ENM TUTORIALS

JAQPOT QUATTRO
API TUTORIAL
Nano-QSAR
Modelling infrastructure

RELEASE DATE:	01/06/16
USE:	How to use all available functionalities of the JaqPot Quattro web application
VERSION:	V.1.0
MAIN AUTHOR:	Georgios Drakakis
PARTNER:	NTUA
CONTACT DETAILS:	hsarimv@central.ntua.gr
AUTHORS:	G. Drakakis, C. Chomenidis, G. Tsiliki, E. Anagnostopoulou, P. Doganis, H. Sarimveis
LICENCE:	CC-BY 4.0

TABLE OF CONTENTS

1. INTRODUCTION

This document provides a tutorial for the application program interfaces (APIs) available in the Jaqpot Quattro modelling infrastructure. The list of functionalities has been made available via the API documentation framework Swagger at http://test.jaqpot.org:8080/jaqpot/swagger/. At this location, users may create datasets containing nanoparticles and properties, apply PMML transformations, create machine learning models and several other functionalities described in detail later in this document.

2. GENERAL

The main menu contains all available APIs available within Jaqpot Quattro, as shown in Figure 1. Each API supports one or more HTTP methods for RESTful services, including GET, POST, PUT, PATCH and DELETE. By clicking on one of the listed APIs, a menu appears listing the allowed HTTP methods and their necessary parameters. All methods and parameters will be discussed in full in the following sections.

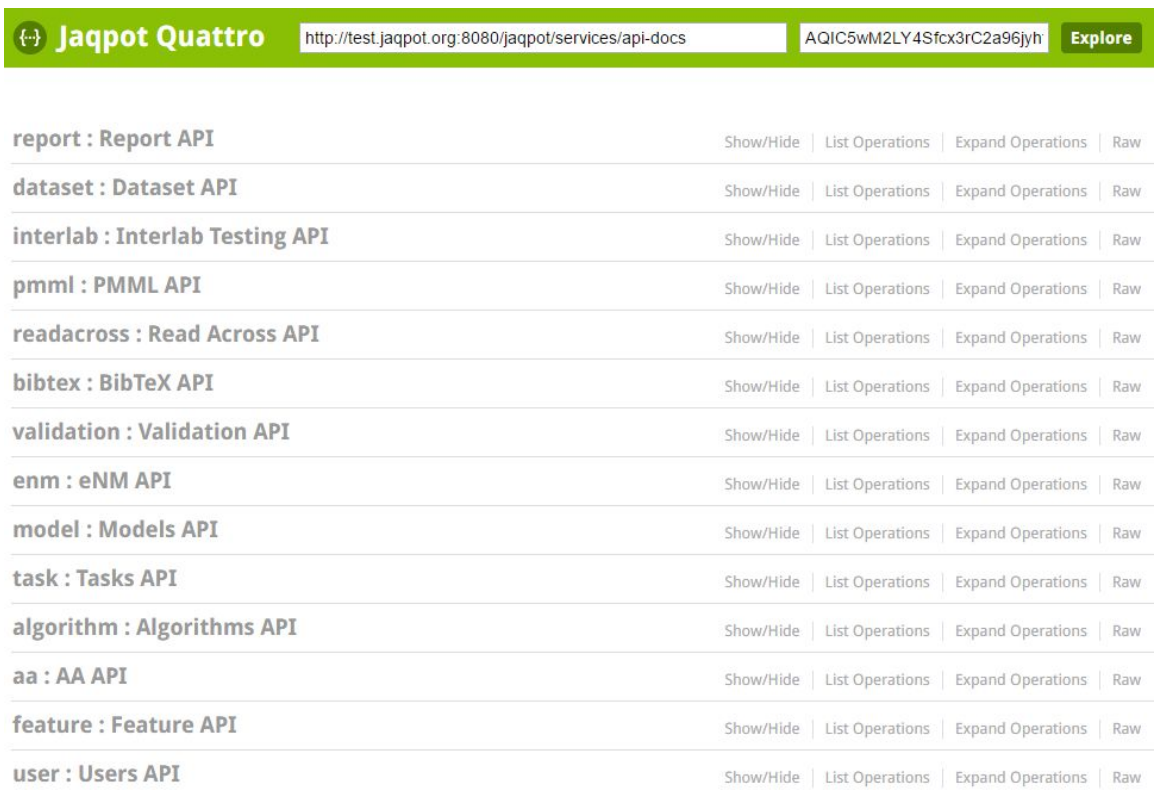

Figure 1: Main menu of APIs available at http://test.jaqpot.org:8080/jaqpot/swagger/

3. REPORT

This API retrieves any Jaqpot outcome visible to the user in report format. This may be a validation report (cross, external) containing performance metrics, or inter-laboratory testing outcomes with action/warning signals for labs estimated to have bias in their measurement procedures. In Figure 2 it can be seen by clicking on the *Report API* that the available HTTP methods are GET, GET by ID and DELETE by ID.

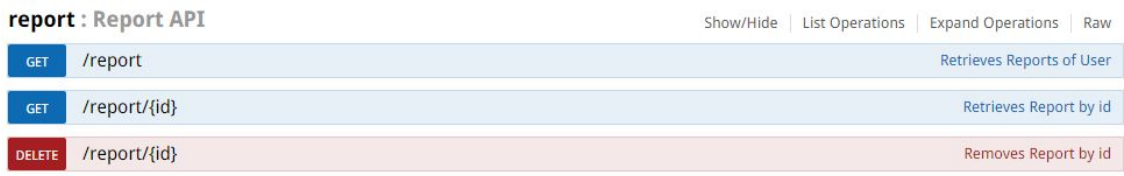

Figure 2: Report API retrieves or deletes report by ID and fetches all visible reports, such as validation, read across or inter-laboratory testing reports.

3.1 GET REPORT BY ID

This API returns a specific report by its ID. Should a user run a task which returns a report ID, by inserting it into the *ID* field and clicking on *Try it out*, he/she would receive the response in JSON format, subject to the authorisation token (*subjectid*) being valid. Shown in Figure 3.

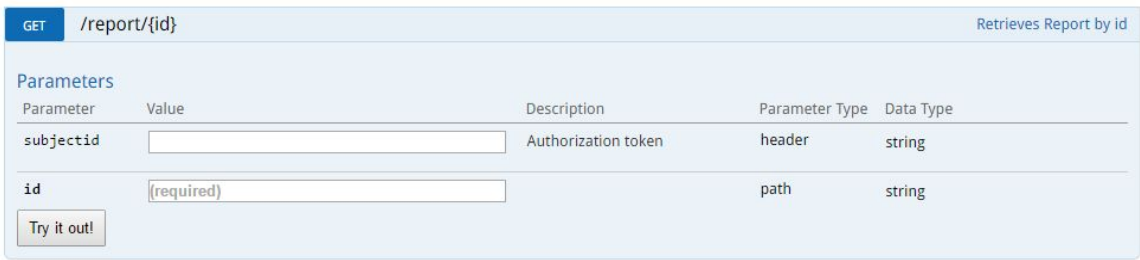

Figure 3: Screenshot of GET Report by ID API.

3.2 DELETE REPORT BY ID

Similar to above, this API deletes a specific report by its ID subject to permissions of the user's authorization token. Shown in Figure 4.

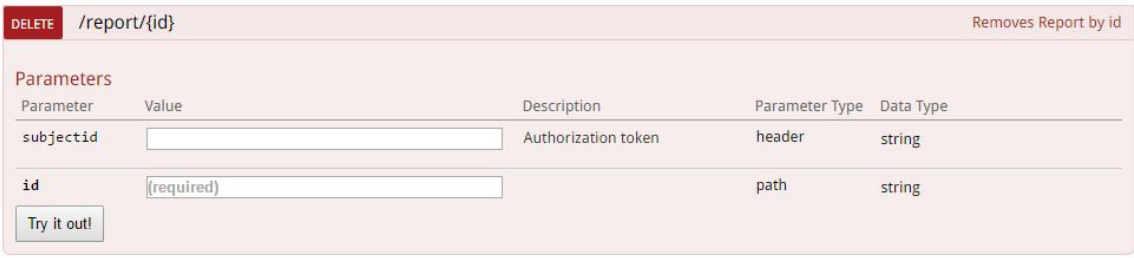

Figure 4: Screenshot of DELETE Report by ID API.

3.3 GET REPORT

This API returns all reports visible to the user in JSON format. Parameters *start* and *max* determine the number of reports to be fetched based on their unique internal ID assigned to them upon creation. Screenshot in Figure 5.

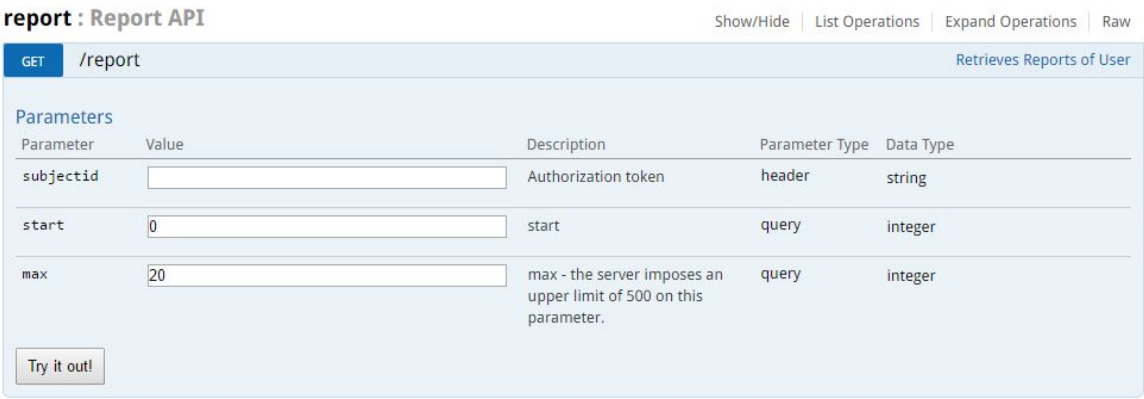

Figure 5: Screenshot of GET all visible reports.

4. DATASET

The available HTTP methods for the Dataset API can be seen below in Figure 6. These options can be individually expanded and called by clicking on the desired method, as shown later in this section.

Figure 6: Available HTTP methods for Dataset API

4.1 GET DATASET BY ID

There are three separate options to retrieve a dataset by ID, namely **/dataset/{id}**, **/dataset/{id}/features** and **/dataset/{id}/meta**. All three require the dataset ID in the path parameter field {id}. **/dataset/{id}/features** returns information on the nanoparticle properties, such as name (i.e. Zeta Potential), units etc. **/dataset/{id}/meta** shows all meta-information such as the dataset owner. These are shown in Figures 7 and 8 respectively.

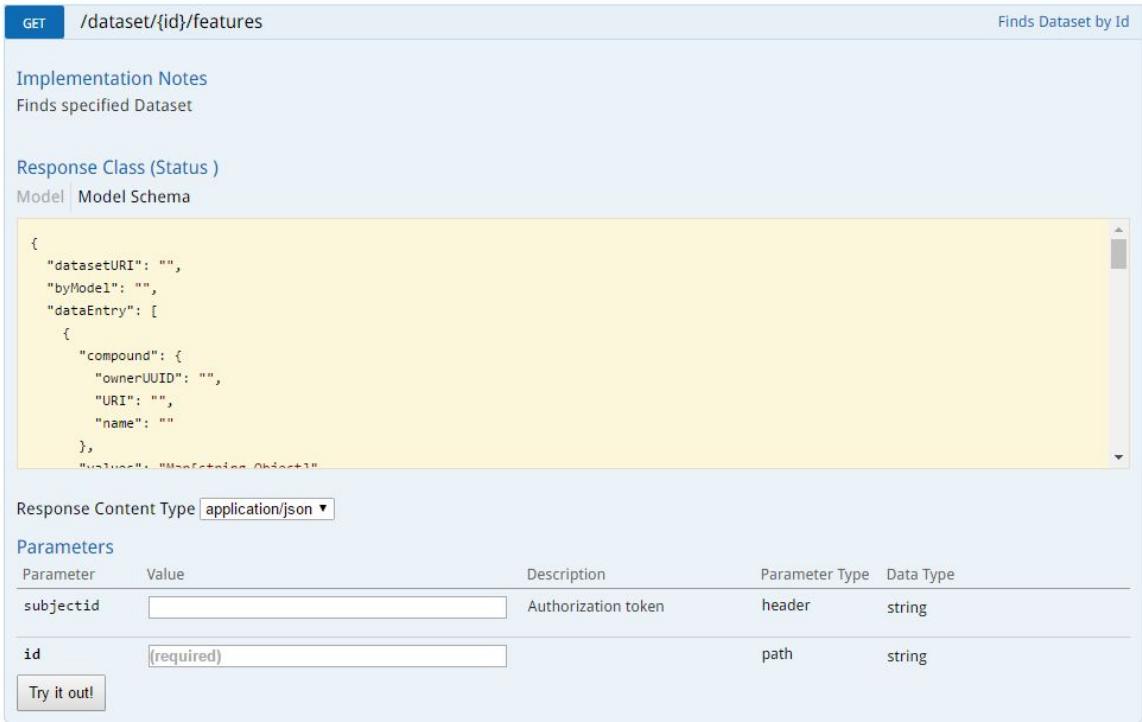

Figure 7: Find dataset features by dataset ID

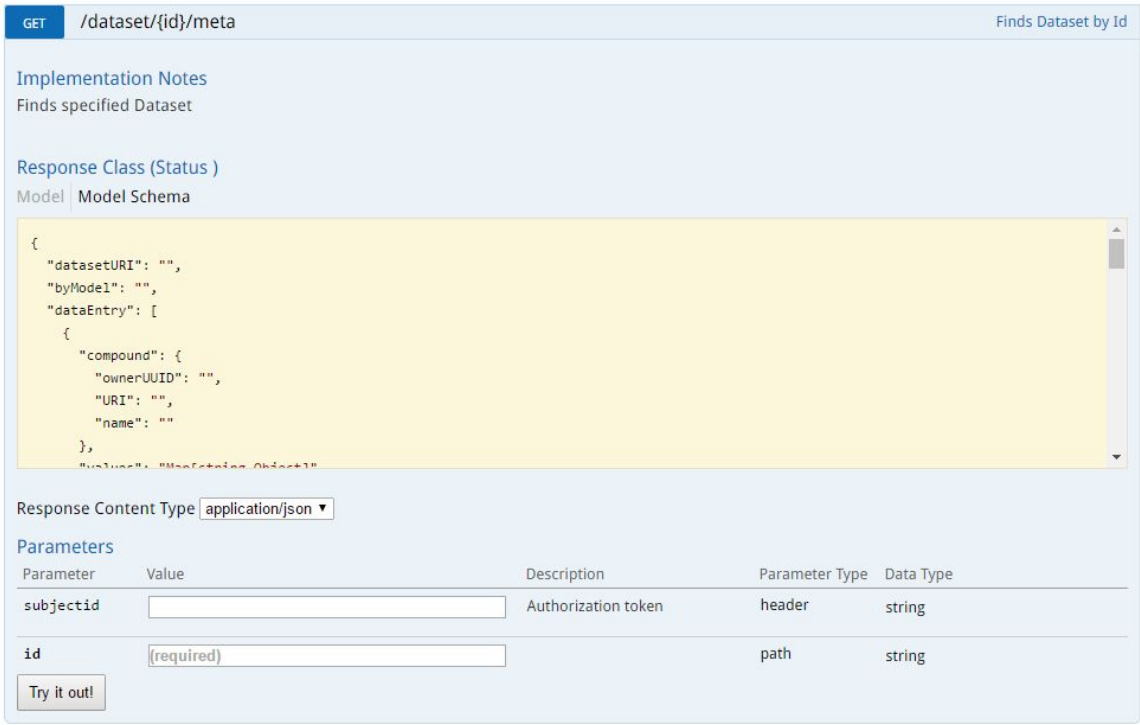

Figure 8: Find dataset meta information by dataset ID

However, **/dataset/{id}** will return the full body of the dataset in addition to meta and property information, but it can be optionally parameterised further by selecting a range of rows (instances) or columns (features/attributes) to be included for paging purposes, and also whether a dataset should be stratified or randomised for cross-validation, and which is the target class for applying modelling techniques after the retrieval/modification of the dataset. The available options are shown in Figure 9.

The query parameters in more detail:

rowStart : The index number of the row from which to start.

rowMax : The number of rows to be retrieved, starting from rowStart.

colStart : The index number of the column from which to start.

colMax : The number of columns to be retrieved, starting from colStart.

stratify : Has 2 options, 'random' and 'normal'. Random will return the dataset with a randomised order of rows based on **seed**. Normal will return the dataset in a stratified order based on **folds** and **target_feature**. What will actually happen internally by this option is that the dataset rows will be sorted ascending on **target_feature**. Then, row buckets will be created each one containing instances with values taken from the full spectrum of **target_feature.** Those buckets will be concatenated together and retrieved again as a single dataset, or as different pages of it.

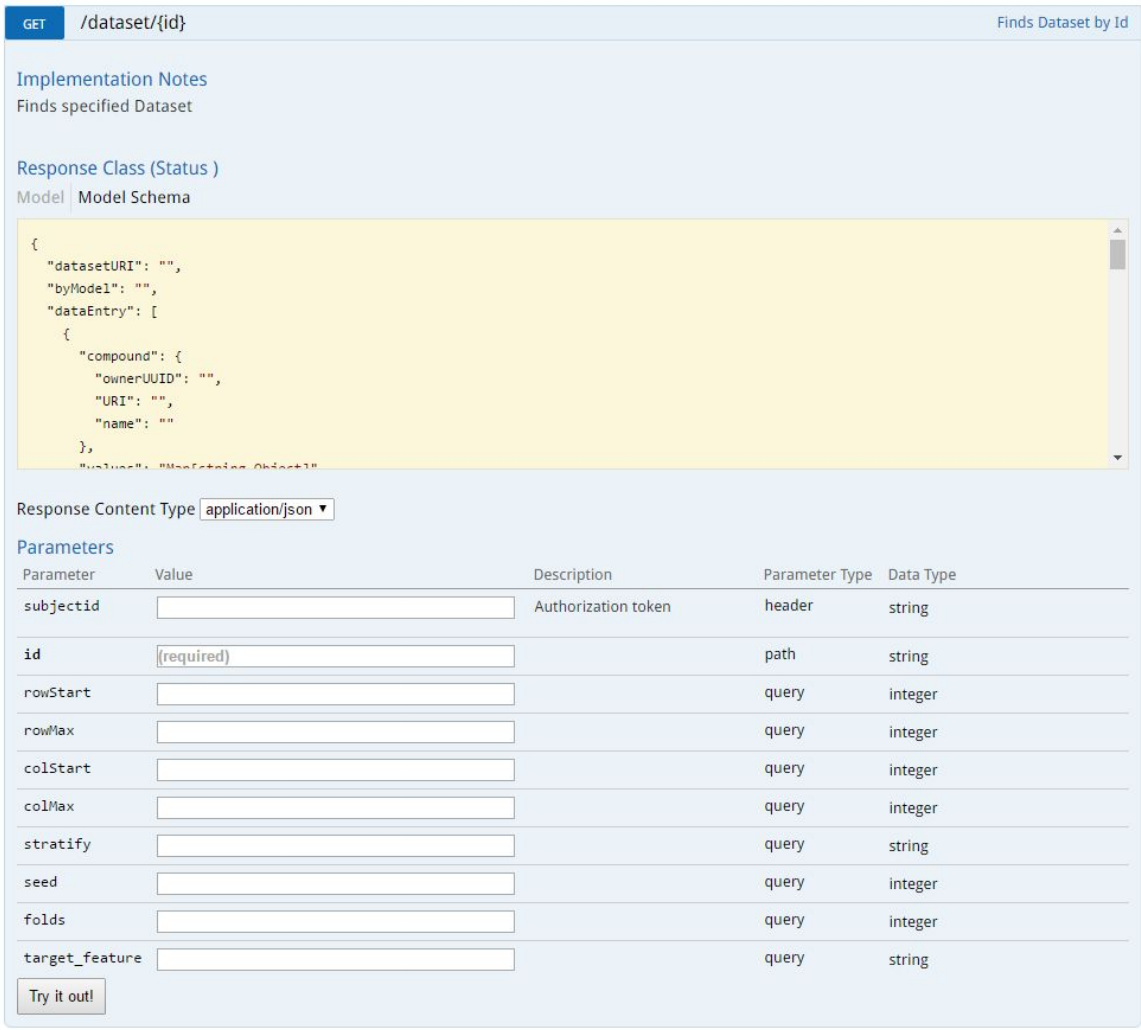

Figure 9: Options available under GET dataset by ID

4.2 DELETE DATASET BY ID

Similar to GET dataset by ID, should a dataset ID be known to the user (and the user has the appropriate permissions on the dataset; *subjectid*), he/she can paste it into the *ID* field and click on *Try it out* thus deleting the dataset. This can be seen in the screenshot shown in Figure 10.

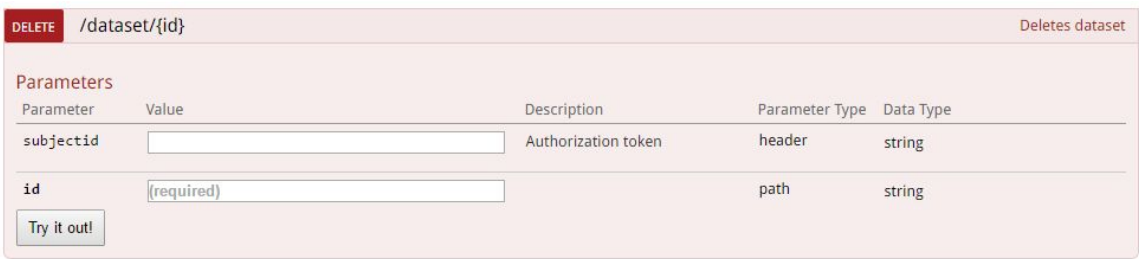

Figure 10: Screenshot of DELETE dataset by ID

4.3 POST DATASET

This option allows the user to create a new dataset in the database. This needs to be in JSON format. In the *body* section there is a template indicating the necessary fields to be filled in. This is shown in Figure 11. Multiple entries for nanoparticles (keyword *compound*) and variables (*entries*) are allowed. All parameters need to be appropriately filled in, such as number of rows and columns (instances and variables). This process is usually called once the JSON file has been created internally by retrieving data stored in the Ambit database.

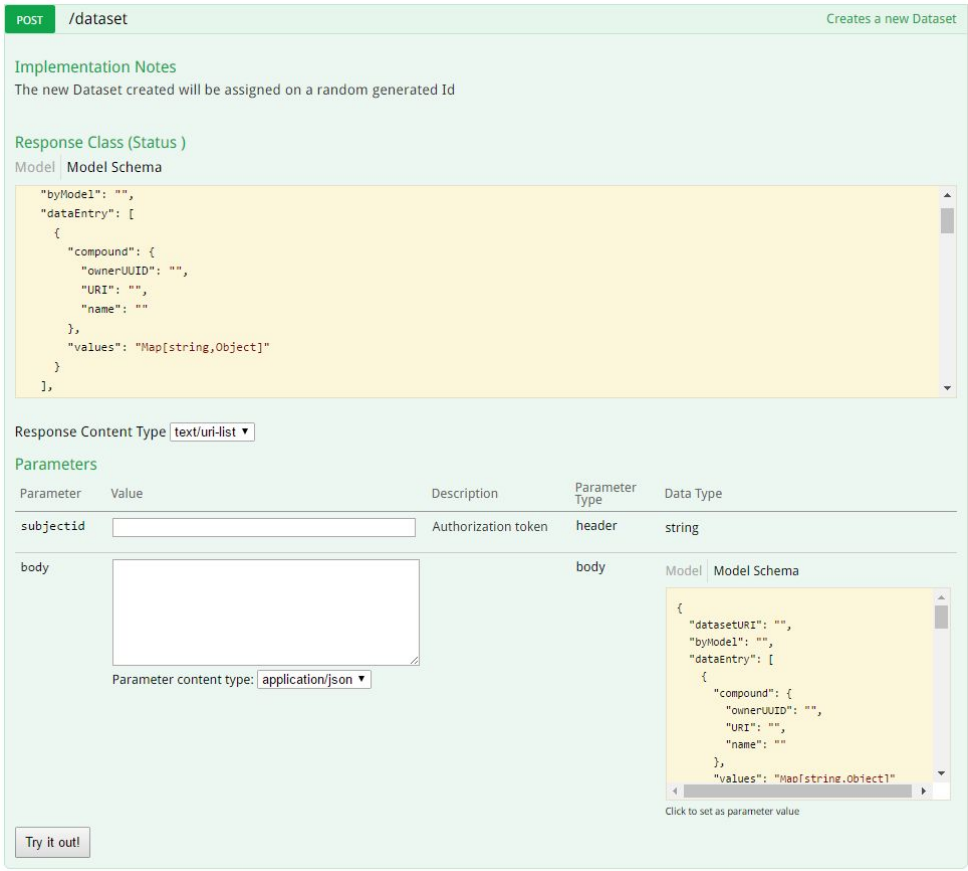

Figure 11: Create a new dataset in Jaqpot using the template in the *body* field of POST dataset.

4.4 GET ALL DATASETS

This HTTP method retrieves datasets based on their ascending IDs. Therefore, should these parameters be set correctly, a user may retrieve all datasets to which he/she has access. Figure 12 shows a screenshot of the Swagger interface and the relevant parameter fields Results can be obtained either in the form of a URI list or as a JSON list as specified by the Accept HTTP header. In the latter case, a list will be returned containing only the IDs of the datasets, their metadata and their ontological classes. The parameter max, which specifies the maximum number of IDs to be listed is limited to 500.

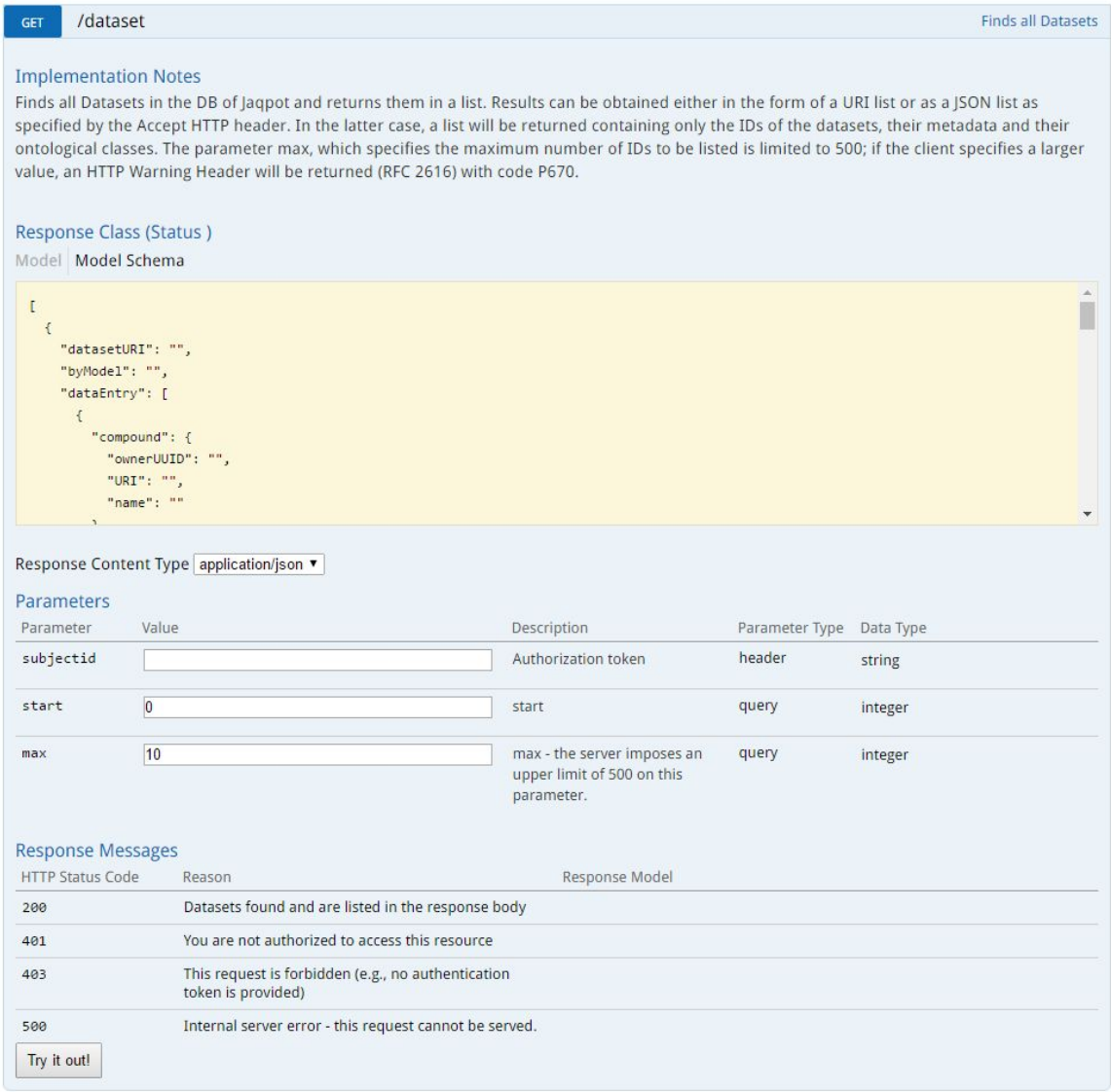

Figure 12: GET dataset method for retrieving datasets by their ascending IDs.

4.5 GET DATASET FEATURED

This method allows a user find Featured datasets in the Jaqpot database and returns them in a list. Results can be obtained either in the form of a URI list or as a JSON list as specified by the Accept HTTP header. In the latter case, a list will be returned containing only the IDs of the datasets, their metadata and their ontological classes. The parameter max, which specifies the maximum number of IDs to be listed is limited to 500; if the client specifies a larger value, an HTTP Warning Header will be returned (RFC 2616) with code P670. Screenshot in Figure 13.

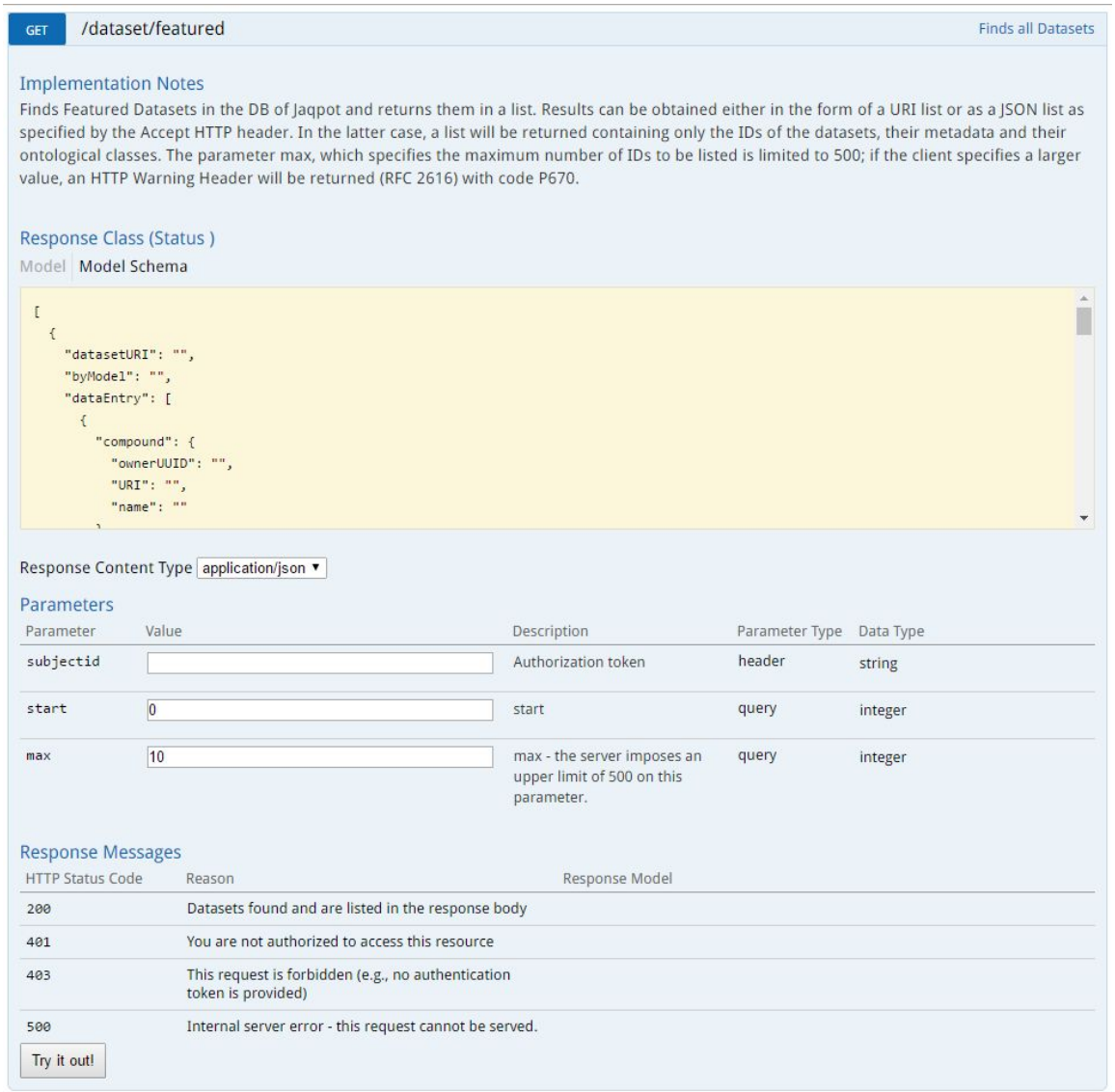

Figure 13: Screenshot of dataset API GET method for counting all datasets.

5. INTERLAB

This API contains one HTTP method (POST) which creates an inter-laboratory testing report which (once created) can also be viewed under the Report API described earlier. This requires a dataset created by a coordinator which includes measurements of different labs on the same nanoparticle and biological endpoint (keyword *prediction feature*). API shown in Figure 14.

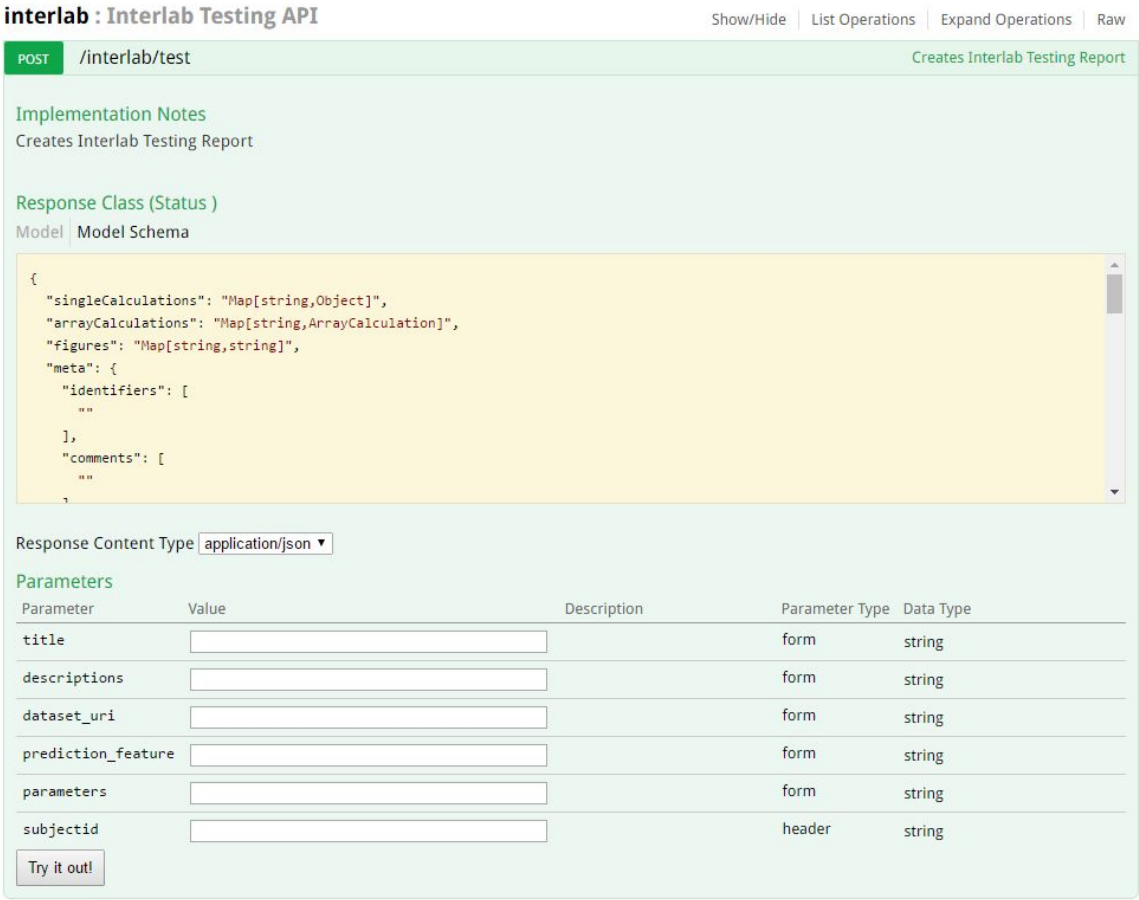

Figure 14: Screenshot of Interlab API and parameters for creating an inter-laboratory testing report.

6. PMML

The PMML API allows the creation and retrieval of PMML transformations and modelling tasks. The specific HTTP methods can be seen in Figure 15, which are namely GET all or specific entries and POST a new PMML or a selection.

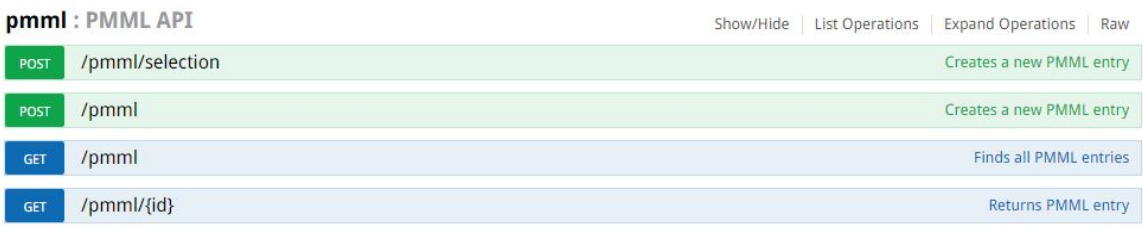

Figure 15: Options available under the Jaqpot PMML API.

6.1 POST PMML

This method allows the user to create a new PMML entry which is assigned a random unique ID. This must be pasted into the body section in JSON format. Figure 16 shows the method options.

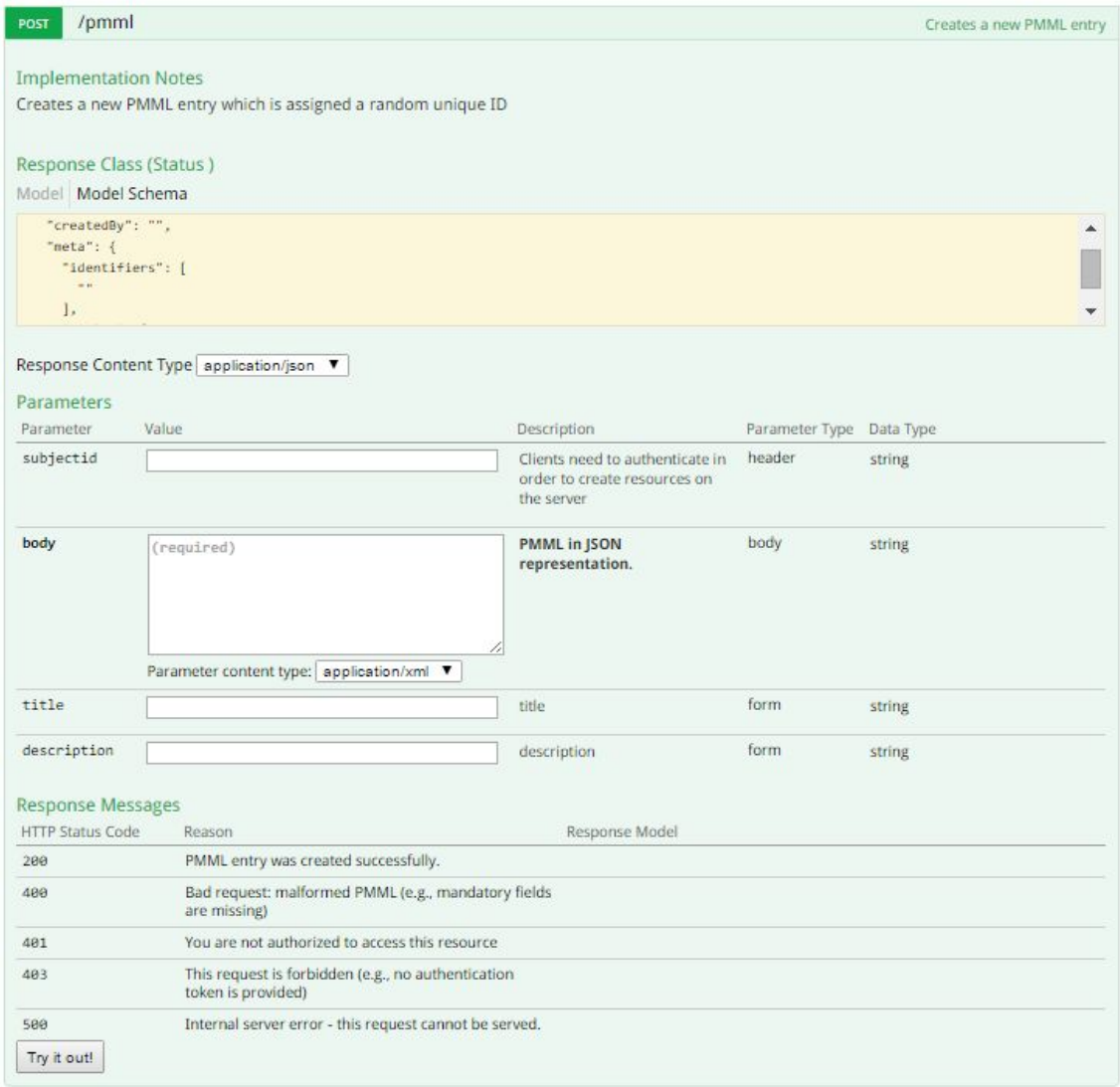

Figure 16: Create a new PMML entry using POST PMML.

6.2 GET PMML

This method finds all PMML entries in the database based on their ascending unique identifier and returns them in list format. Parameter *Start* shows the method where to begin and *Max* after how many to stop retrieving. The method is shown in Figure 17.

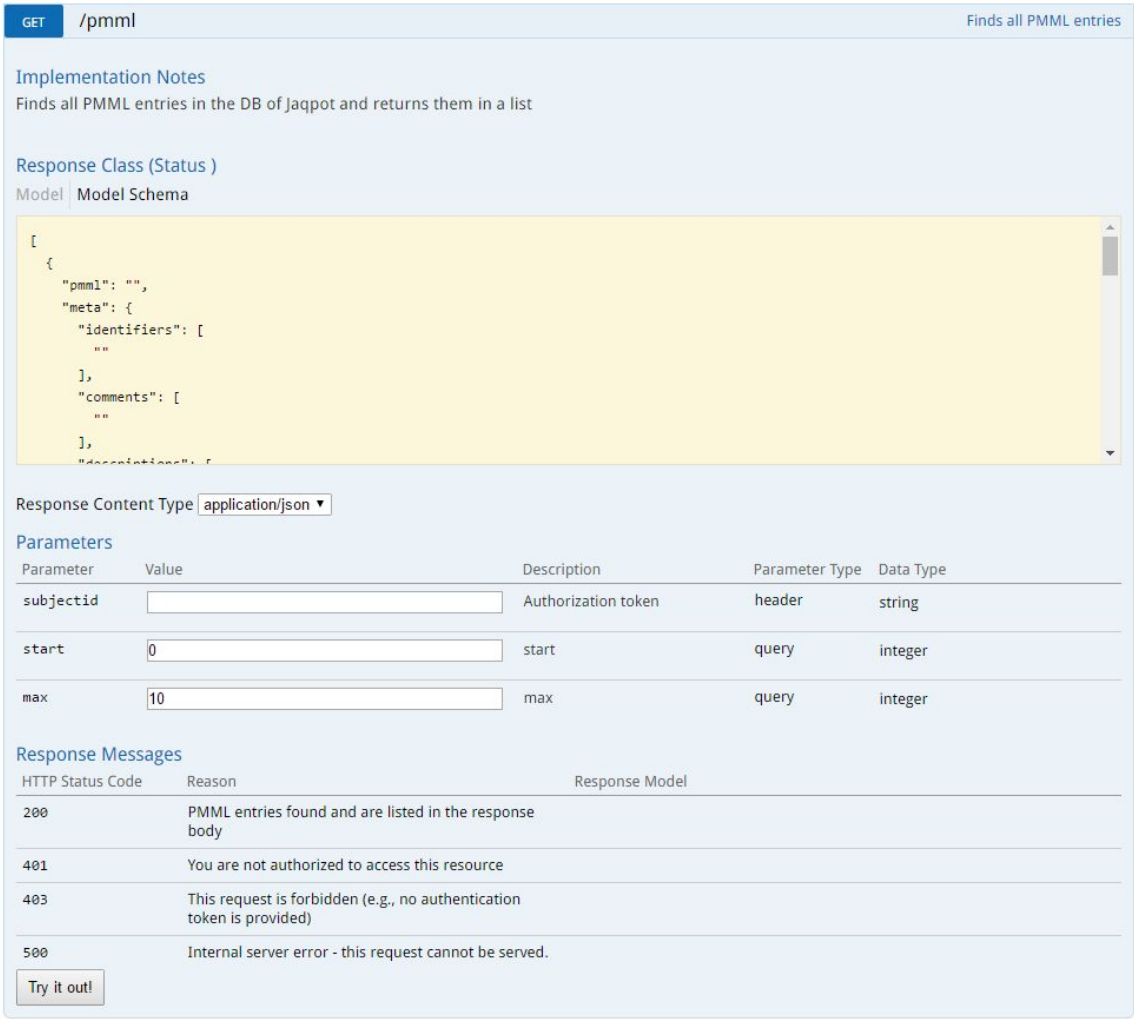

Figure 17: GET PMML method for retrieving all PMML entries in the database.

6.3 POST PMML SELECTION

This method, as with the above POST method creates a new PMML entry that performs feature selection. This requires the *features* field to be filled in. What this field expects is a comma separated list of feature URIs. The service will create a PMML entry based on those URIs, that can be used to filter a Dataset, cleaning it from any feature that does not belong to this list. A screenshot of the method and requirements is shown in Figure 18.

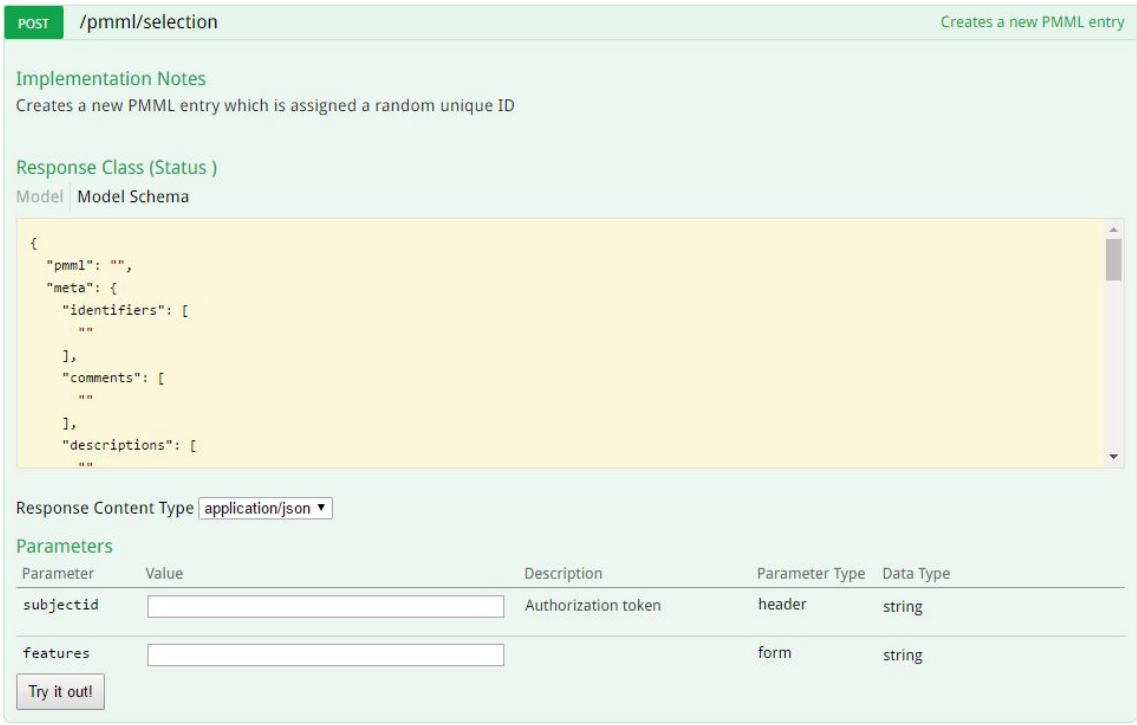

Figure 18: PMML POST method with selection for creating a new PMML entry.

6.4 GET PMML BY ID

This method retrieves a PMML document based on its ID. Screenshot in Figure 19.

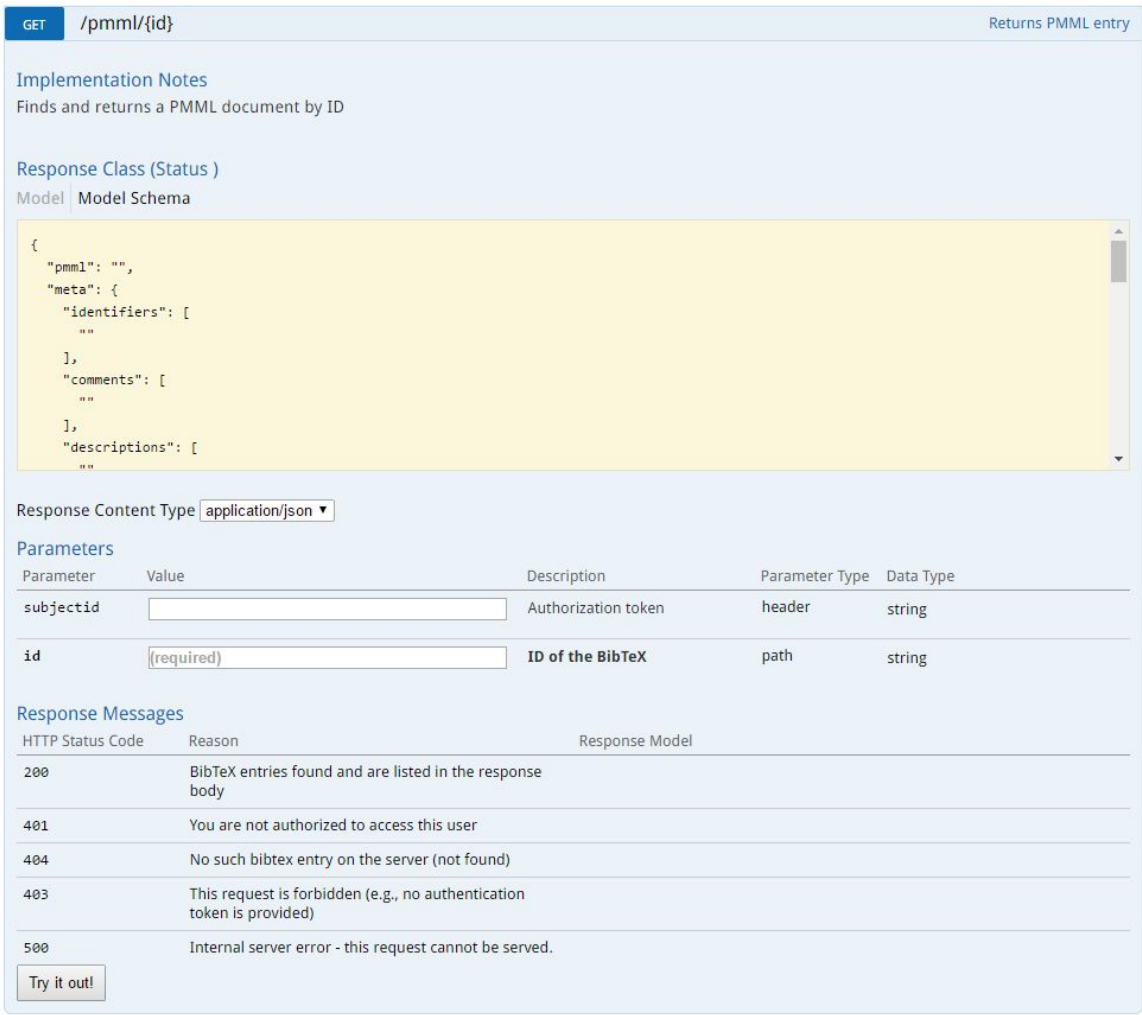

Figure 19: HTTP GET method in PMML API for retrieving PMML document.

7. READ ACROSS

The readacross API allows the user to derive a prediction based on similar nanoparticles in an existing dataset. In the parameters field, readAcrossURIs must be set to the nanoparticles for which the prediction will be carried out. The service responds with a report containing a weighted response based on known nanoparticles and the nearest neighbours found in the dataset. Screenshot shown in Figure 20.

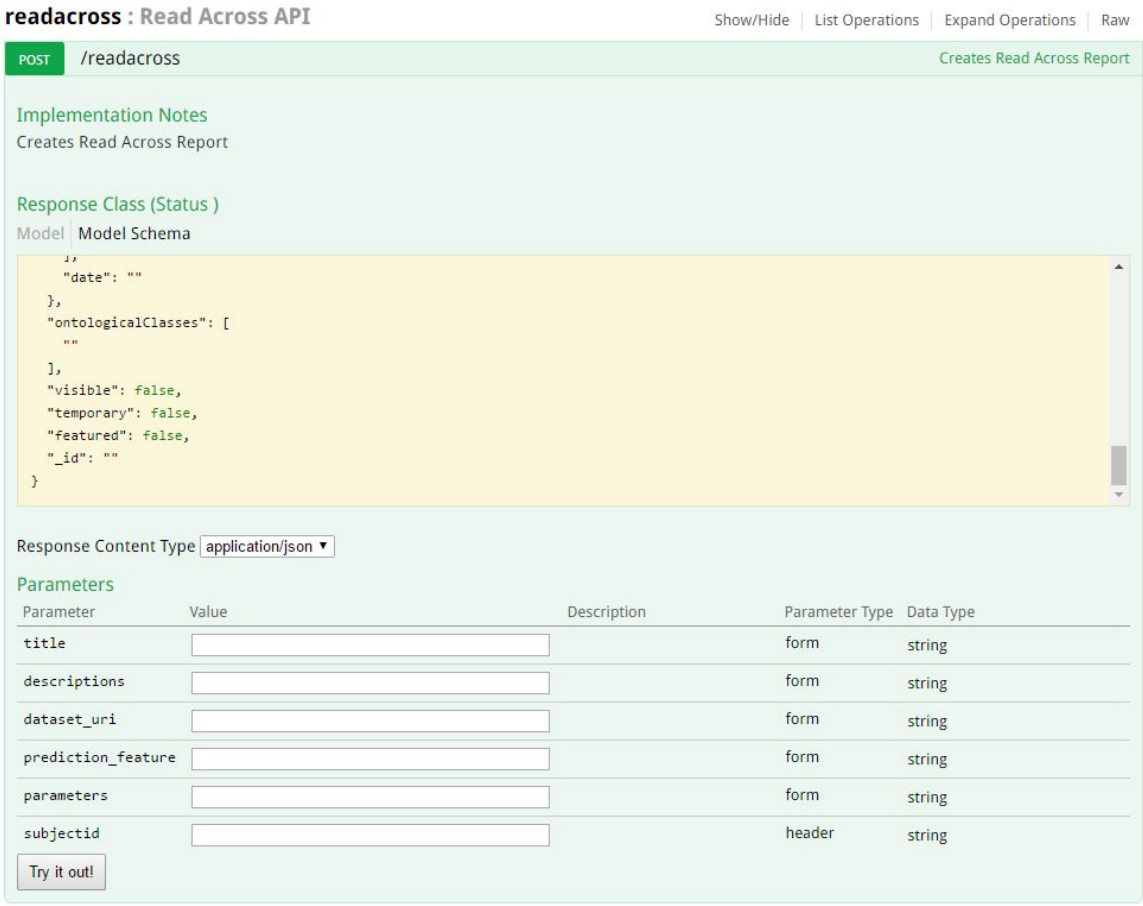

Figure 20: HTTP POST method in readacross API for retrieving prediction report based on similarity metrics.

8. BIBTEX

This API allows a user to handle bibliographic information in Bibtex format. It includes creating new entries, as well as retrieving, editing and deleting existing ones. This is shown in Figure 21.

Figure 21: Bibtex API menu.

8.1 GET BIBTEX

Users may find all bibtex entries in the database using their unique identifiers, as with services already described earlier in this document, using a starting ID and a maximum ID as parameters. Furthermore, they may use a generic search or locate the entry via its creator. This is shown in Figure 22.

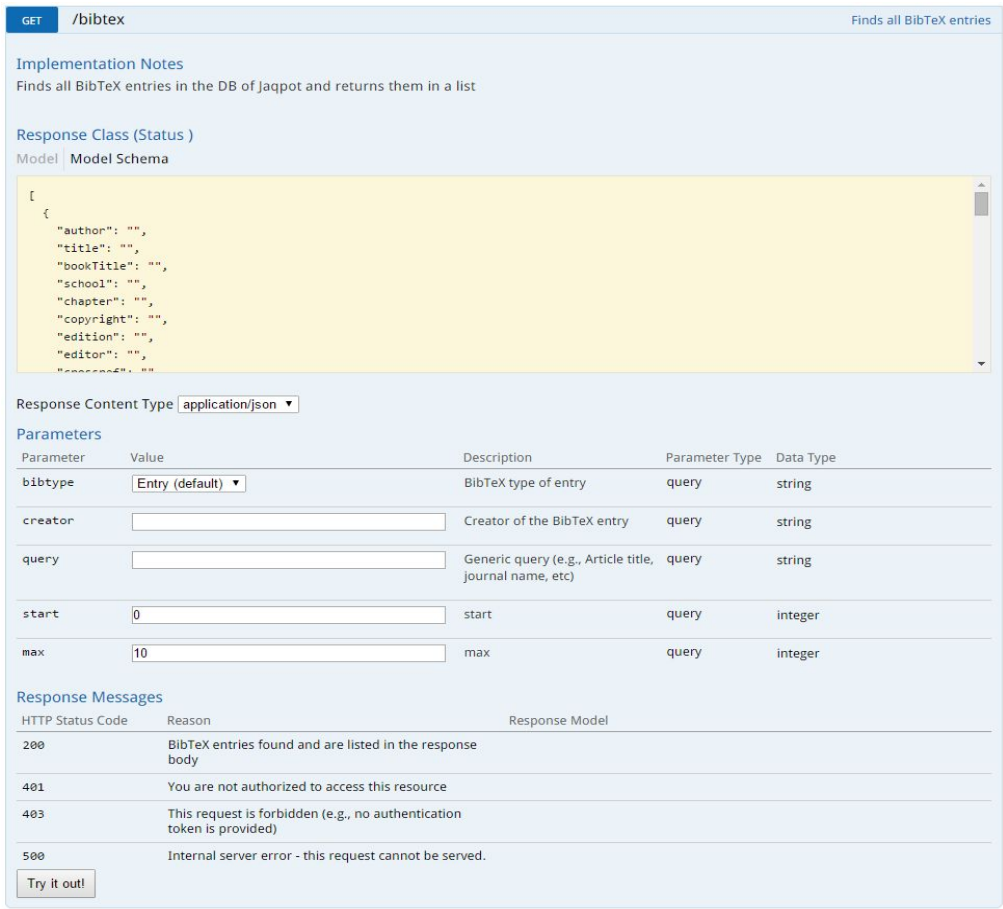

Figure 22: GET method for Bibtex API.

8.2 POST BIBTEX

This method creates a new bibtex entry which is assigned a random identifier. In the body section, a template provides the user with the necessary relevant fields such as author and journal in JSON format. This is shown in Figure 23.

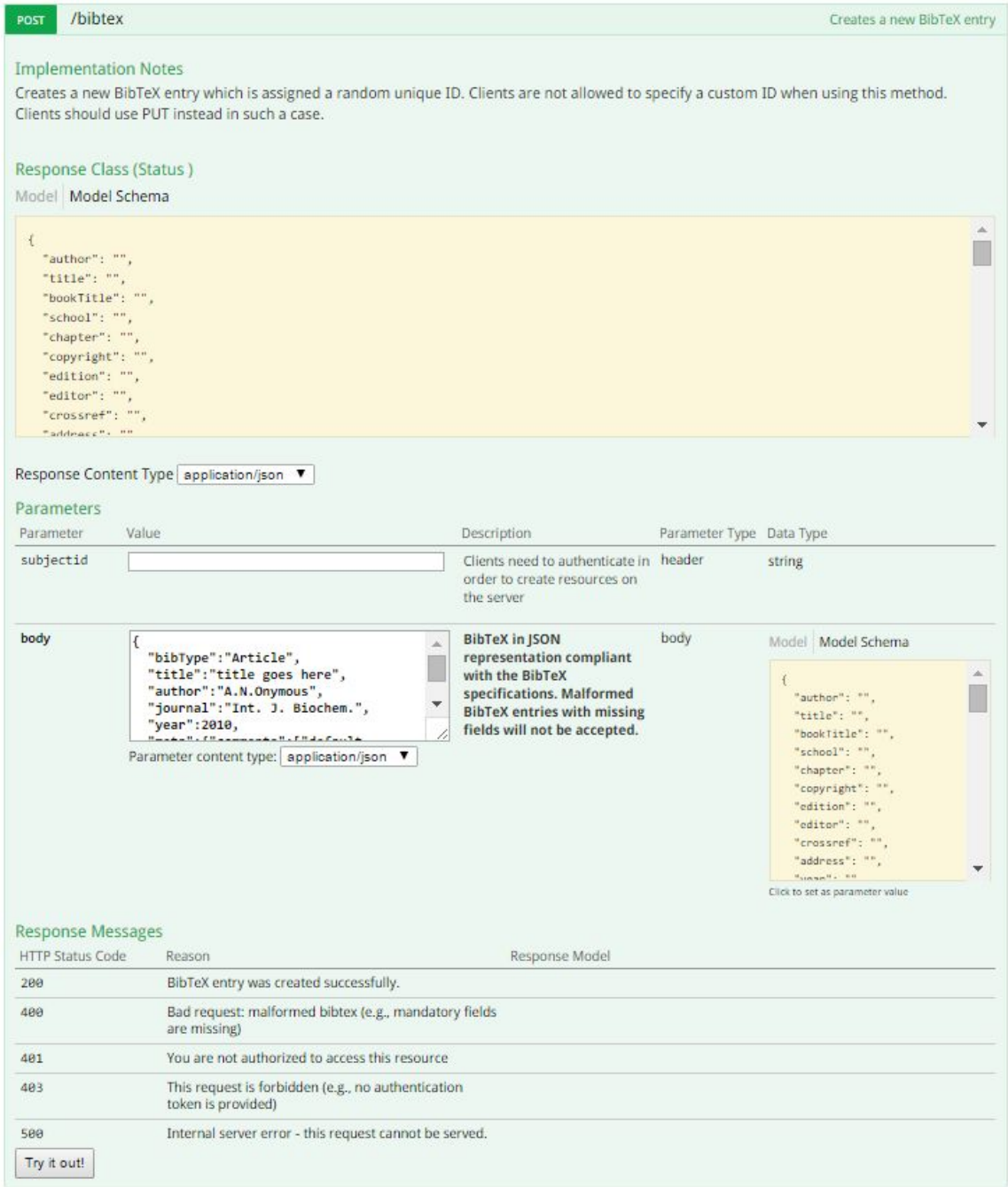

Figure 23: POST method for Bibtex API.

8.3 GET BIBTEX BY ID

Locates and returns a Bibtex entry based on its unique identifier. Screenshot of the HTTP method is shown in Figure 24.

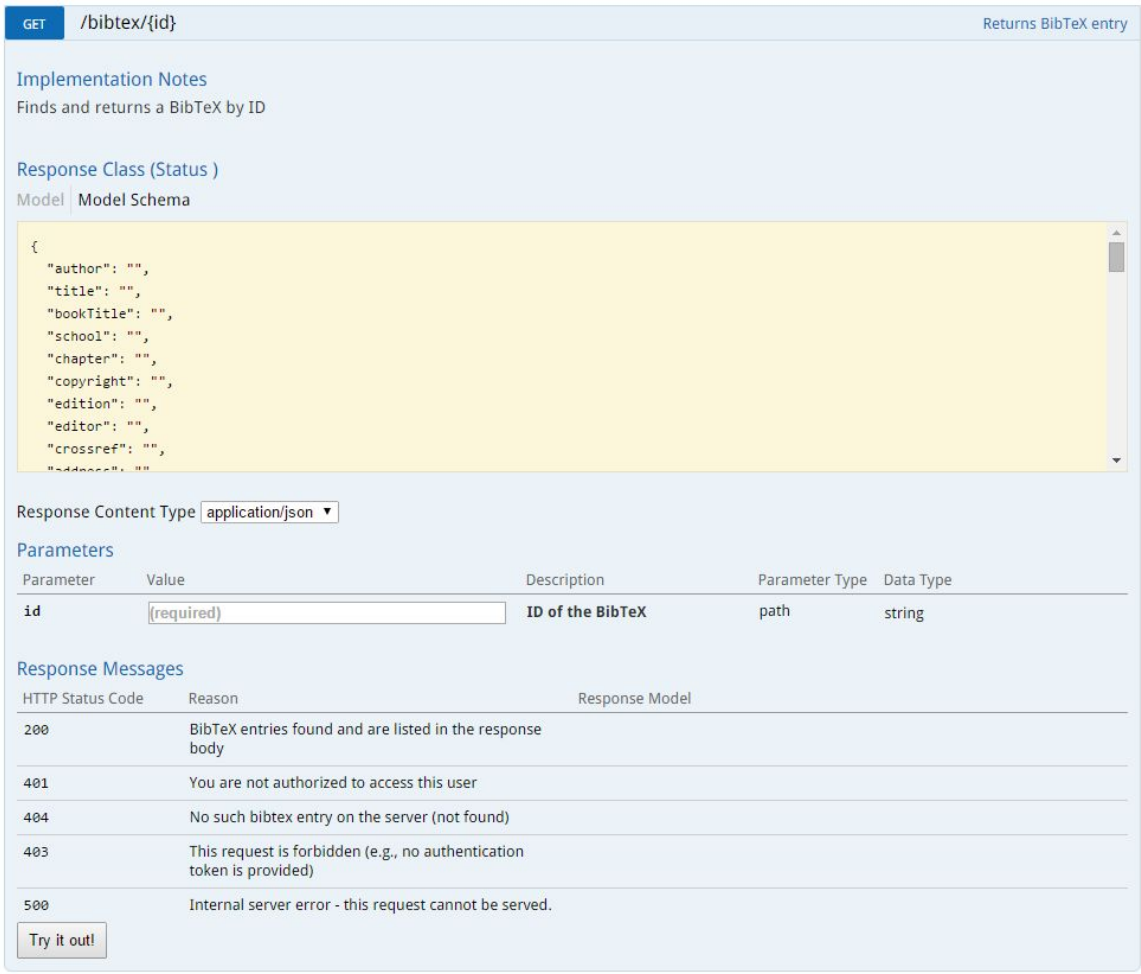

Figure 24: GET Bibtex by ID method for Bibtex API.

8.4 PUT BIBTEX BY ID

Creates a new BibTeX entry at the specified URI. If a BibTeX already exists at this URI, it will be replaced. If, instead, no BibTeX is stored under the specified URI, a new BibTeX entry will be created. Authentication, authorisation and accounting (quota) restrictions may apply. As above, the template is shown in the *body* parameter. Screenshot in Figure 25.

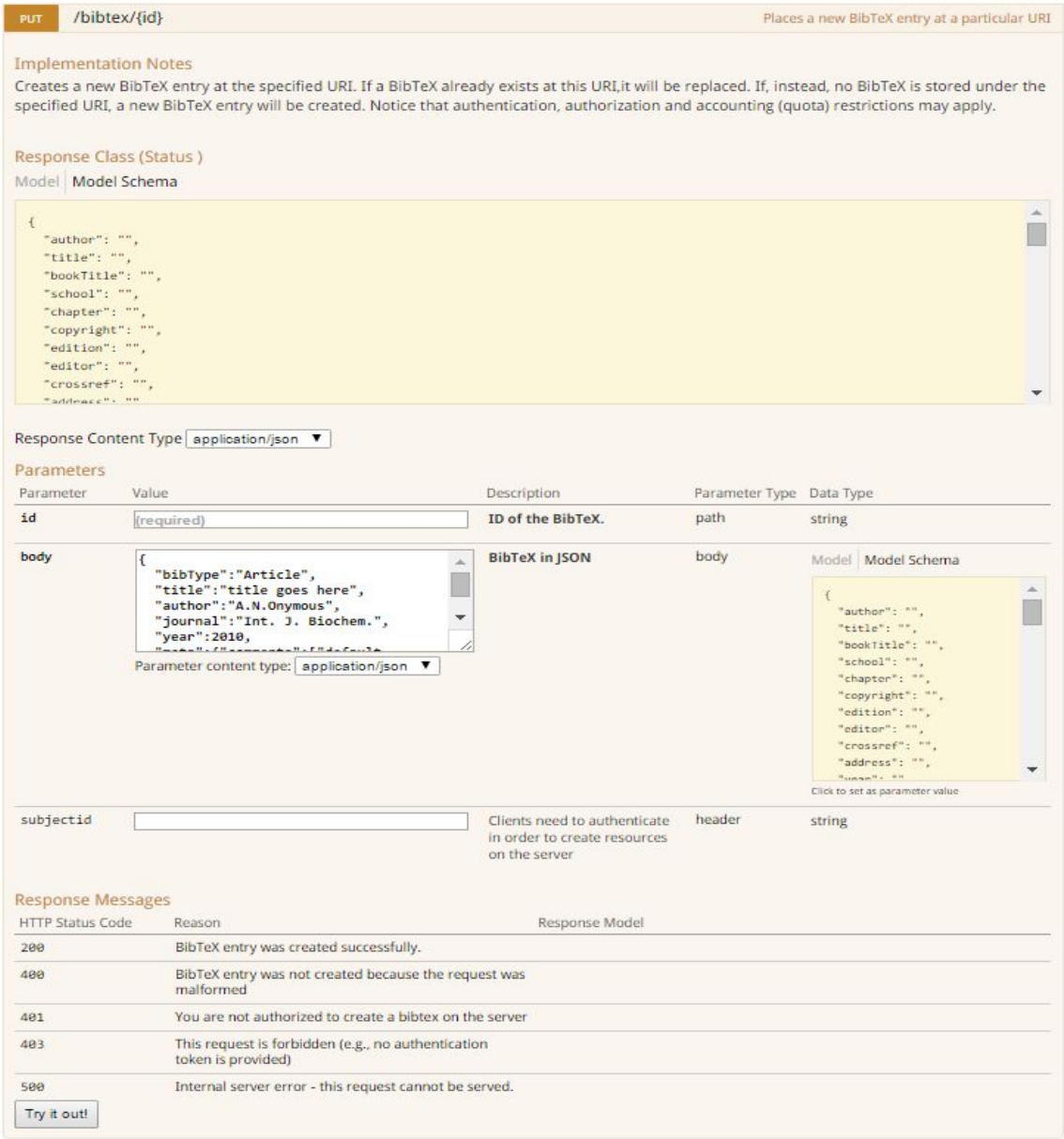

Figure 25: Screenshot of PUT Bibtex entry by ID

8.5 DELETE BIBTEX BY ID

As with GET by ID, this method uses a single parameter to delete a Bibtex entry. The method is idempotent, that is, it can be used more than once without triggering an exception/error. If the BibTeX does not exist, the method will return without errors. Authentication and authorization requirements apply, so clients that are not authenticated with a valid token or do not have sufficient privileges will not be able to delete a BibTeX using this method. This is demonstrated in Figure 26.

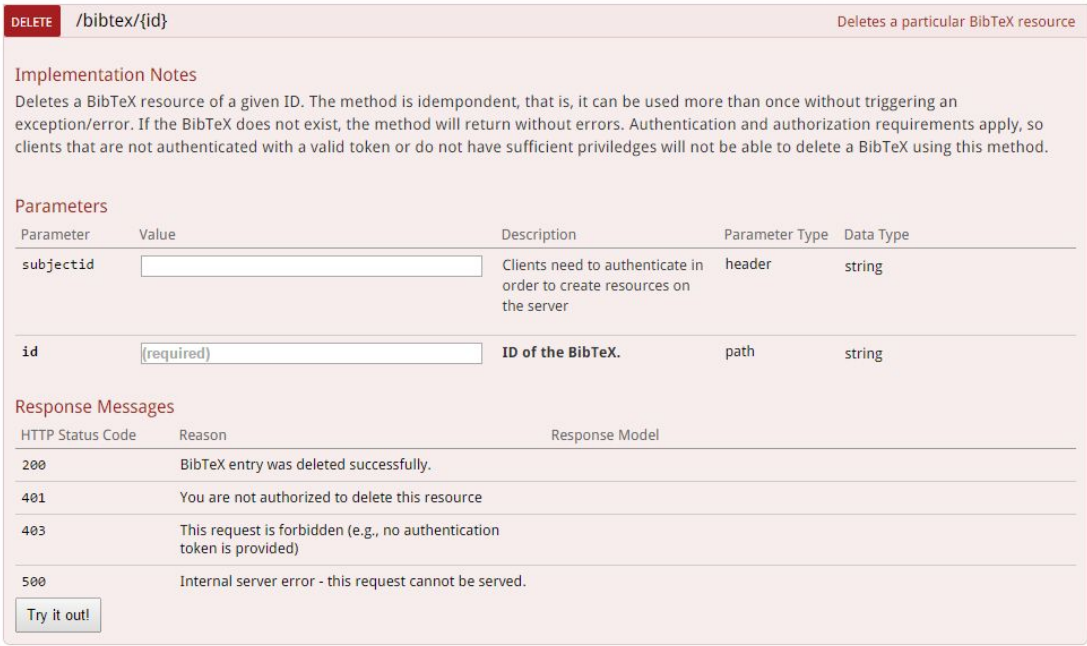

Figure 26: DELETE Bibtex entry by unique identifier

8.6 PATCH BIBTEX BY ID

Users are allowed to edit existing Bibtex entries by providing an existing BibtexID and a new body, as shown in Figure 27. Authentication and authorisation restrictions may apply.

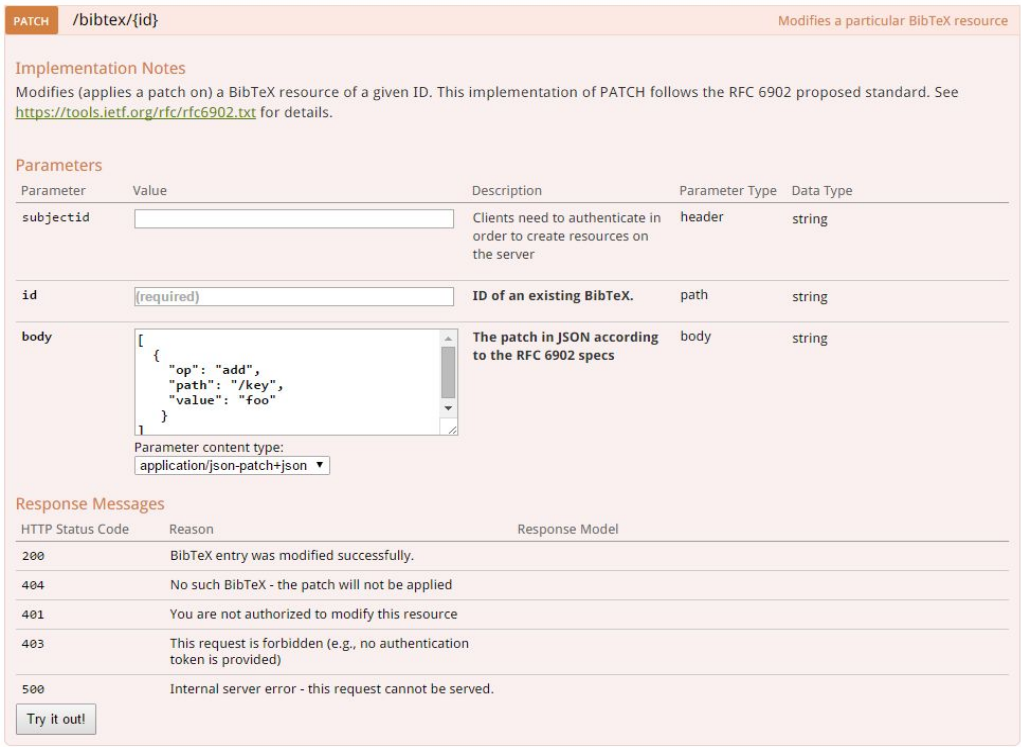

Figure 27: PATCH (modify) Bibtex entry by ID method of Bibtex API.

9. VALIDATION

In the validation API users may submit datasets and models for validation using three POST methods. The available options are cross-validation (stratified or random), training set split and external validation, as shown in Figure 28. All validation methods use the Task system in their execution. The result of the task will always be a Report document.

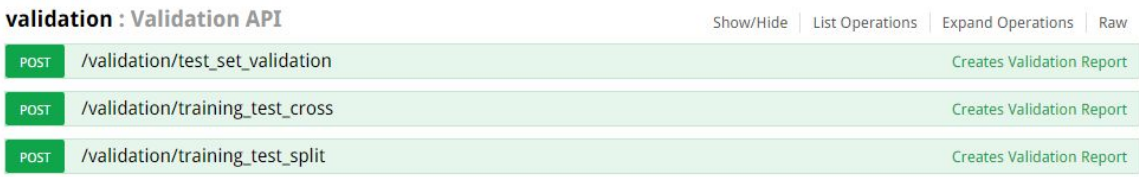

Figure 28: POST options available under validation API.

9.1 POST VALIDATION TEST SET VALIDATION

For this method, a user must have already built or located an existing machine learning model and have created a test set with the same variables as the training set. These need to be provided in the appropriate fields as demonstrated in Figure 29.

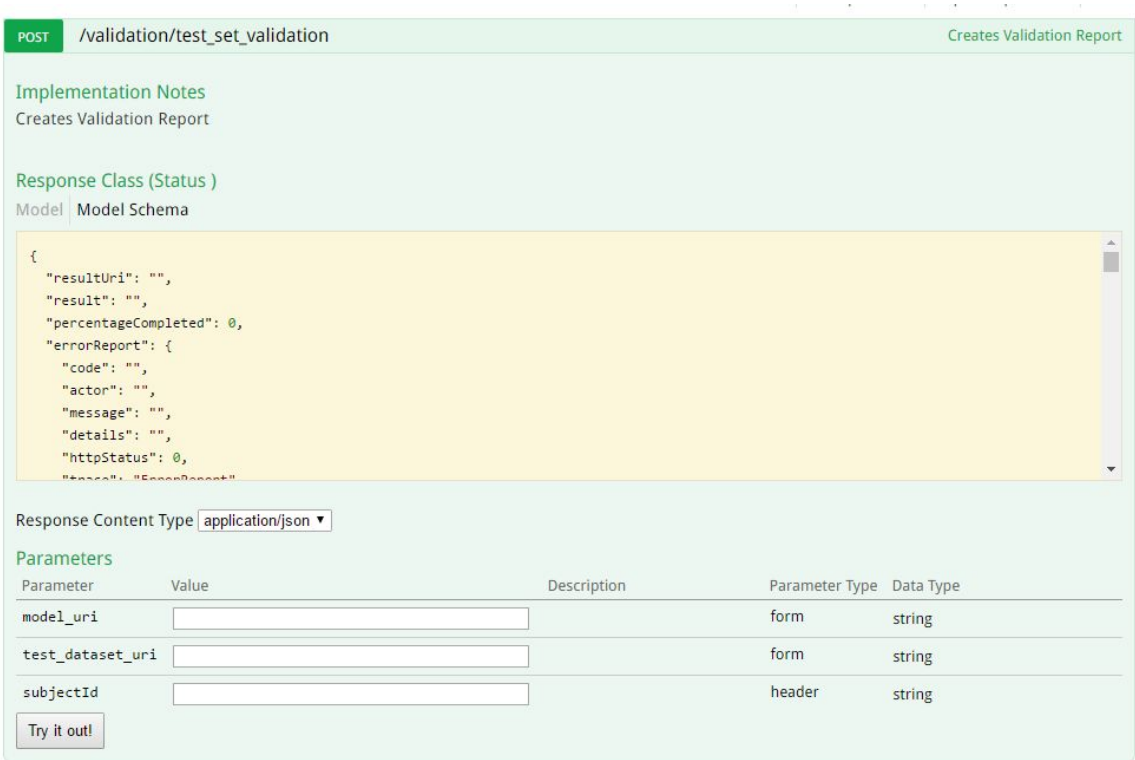

Figure 29: POST method for external test set validation.

9.2 POST VALIDATION TRAINING SET CROSS

Users may choose to cross validate an existing algorithm before building their models. In this case, options need to be provided for all data splits, including scaling or PMML transformations. Furthermore, the dataset, prediction feature and algorithm need to be provided, including potential parameters if any (i.e. PLS with VIP scores requires the number of latent variables to be provided to the algorithm). Finally, the user may choose stratified or random cross-validation, for which he/she will need to provide a random seed, as with commercial machine learning software applications. These options are shown in the screenshot of the method in Figure 30.

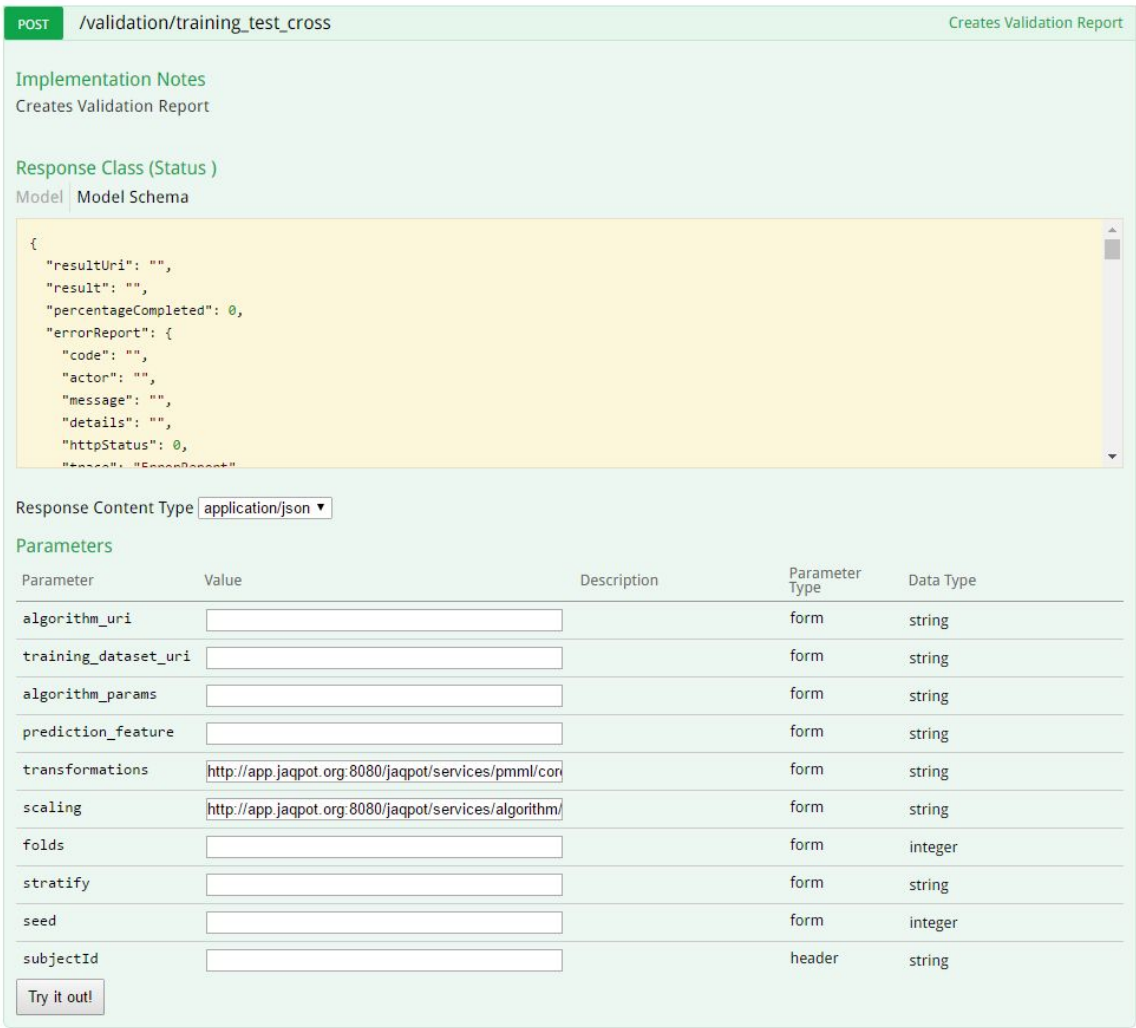

Figure 30: POST method for cross-validation under validation API

9.3 POST VALIDATION TRAINING TEST SPLIT

The last option available in the validation API is to validate an existing algorithm by splitting a dataset into training and test sets on a specified ratio. As above, transformations and scaling need to be provided if required, as well as the algorithm and prediction feature. The method options are shown in Figure 31.

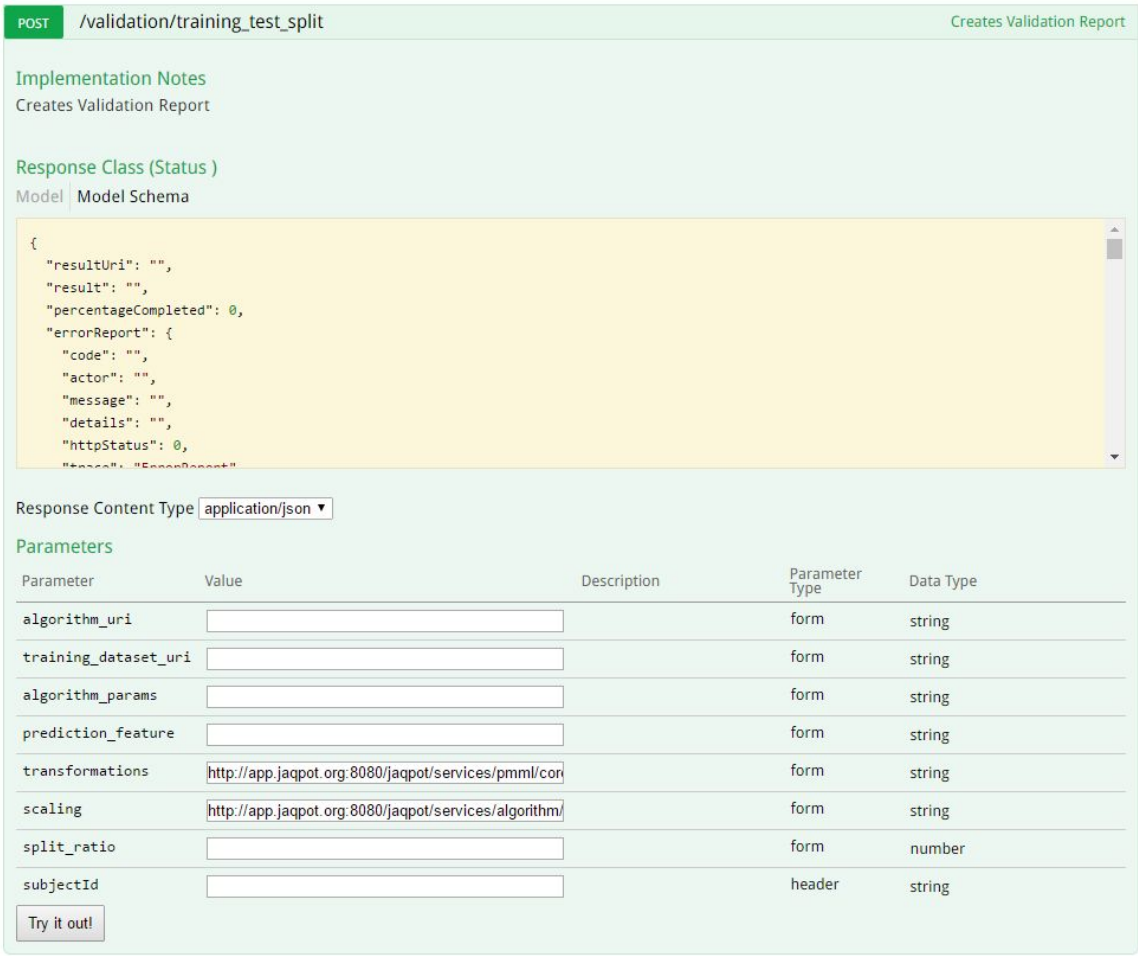

Figure 31: Options for POST training/test split method in validation API.

10. ENM

The eNM (eNanoMapper) API allows the creation of bundles and datasets, but also retrieves properties (experimental) and descriptors (calculated). The menu options are shown in Figure 32.

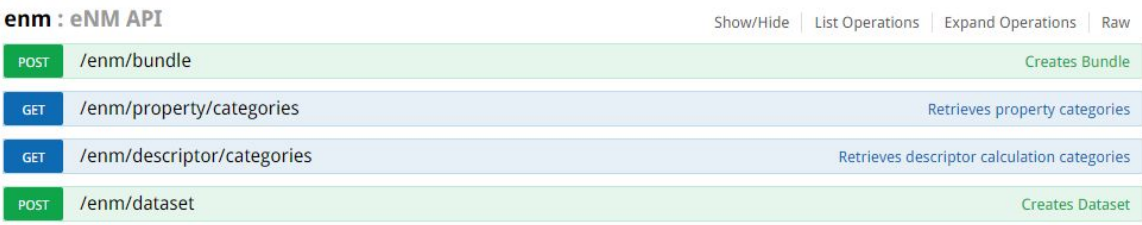

Figure 32: eNM API methods available.

10.1 POST ENM DATASET

This method allows users to create a dataset including calculated descriptors. It works by reading studies from bundles' substances. Users should provide a *body* parameter according to the template provided in JSON format. This is shown in Figure 33.

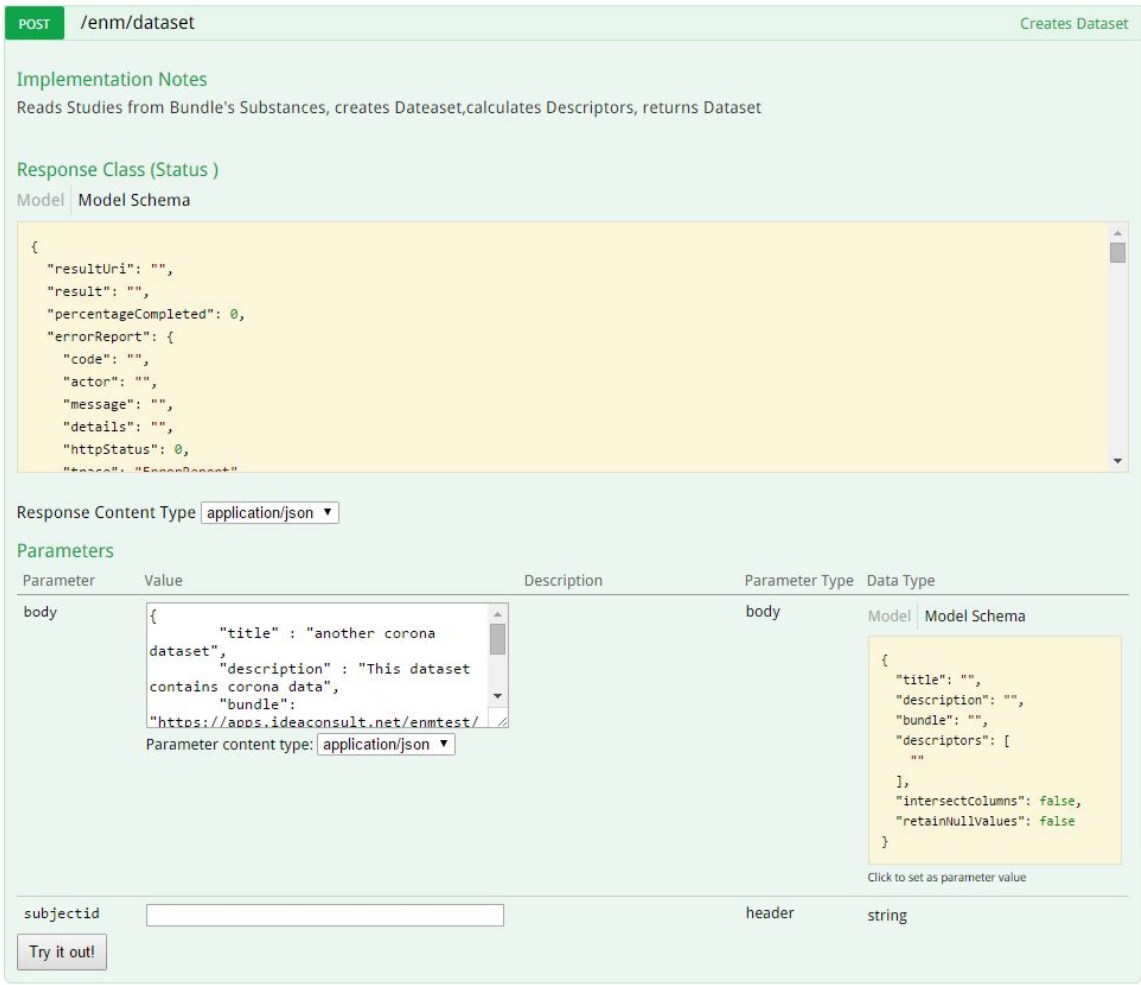

Figure 33: POST eNM dataset method contained in eNM API

10.2 POST ENM BUNDLE

Creates bundle by retrieving nanoparticle entries (keyword *substances*) from a particular owner (substance owner). As above, it request needs to provide a body parameter. If no substances are provided, the system assumes that all available substances from that owner are required. The same goes with properties. Screenshot shown in Figure 34.

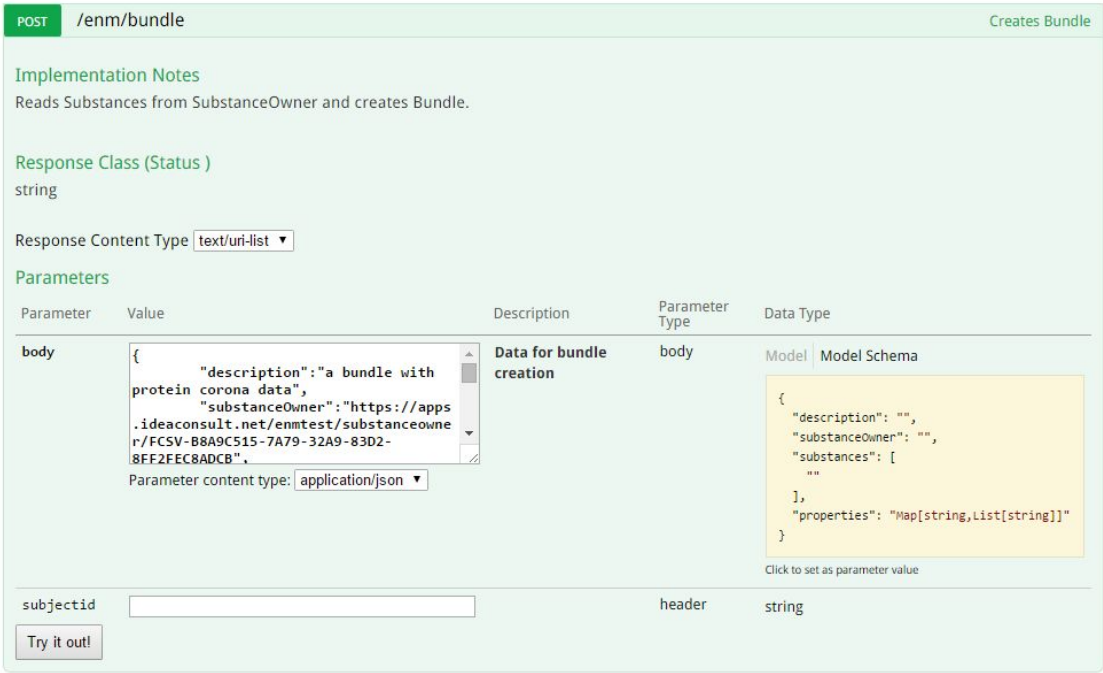

Figure 34: POST eNM bundle method from eNM API

10.3 GET ENM PROPERTY CATEGORIES

Returns all eNM property categories. No parameters necessary, as demonstrated in Figure 35.

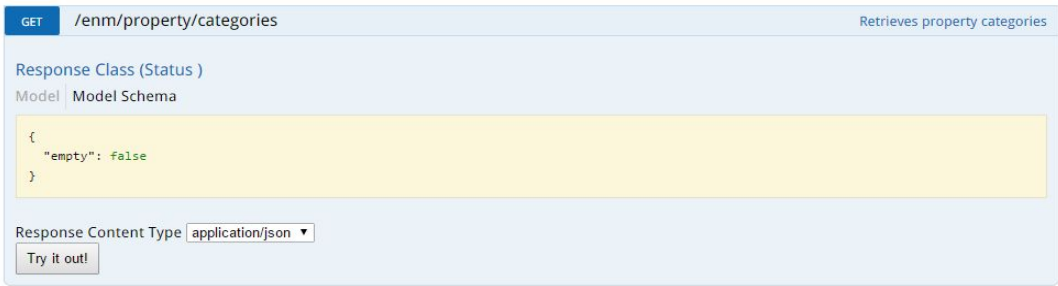

Figure 35: eNM API GET property categories method.

10.4 GET ENM DESCRIPTOR CATEGORIES

As above, for calculated descriptor categories. Screenshot in Figure 36.

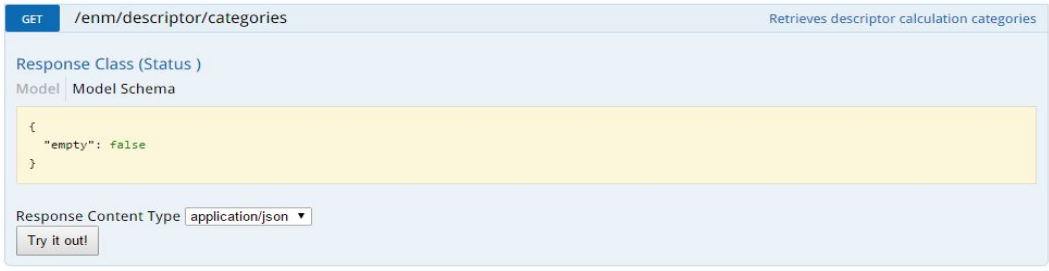

Figure 36: eNM API GET descriptor categories method.

11. MODEL

The Jaqpot model API allows users to retrieve a model and its features, use a model (POST) to create a prediction and delete a model. Each HTTP method is described in detail later in this section. All options available under the model API are shown in Figure 37.

Figure 37: Options available in Jaqpot's model API

11.1 GET MODEL BY ID

Retrieves an existing model by its unique identifier. Users must know the model ID and fill in the appropriate field. By clicking *Try it out* a response is given by the Swagger interface. Shown in Figure 38.

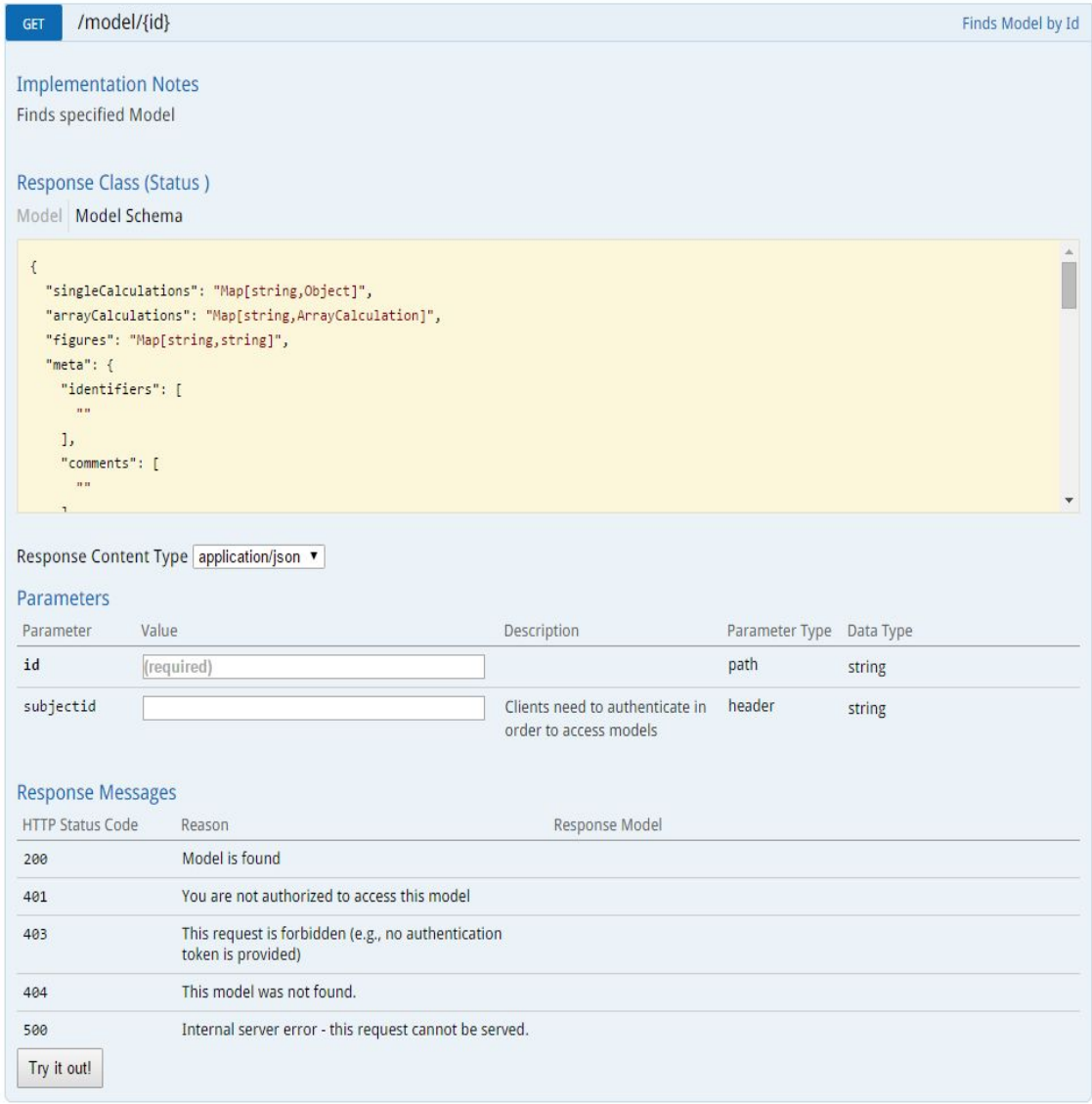

Figure 38: GET model by ID method of model API

11.2 POST MODEL BY ID

Users may use this HTTP method in order to derive a prediction on their data using an existing model. It creates a new dataset with predictions. If transformation or scaling is applied, it replaces columns with transformed/scaled ones, otherwise it expands the input dataset with new predicted columns. Both dataset URI and model ID are required fields. Users may choose to allow the response to be visible to other users or not ("visible" parameter). If "visible" is set to false, it makes a resource invisible to any user. This option is usually set true for transformation, scaling and DOA models. It should be noted that invisible models do not take up user quota. Figure 39 illustrates a screenshot of the POST model by ID method.

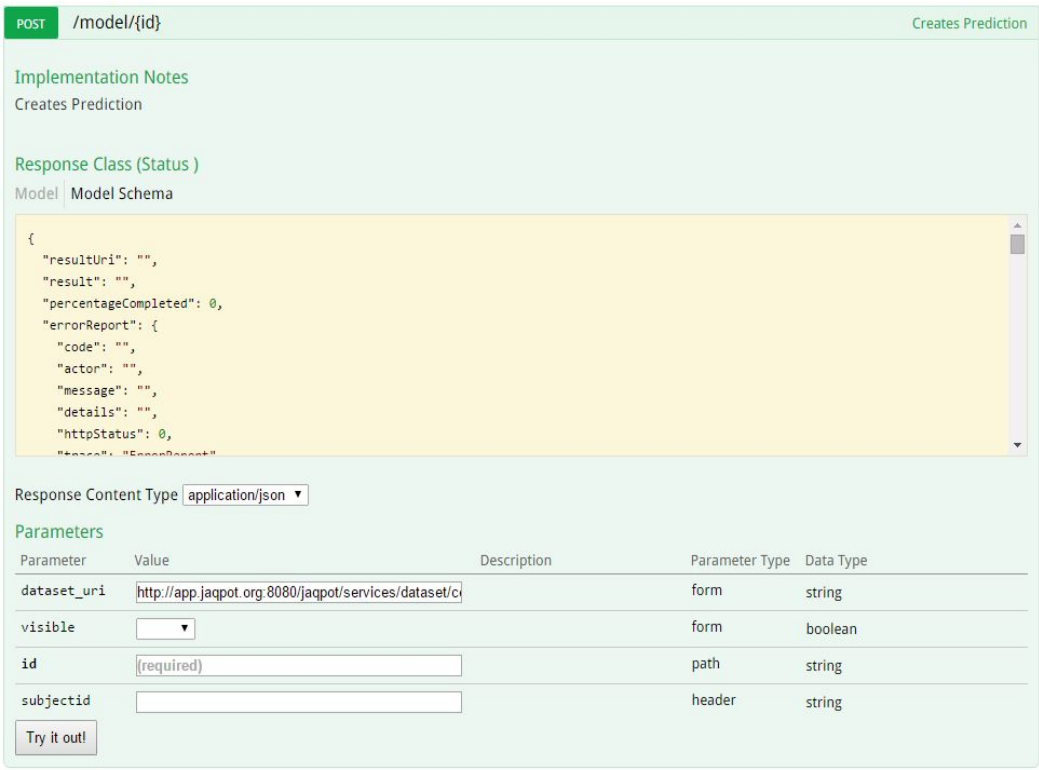

Figure 39: POST model by ID method under the model API

11.3 DELETE MODEL BY ID

Users may delete a model using its unique ID. Authorisation restrictions may apply. Screenshot in Figure 40.

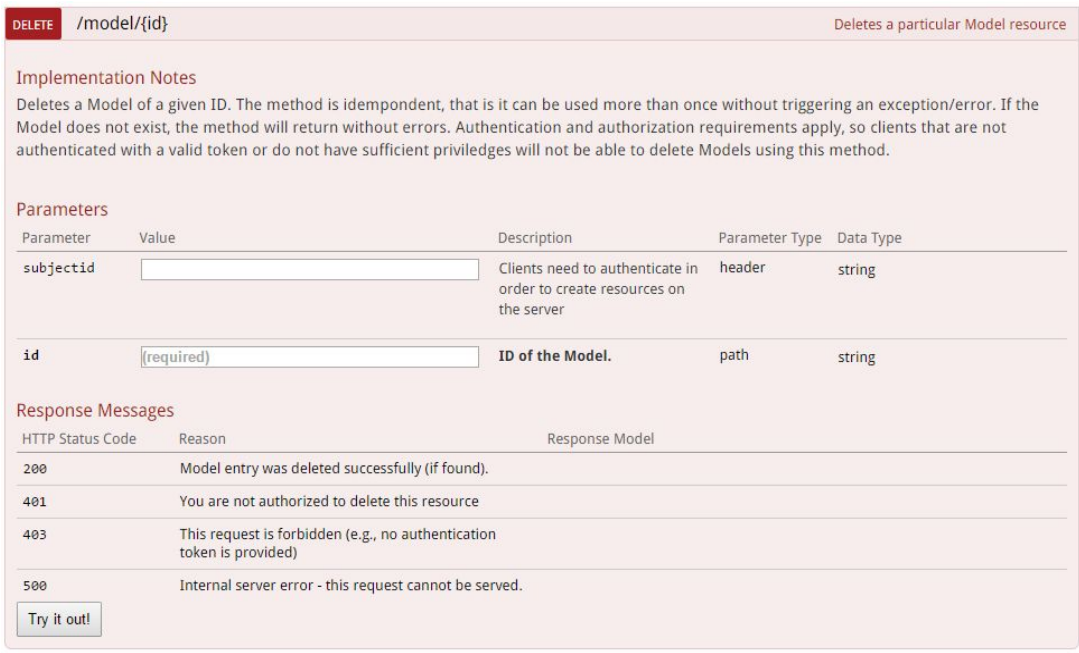

Figure 40: DELETE model by ID method in model API

11.4 GET MODEL

Retrieves all models created by the user in the Jaqpot database based on their ascending IDs. This can be carried out with the start/max options described in full earlier in this document. Screenshot in Figure 41.

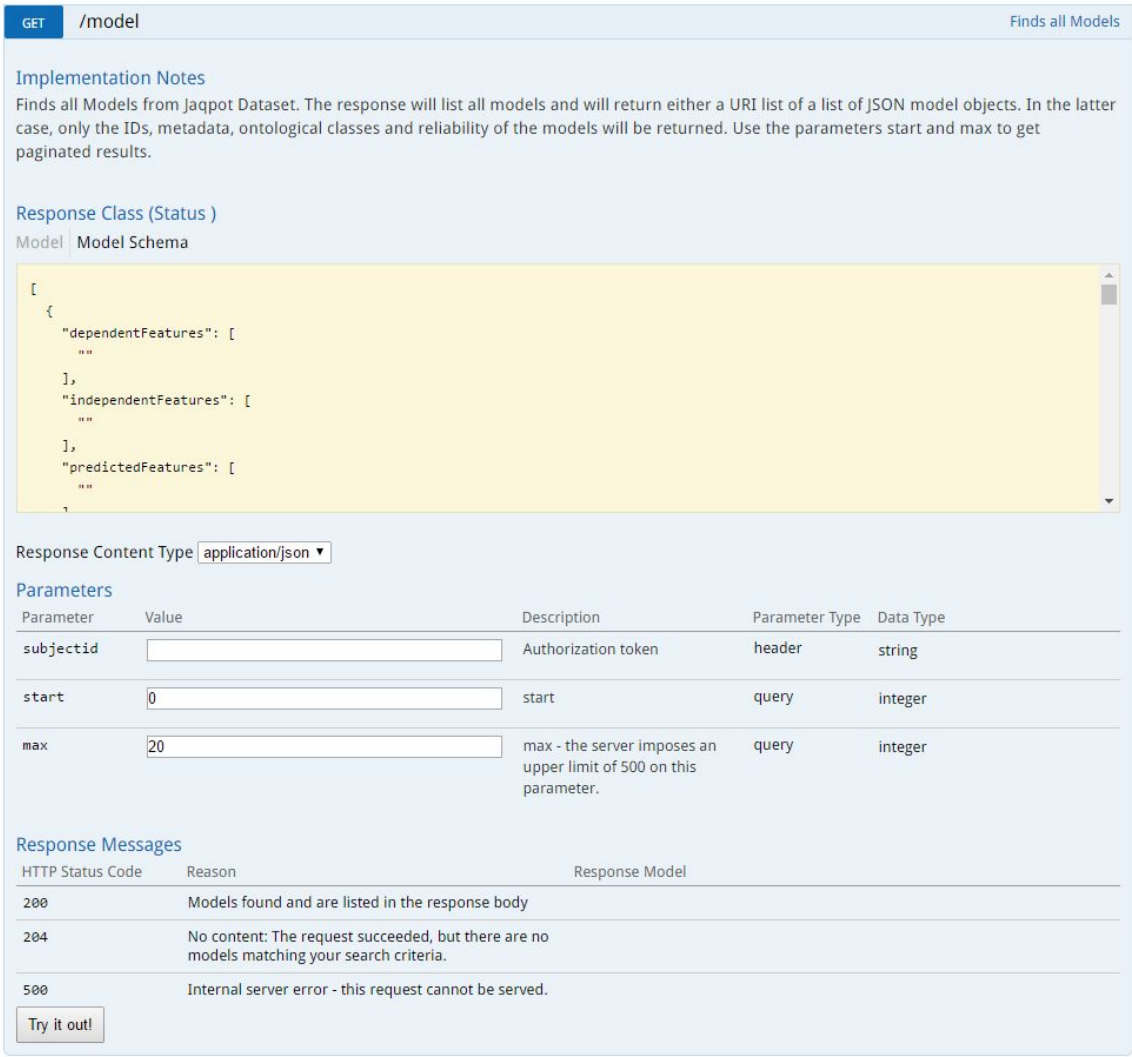

Figure 41: Screenshot of GET model HTTP method from model API

11.5 GET MODEL FEATURED

Finds featured Models from Jaqpot database. The response will list all models and will return either a URI list of a list of JSON model objects. In the latter case, only the IDs, metadata, ontological classes and reliability of the models will be returned. As above, users need to provide the parameters start and max to get paginated results. This is shown in Figure 42.

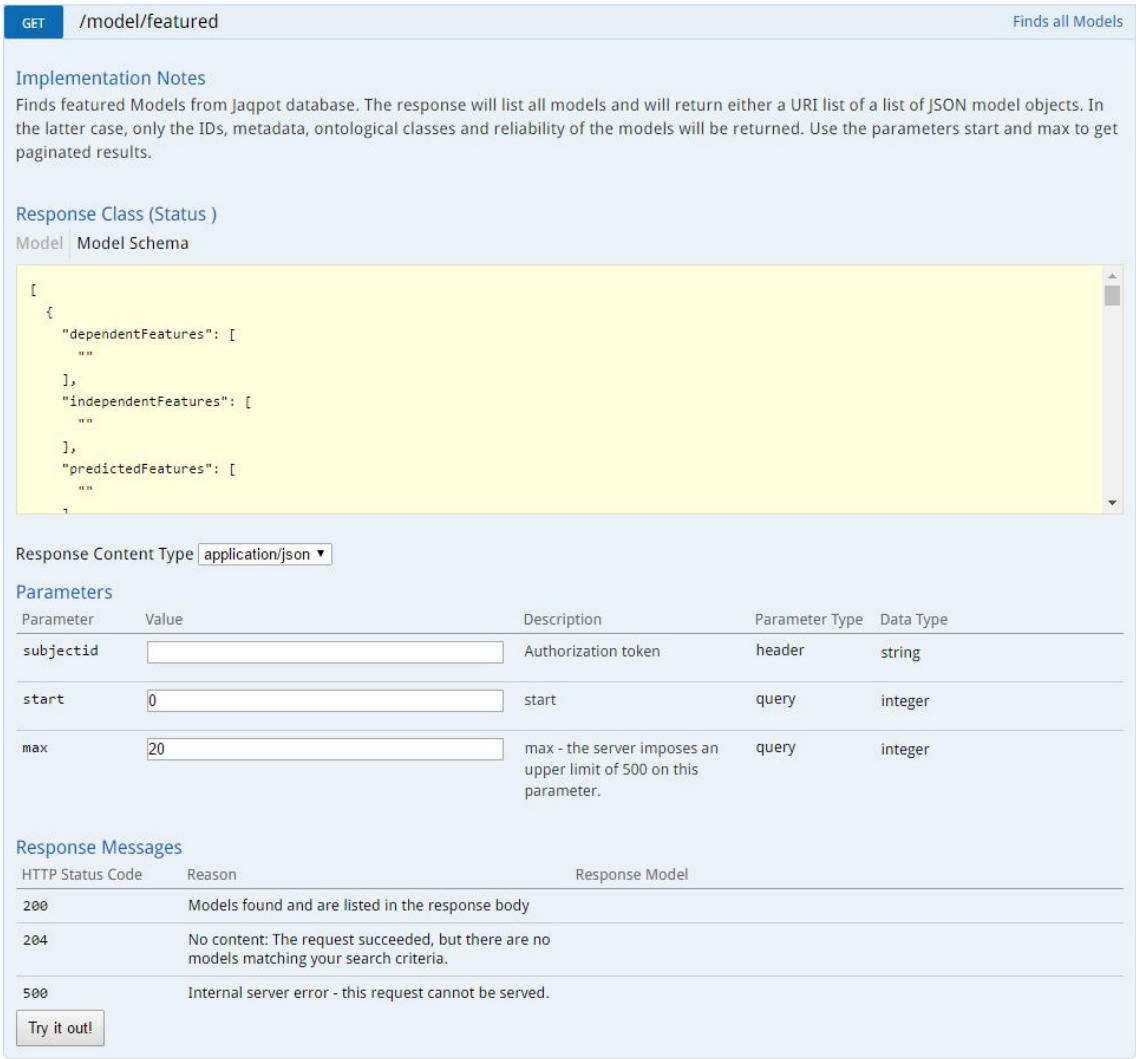

Figure 42: GET model count from model API

11.6 GET MODEL BY ID PMML

Similar to all GET by ID methods, this option requires a unique identifier and returns a model in PMML format, if applicable. Shown in Figure 43.

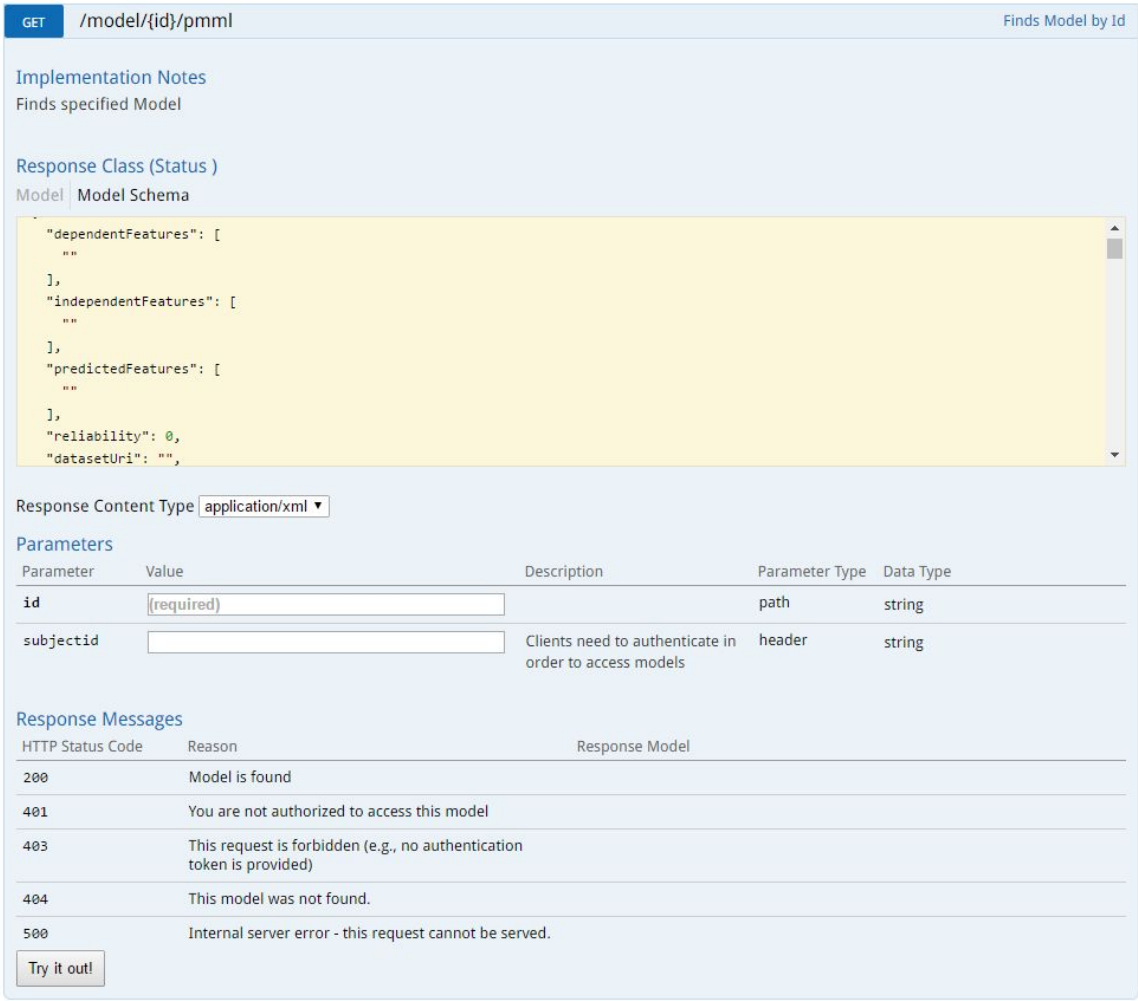

Figure 43: GET PMML model by ID method

11.7 GET MODEL BY ID INDEPENDENT

Lists the independent features of a model. Independent features are the attributes/properties of the dataset used to train the model that the model selected as input. The result is a list of URIs. Screenshot in Figure 44.

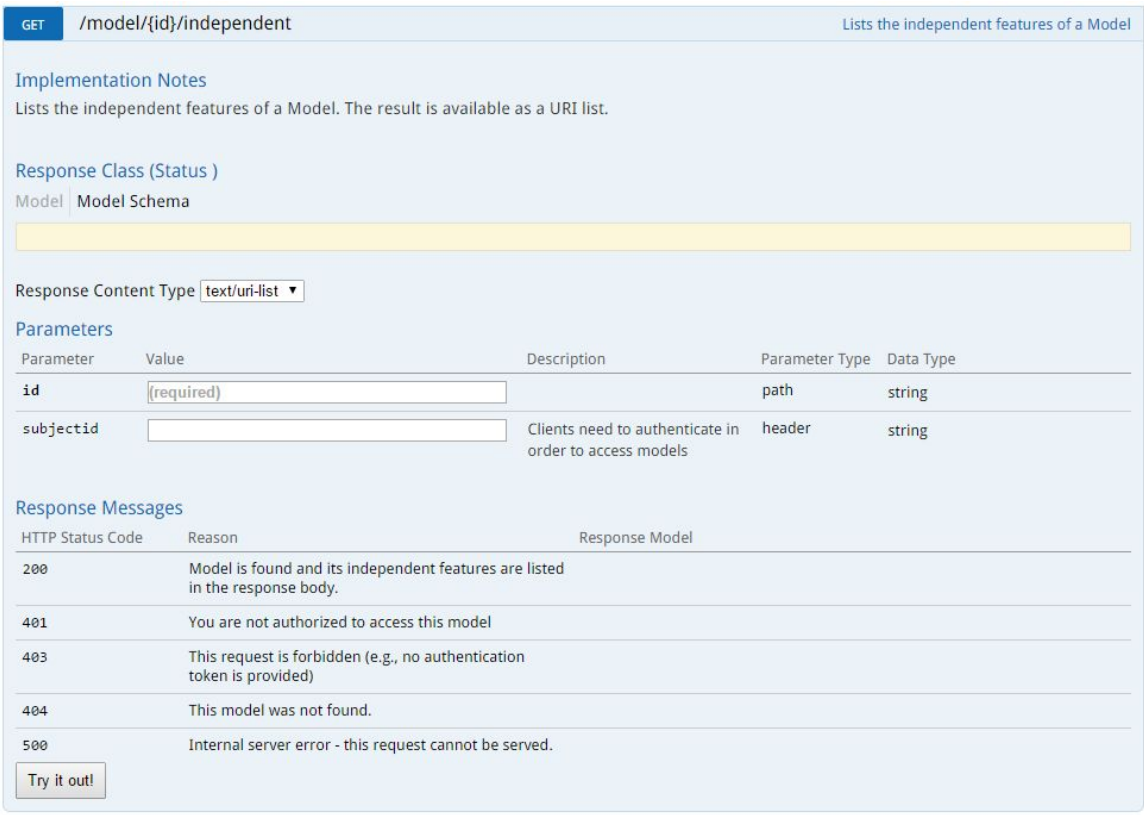

Figure 44: GET independent features from model API

11.8 GET MODEL BY ID DEPENDENT

As above, this returns dependent features of a specific model. Dependent features are the target or class attributes of the model.

11.9 GET MODEL BY ID PREDICTED

As above, this lLists the predicted features of a Model identified by its ID. Predicted features are new features created by the model that will be used to store any predictions the model makes. The result is available as a URI list.

11.10 GET MODEL BY ID REQUIRED

As above, lists the required features of a Model identified by its ID. Required features are the independent features of the first transformation sub-model. More on transformation sub-models in the Algorithm section. The result is available as a URI list.

12. TASK

Most responses in the Jaqpot infrastructure return tasks. These are jobs executed asynchronously and in unique threads in the system. They are started by processes such as model building and

represented with unique IDs. Tasks allow users to monitor their processes. Three options are available under this API which are shown in Figure 41.

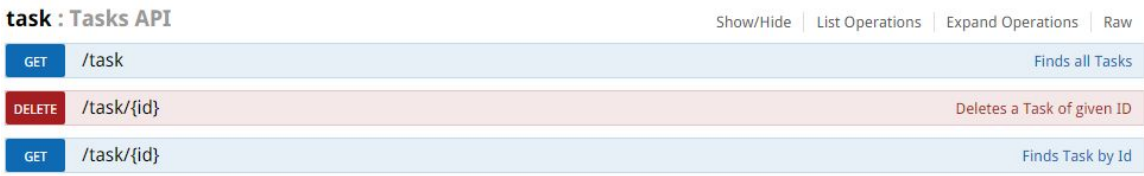

Figure 45: Options available under task API

12.1 GET TASK BY ID

By copying a task ID generated by a process and pasting it into the GET task by ID method, users may monitor the progress of their processes. Screenshot in Figure 46.

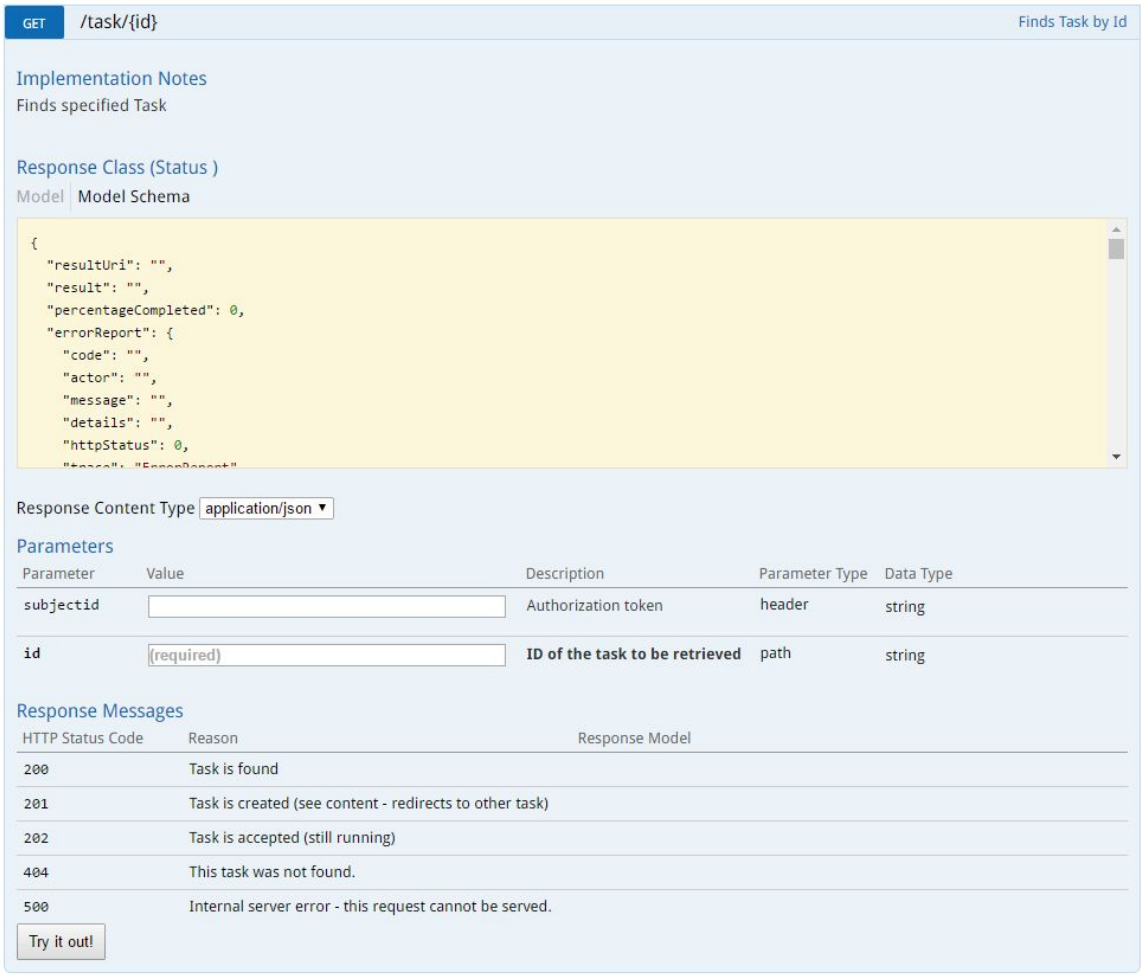

Figure 46: GET task by ID for monitoring the progress of initiated processes

12.2 DELETE TASK BY ID

Cancels a Task given its ID in the URI. When the DELETE method is applied, the task is interrupted and tagged as CANCELLED. Note that this method does not return a response on success. If the task does

not exist, an error report will be returned to the client accompanied by an HTTP status code 404. Note also that authentication and authorisation restrictions apply, so clients need to be authenticated with a valid token and have appropriate rights to be able to successfully apply this method. Screenshot in Figure 47.

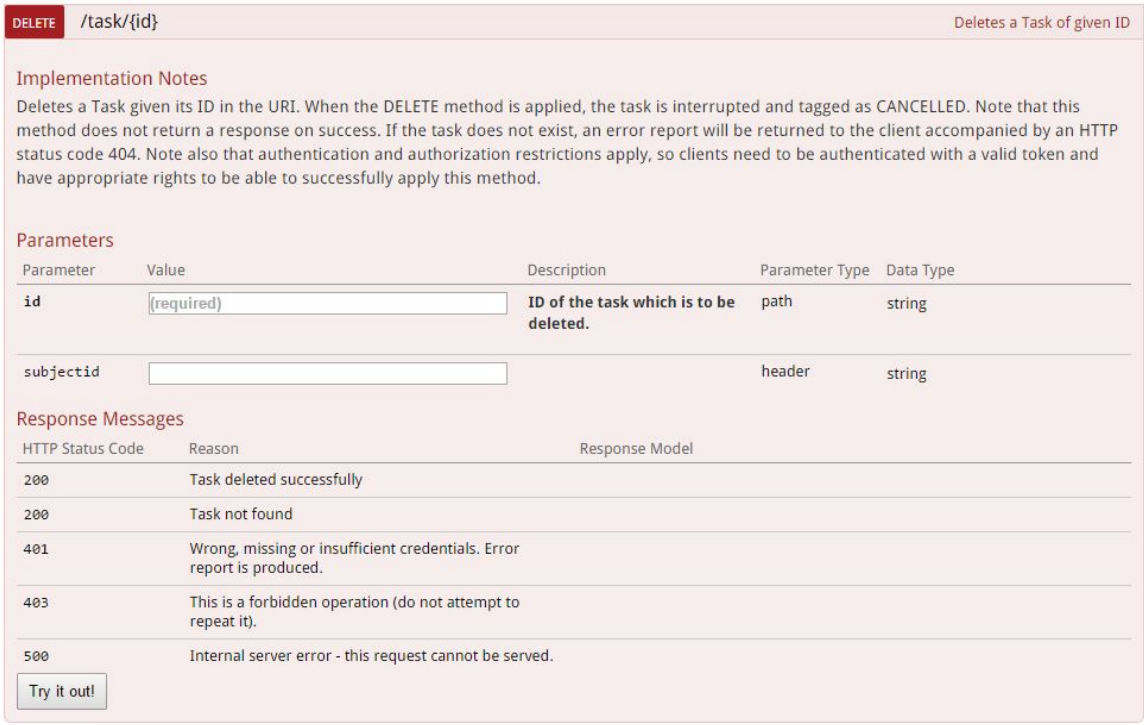

Figure 47: DELETE task by ID under task API

12.3 GET TASK

Returns all tasks from Jaqpot. As with GET methods described above, users may specify the number of tasks to be returned, but also the status of these tasks (running, cancelled etc.). Method is shown in Figure 48.

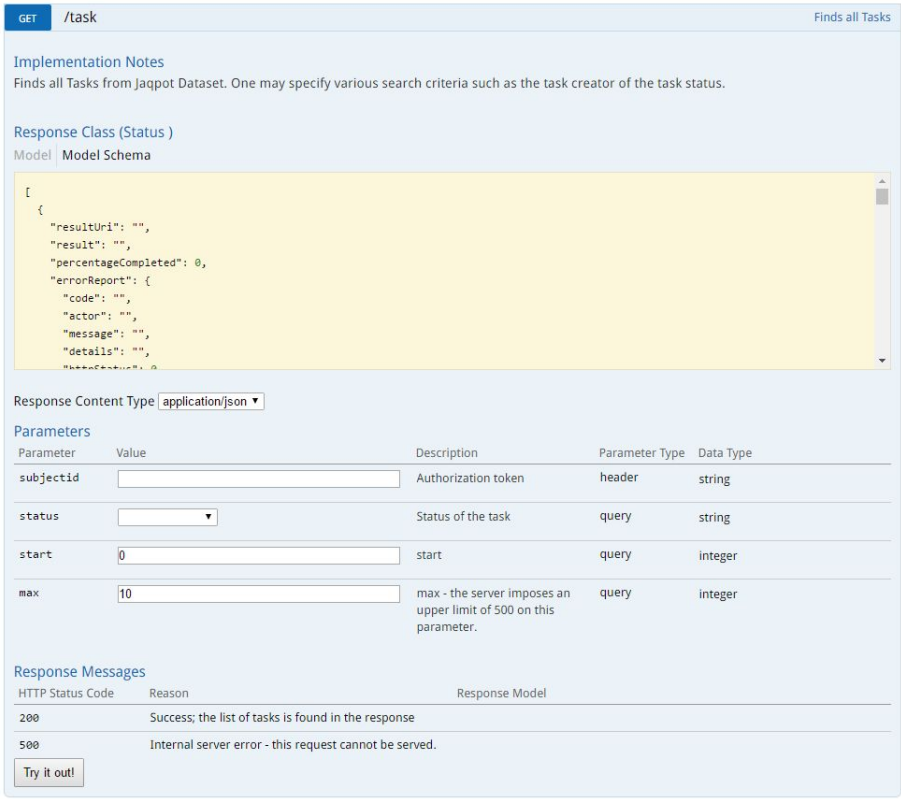

Figure 48: GET task method implemented in task API.

13. ALGORITHM

In this API users can retrieve algorithms, create algorithms, modify algorithm entries and build machine learning models. The options available are shown in Figure 49 and explained in detail later in this section.

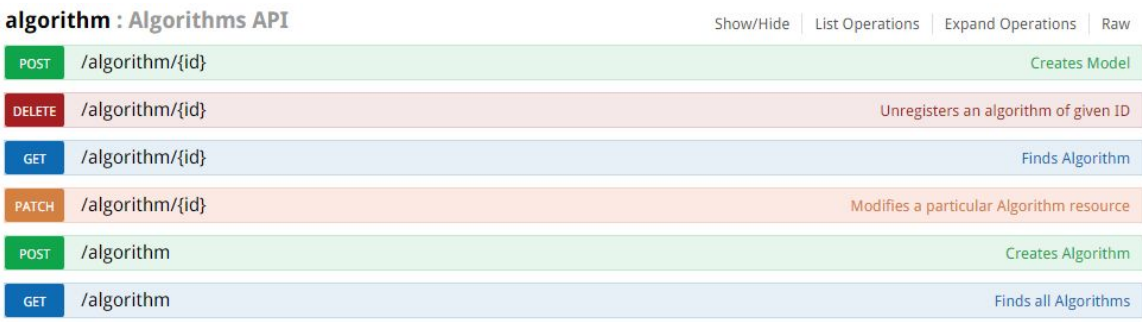

Figure 49: Options available under algorithm API

13.1 GET ALGORITHM BY ID

Users can retrieve a specified algorithm given its unique identifier. Screenshot in Figure 50.

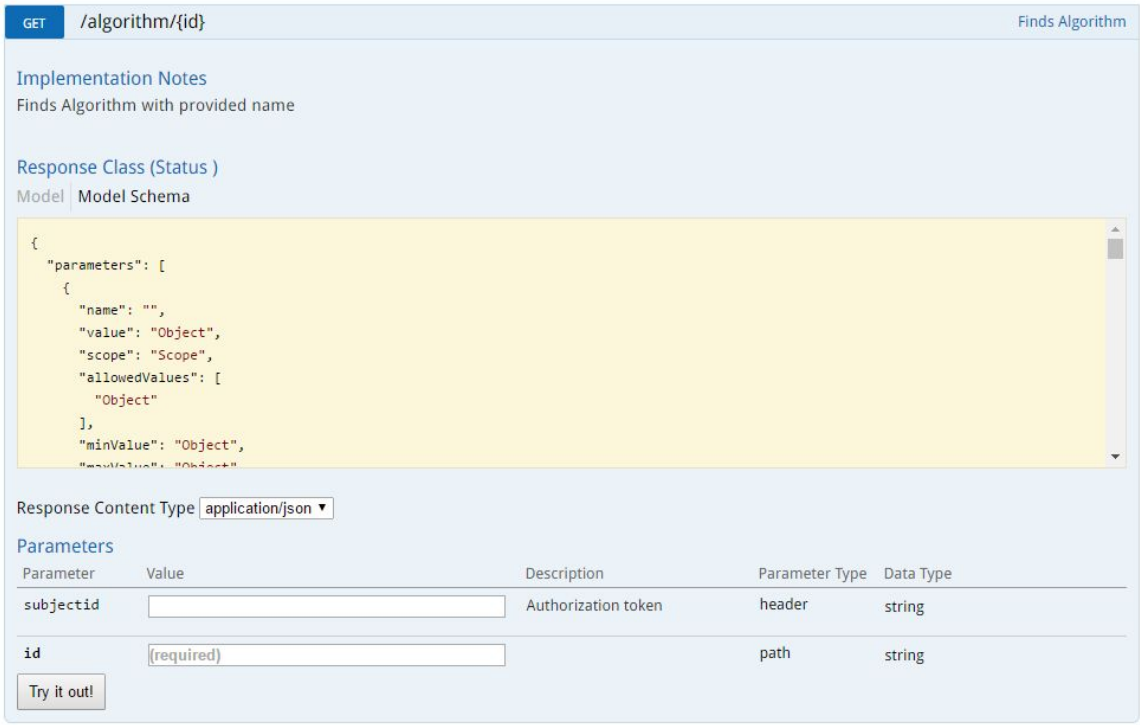

Figure 50: Screenshot of GET algorithm by ID

13.2 POST ALGORITHM BY ID

This method allows users to create machine learning models by supplying dataset and parameters (such as scaling and transformations) to particular algorithm given its ID. All parameters except *DoA* have been described in detail in the above sections.

The **parameters** field must be a JSON string with key/value pairs the algorithm parameters and the desired values. For example parameters for weka-pls algorithm would be: *{"algorithm":"SIMPLS", "components":10}*

The **transformations** field requires the URI of a PMML document. The dataset will be filtered through this PMML and transformed to a new dataset.

The **scaling** field requires the URI of a scaling algorithm. Scaling algorithms supported are http://test.jaqpot.org:8080/jaqpot/services/algorithm/scaling and http://test.jaqpot.org:8080/jaqpot/services/algorithm/standarization.

The former follows the formula:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

while the latter follows the formula:

$$x' = \frac{x - \bar{x}}{\sigma}$$

By selecting any of the transforming parameters, Jaqpot will create a transformation sub-model for each transformation and apply that model to the training dataset. Those models will be stored along with the final model. When a prediction is required from the model, the prediction dataset must be transformed as dictated by each transformation model before reaching the final model. Thus, in order for a model to work, the features required in the prediction dataset must be no other than the independent features of the first transformation sub-model.

DoA stands for domain of applicability and users may specify the method to be applied such as *leverage*, in order to see if their future predictions will be within the scope of the derived model. Screenshot in Figure 51.

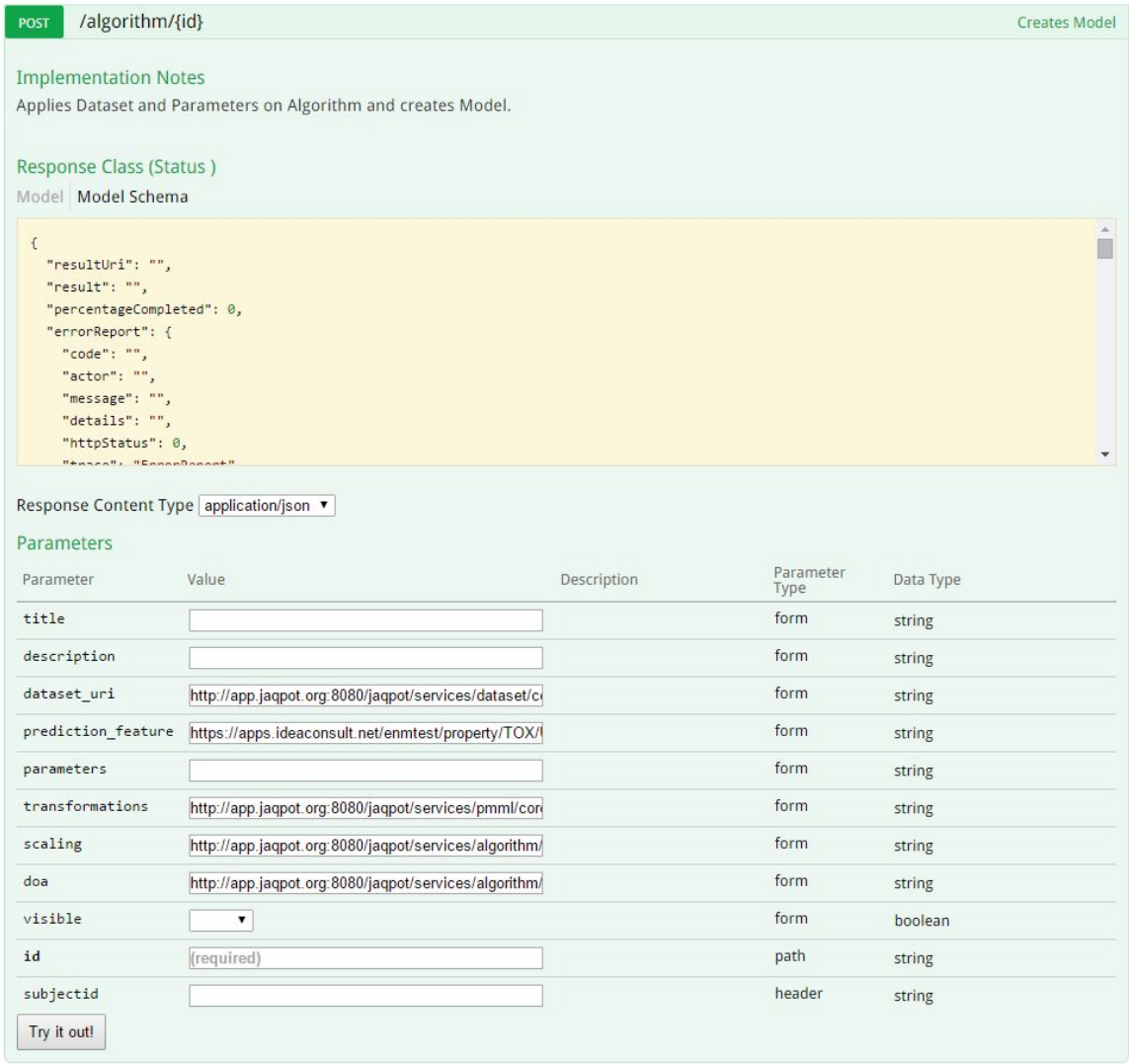

Figure 51: POST algorithm by ID for creating models under algorithm API

13.3 DELETE ALGORITHM BY ID

Users may delete an algorithm given its ID. authorization restrictions apply. Method and parameters shown in Figure 52.

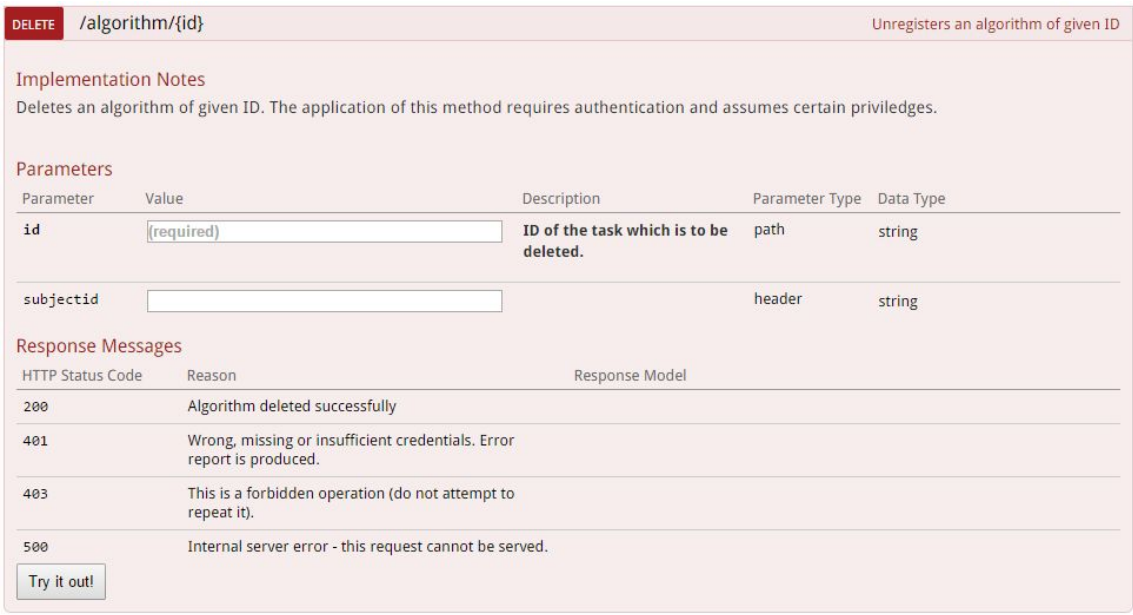

Figure 52: DELETE algorithm by ID method

13.4 PATCH ALGORITHM BY ID

Modifies (applies a patch on) an Algorithm resource of a given ID. This implementation of PATCH follows the RFC 6902 proposed standard. See https://tools.ietf.org/rfc/rfc6902.txt for details. Modification must be in JSON format. Illustrated in Figure 53.

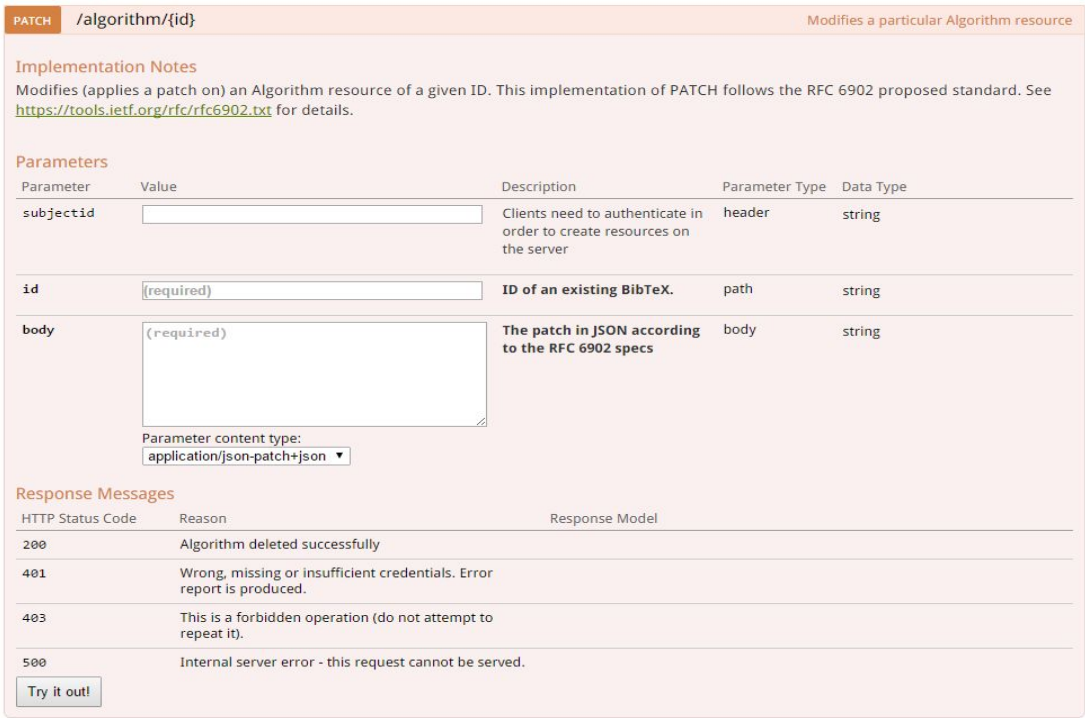

Figure 53: PATCH algorithm by ID screenshot.

13.5 GET ALGORITHM

Finds all algorithms in Jaqpot. Users may use start/max parameters described in the above sections or ontological class (clustering, regression or classification). Screenshot in Figure 54.

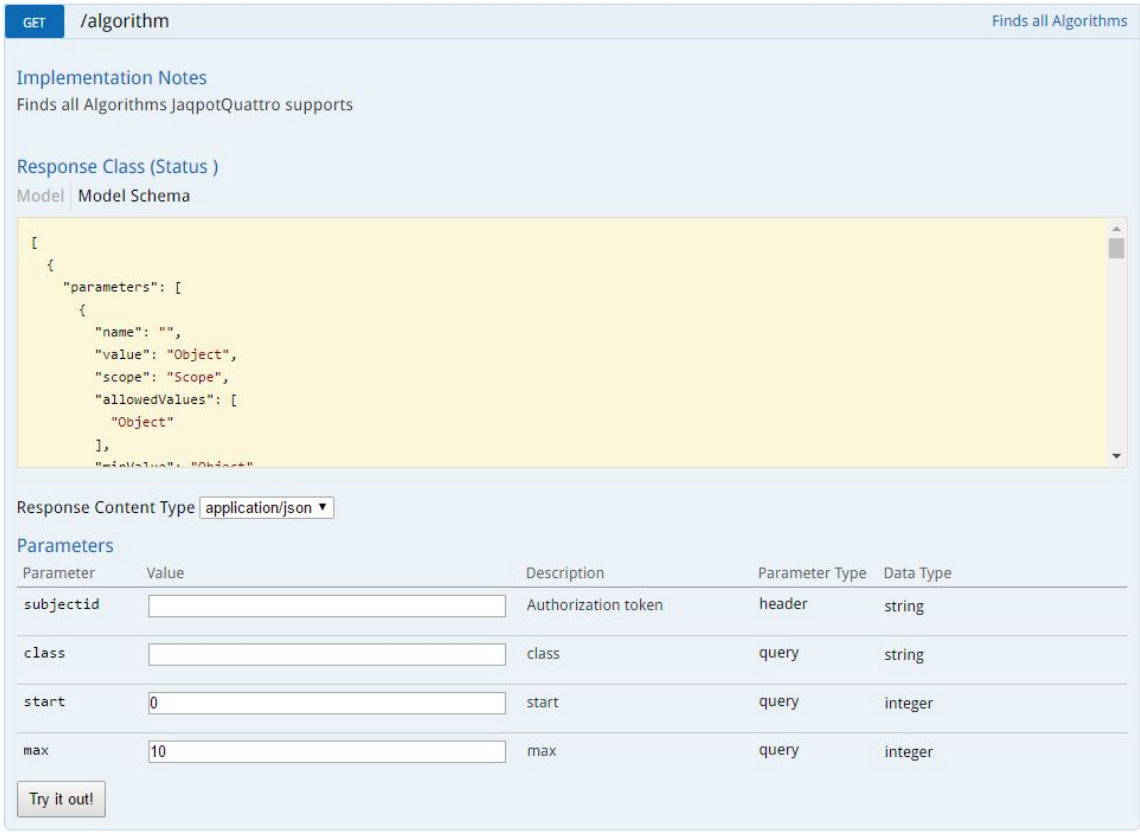

Figure 54: GET algorithm method in algorithm API

13.6 POST ALGORITHM

Registers a new JPDI-compliant algorithm service. When registering a new JPDI-compliant algorithm web service it is crucial for users to propertly annotate their algorithm with appropriate ontological classes following the OpenTox algorithms ontology.

For instance, a Clustering algorithm must be annotated with ot:Clustering. It is also important for discoverability to add tags to the algorithm using the meta.subjects field. An example is provided below. The *body* parameter is provided with a template for posting the algorithm, which must be in JSON format.

The remaining parameters title, description etc. have been discussed previously. Figure 55 shows a screenshot of the method.
Finally, templates for handling training and test/prediction requests in both python and R are provided in Appendix A.

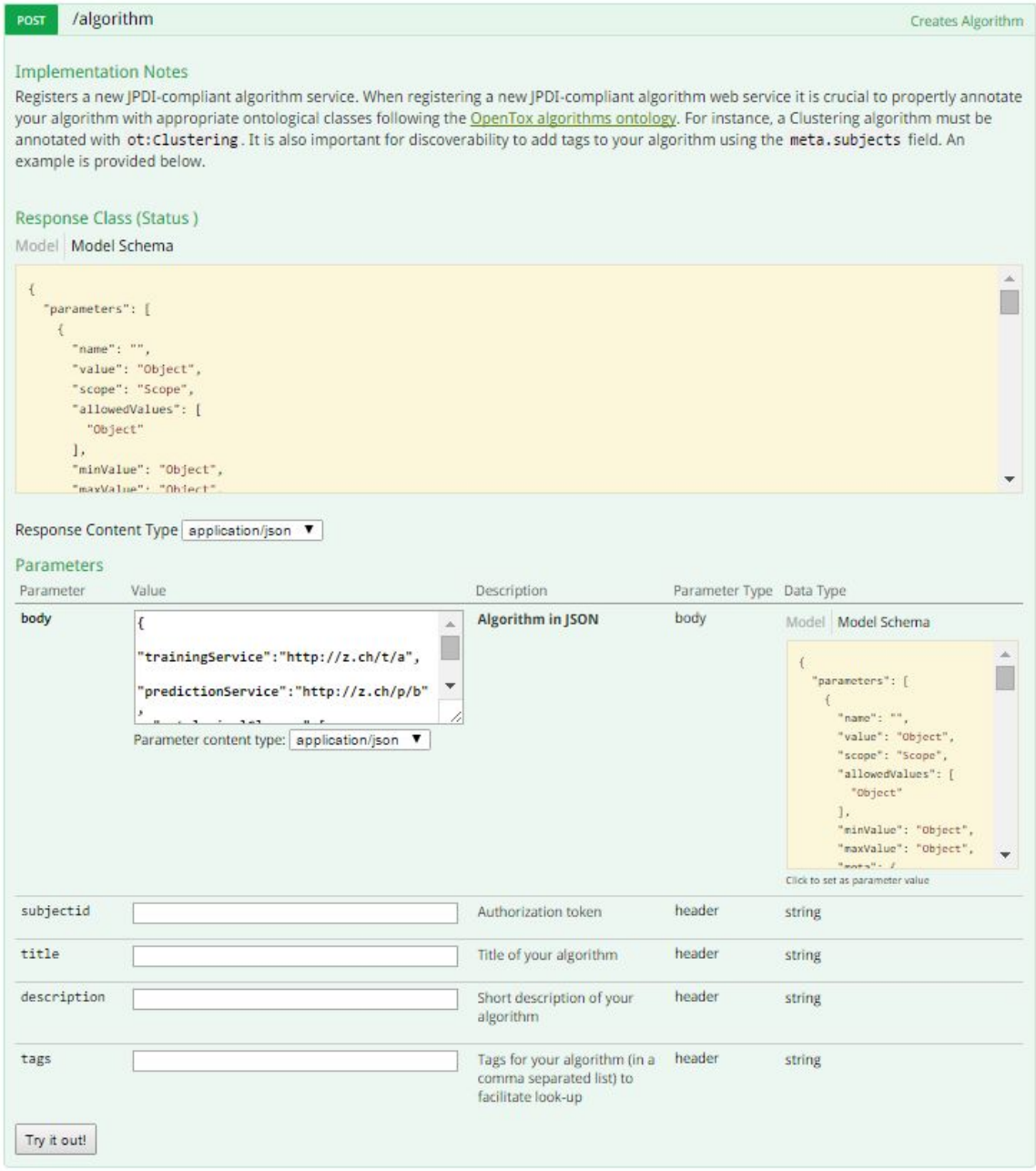

Figure 55: POST algorithm method for creating a new algorithm in Jaqpot

14. AA

This service allows users to login/logout and checks whether a token is valid, as well as whether a particular user given a token is authorised to proceed to use a particular URI (data or method). Main menu shown in Figure 56.

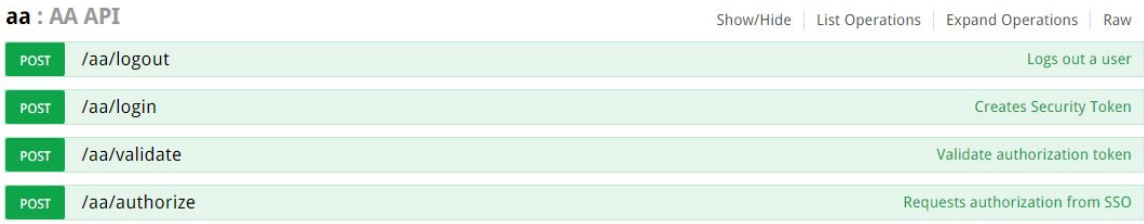

Figure 56: Options available under AA API

14.1 POST AA LOGIN

In order to use the eNanoMapper infrastracture a user must obtain a security token. In order to achieve this they should provide their credentials using this service in Figure 57.

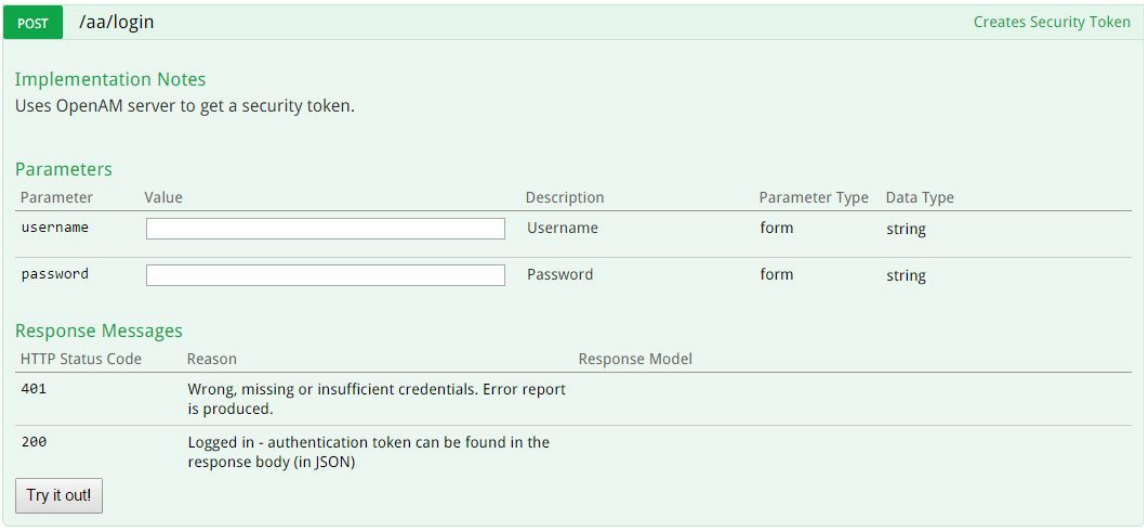

Figure 57: POST AA login method

14.2 POST AA LOGOUT

This method invalidates an active token and logs out its corresponding user, shown in Figure 58.

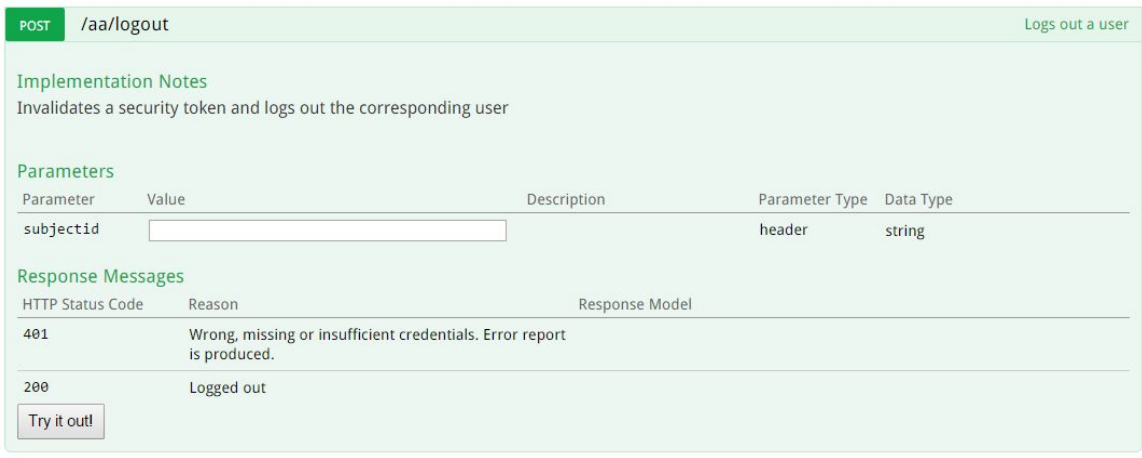

Figure 58: POST AA logout method

14.3 POST AA VALIDATE

As above. Checks whether an authorisation token is valid.

14.4 POST AA AUTHORIZE

This method assesses if the user corresponding to the provided token is authorised to apply a method to a particular URI. The method parameters are detailed in Figure 59.

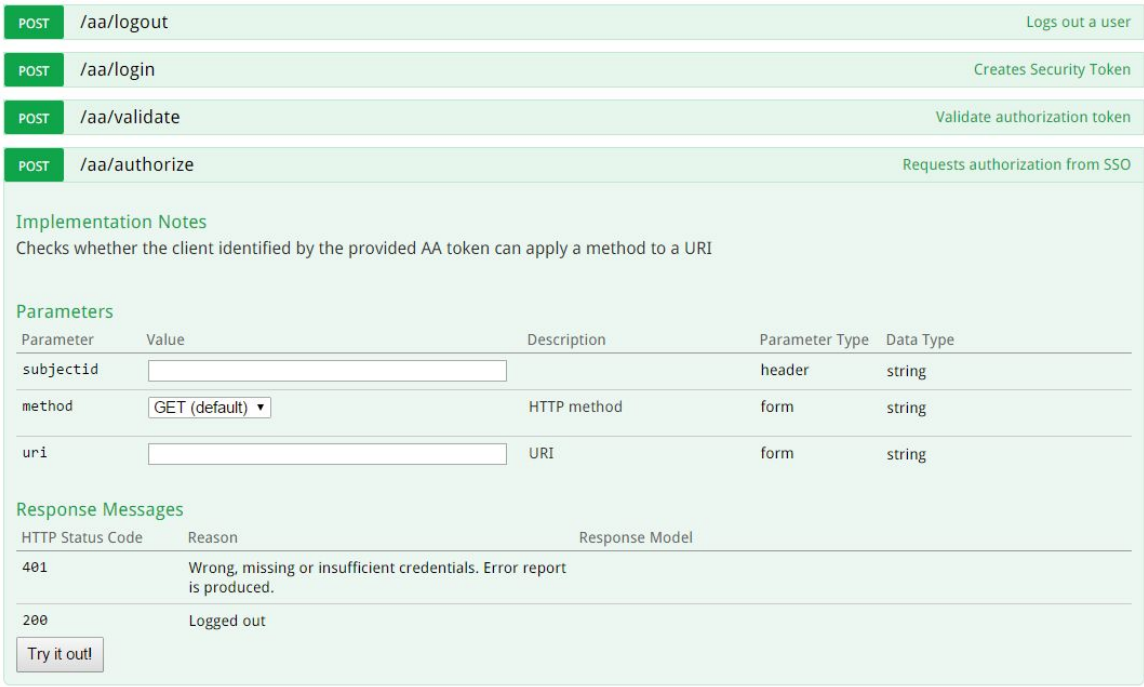

Figure 59: POST AA authorize method

15. FEATURE

The Feature API allows a user to create, edit, find and delete nanoparticle properties located in the Jaqpot database. More specifically, users may create a new feature using POST, locate a single existing feature in the Jaqpot database using its unique ID (URI) or list all available features using GET, modify an existing URI using PUT and finally delete a particular feature using its uique ID under DELETE. Menu shown in Figure 60.

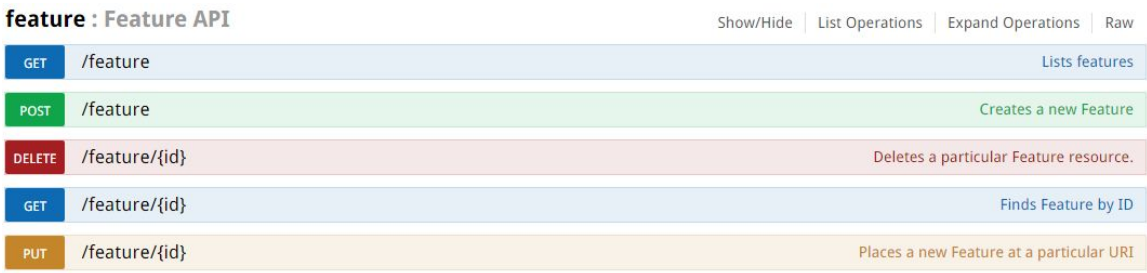

Figure 60: Available options under Feature API

15.1 GET FEATURE

Lists Feature entries in the DB of Jaqpot and returns them in a list. Results can be obtained either in the form of a URI list or as a JSON list as specified by the Accept HTTP header. In the latter case, a list will be returned containing only the IDs of the features, their metadata and their ontological classes. The parameter max, which specifies the maximum number of IDs to be listed is limited to 500; if the client specifies a larger value, an HTTP Warning Header will be returned (RFC 2616) with code P670. Users may also use generic queries. Screenshot in Figure 61.

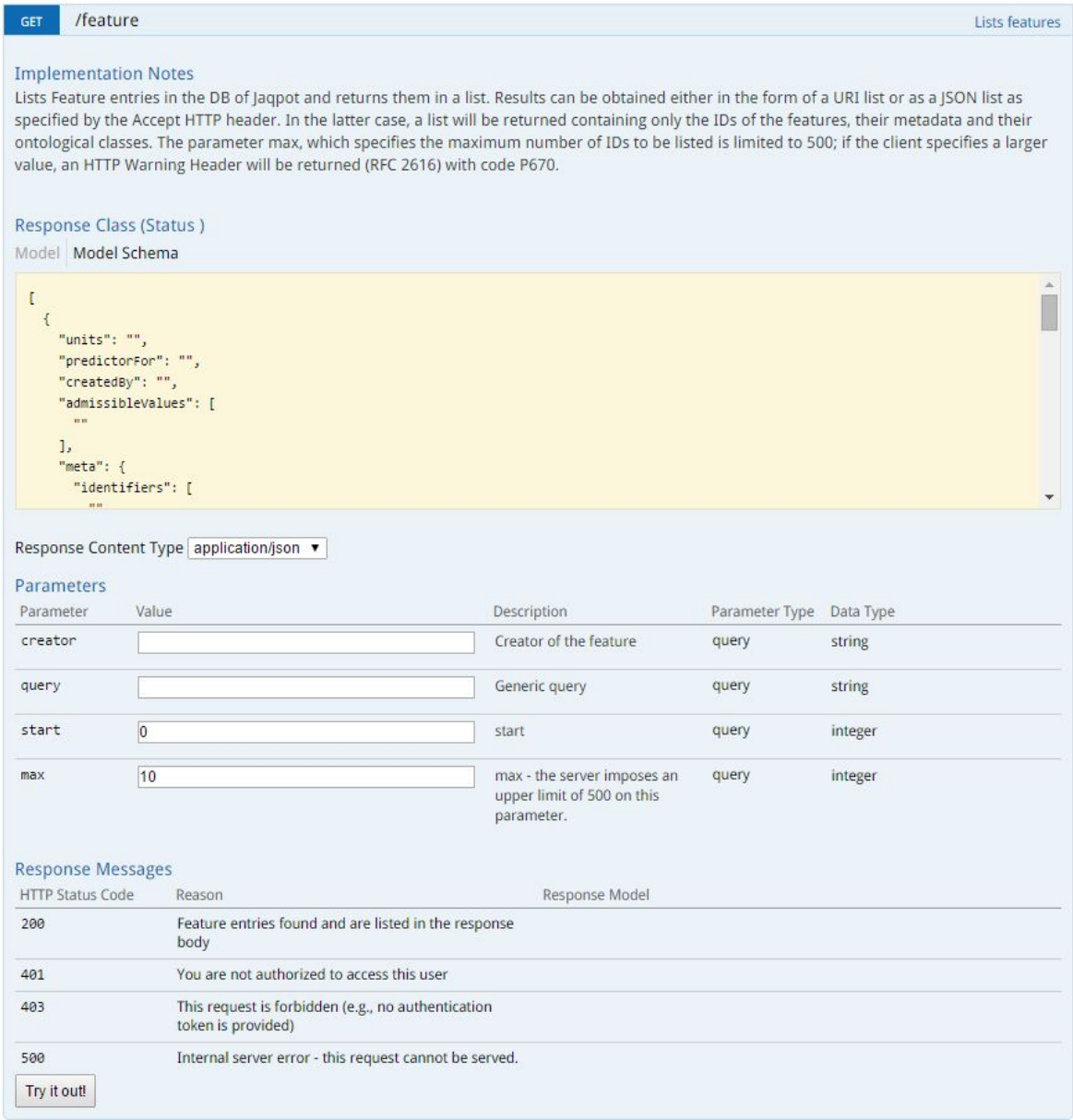

Figure 61: GET feature method under feature API

15.2 POST FEATURE BY ID

Creates a new feature which is assigned a random unique ID. When creating a new feature, clients must wary not only for its syntactic correctness, but also for its semantic completeness. It is strongly recommended to add a comprehensive and identifying title to your feature using the meta.titles field, to add a description in meta.descriptions and also to add a list of tags in meta.subjects that will facilitate the discoverability of your features later. Additionally, all features should be annotated with appropriate ontological classes (from the OpenTox ontology), such as ot:Feature, ot:NumericFeature and ot:NominalFeature. Features that are created as prediction features for a model or are descriptors that can be calculated using a descriptor calculation web service should be linked to this/these service(s) using meta.hasSources. Finally, nominal features should define their admissible values in admissibleValues. Malformed feature documents will not be accepted by the server and an

error report will be generated and returned to the client. Notice also that authentication, authorisation and accounting restrictions may apply. Show in Figure 62.

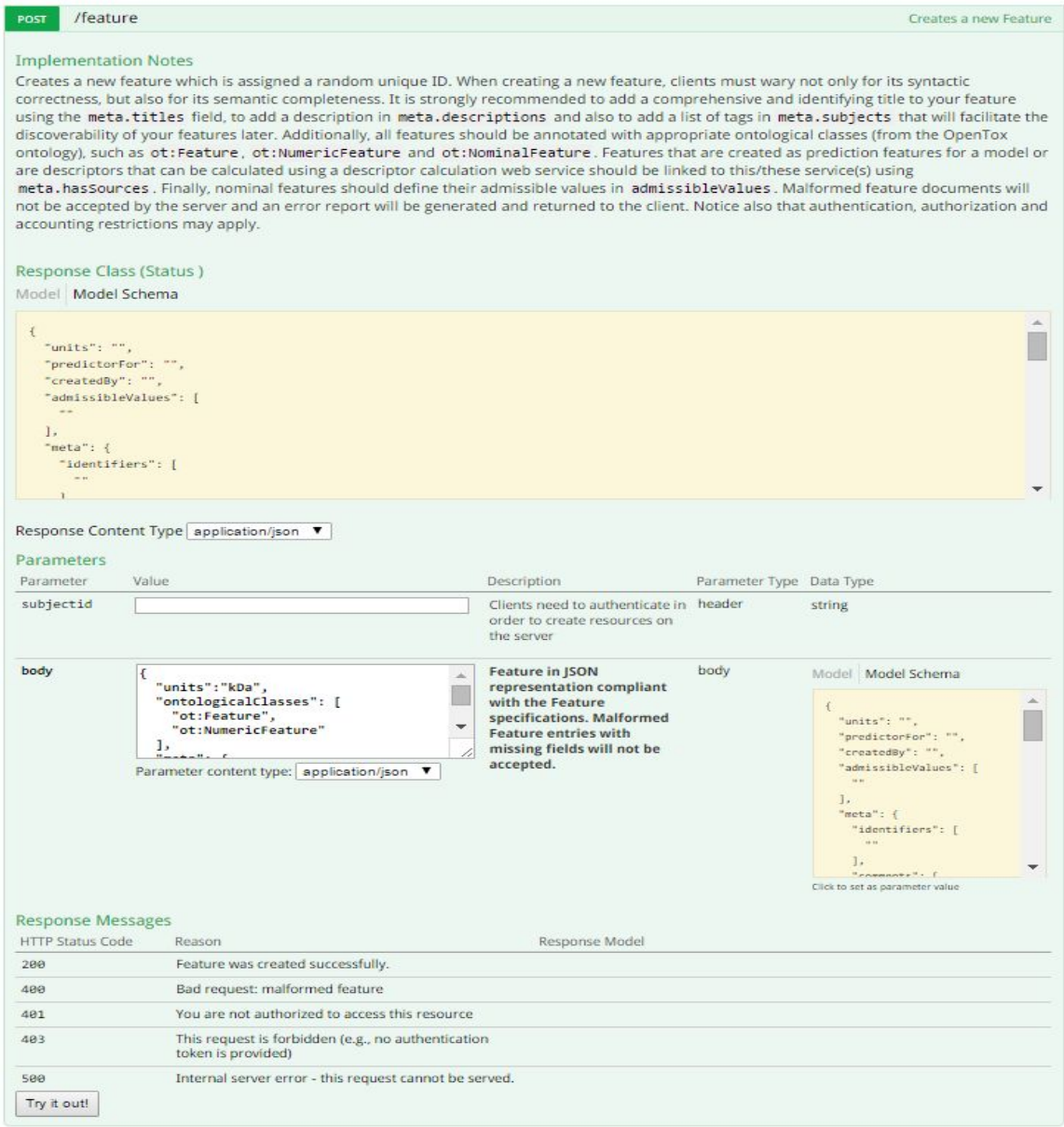

Figure 62: POST feature method for creating a new property in the feature API

15.3 DELETE FEATURE BY ID

Deletes a Feature of a given ID. The method is idempotent, that is, it can be used more than once without triggering an exception/error. If the Feature does not exist, the method will return without errors. Authentication and authorisation requirements apply, so clients that are not authenticated with a valid token or do not have sufficient privileges will not be able to delete a Feature using this method. Screenshot in Figure 63.

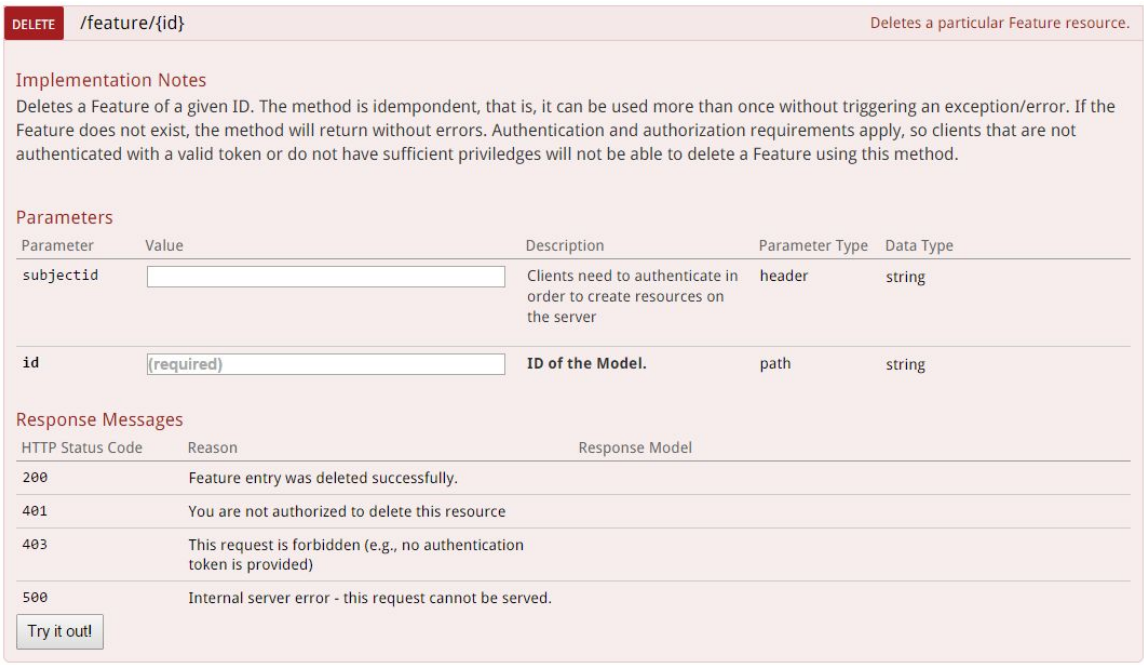

Figure 63: DELETE feature by ID service

15.4 GET FEATURE BY ID

As GET methods described in previous sections, returns feature given its ID.

15.5 PUT FEATURE BY ID

Creates a new Feature entry at the specified URI. If a Feature already exists at this URI, it will be replaced. If, instead, no Feature is stored under the specified URI, a new Feature entry will be created. Notice that authentication, authorisation and accounting (quota) restrictions may apply. User needs to provide the body parameter with a valid JSON according to the template. Screenshot in Figure 64.

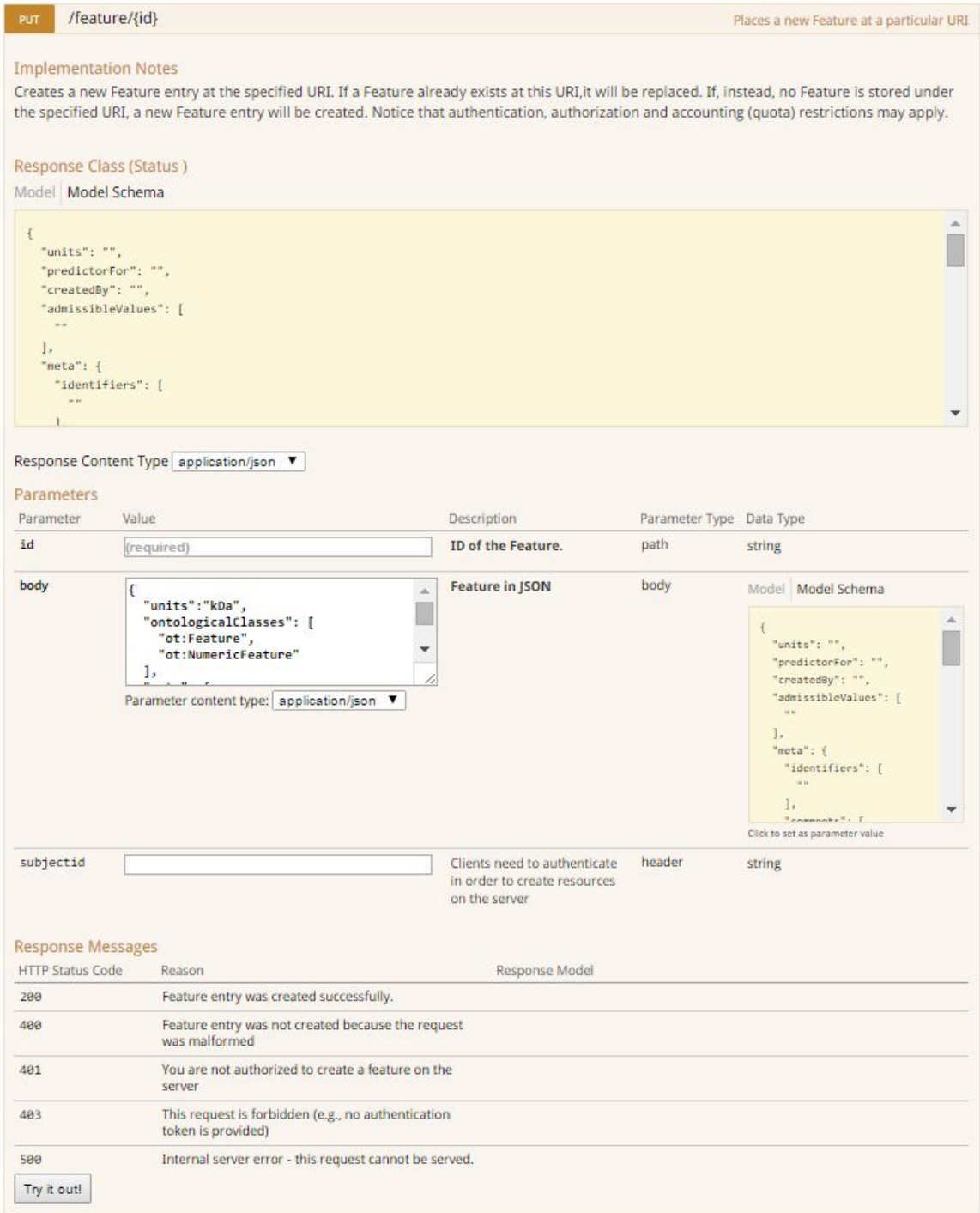

Figure 64: PUT feature by ID

16. USER

The User API consists of three GET methods, one for finding all users, one for finding a single user and another for checking the quota of a particular user. Available methods shown in Figure 65.

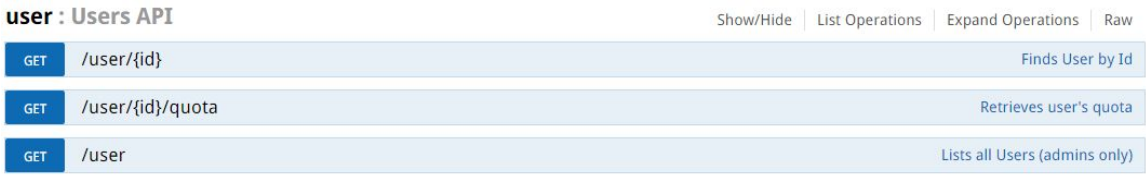

Figure 65: GET methods available for user API - GET user, user by ID and user quota by ID.

16.1 GET USER BY ID QUOTA

Returns user's quota given the user's ID. Authenicated users can access only their own quota. Jaqpot administrators can access the quota of all Jaqpot users. Screenshot in Figure 66.

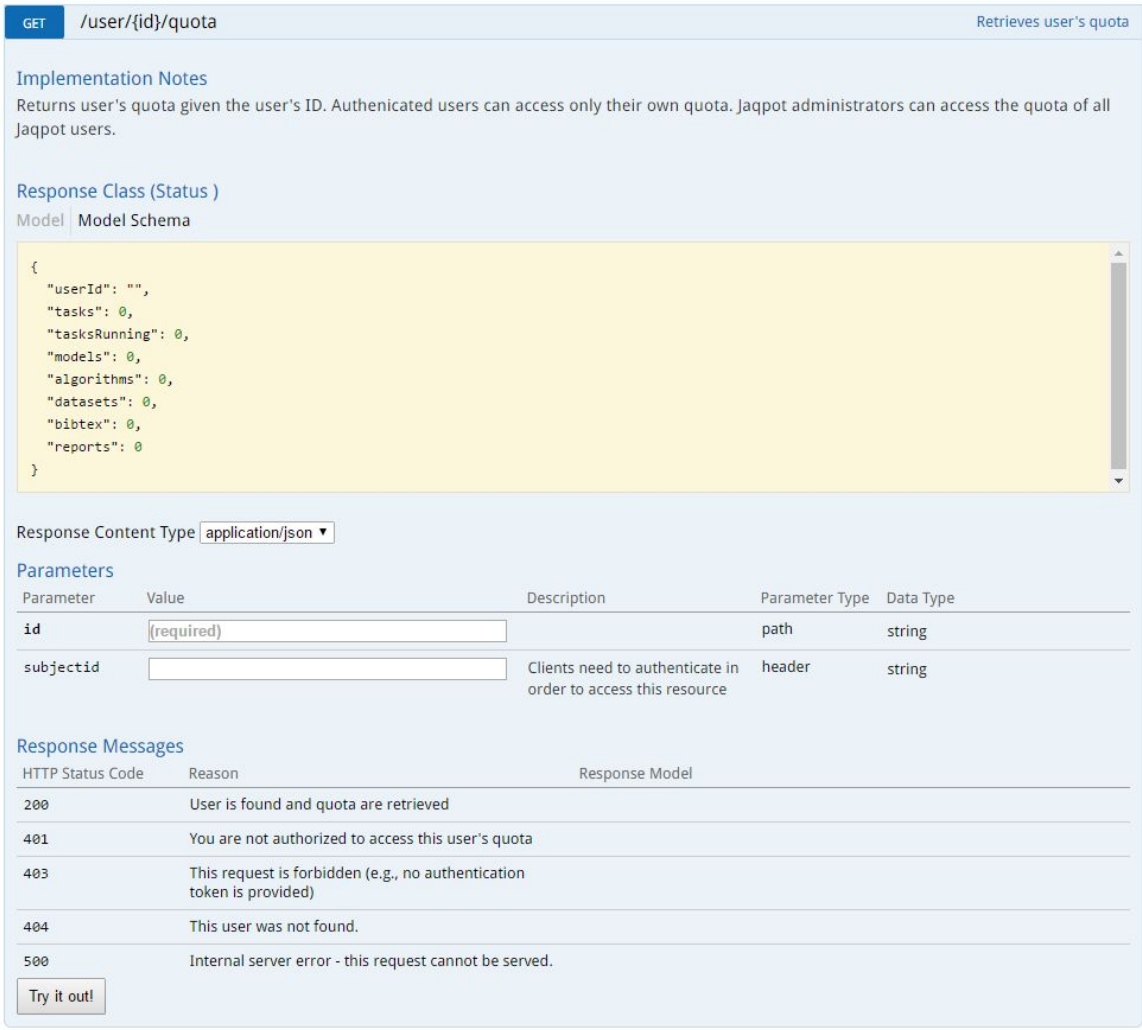

Figure 66: GET user quota by ID web service.

16.2 GET USER BY ID

Returns details of user given his/her unique identifier. Screenshot in Figure 67.

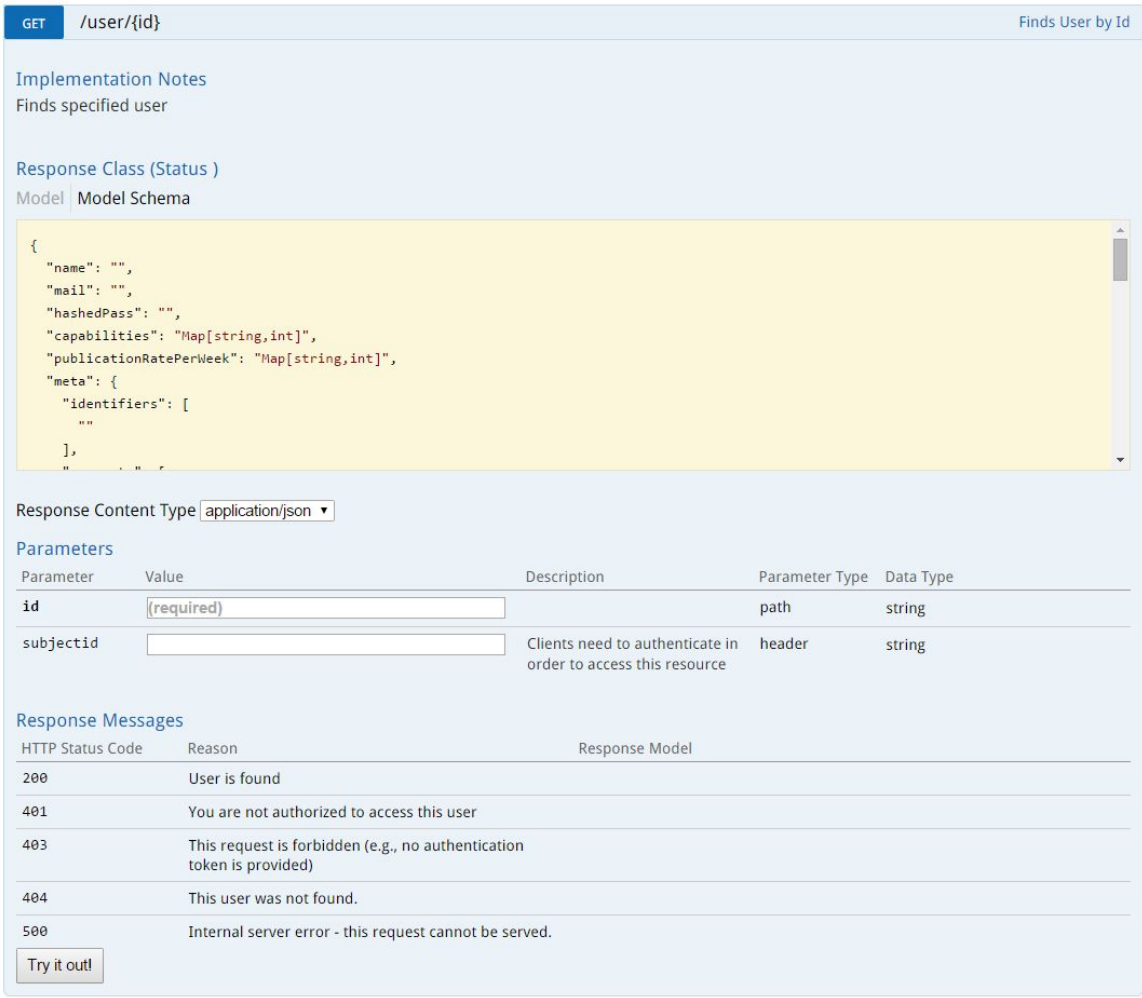

Figure 67: GET user by ID method

16.3 GET USER

Lists all Users of Jaqpot. This operation can only be performed by the system administrators. Shown in Figure 68.

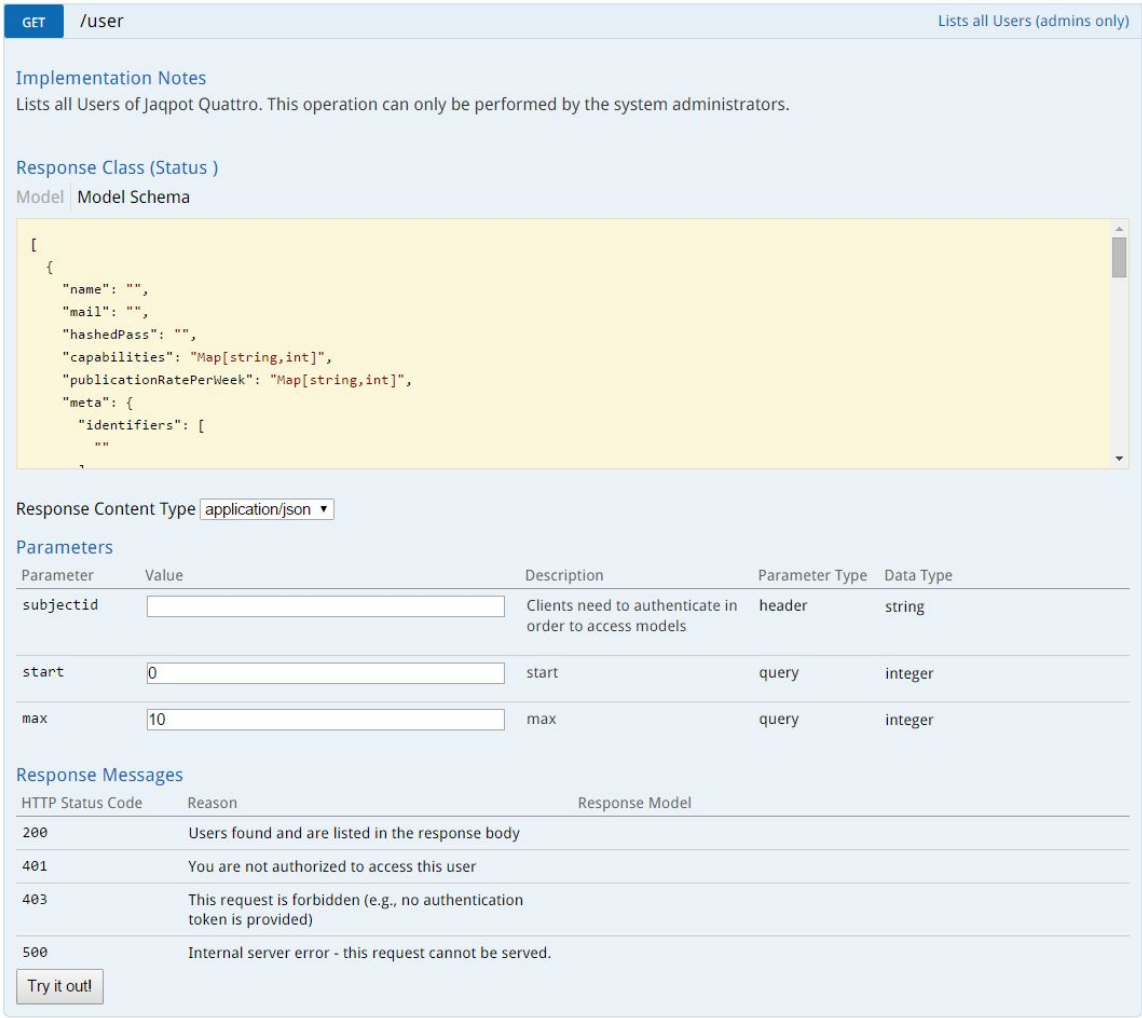

Figure 68: GET all users screenshot under user API

17. ACKNOWLEDGMENTS

The eNanoMapper project is funded by the European Union's Seventh Framework Program for research, echnological development and demonstration (FP7NMP2013SMALL7) under grant agreement No. 604134.

18. REFERENCES

- Walkey et al., Protein Corona Fingerprinting Predicts the Cellular Interaction of Gold and Silver Nanoparticles, *ACS Nano* **8** (3), 2439-2455 (2014)

19. KEYWORDS

NanoQSAR modelling, data mining, model validation, Application programming interface, API

20. APPENDIX A

```
# All JaqpotQuattro opencpu R packages need to include two functions: a model function where the model is built and
# a prediction function where predictions are made. Obviously the latter's input is the former's output.

# Include all required packages, 'jsonlite' and 'RCurl' are mandatory.

require(jsonlite)
require(RCurl)
require(pmml)

# Read-in data for the model-function.
#
# Data should be a list with three slots, i.e. "dataset", "predictionFeature", "parameters"
# dataset should be a list with slots "datasetURI" and "dataEntry"; the data are stored in dataEntry, a data.frame of 2 data.frame objects,
# where dataEntry[,1] is a data.frame of (number of compounds) x 1 storing the names of the compounds and
# dataEntry[,2] is a data.frame of (number of compounds) x (number of features) storing the data values
# parameters is again a list including all parameter values expected by the algorithm, e.g.
# parameters=list(nTrials=c(11),form='linear',r2.threshold=0.8)
#
#dat2<- list(dataset=<dataset>,predictionFeature=<Name or URL of the predictionFeature>,parameters=<list of parameters>)
#sink("training_experimental_design.json")
#cat(toJSON(dat2))
#sink()

#dat4<- fromJSON('training_expDesign.json')
#dat1<- dat4$dataset
#dat1p<- dat6$dataset
#save(dat1,file='dat1.rda')
```

```
# Include any auxiliary R functions here.
#

r2.funct.lm<- function(y,y.new){#y==y, y.new=predicted
#only for lm with intercept
        x.in<- cor(y,y.new)^2
        return(x.in)
}

# Model - function

exp.design.xy<- function(dataset,predictionFeature,parameters){
        #dataset:= list of 2 objects -
        #datasetURI:= character sring, code name of dataset
        #dataEntry:= data.frame with 2 columns,
        #1st:name of compound,2nd:data.frame with values (colnames are feature names)
        #predictionFeature:= character string specifying which is the prediction feature in
dataEntry,
        #parameters:= list with parameter values-- here awaits a list with four elemnts:
        #nTrials (a numeric value indicating number of trials suggested, if 0 then an estimated
number is suggested),
        #criterion (a character value to indicate which optimal deisgn to apply. Possible values are
'D', 'A', 'I'. Default is 'D'),
        #form (a string indicating the formula of the deisgn- 'linear','quad','cubic','cubicS')),
        #r2.threshold (numeric value indicating the r2 threshold value. If the data supplied
provides r2 value greater
        #than the threshold value, a stop message is returned.).

        dat<- dataset$dataEntry[,2] # data table

        dat1.yind<- predictionFeature # string to indicate dependent variable

        depend.variable<- which(colnames(dat) %in% dat1.yind)

        ind.dat<- colnames(dat)

        <THE BODY OF YOUR FUNCTION GOES HERE>

        # if prediction features are created (a new column in the data) then they need to be named
        pred.name<- c('suggestedTrials',parameters$newY)

        # serialized raw model -anything that needs to be called by the prediction function below
(suggest.trials.xy)
        suggested.trials.ser<- serialize(list(Trials=suggested.trials),connection=NULL)

        desD.res<- list(design=desD.int$design,selected.rows=desD.int$rows,
        norm.var=desD.int$Ge,confounding.effect=desD.eval$diagonality,
        r.squared=ifelse(length(depend.variable)!=0,r2,NA),
```

```
        adj.r.squared=ifelse(length(depend.variable)!=0,adj.r2,NA),
        verbal.notes=verbal.in,predictedFeatures=pred.name)#or 'NA'?

        # the final result is a list as seen below.
        #
        # additionalInfo is the place to include any information possibly useful to the user
        # at this stage, its only mandatory to the prediction function below is the
additionalInfo$predictedFeatures that must be included.
        m1.ser.list<-
list(rawModel=suggested.trials.ser,pmmlModel=NULL,independentFeatures=ind.dat,
        predictedFeatures=pred.name,#NULL,
        additionalInfo=desD.res)


   return(m1.ser.list)

}

# Prediction - function
# Input data for the prediction-function:
# dataset (original data as in model-function), rawModel (as returned by the model-function),
additionalInfo (as returned by the model-function)
# In the example below, a vector (sug.trials) is returned as a list of lists. The returned list should
have a slot called 'predictions' where
# the list of lists is stored.

suggest.trials.xy<- function(dataset,rawModel,additionalInfo){
        #dataset:= list of 2 objects -
        #datasetURI:= character sring, code name of dataset
        #dataEntry:= data.frame with 2 columns,
        #1st:name of compound,2nd:data.frame with values (colnames are feature names)
        #rawModel:= numeric vector showing experimental design results
        #additionalInfo:= list with summary statistics, returns design matrix with
        #suggested trials for y (whether or not y was originally supplied)

        dat1.m<- rawModel
        dat1.m<- base64Decode(dat1.m,'raw')
        sug.trials<- unserialize(dat1.m)

        sug.trials<- sug.trials$Trials

        sug.name<- additionalInfo$predictedFeatures

        for(i in 1:length(sug.trials)){
                if(length(sug.name)>1){w1<- data.frame(sug.trials[i],NA)}else{w1<-
data.frame(sug.trials[i])}
                colnames(w1)<- sug.name
                if(i==1){p7.1<- list(unbox(w1))
```

```
                }else{
                        p7.1[[i]]<- unbox(w1)
                }
        }
        p7.2<- list(predictions=p7.1)


        return(p7.2)
}

# Use package.skeleton to wrap-up the R package and proceed with building/testing the package.
#package.skeleton(list=c('exp.design.xy','suggest.trials.xy','dat1','dat1p','r2.funct'),name='ExpDesig
nPkg')
```

Supplementary Figure 1. R template for handling Jaqpot request

```
#!flask/bin/python

from flask import Flask, jsonify, abort, request, make_response, url_for
# import json, pickle, base64
# suggested imports: numpy, scipy, sklearn, math

app = Flask(__name__, static_url_path = "")

def getJsonContentsTrain (jsonInput):
   # User must look at JSON request for the required fields.
   # Here is an example for training dataset.
   # The same goes for testing - see next function.
   try:
      dataset = jsonInput["dataset"]
      predictionFeature = jsonInput["predictionFeature"]
      parameters = jsonInput["parameters"]
      datasetURI = dataset.get("datasetURI", None)
      dataEntry = dataset.get("dataEntry", None)
      variables = dataEntry[0]["values"].keys()
      variables.sort()
      datapoints =[]
      target_variable_values = []
      for i in range(len(dataEntry)):
         datapoints.append([])

      for i in range(len(dataEntry)):
         for j in variables:
            if j == predictionFeature:
               target_variable_values.append(dataEntry[i]["values"].get(j))
```

```
            else:
                datapoints[i].append(dataEntry[i]["values"].get(j))
        variables.remove(predictionFeature)

    except(ValueError, KeyError, TypeError):
        print "Error: Please check JSON syntax... \n"
    return variables, datapoints, predictionFeature, target_variable_values, parameters

def getJsonContentsTest (jsonInput):
    try:
        dataset = jsonInput["dataset"]
        rawModel = jsonInput["rawModel"]
        additionalInfo = jsonInput["additionalInfo"]
        datasetURI = dataset.get("datasetURI", None)
        dataEntry = dataset.get("dataEntry", None)
        predictionFeature = additionalInfo[0].get("predictedFeature", None)
        variables = dataEntry[0]["values"].keys()
        variables.sort()
        datapoints =[]
        for i in range(len(dataEntry)):
            datapoints.append([])
        for i in range(len(dataEntry)):
            for j in variables:
                datapoints[i].append(dataEntry[i]["values"].get(j))
    except(ValueError, KeyError, TypeError):
        print "Error: Please check JSON syntax... \n"
    return variables, datapoints, predictionFeature, rawModel

def my_training_function("""NECESSARY INPUT"""):
    # do calculations here
    # encode model in base64
    return my_model_in_base64_format

def my_test_function("""NECESSARY INPUT"""):
    # decode model from base64
    # do calculations here
    return my_predictions

@app.route('/.../myAlgorithm/train', methods = ['POST'])
def create_task_myAlgorithm_train():
    # copied to body for handling chunked input
    if not request.environ['body_copy']:
        abort(500)
    readThis = json.loads(request.environ['body_copy'])
    variables, datapoints, predictionFeature, target_variable_values, parameters =
getJsonContentsTrain(readThis)
    my_model_in_base64_format = my_training_function("""NECESSARY INPUT""")
    predictedString = predictionFeature + " predicted"
```

```
    task = {
        "rawModel": my_model_in_base64_format,
        "pmmlModel": "IF applicable",
        "additionalInfo" : [{'predictedFeature': predictedString}],
        "independentFeatures": variables,
        "predictedFeatures": [predictedString]
    }
    jsonOutput = jsonify( task )
    return jsonOutput, 201

@app.route('/.../myAlgorithm/test', methods = ['POST'])
def create_task_myAlgorithm_test():
    if not request.environ['body_copy']:
        abort(500)
    readThis = json.loads(request.environ['body_copy'])
    variables, datapoints, predictionFeature, rawModel = getJsonContentsTest(readThis)
    my_predictions = my_test_function("""NECESSARY INPUT""")
    task = {
        "predictions": my_predictions
    }
    jsonOutput = jsonify( task )
    return jsonOutput, 201

# Middleware for chunked input
#from
http://stackoverflow.com/questions/14146824/flask-and-transfer-encoding-chunked/21342631
class WSGICopyBody(object):
    def __init__(self, application):
        self.application = application

    def __call__(self, environ, start_response):
        from cStringIO import StringIO
        input = environ.get('wsgi.input')
        length = environ.get('CONTENT_LENGTH', '0')
        length = 0 if length == '' else int(length)
        body = ''
        if length == 0:
            environ['body_copy'] = ''
            if input is None:
                return
            if environ.get('HTTP_TRANSFER_ENCODING','0') == 'chunked':
                size = int(input.readline(),16)
                while size > 0:
                    temp = str(input.read(size+2)).strip()
                    body += temp
                    size = int(input.readline(),16)
        else:
            body = environ['wsgi.input'].read(length)
```

```
        environ['body_copy'] = body
        environ['wsgi.input'] = StringIO(body)

        app_iter = self.application(environ, self._sr_callback(start_response))

        return app_iter

    def _sr_callback(self, start_response):
        def callback(status, headers, exc_info=None):
            start_response(status, headers, exc_info)
        return callback

if __name__ == '__main__':
    app.wsgi_app = WSGICopyBody(app.wsgi_app)
    app.run(host="0.0.0.0", port = 5000, debug = True)
```

Supplementary Figure 2. Python example template for handling Jaqpot request