Crowdsourcing framework
Json API documentation

August 12, 2015

Chapter 1

GET interface

This section will explain how to use the get json interface. The example figures will be constitued by the input code on the left, and the output given by the platform on the right.

1.1 Access the interface

To access the interface, you will have to send a POST request to :

> <framework root>/json/?page=get

The POST data must content only json, and the request *Content-type* header must be set to *application:json*.

1.2 login

All json requests must contain a *login* object to authenticate yourself as an administrator of the platform.

```
{
  "login":{
    "id":"admin",
    "password":"admin"
  }
}
```

```
{
  "code": 9,
  "message": "Json query syntax incorrect"
}
```

This input alone will return an error, as the script expects a request. It is, however, required to include it in any request.

Figure 1.1: Example of login code

You can notice here the structure of any return from the platform : a return code that is 0 if there was no error, a positive integer if there was one.

1.3 return codes table

code	meaning	help
0	query successful	Everything went good.
1	Unable to parse JSON into '.mysqlInfo' file.	Check the json syntax of the '.mysqlInfo' file.
2	Unable to connect to mysql.	Check that the login information in the '.mysqlInfo' file is correct; check that the myql server is reachable.
3	Invalid input data.	The data transmitted (GET or POST) is incorrect. See the error message for more details.
5	input data is not json or header is incorrect.	Check that the http request sends correct json, set the 'Content-Type' header to 'application/json'.
6	Unable to read the platform configuration files.	Check that the platform has been initialized, if not go to <root>/setup - Check that '.mysqlInfo' and '.fmwName' files are readable by the web server.
7	Error while executing a sql request.	
9	Json request sytax incorrect.	The Json query received by the iterface was incorrect. Check its syntax and compare it with the example.
10	Return size greater than specified maximum.	The query you made on the 'get' interface generated data heavier than the 'max_size' specified (10 MB if not specified in the request). Try using better filters.

1.4 Make a request

The request is made in a *request* object that contains at least a *type* field.

```
{
  "login":{
    "id":"admin",
    "password":"admin"
  },
  "request":{
    "type":"worker"
  }
}
```

```
[
    {
        "id": "1",
        "username": "worker",
        "password": "$2y$10$IuB4WsobsGdGNCo1ia",
        "birthdate": "1993-07-04"
    },
    {
        "id": "2",
        "username": "worker2",
        "password": "$2y$10$p0U3pW52oWaFN4qESD",
        "birthdate": "2000-05-04"
    }
]
```

Figure 1.2: Example minimal request

As shown on the example above, the return is an array of the matching data.

1.5 Specifying return fields

If you don't need all the return fields, you can use the following syntax to specify the ones you want :

```
{
  "login":{
    "id":"admin",
    "password":"admin"
  },
  "request":{
    "type":"worker",
    "fields":[
      "id",
      "username"
    ]
  }
}
```

```
[
    {
        "id": "1",
        "username": "worker"
    },
    {
        "id": "2",
        "username": "worker2"
    },
    {
        "id": "3",
        "username": "worker3"
    }
]
```

Figure 1.3: Example request with return fields selection

1.6 Filtering the results

The results can be filtered using the following syntax :

```
{
  "login":{
    "id":"admin",
    "password":"admin"
  },
  "request":{
    "type":"worker",
    "fields":[
      "id",
      "birthdate"
    ],
    "filters":[
      "DATEDIFF(NOW(),birthdate)>(20*365)"
    ]
  }
}
```

```
[
    {
        "id": "1",
        "birthdate": "1993-07-04"
    },
    {
        "id": "3",
        "birthdate": "1960-10-22"
    }
]
```

Figure 1.4: Example request with results filtering

The example above selects only the users whose age is greater than 20. The filters must be written with SQL syntax. There can be as many filters as needed. Fields selection and filters can be used together or separately.

1.7 sub-results

Some types (*task*, *questions* and *contributions*) have sub-results. Fields selection and results filtering can be applied to these sub-results by adding sub-objects to the *fields* or *filters* objects as shown in the following example :

```
{
  "login":{
    "id":"admin",
    "password":"admin"
  },
  "request":{
    "type":"task",
    "fields":{
      "0":"name",
      "questions":{
        "0":"question",
        "answers":"false"
      }
    }
  }
}
```

```
[
    {
        "id": "1",
        "name": "test_task 1",
        "questions": [
            {
                "id": "1",
                "question": "How are you ?"
            },
            {
                "id": "2",
                "question": "is this picture
                    correctly displayed?"
            }
        ]
    },
    {
        "id": "2",
        "name": "assignTest",
        "questions": [
            {
                "id": "3",
                "question": "test"
            }
        ]
    }
]
```

Figure 1.5: Example request with results filtering

As shown, you can also use the *"false"* string to ommit a sub-result. Please note that the *answers* subtype only accepts *"false"* as a fields selection, since it has to few fields for this feature to make sense.
Finally, please note that the *id* field will always be returned, since the framework needs it.

Chapter 2

SET interface

2.1 WARNING

This interface allows to insert data directly in the database, so be careful with it. Inserting incorrect data can introduce bad bugs in the framework.

2.2 Access the interface and login

The SET interface can be accessed the same way as the GET interface :
To access the interface, you will have to send a POST request to :

<framework root>/json/?page=set

The POST data must content only json, and the request *Content-type* header must be set to *application:json*.
All json requests must contain a *login* object to authenticate yourself as an administrator of the platform.

2.3 edit existing content

Inserting new content is simply done with the following syntax : You need to specify the type and the id of the

```
{
    "login":{
        "id":"admin",
        "password":"admin"
    },
    "request":{
        "type":"worker",
        "id":"1",
        "username":"user10",
        "password":"test"
    }
}
```

```
{
    "code": 0,
    "message": "Row edition OK"
}
```

Figure 2.1: Example edition request

content to edit. Then just set all the fields you want to edit. Unspecified fields will simply remain unchanged.

2.4 Insert new contents

The following syntax is used to insert new rows into the database :

```
{
    "login":{
        "id":"admin",
        "password":"admin"
    },
    "request":{
        "type":"worker",
        "values":[
            {
                "username":"user20",
                "password":"test",
                "birthdate":"2010-10-10"
            },
            {
                "username":"user21",
                "password":"test"
            }
        ]
    }
}
```

```
{
    "code": 0,
    "message": "successfully inserted 2 rows."
}
```

Figure 2.2: Example insertion request

Values must be an unnamed array (defined with [ ] in JSON).
The script uses the first object in *values* to define the fields used in the sql query. This means that if all the objects to insert do not have the same lenght, the longest object should always be placed first :

```
{
    "login":{
        "id":"admin",
        "password":"admin"
    },
    "request":{
        "type":"worker",
        "values":[
            {
                "username":"user21",
                "password":"test"
            },
            {
                "username":"user20",
                "password":"test",
                "birthdate":"2010-10-10"
            }
        ]
    }
}
```

```
{
    "code": 0,
    "message": "successfully inserted 2 rows."
}
```

This request works, but the *birthdate* field of *user20* will be ignored by the script.

Figure 2.3: Example of a misformed insertion request

Chapter 3

DELETE interface

3.1 WARNING

This interface deletes content directly from the database, so be careful about what you do with it. Please note that, due to the database's structure, deleting content of certain types will *automatically* delete sub-contents linked to it. For instance, deleting a task will delete all its questions, but also the answers and contributions linked to these questions. Any deletion will be irrecoverable.

3.2 access the interface

The access is the same as before. You will have to send a POST request to :

<framework root>/json/?page=delete

The POST data must content only json, and the request *Content-type* header must be set to *application:json*.

All json requests must contain a *login* object to authenticate yourself as an administrator of the platform.

3.3 Delete a row

Deleting content can be done using the following syntax :

```
{
    "login":{
        "id":"admin",
        "password":"admin"
    },
    "delete":{
        "type":"worker",
        "id":5
    }
}
```

```
{
    "code": 0,
    "message": "row successfully deleted"
}
```

Figure 3.1: Example of a misformed insertion request