**PDS JavaScript API Programming Guide**

**1. Introduction.**

There are several ways to get access to the data provided by the NASA's Planetary Data System (PDS). One of them is by using web service developed by TopCoder. This document describes *the JavaScript API to the PDS web service*.

To get access to the PDS functionality the following JavaScript source files should be added to the web page using <script> html tag:

* **cxf-utils.js**
* **PlanetaryDataSystemService.js**
* **Common.js**

In the provided distribution they are located in the ***js-api***directory.

The basic algorithm to start with PDS in your JavaScript:

*Step1*. Create pds object:

var PDS = new PlanetaryDataSystem()

*Step2*. Set url property of the PDS object (it's an url of the web service):

PDS.url = WEB\_SERVICE\_URL

NOTE: WEB\_SERVICE\_URL constant is defined in Common.js

*Step3*. Invoke the operation of interest on the PDS object:

PDS.getMissionsInfo(onSuccess, onError, page)

NOTE: This example shows getMissionsInfo function call, the API reference for *all* operations from the PDS object are provided further in this document.

Unless the operation is **one way** (i.e. there is no return value), the function will take two callback parameters, plus the parameters for the operation itself. The first two parameters are the success callback and the error callback. OneWay operations have no callbacks; they are "fire and forget."

The success callback is called back with one parameter: the JavaScript object corresponding to the response. The demos provided with the distribution and the *operations reference* describe how the response object, which contains all the data returned by the operation, should be used.

**2. Entities types and operations overview.**

The data in PDS are stored in datasets. The dataset from the physical standpoint is just a directory with files. Some of these files are used to describe the structure of the dataset itself (metadata) and the others are the 'real data' (like images and tables).

The PDS interface defines the following entities (objects) that describe metadata/data stored in dataset:

* **TargetType** - each target belongs to one or more target types. It's just a way to categorize targets.
* **Target** - in most cases target is a space object (like planet, comet, etc.) for which data was collected and presented as dataset.
* **Mission -** describes the scientific/commercial mission for which the dataset was created.
* **Instrument -** describes the scientific instrument which was used to collect/measure data for the dataset.
* **DataSet** - the object that represents the entire dataset. It contains references to other entities like Targets, Missions, Instruments, DataFiles.
* **DataFile** - represents the ascii/binary files from the dataset. This file is either a file that contains dataset's data (for example, image file) or a file with some auxiliary data (for example, readme file with some notes).

The operations provided by the PDS JavaScript interface can be grouped based on type of the entity they work with. Also there are search operations which works with entities of all types.

TargetType operations:

* **getTargetTypesInfo**
* **getTargetType**

Target operations:

* **getTargetsInfo**
* **getTarget**

Mission operations:

* **getMissionsInfo**
* **getMission**

Instrument operations:

* **getInstrumentsInfo**
* **getInstrument**

DataSet operations:

* **getDataSetsInfo**
* **getDataSet**
* **getDataSetRelatedEntitites**

DataFile operations:

* **getDocumentsInfo**
* **getImagesInfo**
* **getDataFile**

Search operations:

* **searchEntities**
* **searchEntitiesByType**
* **searchDataSetsByCriteria**

**3. Common parameters.**

There are a few parameters of the PDS operations that have the same meaning for all operations where they are used. This section describes such parameters (id, page and restriction).

**id** - this parameter represents *identifier* of the corresponding entity. For example, it can be dataset id, mission id, instrument id, etc. It takes on positive integer values like 1, 5, 23.

**page** - is an *optional* parameter that defines the range of data that should be retrieved. Consider an operation that returns a list of 1000 elements. Without using *page* parameter all 1000 elements will be returned. When the *page* parameter is specified the data is grouped into pages and only the selected page is returned. For example, you can specify that each page contain 20 elements. In this case you will have 1000 / 20 = 50 pages. To get the necessary data in addition to setting the page size, the page number should also be defined (from 1 to 50 in our example).

*Functions of the page object:*

***setItemsPerPage***(itemsPerPage) - sets elements count per page. itemsPerPage parameter is a positive integer value.

***setPageNumber***(pageNumber) - sets the page number of interest. pageNumber parameter is a positive integer value.

Please, note, page is an optional parameter. When it is not specified (nil or undefined) then all elements will be returned by the corresponding operation.

Example: JavaScript code to create page object that defines 20 elements per page and that selects the second page:

var page = new Page();

page.setItemsPerPage(20);

page.setPageNumber(2);

**restriction** - is an *optional* parameter that imposes additional restrictions on the PDS queries. For example, *getInstrumentsInfo* operation returns list of *all* available instruments from the PDS database (let's assume that page parameter is not specified). By using restriction parameter you can retrieve only those instruments that have some property you are interested in.

*Functions of the restriction object:*

***setRestrictionEntityId***(restrictionEntityId) - sets the id of the entity that is used to restrict results.

***setRestrictionEntityClass***(entityClassName) - sets the type of the entity which id is set with *setRestrictionEntityId* function. Only the following string values are allowed as the value of *entityClassName* parameter:

* 'TargetType'
* 'Target'
* 'Mission'
* 'Instrument'
* 'DataSet'
* 'DataFile'

Why do we need restriction and how to use it ? Let's consider one of the operation from *getXXXInfo* family, for example, *getDataSetsInfo*. When we do not specify restriction parameter (null or undefined) then this operation will retrieve info about all available datasets (also optionally we can specify page but it's not important in this context). The restriction allows us to apply additional restriction to the query. If we set restriction to target T1 (by setting the id of the T1 target and 'Target' class for restriction parameter) then all dataset entities returned by *getDataSetsInfo* will be the datasets that corresponds to the measurements that were made for T1 target. In the same way we can get datasets for some particular mission, or another example get all instrument (getInstrumentsInfo) that belongs to some mission or get all data files (getDataFilesInfo) that belongs to the given dataset, etc.

For the given entity only some specific entity types can be used for restriction. Here is the restriction hierarchy*:* ***TargetType******-> Target -> Mission -> Instrument -> DataSet -> DataFile***. *The entity can be restricted only by the entities that go before it*. For example, in getMissionsInfo we can use Target and TargetType for restriction, for getTargetTypeInfo we can't specify any restriction and for getDataFilesInfo we can set any other entity type as a restriction (except DataFile class itself).

Example: JavaScript code to create restriction object that restricts results to belong to the mission with id 3:

var restriction = new Restriction();

restriction.setRestrictionEntityId(3);

restriction.setRestrictionEntityClass('Mission');

**4. Operations reference.**

This section gives detailed description of all operations presented in section 2 'Entities types and operations overview'. The operations here are divided into groups by the similarity of theirs functionality.

As described in section 1 'Introduction' the first parameter of each operation is a success callback and the second is an error callback. Here we imply that each operation uses these callbacks and do not mention explicitly about this when describing each operation (but we will show the usage of callbacks in examples). The other parameters that provide info to the operation are described explicitly.

1. *getXXXInfo* operations:

* **getTargetTypesInfo**
* **getTargetsInfo**
* **getMissionsInfo**
* **getInstrumentsInfo**
* **getDataSetsInfo**
* **getDocumentsInfo**
* **getImagesInfo**

Each operation from this group provides info about entities of the corresponding type. For example, getTargetInfo gets info about available targets, getInstrumentsInfo gets info about available instruments. The two optional parameters can be passed to each of these operations: page and restriction. If none of them is specified then the operation will get info about all entities in the PDS database of the corresponding type.

The info returned contains the identifier and the name of each matching object.

Example1: the following JavaScript shows how to print names and ids of all TargetTypes:

// invoke getTargetTypesInfo operation

PDS.getTargetTypesInfo(onSuccess, onError);

// success callback

function onSuccess(response) {

var results = response.getReturn().getResults();

for (var i = 0; i < results.length; i++) {

var entityInfo = results[i];

console.log('id = ' + entityInfo.getId() + ' name = ' + entityInfo.getName());

}

}

// error callback

function onError(error) {

alert('error ' + error);

}

Example2: print all missions names which were accomplished for target with id 1 (the onSuccess and onError callbacks are from example 1):

// create restriction object

var restriction = new Restriction();

restriction.setRestrictionEntityId(1);

restriction.setRestrictionEntityClass('Target');

// invoke getMissionsInfo operation

PDS.getMissionsInfo(onSuccess, onError, null, restriction);

Example3: print names of the first 10 targets from the PDS database:

// create page object

var page = new Page();

page.setItemsPerPage(10);

page.setPageNumber(1);

// invoke getTargetsInfo operation

PDS.getTargetsInfo(onSuccess, onError, page);

NOTE: The getDocumentsInfo returns info about all DataFiles in the PDS database. On the other hand getImagesInfo returns info only about those DataFiles that represent image files.

The ***1-entities-enumeration*** demo from the distribution shows entities enumeration techniques described here.

2. *getXXX* operations:

* **getTargetType**
* **getTarget**
* **getMission**
* **getInstrument**
* **getDataSet**
* **getDataFile**

Given the entity id these operations return the object that represents the entity of the corresponding type. Compared to getXXXInfo operations which enumerate entities (return only ids and names of *multiple* entities), the operations from this group return all available info for a *single* entity with the given id. For example, depending on the entity type the following additional data can be provided: description, references, start/stop date, etc.

In the success callback of each of the getXXX method the entity object can be retrieved from the response in the following way:

var entity = response.getReturn();

The ***2-get-entity*** demo from the distribution shows how to get entities of various types.

The following is the description of each method of each entity:

TargetType methods:

* getId() - returns target type id
* getName() - returns target type name

Target methods:

* getId() - returns target id
* getName() - returns target name
* getTypes() - returns array of TargetTypes associated with this target

Mission methods:

* getId() - returns mission id
* getName() - returns mission name
* getStartDate() - returns start date of the mission
* getEndDate() - returns end date of the mission
* getDescription() - returns textual description of this mission
* getReferences() - returns an array of references associated with this mission

Instrument methods:

* getId() - returns instrument id
* getName() - returns instrument name
* getType() - returns instrument type string
* getTextId() - returns textual identifier of this instrument
* getDescription() - returns textual description of this instrument
* getHosts() - returns an arrray of hosts objects associated with this instrument
* getReferences() - returns an array of references associated with this instrument

DataSet methods:

* getId() - returns dataset id
* getName() - returns dataset name
* getTextId() - returns textual identifier of this dataset
* getStartDate() - returns start date
* getStopDate() - returns stop date
* getDescription() - returns textual description of this dataset
* getReferences() - returns an array of references associated with this dataset
* getVolumes() - returns an array of volume objects associated with this dataset
* getTargtets() - returns an array of target objects associated with this dataset
* getMissions() - returns an array of mission objects associated with this dataset
* getInstruments() - returns an array of instrument objects associated with this dataset

DataFile methods:

* getId() - returns datafile id
* getName() - returns datafile name
* getContent() - returns content of the ascii file. Returns null for binary files.

The following objects are not exposed directly via PDS interface but they can be returned by other methods: Volume, Host, Reference.

Volume methods (volume is a unit of datasets organization in the PDS archive):

* getId() - returns volume id
* getName() - returns volume name
* getDescription() - volume description
* getTextId() - volume textual identifier
* getSetTextId() - returns setTextId (not sure what does it mean)
* getSeriesName() - returns series name

Host methods (in most cases host is a spacecraft which was used for the particular mission):

* getId() - returns host id
* getName() - host name
* getTextId() - host textual identifier
* getReferences() - returns an array of references associated with this host

Reference methods (reference object represents a bibliographic reference to some book or scientific paper):

* getId() - returns reference id
* getKeyTextId() - reference textual identifier
* getDescription() - returns textual description of this reference

3. search operations:

* **searchEntities**
* **searchEntitiesByType**
* **searchDataSetsByCriteria**

The search operations allow to search for the entities by specifying text patterns from the entity name, search criteria, restrictions, etc.

*Please, note, that search operations for found entities return only info object, which contains entity's id and name - not the entire entity itself (the same like getXXXInfo operations). To get full entity object use getXXX operations to get entity by its id.*

The ***3-search*** demo from the distribution shows how use search operations.

The most simple search operation is *searchEntities*. It accepts text pattern and optionally page object and returns all the entities which have the specified text pattern in its name.

Signature: searchEntities(onSuccess, onError, textPattern, page)

The *searchEntitiesByType* operation provides more parameters to control search process. It searches for the entities of the specified type which contain the given text pattern in its names. The operation can accept optional page and restriction parameters.

Signature: searchEntitiesByType(onSuccess, onError, entityType, textPattern, page, restriction)

The *entityType* parameter can take on the following values:

* 'TargetType'
* 'Target'
* 'Mission'
* 'Instrument'
* 'DataSet'
* 'DataFile'

The *searchDataSetsByCriteria* operation returns info about datasets that satisfy the given criteria. The criteria are specified as the SearchCriteria object.

Signature: searchDataSetsByCriteria(onSuccess, onError, searchCriteria, page)

The result of this operation is the datasets that match all the criteria defined by SearchCriteria object.

To set criteria the SearchCriteria object provides the following functions:

* setDataSetId(id) - sets id of the dataset of interest. The result of the search will be a dataset with the given id.
* setStartDate(dateStr) - sets start date. The result of the search will be the datasets with start date after the date defined by dateStr parameter.
* setStopDate(dateStr) - sets stop date. The result of the search will be the datasets with stop before the date defined by dateStr parameter.
* setTargetTypes(targetTypeNamePatterns) - sets arrays of target type name patterns. The result of the search will be datasets which belong to target types with names that match at least one pattern from *targetTypeNamePatterns* array.
* setTargets(targetNamePatterns) - sets arrays of target name patterns. The result of the search will be datasets which belong to targets with names that match at least one pattern from *targetNamePatterns* array.
* setMissions(missionNamePatterns) - sets arrays of mission name patterns. The result of the search will be datasets which belong to missions with names that match at least one pattern from *missionNamePatterns* array.
* setInstruments(instrumentNamePatterns) - sets arrays of instrument name patterns. The result of the search will be datasets which belong to instruments with names that match at least one pattern from *instrumentNamePatterns* array.
* setInstrumentHosts(instrumentHostNamePatterns) - sets arrays of instrument host name patterns. The result of the search will be datasets which belong to instrument hosts with names that match at least one pattern from *instrumentHostNamePatterns* array.

4. other operations:

* **getDataSetRelatedEntities**
* **getPreviewImageURL**

The *getDataSetRelatedEntities* operation returns entities of the specified type that has relation to the given dataset. For example, for Mission entity type it's all missions in which the dataset was created (in most cases it's a single mission), for Instrument entity type it's all instruments which was used to measure data for the given dataset and so on.

Signature: getDataSetRelatedEntities(onSuccess, onError, dataSetId, entityType)

The *entityType* parameter can take on the following values:

* 'TargetType'
* 'Target'
* 'Mission'
* 'Instrument'

The *getPreviewImageURL* operation returns URL of the preview image that corresponds to the image defined by DataFile with the given id (imageDataFileId).

Signature: getPreviewImageURL(, onError, dataSetId, imageDataFileId)