our case *web resources* are either data sets (identified by a given URI) or the results of operations on data sets themselves (filtering, processing, resource updates etc.).

In a REST web service, requests made to a resource's URI will elicit a response that may be in XML, HTML, JSON or some other defined format. The response may confirm that some alteration has been made to the stored resource, and it may provide hypertext links to other related resources or collections of resources.

All of the business services currently available within the new IOTC system are accepting parameters and returning resources either as XML or JSON. A limited number of services can also produce results as CSV files, for convenience's sake.

Most of the operation that external users are expected to perform on the available resources are either categorized as *data filtering* or *data processing* operations and in both cases these are not supposed to alter the state of a given resource: this means that unless under very specific circumstances, external users have *read-only* access to all the information stored within the IOTC database and the results of services invocations will be the actual data returned by the service and / or a meaningful HTTP/1.1 status code (`200 - OK`, `204 - No Content`, `404 - Not Found`, `403 - Access Forbidden` etc.).

The set of REST services available to end users is constantly expanding, so as to provide the required business logic to perform common management operations (mainly targeted at internal users interactively invoking the remote services through a dedicated user interface) and general purpose data extraction and processing tasks.

The overall rationale of this approach is that by means of the common HTTP *verbs* (`GET`, `PUT`, `POST`, `DELETE`) *service consumers* (interactive users or scripts and procedures in any programming language) could perform the required operations on resources identified by their URI.

Depending on the complexity of the task, business services expect input parameters to be provided in three ways:

- As *query parameters* (appended to the resource URI) – e.g.: `GET some/remote/resource?`**`name=foo`**
- As *path parameters* (constituting portion of the resource URI) – e.g.: `GET some/other/resource/named/`**`foo`**
- As an *additional payload* for the request itself – e.g. `POST yet/another/complex/resource` with a JSON / XML payload modelling the required inputs such as **`{ expired: false, providedBy: ["me", "myself", "I" ] }`**

The content of a code list, which under this paradigm is considered a *resource* just like an image displayed on a web page, will be identified by a unique URI such as:

`http://statistics.iotc.org/rest/services/reference/codelists/retrieve/Species`

Retrieving the content of a codelist / resource is as simple as issuing an HTTP `GET` request to URIs similar to the one above (in this case, it's just a matter of clicking on the link to get the XML representation of the current Species codelist content in your browser – assuming that you can authenticate your request with proper credentials).

HTTP verbs semantics

When HTTP/1.1 is used to access REST resources and operations (as it is most common, including in this case) the semantic of the HTTP verbs should represent the type of action to be performed, according to the following rules:

- `GET` – to perform *idempotent* operations on a resource or on a collection of resources (e.g. retrieve an occurrence of a resource or its entirety, filter the content of a resource etc.). Inputs are either provided as query parameters or as path parameters (see above);

- `PUT` – to create one or multiple occurrences of a resource. Inputs are either provided as query parameters, as path parameters or as an additional payload attached to the request (that might include a proper representation of the resource to be created, as a JSON or XML document). `PUT` operations are not *idempotent* as they will change the state of a given resource, e.g. adding a new occurrence;
- `POST` – usually, to update one or multiple occurrences of a resource or to perform *idempotent* operations on a resource that require complex inputs that cannot be provided as either *query* parameters or *path* parameters. POST operations are usually *not idempotent* (e.g. update an existing resource), except when they're used as per the second scenario, to provide an additional payload that due to HTTP/1.1 specifications *cannot* be submitted with a simple `GET` request;
- `DELETE` – to remove one or multiple occurrences of a resource. Inputs are either provided as query parameters, as path parameters or as an additional payload attached to the request (that might include a proper representation of the resource to be created, as a JSON or XML document). `DELETE` operations are not *idempotent* as they will change the state of a given resource, e.g. removing an existing occurrence.

Restricting user access to resources

In order to account for data confidentiality issues and at the same time limit the set of operations that external users can perform on the system, access to all of the available REST services is restricted to clients provided with either their own set of access credentials or a dedicate API key.

The difference between performing a full authentication as opposed to using an API key for direct invocation of any remote service is in the need (or not) to have a user session available. For all of the tasks that external users are expected to perform, an API key is more than adequate as no long-standing and *stateful* sequence of operations is supposed to be made available outside the Secretariat.

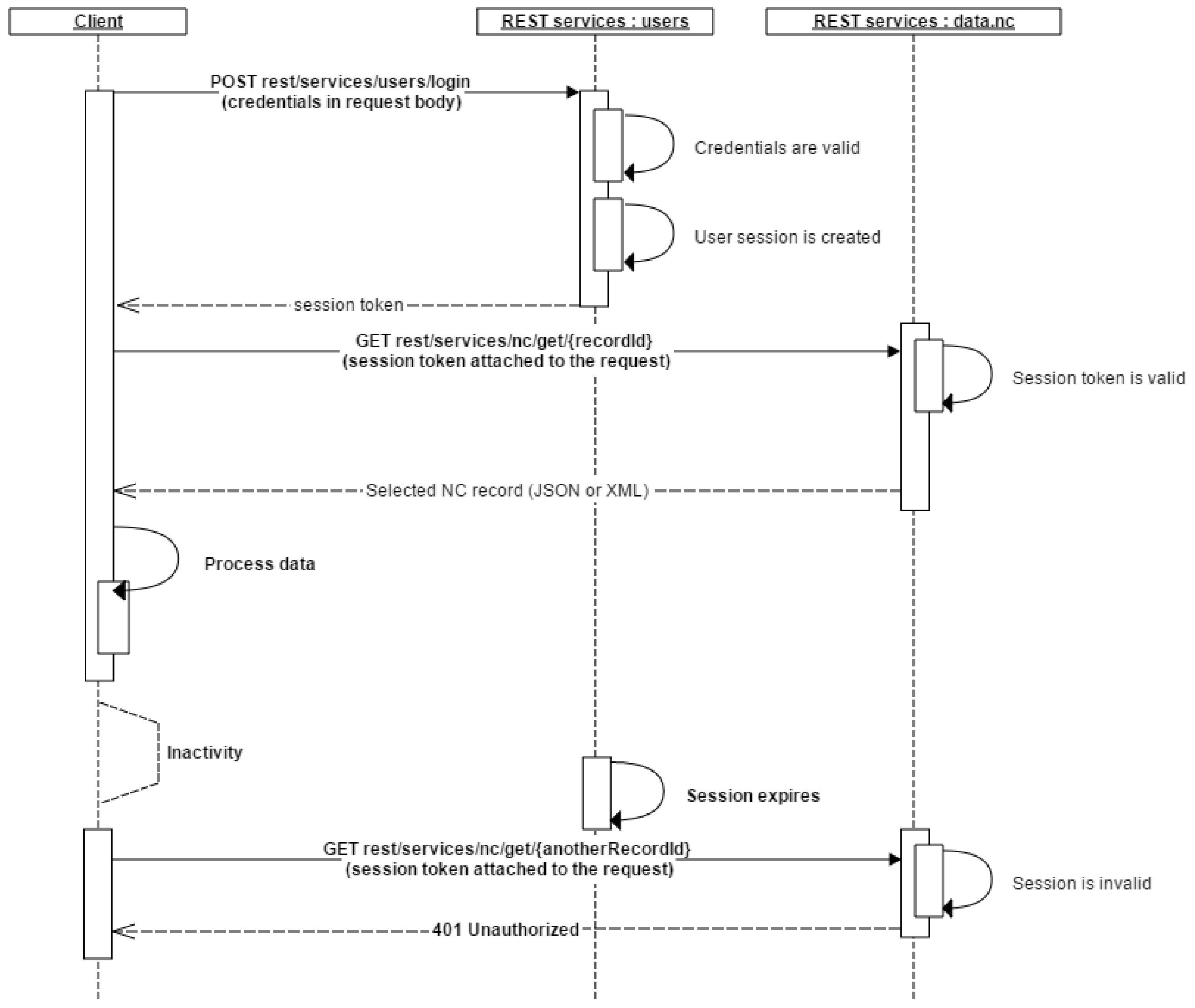

Client
REST services : users
REST services : data.nc
POST rest/services/users/login
(credentials in request body)
Credentials are valid
User session is created
session token
GET rest/services/nc/get/{recordId}
(session token attached to the request)
Session token is valid
Selected NC record (JSON or XML)
Process data
Inactivity
Session expires
GET rest/services/nc/get/{anotherRecordId}
(session token attached to the request)
Session is invalid
401 Unauthorized

Figure 1. Sequence diagram of possible interactions with REST services using a session token

Figure 1 shows the sequence of requests / response interactions that a client has to perform in order to authenticate onto the system and subsequently invoke any of the operations at his availability during the validity period of a user session.

The first step requires clients to authenticate on the system by providing their personal credentials to a dedicated *login* service. This operation results in the creation of a user session (if the login is successful) and in the provision of a *session token* to the invoker. This same token should be eventually used to perform any subsequent interaction.

User sessions have a timeout of 30 minutes, which is reset at each successful interaction: if no activity is performed during a time frame longer than the session timeout, the provided session token expires and cannot be reused. The REST services will reply with an HTTP `401 - Unauthorized` response code to requests providing an invalid or expired session token.

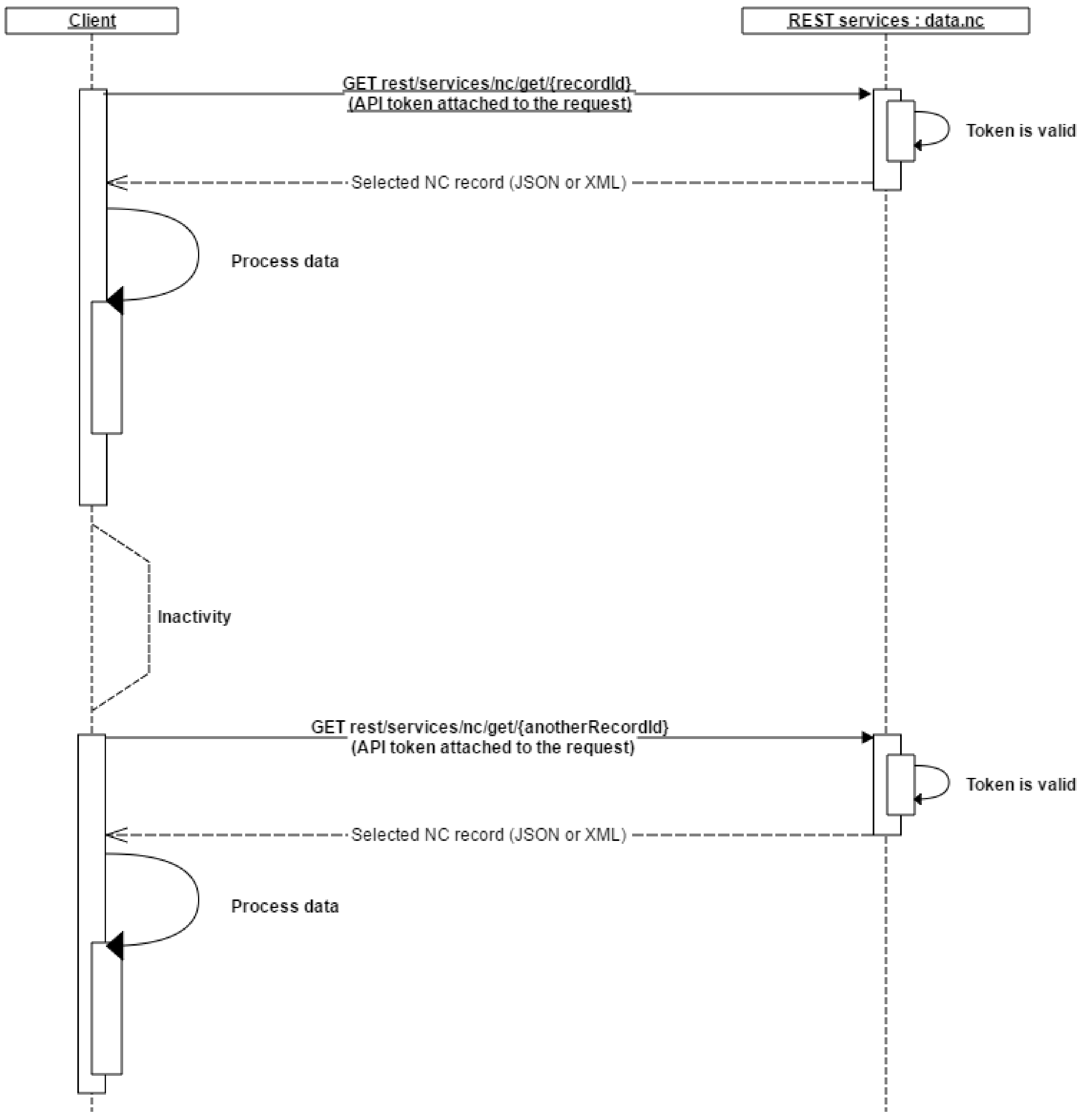

Client
REST services : data.nc
GET rest/services/nc/get/{recordId}
(API token attached to the request)
Token is valid
Selected NC record (JSON or XML)
Process data
Inactivity
GET rest/services/nc/get/{anotherRecordId}
(API token attached to the request)
Token is valid
Selected NC record (JSON or XML)
Process data

Figure 2. Sequence diagram of possible interactions with REST services using an API token

Figure 2 shows the sequence of request / response interactions that a client has to perform in order to invoke any of the operations at his availability by using a specific API key.

As opposed to the session-based interaction scenario, users with an API key at their availability do not need to authenticate onto the system. API keys could be used without restrictions in time (unless they are disabled for specific reasons) and they do not require nor imply the creation of a user session.

While this is generally not a problem with most operations, under some circumstances (e.g. the creation and manipulation of disaggregated nominal catches) a user session is required: in this scenario, users need to provide a valid session token for any interaction with the system.

Session tokens or API keys are associated to specific user roles and capabilities, and these are transparently checked by the system anytime an operation (including *idempotent* ones) is triggered.

When the provided session token or API key belongs to a user whose roles and capabilities are not perfectly matching those required by the target operation, the system will respond with an HTTP/1.1 status code `403 – Forbidden`.

Getting a proper set of credentials or an API key is not yet an automated process, as it is subject to approval by the system managers: for the time being, users are expected to issue this request through the dedicated statistics @iotc.org e-mail address, specifying their personal details, the types of operations they would like to perform on the data and the reason for their request.

Once acknowledged, proper credentials and / or API keys will be provided back to the requesting user.

REST services expects session tokens or API keys as part of each received requests in two possible and alternative ways (depending on the stateful nature of the operation and on end users' convenience):

- As *query parameters*: e.g.

```
GET /rest/services/data/nc/get/666? X-IOTC-STATS-API-key=<your API key>&anyOtherParameter=value&…
```

- As *HTTP headers* (with the same names as the expected query parameters)

The actual names of the session token or API key to be provided are:

- `X-IOTC-STATS-API-key` for the API key or
- `X-IOTC-STATS-User-Session-Token` for the session token

In any case, failing in providing either the session token or the API key during a request will result in the system replying with an HTTP/1.1 `400 – Bad request` response code.

Common operations available to external users

Reference data

Reference data consists of two different categories of data sets:

- *Codelists* – that represent the commonly used standard references attached to all other datasets (e.g. Species, Fleets, Fishing grounds etc.);
- *Other reference data* – that account for heterogeneous types of information such as relationships between codelists (Fishing grounds aggregations), configuration tables and other miscellaneous items.

Services available to end users in this category are:

- Listing all the available reference data;
- Describing an existing reference data type (attribute names, type, size and mandatory status for each field);
- Retrieving the content of a given reference data, either in its entirety or limited to a given number of items;
- Retrieving the content of a specific occurrence for a given reference data;
- Any other CRUD (*Create, Read, Update and Delete*) operation on the data – usually not available to external users.

Normally, end users do not need to explicitly access these types of services if not to retrieve the information that might be used – on the client side – to populate specific component of a user interface (e.g. a drop-down list) or similar.

Nominal catch

Nominal catch services can be used to perform the basic set of CRUD operations on the nominal catch dataset or process its content accordingly.

Services available to end users in this category is:

- Checking whether nominal catch records exist for a given strata (year / quarter / fleet / fishery / species / area);
- Counting the number of nominal catch records matching specific filtering criteria;
- Returning all nominal catch records matching specific filtering criteria, either in their entirety or limited to a given number of records;
- Downloading all nominal catch records that can be publicly disseminated in a CSV format;

- Producing nominal catch reports by species / fishery / area;
- Producing nominal catch summaries according to different filtering and grouping criteria;
- Performing the nominal catch disaggregation process on a subset of the available records;
- Finalizing results of the nominal catch disaggregation process;
- Any other CRUD (*Create, Read, Update and Delete*) operation on the data, including the bulk upload of nominal catch records – usually not available to external users.

Catch and effort

Catch and effort services include common operations that can be performed on both effort and catch records, including the usual basic set of CRUD operations. An additional set of dataset specific services allows end users to perform spatial reallocations of catch data and compute summaries for efforts, catches and (nominal) CPUEs.

Among the currently available services we have:

- Checking whether nominal effort / catch records exist for a given strata (year / month / fleet / fishery / area / species / unit);
- Returning all nominal effort / catch records matching specific filtering criteria, either in their entirety or limited to a given number of records;
- Downloading all catch and effort records that can be publicly disseminated in a CSV format;
- Producing effort / catch reports by species / fishery / area;
- Producing effort / catch / CPUE summaries according to different filtering and grouping criteria;
- Performing spatial reallocations of effort / catch / CPUE data;
- Producing spatial and temporal reallocations of catch data;
- Finalizing results of spatial and temporal catch data;
- Any other CRUD (*Create, Read, Update and Delete*) operation on the data, including the bulk upload of nominal catch records – usually not available to external users.

Size frequency

Size and frequency services include common operations that can be performed on both samples and size-distribution records, including the usual basic set of CRUD operations. An additional set of dataset specific services allows end users to perform spatial reallocations and conversion of size and frequency data and compute size-distribution summaries for samples and size-distribution records.

Among the currently available services we have:

- Checking whether size and frequency records exist for a given strata (year / month / fleet / fishery / area / species / unit);
- Returning all sample / size-distribution records matching specific filtering criteria, either in their entirety or limited to a given number of records;
- Downloading all size-distribution records (converted to the default measure units by species) that can be publicly disseminated in a CSV format;
- Producing size-distribution summaries by fleet / fishery / species / measure unit;
- Performing spatial reallocations of size-frequency data;
- Producing spatial and temporal reallocations of converted size-frequency data;
- Finalizing results of spatial and temporal reallocations of converted size-frequency data;

Any other CRUD (*Create, Read, Update and Delete*) operation on the data, including the bulk upload of nominal size-frequency records – usually not available to external users.

Other

A few other uncategorized services are currently available to end users as well. These include the following:

- Browsing fishing grounds geometries and retrieve their WKT definition
- Identifying fishing grounds that are only partially overlapping with the Indian Ocean area, and to a very limited extent (suited for identifying fishing grounds that are either misplaced or fully within the mainland)
- Reconciliation of raw, possibly misspelled data against common codelists (e.g. species names or scientific names, gear names etc.) – Under development

Invoking the remote services

While the REST specification (in terms of data exchange and operations' invocation protocol) is text-based and language agnostic, being largely based on the HTTP/1.1 specification and on some common format for structured data representation (JSON, XML), the way in which REST services could be effectively and programmatically invoked depends on the specific programming language adopted by end users.

We will now provide some non-exhaustive examples showing how a REST service request could be triggered by some of the most commonly used programming languages, as well as examples demonstrating how the produced results could be effectively used for further computations.

Within these examples, notwithstanding the possibility of providing data and accepting results either as JSON or XML, we will just focus on JSON data inputs and outputs. In any case, switching to an XML representation of these same set of information should be pretty straightforward in most languages.

Within an R script

Assuming that an API key is available for testing purposes (we will provide a temporary one during the meeting), in order to enable an R script to perform a REST call and process the returned results as JSON it is necessary to install the *httr* and *jsonlite* packages through the usual:

```
install.packages(”httr”)

install.packages(”jsonlite”)
```

Once these packages are made available to the R environment, a possible way to invoke any of the REST services is defined by the following steps (text colours are mapped to the corresponding section in the sample R script in Listing 1):

1. Create the JSON payload to provide input to the REST service. This might not be strictly needed (e.g. services invoked through GET verbs do not expect a payload) and at the time we write, this step can be easily performed by simple string concatenation;
2. Prepare the HTTP request targeting the chosen endpoint through the corresponding HTTP verb, adding all the required HTTP headers that account for:
   2.1. Specification of the accepted response content types (we'll assume to be accepting JSON documents as responses);
   2.2. Specification of the payload content type (if requested). We'll assume to be sending JSON documents as payload, in case;
   2.3. Any other authentication header (as per previous specifications: these might be either a session token or an API key header).
3. Perform the HTTP request;
4. Wait for the response and evaluate the response status, halting in case an HTTP error code is returned;
5. Transform the result from a JSON document to an R object
6. Process the result (as an R object)

An actual example of REST services invocation is the following

```
library(httr)
library(jsonlite)
...
payload = paste0(
        '{',
                '"yearFrom": 1975, ',
                '"yearTo": 2015, ',
                '"includePreliminary": true, ',
                '"includeConfidential": false, ',
                '"fleets": [], ',
                '"fisheries": ["PS", "LL"], ',
                '"species": ["ALB", "BET", "SKJ", "SWO", "YFT"] ',
        '}'
)

response <- POST("http://statistics.iotc.org/rest/services/data/ce/catch/reallocation/byArea",
            accept_json(),
            add_headers("Content-Type" = "application/json"),
            add_headers("X-IOTC-STATS-API-key" = "7b5147f8546c73437a066bb9e0d086f0"), body = payload)

stop_for_status(response)
data <- fromJSON(content(response, "text"))
...
```

Listing 1. Example of REST service invocation within an R script

The example in Listing 1 demonstrates how to create a JSON payload to provide filtering criteria to the catch-and-effort spatial reallocation service. The actual JSON payload resulting from the string concatenation will be the following:

```
{
      yearFrom: 1975,
      yearTo: 2015,
      includePreliminary: false,
      includeConfidential: false,
      fisheries: [ "PS", "LL" ],
      species: [ "ALB", "BET", "SKJ", "SWO", "YFT" ]
}
```

Listing 2. Example of JSON payload in its raw format

JSON documents, as is the case of this sample payload, do not require carriage returns or separating spaces between tokens, therefore this payload is supposed to be fully equivalent to:

```
{yearFrom:1975,yearTo:2015,includePreliminary:false,includeConfidential:false,fisheries:["PS","LL"],species:["ALB","BET","SKJ","SWO","YFT"]}
```

Listing 3. Example of equivalent JSON payload in its raw format (all spaces and indentation removed)

It's up to end users to decide which forms is best suited to their needs when producing the JSON payload for similar services: as long as the JSON syntax is respected, any choice is absolutely equivalent.

In this sample case, the payload represents the inputs required to configure the remote service execution: in particular, with the sample payload we are asking the service to limit its extent of application to catch-and-effort data reported between 1975 and 2015 (included) by PS or LL fisheries and related to any of the five major species (ALB, BET, SKJ, SWO and YFT). Additionally, we're also specifying that we do not want preliminary or confidential data as part of the result.

Each service has different requirements in terms of the input data that it's expected to accept: in a following section we will provide all the necessary information to access the formal specifications of all currently available services.

The second component within this sample listing is actually responsible to prepare the HTTP request (a POST request in this case) by providing the target service endpoint URL and explicitly specifying that the caller is accepting JSON documents as response ("`accept_json()`") while at the same time informing the remote services that the request contains a JSON payload ("`add_headers("Content-Type" = "application/json")`").

Access credentials are provided as the additional HTTP header "`X-IOTC-STATS-API-key`" whose value is set to a (supposedly) valid API key.

Finally, the body of the request is set to the actual JSON payload that was prepared before the invocation: the following lines show how the R script can be instructed to wait for the server response (and possibly halt in case an HTTP error is returned – "`stop_for_status(response)`") and transform the returned JSON response into an R object ("`data <- fromJSON(content(response, "text"))`").

At this stage, if the service invocation went well, invoking users will have the returned data available for further processing within the R script in the `data` object.

The exact syntax to invoke the REST service and process the result is library-specific (in this case, the actual syntax expected by the `httr` and `jsonlite` libraries is used). Different libraries might have different requirements and procedures to provide the same functionalities.

Additionally, the example is showing the possible *worst-case scenario*, in which a REST call should be configured by providing an explicit payload – with all the implications in terms of string concatenation that we explained earlier.

Simpler services (mostly those performed through GET requests or by the adoption of *query parameters* and / or *path parameters* as inputs) do not require this initial processing step.

Within Javascript code

Writing Javascript client code to access REST services is somehow simpler than in the case of R. This because Javascript already provides plenty of pre-packaged libraries that greatly simplify the work of dealing with HTTP asynchronous requests in a standardized and convenient form. Moreover, both JSON and XML could be natively processed by Javascript, thus giving direct access to the results without additional burden.

One of the main issues encountered by Javascript developers when designing dynamic web pages whose content is retrieved from a remote endpoint is the *Same Origin Policy* (SOP) constraint that in principle prevents processing asynchronous remote requests directed to hosts other than the source of the web page triggering the request.

For this reason, in order to extend the level of access for the provided REST services, these are hosted on a server that enforces the so-called *Cross Origin Resource Sharing* (CORS) recommendation – a W3C consortium standard technology that enables remote requests to be safely processed on the client side, regardless of the targeted remote endpoint.

In practical terms, this has no direct implication on developers and consumers of the REST services, if not for the possibility to embed functionalities provided by the remote services in whatever web site they are managing.

In the following Javascript example, we assume that a recent implementation of the jQuery library is loaded within the browser executing the test code: jQuery is a lightweight and feature-rich cross-browser library that standardizes most of the common tasks required by developers, including convenient management procedures for REST services invocation.

Its last version can be downloaded from https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js or directly embedded in a web page as a reference to remote Javascript source file.

The configuration steps are very similar to those already detailed in the R script example (text colors are mapped to the corresponding section in the sample script in Listing 4):

1. Create the JSON payload to provide input to the REST service. This might not be strictly needed (e.g. services invoked through GET verbs do not expect a payload) and in any case it can be done in a straightforward way thanks to Javascript native JSON support;
2. Prepare the HTTP request targeting the chosen endpoint through the corresponding HTTP verb, adding all the required HTTP headers that account for:
   2.1. Specification of the accepted response content types (we'll assume to be accepting JSON documents as responses);
   2.2. Specification of the payload content type (if requested). We'll assume to be sending JSON documents as payload, in case;
   2.3. Any other authentication header (as per previous specifications: these might be either a session token or an API key header).
3. Perform the HTTP request;
4. Wait for the response and properly handle success or error HTTP codes;
5. In case of success, process the JSON results

```
<script type="text/javascript">
        …
        var payload = {
                yearFrom: 1975,
                yearTo: 2015,
                includePreliminary: false,
                includeConfidential: false,
                fisheries: [ "PS", "LL" ],
                species: [ "ALB", "BET", "SKJ", "SWO", "YFT" ]
        };
        $.ajax({
                type: "POST",
                url: "http://statistics.iotc.org/rest/services/data/ce/catch/reallocation/byArea",
                data: JSON.stringify(payload),
                beforeSend: function(request) {
                   request.setRequestHeader("X-IOTC-STATS-API-key", "7b5147f8546c73437a066bb9e0d086f0");
                },
                error: function(jqXHR, textStatus, errorThrown) {
                        //Handle the error...
                },
                success: function(result, status, jqXHR) {
                        process(result);
                },
                contentType: "application/json",
                dataType: "json"
        });
        …
</script>
```

Listing 4. Example of REST service invocation within Javascript

Understanding the way in which jQuery can be integrated within Javascript code and most of all the proper way to process the results and handle potential errors is out of the scope of this document (any Javascript developer could easily find training resources online).

Nevertheless, one clear advantage of Javascript over R is its native support for JSON documents manipulation that results in little-to-no additional code to produce JSON payload and process JSON results.

Within Java code

Java, as opposed to both R and Javascript, is a compiled and typed language. For this reason, Java clients need to have direct references to all the *data transfer objects* that model either a JSON payload or a JSON response.

As part of the effort in revising and redesigning the core information systems within the Secretariat, a pre-compiled set of libraries will be made available in the near future so that any Java developer wishing to get access to the REST services within his / her code could simply plug-in these pre-packaged libraries and integrate them within its own business logic.

Examples

To further complement the discussions about the advantages and peculiarities of REST services made available to end users, we will showcase a few examples (either in R or in Javascript) demonstrating how through a few line of codes these services could be successfully plugged in existing scripts and procedures to provide data visualization and processing functionalities by directly accessing the information available at the Secretariat.

The source code for all these examples is available for download under the WPDCS12 web pages (within the “Dataset” category).

For users to be able to execute the R scripts, a proper R environment with all the dependencies specified within the scripts must be available. The Javascript example, conversely, can be executed by simply opening the sample web page within any recent browser.

As a general remark, users are advised that Internet connection speed from the client location to the Secretariat network might affect response times.

R-examples

The first, basic examples of REST services invocation through R are built through the same techniques demonstrated in the previous sections.

String concatenation is used to prepare the JSON payload and results of the POST request to the nominal catch summary endpoint (`data/nc/summary/results`) are then plotted as bar charts using different criteria.

Plotting nominal catches as a bar chart

The service input payload is configured in order to request summary nominal catch values for Yellowfin tuna in the years from 1970 to 2015 by year and fishery type. Results are plotted using years as category for the X-axis and fishery type (either **AR**tisanal, **S**emi **I**ndustrial or **IN**dustrial) to break down catches by year (Figure 3).

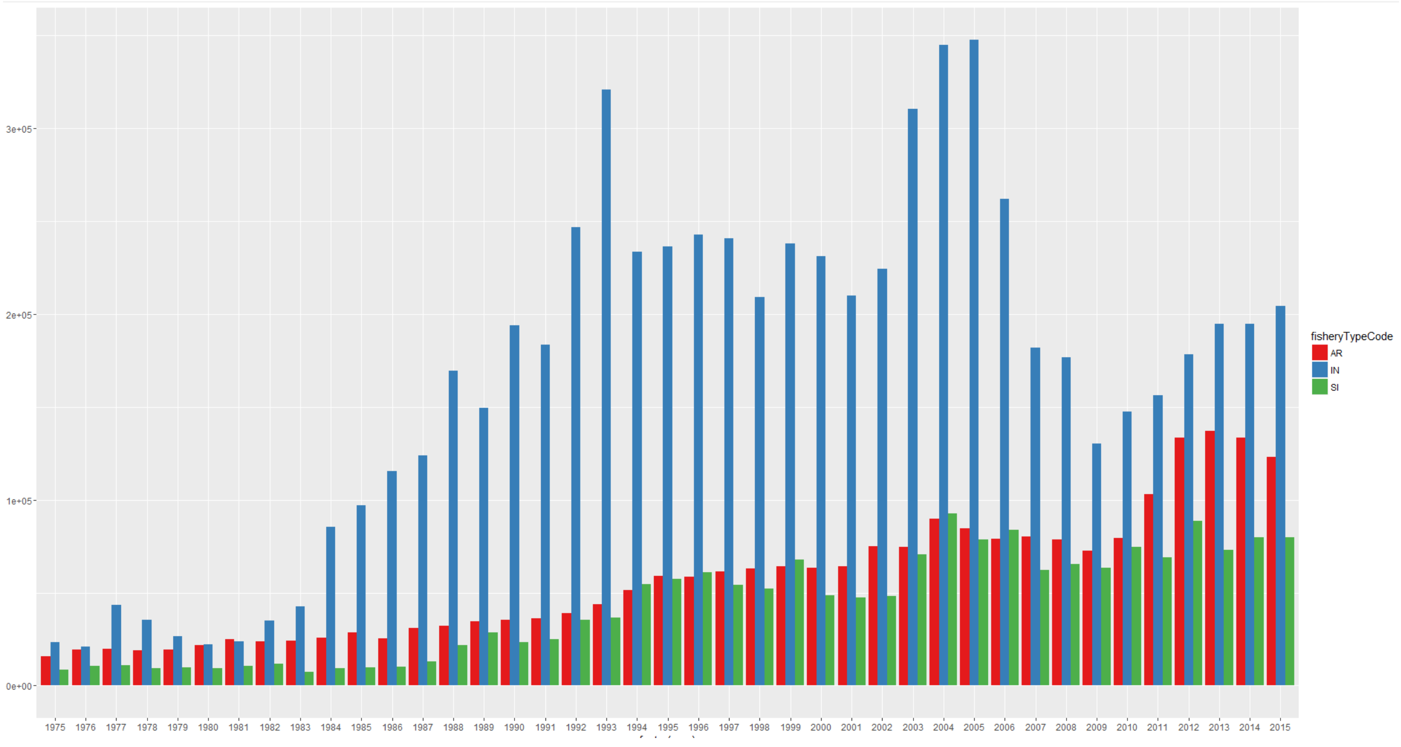


Figure 3. YFT nominal catches (1970 – 2015) by year and fishery type

Raw JSON payload:

```
{
	"fromYear": 1975,
	"toYear": 2015,
	"includePreliminary": false,
	"includeConfidential": false,
	"species": ["YFT"],
	"iotcSpecies": true,
	"groupByFisheryType": true,
	"groupByYear": true
}
```

Source code: (`IOTC-2106-WPDCS12-DT01/NC_summary.r`)

```
library(ggplot2)
library(httr)
library(jsonlite)

endpoint <- "http://statistics.iotc.org/rest/services/data/nc/summary/results"
api_key <- "f6ec6dd32798ea96ae552bf34201ff17"

payload = paste0(
	'{',
		'"fromYear": 1975, ',
		'"toYear": 2015, ',
              '"includePreliminary": false, ',
              '"includeConfidential": false, ',
              '"species": ["YFT"], ',
              '"iotcSpecies": true, ',
              '"groupByFisheryType": true, ',
              '"groupByYear": true',
       '}'
)

response <- POST(endpoint,
                 accept_json(),
                 add_headers("Content-Type" = "application/json"),
                 add_headers("X-IOTC-STATS-API-key" = api_key),
                 body = payload)

stop_for_status(response)
data <- fromJSON(content(response, "text"))
head(data)

ggplot(data,
       aes(factor(year), catches, fill = fisheryTypeCode)) +
       geom_bar(stat="identity", position = "dodge") +
       scale_fill_brewer(palette="Set1")
```

Listing 5. R script to plot nominal catches as a bar chart

Service API: http://statistics.iotc.org/api-browser/#!/data.nc.summary/summary

Plotting nominal catches as a stacked bar chart

The service input payload is configured in order to request summary nominal catch values for all IOTC species in the years from 1970 to 2015 by year and fishery group. Results are plotted using years as category for the X-axis and fishery group to group catches by year (Figure 4).

Source R script: `NC_Summary2.r`

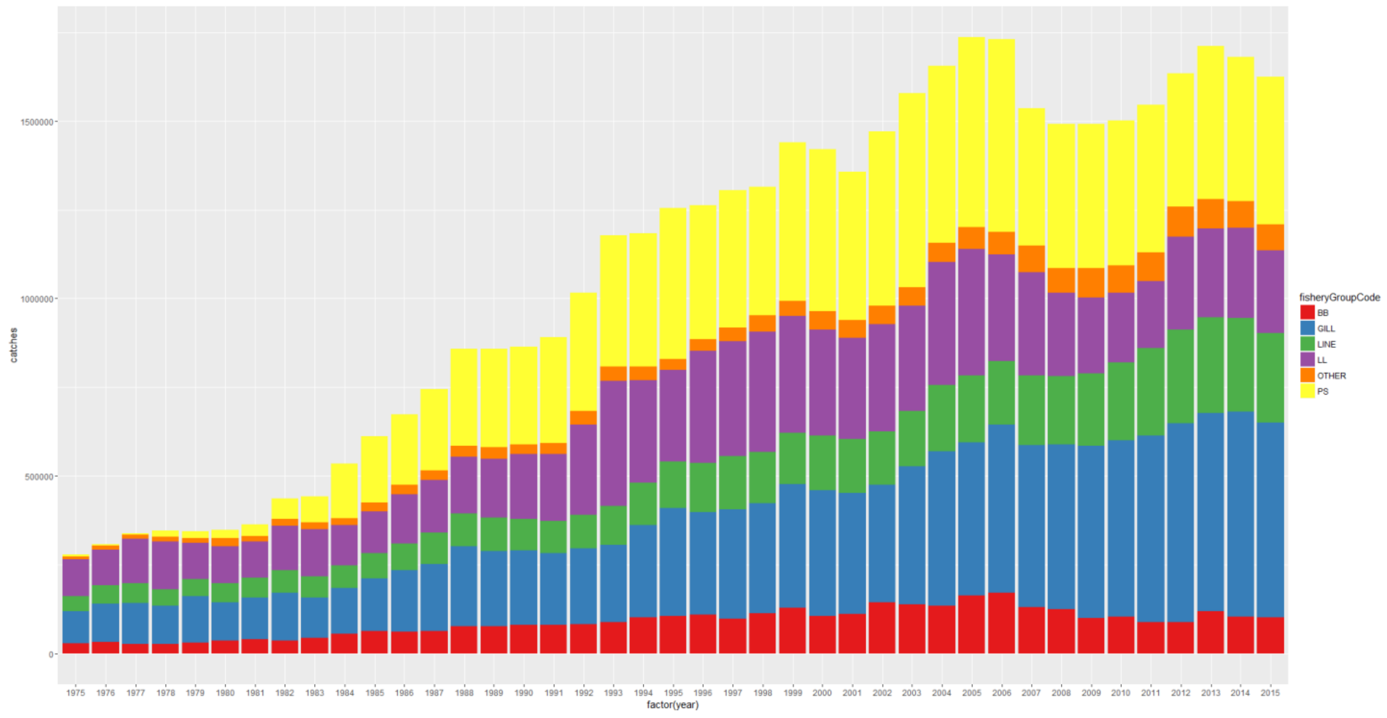


Figure 4. All IOTC species nominal catches (1970 – 2015) grouped by year and fishery group

Raw JSON payload:

```
{
       "fromYear": 1975,
       "toYear": 2015,
       "includePreliminary": false,
       "includeConfidential": false,
       "iotcSpecies": true,
       "groupByFisheryGroup": true,
       "groupByYear": true
}
```

Source code: (`IOTC-2106-WPDCS12-DT01/NC_summary2.r`)

```
library(ggplot2)
library(httr)
library(jsonlite)

endpoint <- "http://statistics.iotc.org/rest/services/data/nc/summary/results"
api_key <- "f6ec6dd32798ea96ae552bf34201ff17"

payload = paste0(
       '{',
              '"fromYear": 1975, ',
              '"toYear": 2015, ',
              '"includePreliminary": false, ',
              '"includeConfidential": false, ',
              '"iotcSpecies": true, ',
              '"groupByFisheryGroup": true, ',
              '"groupByYear": true ',
       '}'
)

response <- POST(endpoint,
                 accept_json(),
                 add_headers("Content-Type" = "application/json"),
                 add_headers("X-IOTC-STATS-API-key" = api_key),
                 body = payload)

stop_for_status(response)
data <- fromJSON(content(response, "text"))
head(data)

ggplot(data,
       aes(factor(year), catches, fill = fisheryGroupCode)) +
       geom_bar(stat="identity", position = "stack") +
       scale_fill_brewer(palette="Set1")
```

Listing 6. R script to plot nominal catches as a stacked bar chart

Service API: `http://statistics.iotc.org/api-browser/#!/data.nc.summary/summary`

Plotting reallocated catches on a map

An example of the added value provided by current REST services is the possibility to spatially reallocate catches (from the catch-and-effort dataset) to display catch values over grids of a given resolution that not necessarily might coincide with the spatial resolution of the original dataset.

The exploited REST service (`data/ce/catch/reallocation/byArea`) is currently filtering the dataset based on the provided criteria and further reallocating identified catch quantities over grids of the target resolution, based on the overlapping area detected between the original fishing grounds (for which data is available) and the target grids (Figure 5).

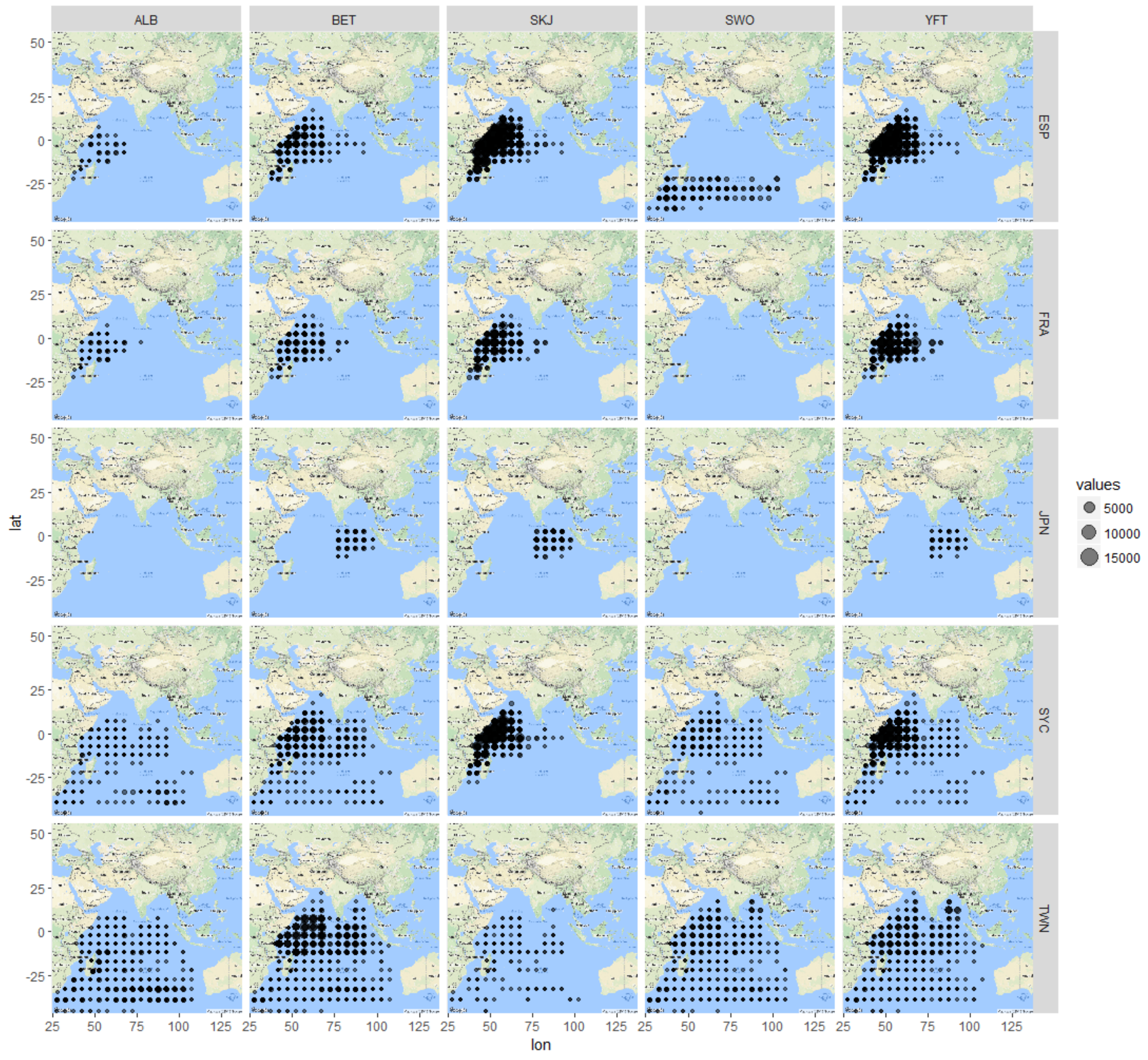


Figure 5. Catches in MT from the main 5 species, plotted over 5x5 degrees grid and faceted by species and fleet, for the years between 2010 and 2015 and ESP, FRA, JPN, SYC and TWN longliners or purse-seines

Raw JSON payload:

```
{
       "fromYear": 2010,
       "toYear": 2015,
       "resolution": 2,
       "includePreliminary": false,
       "includeConfidential": false,
       "fleets": ["TWN", "JPN", "ESP", "FRA", "SYC"],
       "fisheries": ["PSFS", "PSLS", "LL", "LLSW"],
       "groupByFleet": true,
       "groupBySpecies": true,
       "species": ["ALB", "BET", "SKJ", "SWO", "YFT"],
       "unit": "MT",
       "values": true
}
```

Source code: (`IOTC-2106-WPDCS12-DT01/CE_reallocation.r`)

```
library(ggmap)
library(httr)
library(jsonlite)

endpoint <- "http://statistics.iotc.org/rest/services/data/ce/catch/reallocation/byArea"
api_key <- "f6ec6dd32798ea96ae552bf34201ff17"

payload = paste0(
       '{',
              '"fromYear": 2010, ',
              '"toYear": 2015, ',
              '"resolution": 2, ',
              '"includePreliminary": false, ',
              '"includeConfidential": false, ',
              '"fleets": ["TWN", "JPN", "ESP", "FRA", "SYC"], ',
              '"fisheries": ["PSFS", "PSLS", "LL", "LLSW"], ',
              '"groupByFleet": true, ',
              '"groupBySpecies": true, ',
              '"species": ["ALB", "BET", "SKJ", "SWO", "YFT"], ',
              '"unit": "MT", ',
              '"values": true',
       '}'
)

response <- POST(endpoint,
                 accept_json(),
                 add_headers("Content-Type" = "application/json"),
                 add_headers("X-IOTC-STATS-API-key" = api_key),
                 body = payload)

stop_for_status(response)
data <- fromJSON(content(response, "text"))
head(data)

map <- get_map(location = 'Sri Lanka', zoom = 3)
ggmap(map) + geom_point(aes(x = lon, y = lat, size = values), data = data, alpha = .5) +
facet_grid(fleetCode ~ speciesCode)
```

Listing 7. R script to plot reallocated catches on a faceted map

Service API: http://statistics.iotc.org/api-browser/#!/data.ce.catch.reallocation/byArea

Plotting reallocated catches on a map (restricted to a specific EEZ)

A further demonstration of the added value provided by current REST services is built on top of the previous example (plotting reallocated catches on a map) that is extended in order to increase the spatial resolution (1x1 degrees grids) and eventually limit the identification of filtered catch records to those reported in the EEZ of Seychelles only.

The same REST service (`data/ce/catch/reallocation/byArea`) as in the previous example is exploited, with the additional specification of an increased grid resolution and the specification of the area (Seychelles EEZ) to which results should be limited (Figure 6).

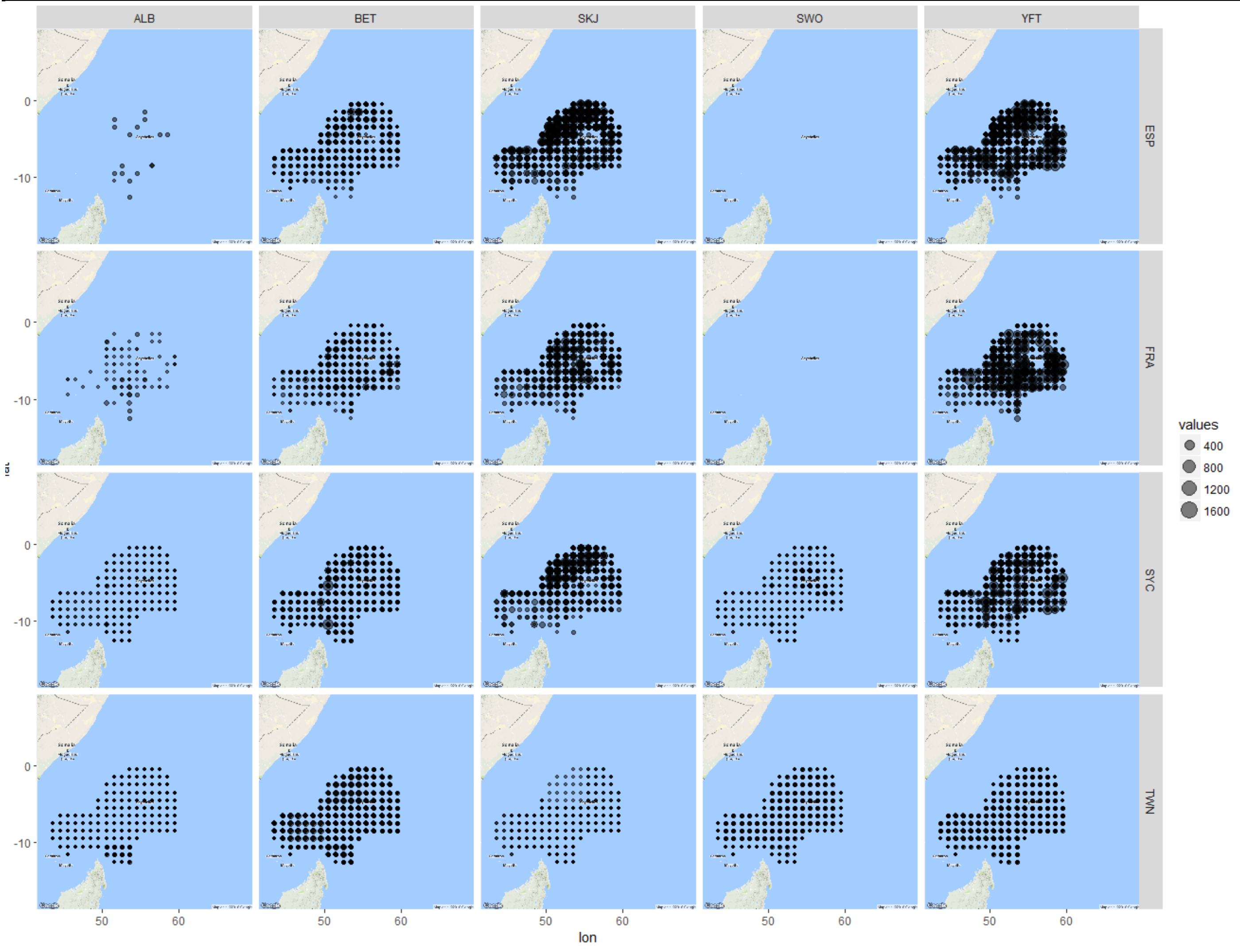


Figure 6. Catches in MT from the main 5 species, plotted over 1x1 degrees grid and faceted by species and fleet, for all years between 2010 and 2015 limited to ESP, FRA, JPN, SYC and TWN longliners or purse-seines. Data is restricted to catches recorded in the EEZ of Seychelles only

Raw JSON payload:

```
{
       "fromYear": 2010,
       "toYear": 2015,
       "resolution": 1,
       "includePreliminary": false,
       "includeConfidential": false,
       "fleets": ["TWN", "JPN", "ESP", "FRA", "SYC"],
       "fisheries": ["PSFS", "PSLS", "LL", "LLSW"],
       "groupByFleet": true,
       "groupBySpecies": true,
       "species": ["ALB", "BET", "SKJ", "SWO", "YFT"],
       "unit": "MT",
       "intersectionAreas": [ "IRSYCEZ" ],
       "values": true
}
```

Source code: (`IOTC-2106-WPDCS12-DT01/CE_reallocation2.r`)

```
library(ggmap)
library(httr)
library(jsonlite)

endpoint <- "http://statistics.iotc.org/rest/services/data/ce/catch/reallocation/byArea"
api_key <- "f6ec6dd32798ea96ae552bf34201ff17"
```

```
payload = paste0(
       '{',
              '"fromYear": 2010, ',
              '"toYear": 2015, ',
              '"resolution": 1, ',
              '"includePreliminary": false, ',
              '"includeConfidential": false, ',
              '"fleets": ["TWN", "JPN", "ESP", "FRA", "SYC"], ',
              '"fisheries": ["PSFS", "PSLS", "LL", "LLSW"], ',
              '"groupByFleet": true, ',
              '"groupBySpecies": true, ',
              '"species": ["ALB", "BET", "SKJ", "SWO", "YFT"], ',
              '"unit": "MT", ',
              '"intersectionAreas": [ "IRSYCEZ" ], ',
              '"values": true',
       '}'
)

response <- POST(endpoint,
                 accept_json(),
                 add_headers("Content-Type" = "application/json"),
                 add_headers("X-IOTC-STATS-API-key" = api_key),
                 body = payload)

stop_for_status(response)
data <- fromJSON(content(response, "text"))
head(data)

map <- get_map(location = 'Seychelles', zoom = 3)
ggmap(map) + geom_point(aes(x = lon, y = lat, size = values), data = data, alpha = .5) +
facet_grid(fleetCode ~ speciesCode)
```

Listing 8. R script to plot reallocated catches (limited to a specific area) on a faceted map

Service API: http://statistics.iotc.org/api-browser/#!/data.ce.catch.reallocation/byArea

Javascript examples

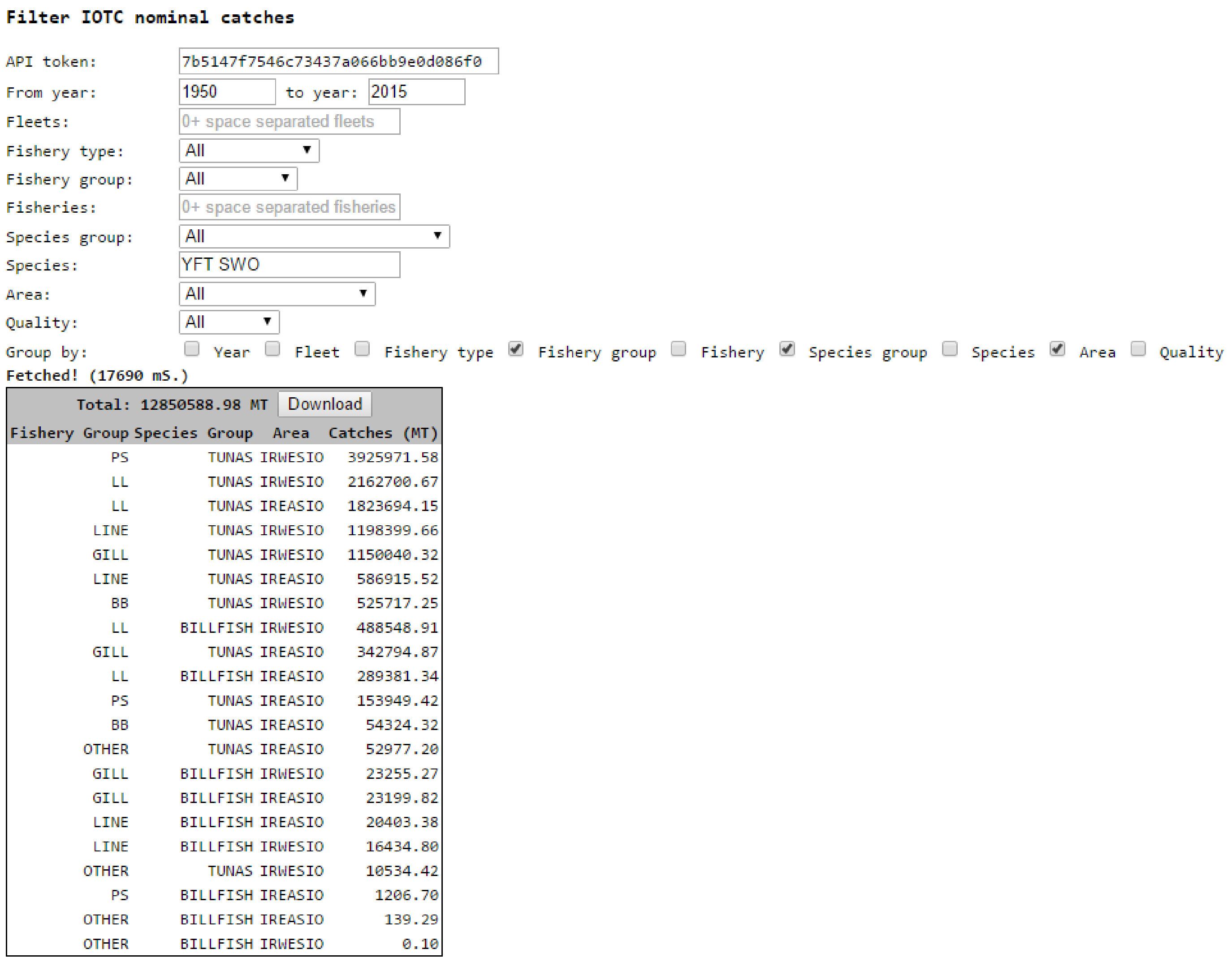
Accessing the REST services within Javascript in a CORS-compliant environment, enables users to design complex interfaces for data retrieval and manipulation with just a few lines of HTML and Javascript code.

As opposed to R examples, this Javascript demo does not require any specific software to be installed on the client machine: a modern, W3C compliant browser (Google Chrome, Mozilla Firefox, Apple Safari or Microsoft Edge) and a working Internet connection is all that is required to perform the test.

Dynamically filter nominal catches with different level of groupings

This example expands the same REST service as shown in the first R script (`data/nc/summary/results`) by providing interactive tools for the configuration of the JSON payload used to drive the service execution. In this particular case, results returned as a JSON object are used to build a dynamic table reflecting the filtered content and the level of grouping set by the user.

Similar tables, with minimal configuration capabilities, could be easily embedded within third-party web pages and therefore provide a *live* account of the status of the data available at the Secretariat.



Filter IOTC nominal catches

API token:	7b5147f7546c73437a066bb9e0d086f0
From year:	1950 to year: 2015
Fleets:	0+ space separated fleets
Fishery type:	All
Fishery group:	All
Fisheries:	0+ space separated fisheries
Species group:	All
Species:	YFT SWO
Area:	All
Quality:	All
Group by:	☐ Year ☐ Fleet ☐ Fishery type ☑ Fishery group ☐ Fishery ☑ Species group ☐ Species ☑ Area ☐ Quality

Fetched! (17690 mS.)

Total: 12850588.98 MT Download

Fishery Group	Species Group	Area	Catches (MT)
PS	TUNAS	IRWESIO	3925971.58
LL	TUNAS	IRWESIO	2162700.67
LL	TUNAS	IREASIO	1823694.15
LINE	TUNAS	IRWESIO	1198399.66
GILL	TUNAS	IRWESIO	1150040.32
LINE	TUNAS	IREASIO	586915.52
BB	TUNAS	IRWESIO	525717.25
LL	BILLFISH	IRWESIO	488548.91
GILL	TUNAS	IREASIO	342794.87
LL	BILLFISH	IREASIO	289381.34
PS	TUNAS	IREASIO	153949.42
BB	TUNAS	IREASIO	54324.32
OTHER	TUNAS	IREASIO	52977.20
GILL	BILLFISH	IRWESIO	23255.27
GILL	BILLFISH	IREASIO	23199.82
LINE	BILLFISH	IREASIO	20403.38
LINE	BILLFISH	IRWESIO	16434.80
OTHER	TUNAS	IRWESIO	10534.42
PS	BILLFISH	IREASIO	1206.70
OTHER	BILLFISH	IREASIO	139.29
OTHER	BILLFISH	IRWESIO	0.10

Figure 7. Using Javascript to produce a dynamic web page to let users filter and group nominal catch records through multiple criteria (results shown for YFT and SWO catches grouped by fishery group, species group and area)

The source code for this example is available as `IOTC-2106-WPDCS12-DT02/NC_query.html`

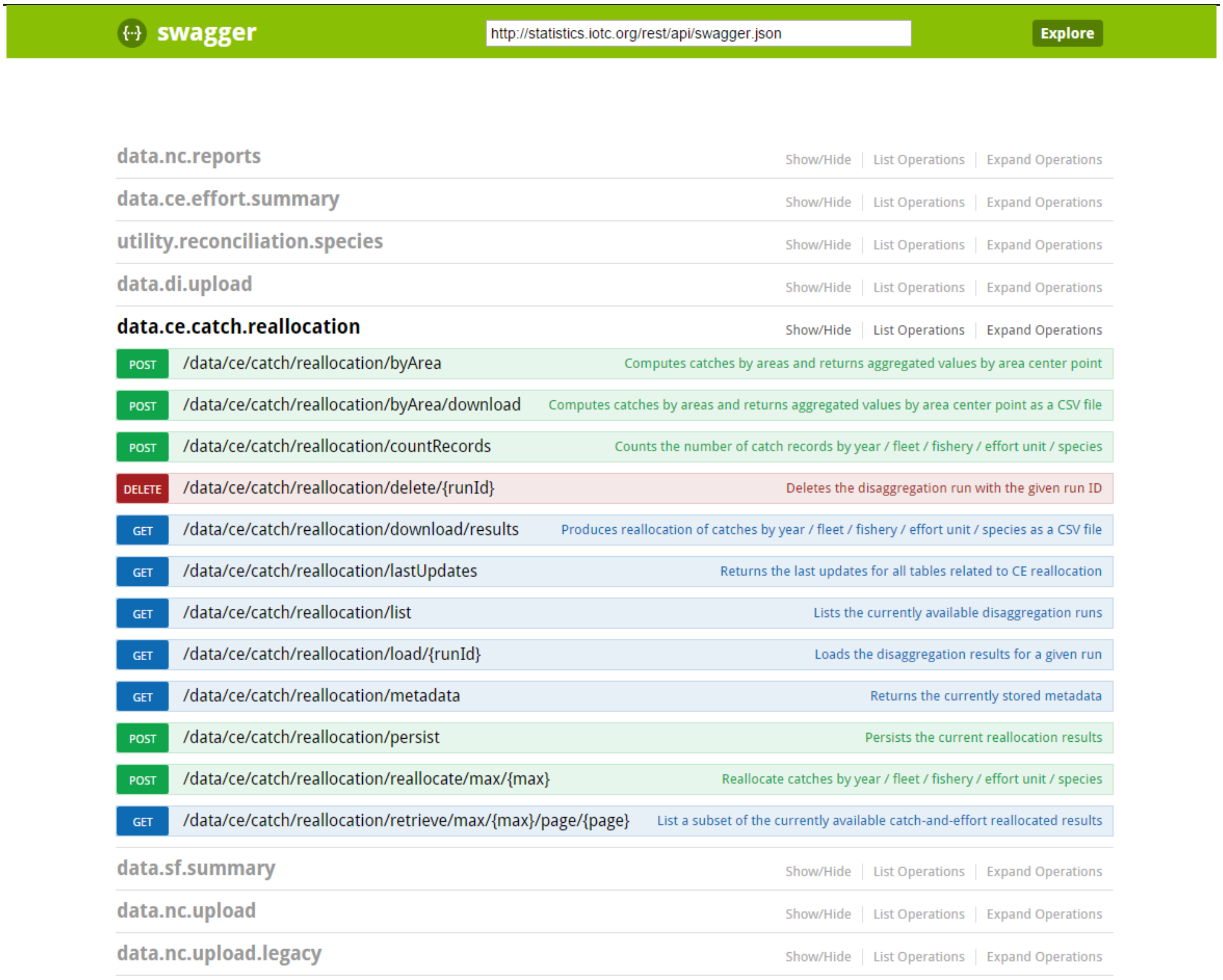
Documentation

Detailed and formal descriptions of each service currently exposed by the system – in terms of resource endpoints, required HTTP verbs, input parameters (if any), response object, response code and access limitations – is automatically generated on the basis of the current service specifications and available through the following public URL:

`http://statistics.iotc.org/api-browser`

The services' API specification is accessible to all users, including non-authenticated ones, and is available as a dynamic web page (powered by *Swagger*) through which users can browse the actual specifications and at the same time invoke any of the services should they have valid user credentials or API keys available.

Although from the end users' point of view this kind API specification is the only information required in order to be able to correctly invoke the remote services and understand their behaviour, a separate user manual would be provided in the near future.


swagger
http://statistics.iotc.org/rest/api/swagger.json
Explore
Show/Hide | List Operations | Expand Operations

data.nc.reports

data.ce.effort.summary

utility.reconciliation.species

data.di.upload

data.ce.catch.reallocation

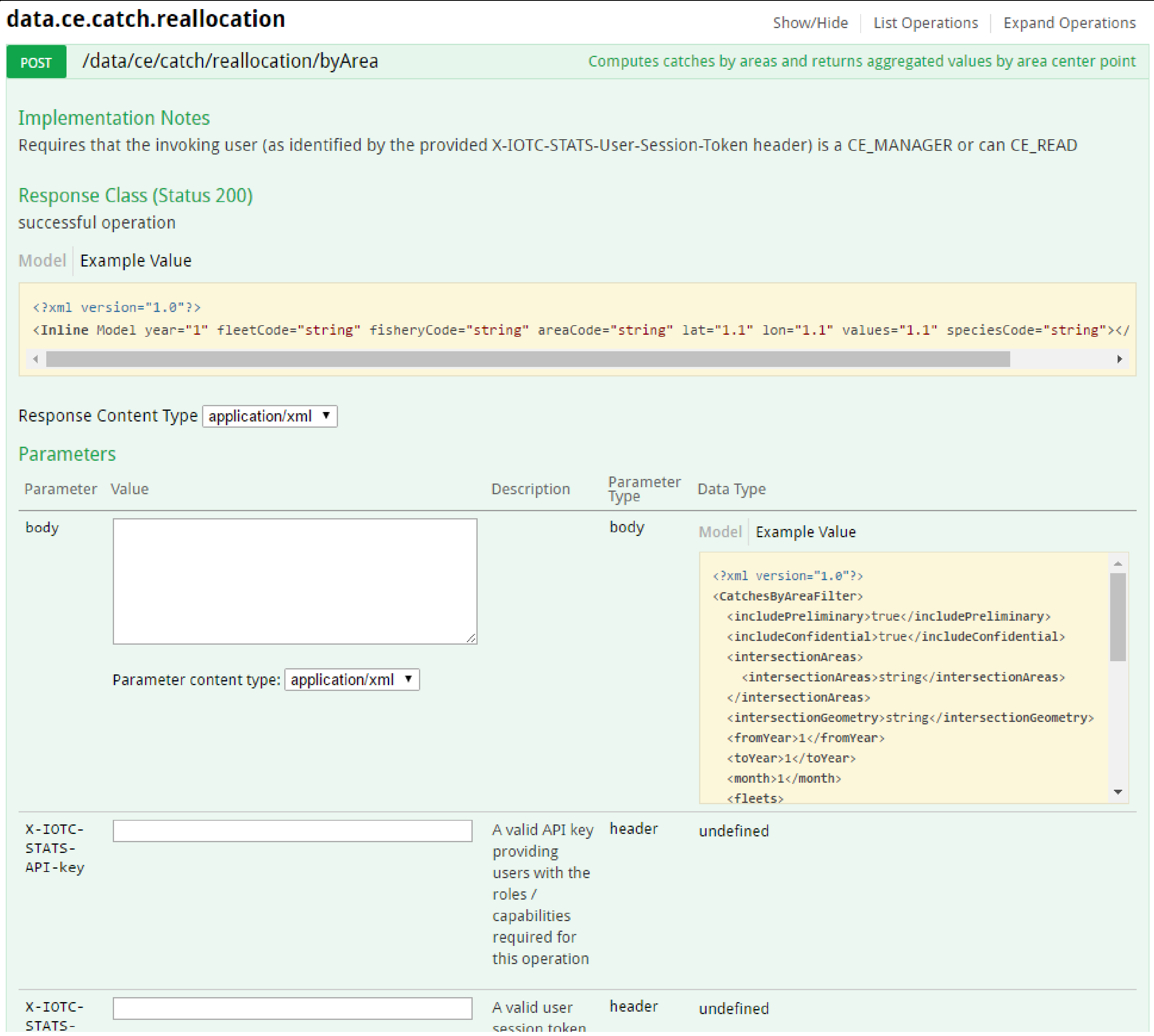
POST	/data/ce/catch/reallocation/byArea	Computes catches by areas and returns aggregated values by area center point
POST	/data/ce/catch/reallocation/byArea/download	Computes catches by areas and returns aggregated values by area center point as a CSV file
POST	/data/ce/catch/reallocation/countRecords	Counts the number of catch records by year / fleet / fishery / effort unit / species
DELETE	/data/ce/catch/reallocation/delete/{runId}	Deletes the disaggregation run with the given run ID
GET	/data/ce/catch/reallocation/download/results	Produces reallocation of catches by year / fleet / fishery / effort unit / species as a CSV file
GET	/data/ce/catch/reallocation/lastUpdates	Returns the last updates for all tables related to CE reallocation
GET	/data/ce/catch/reallocation/list	Lists the currently available disaggregation runs
GET	/data/ce/catch/reallocation/load/{runId}	Loads the disaggregation results for a given run
GET	/data/ce/catch/reallocation/metadata	Returns the currently stored metadata
POST	/data/ce/catch/reallocation/persist	Persists the current reallocation results
POST	/data/ce/catch/reallocation/reallocate/max/{max}	Reallocate catches by year / fleet / fishery / effort unit / species
GET	/data/ce/catch/reallocation/retrieve/max/{max}/page/{page}	List a subset of the currently available catch-and-effort reallocated results

data.sf.summary

data.nc.upload

data.nc.upload.legacy

Figure 8. Example of online documentation showing all the available REST services (and the expected HTTP verbs) for the catch-and-effort catch reallocation features



data.ce.catch.reallocation

/data/ce/catch/reallocation/byArea

Computes catches by areas and returns aggregated values by area center point

Implementation Notes

Requires that the invoking user (as identified by the provided X-IOTC-STATS-User-Session-Token header) is a CE_MANAGER or can CE_READ

Response Class (Status 200)

successful operation

```
<?xml version="1.0"?>
<Inline Model year="1" fleetCode="string" fisheryCode="string" areaCode="string" lat="1.1" lon="1.1" values="1.1" speciesCode="string"></
```

Response Content Type: application/xml

Parameters

Parameter	Value	Description	Parameter Type	Data Type
body	Parameter content type: application/xml		body	Model Example Value (see below)
X-IOTC-STATS-API-key		A valid API key providing users with the roles / capabilities required for this operation	header	undefined
X-IOTC-STATS-		A valid user session token	header	undefined

```
<?xml version="1.0"?>
<CatchesByAreaFilter>
  <includePreliminary>true</includePreliminary>
  <includeConfidential>true</includeConfidential>
  <intersectionAreas>
    <intersectionAreas>string</intersectionAreas>
  </intersectionAreas>
  <intersectionGeometry>string</intersectionGeometry>
  <fromYear>1</fromYear>
  <toYear>1</toYear>
  <month>1</month>
  <fleets>
```

Figure 9. Example of online documentation showing the required inputs, produced outputs and overall specifications of the REST service exploited by the last two R examples provided.

Future developments

- Improvements in the REST services documentation: the goal is to ensure that the *Swagger* generated API documentation is fully aligned with the services specifications and informative enough to enable clients understand how to prepare payloads for service invocation and handle the results;
- Provisions of pre-packaged client libraries (R, Javascript, Python etc.) for easier and more convenient access to the remote services. The goal is to provide end-users with dedicated libraries providing access (in any of the languages of choice) to the full set of functionalities exposed by the REST services in a convenient way (e.g. removing the need to prepare the JSON payload as a tedious exercise of string concatenation in R). The revised WPDCS program of work lists funding available for this purpose, although the activity still need to be endorsed by the SC;
- Addition of new data processing functionalities: this evolution might come either as a consequence of new business case identified during the development of the Integrated IOTC database management tool or by direct communication with end users requesting specific (and not yet implemented) functionalities;

- Provisions of code-snippets for the incorporation of charts and tables within third party websites and applications: this is in line with common approaches aimed at providing users with pre-packaged widgets that can easily be dropped in existing HTML pages (with few lines of code and with minimal configuration) and seamlessly integrate with the REST services for the retrieval and display – mostly as charts or tables – of relevant statistical information extracted from the IOTC database.

References

- IOTC-2016-WPDCS12-25_Rev1: *Improving the core IOTC data management processes* – IOTC Secretariat (2016), 12th Session of the Working Party on Data Collection and Statistics, Victoria, Seychelles
- REpresentational State Transfer (REST) – R. Thomas (2000), *Architectural Styles and the Design of Network-based Software Architectures* – Chapter 5 (Ph.D.). University of California, Irvine http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- URI – *Uniform Resource Identifier*: https://en.wikipedia.org/wiki/Uniform_Resource_Identifier
- HTTP/1.1 Status Code Definition: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
- XML – *eXtensible Markup Language*: https://en.wikipedia.org/wiki/XML
- JSON – *JavaScript Object Notation*: https://en.wikipedia.org/wiki/JSON
- SOP – *Same Origin Policy*: https://en.wikipedia.org/wiki/Same-origin_policy
- CORS – *Cross-Origin Resource Sharing*: https://en.wikipedia.org/wiki/Cross-origin_resource_sharing
- API – *Application Programming Interface*: https://en.wikipedia.org/wiki/Application_programming_interface