

MSL Image Explorer

Thanks for downloading the demo. This tutorial will provide details on how things work behind the scenes with the MSL Image Explorer. With this app you can easily store and index the images of Mars published by the NASA Jet Propulsion Laboratory. In the process of using this app, you will learn how to use a few DynamoDB API calls. We will show code snippets in JavaScript. This app also provides an example Java application to show how to store JSON data in DynamoDB.

Before you get started, your machine will need to support the following:

- a) Java 1.7+
- b) NodeJS
- c) Maven

Let's Explore!

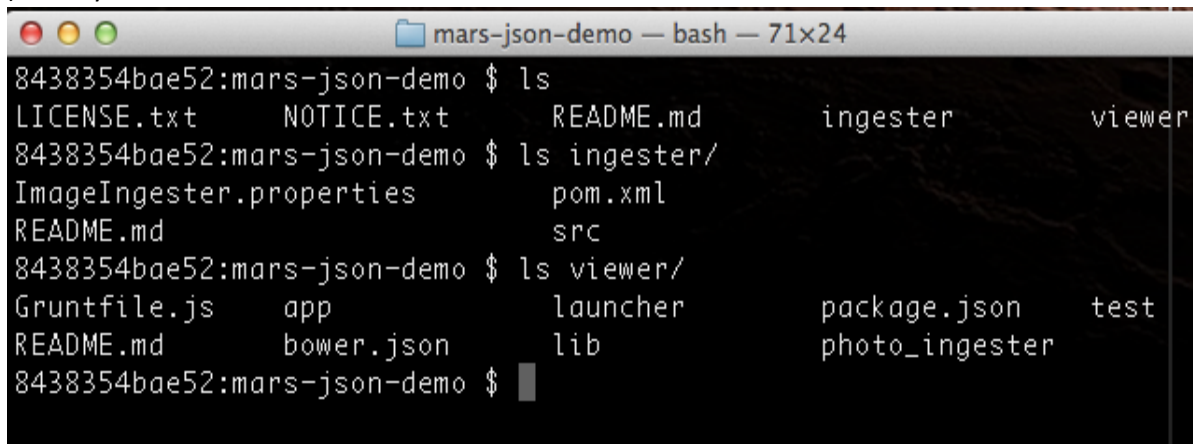
1) Download all the files in the demo:

- a. Option 1, you can get it from GitHub

`git clone https://github.com/aws-labs/aws-dynamodb-mars-json-demo.git`

- b. Option 2, you can download it from this [link](#)

2) Once you extract all the files - Your files should look like this:



```
mars-json-demo — bash — 71x24
8438354bae52:mars-json-demo $ ls
LICENSE.txt  NOTICE.txt  README.md    ingester     viewer
8438354bae52:mars-json-demo $ ls ingester/
ImageIngester.properties  pom.xml
README.md                  src
8438354bae52:mars-json-demo $ ls viewer/
Gruntfile.js  app      launcher  package.json  test
README.md     bower.json  lib       photo_ingester
8438354bae52:mars-json-demo $
```

3) Use the following commands to build the application:

- a. Image Ingester – Go to the `./Ingester` folder and run the following command:

```
mvn clean install
```

- b. Image Viewer – Go to you the `./viewer` directory and run the following commands:

```
sudo npm install -g grunt-cli bower
```

```
npm install
```

```
bower install
```

4) Before we launch the app, let's take a look at this properties file called `ImageIngester.properties` in the `./ingester/` folder. Some interesting settings here are:

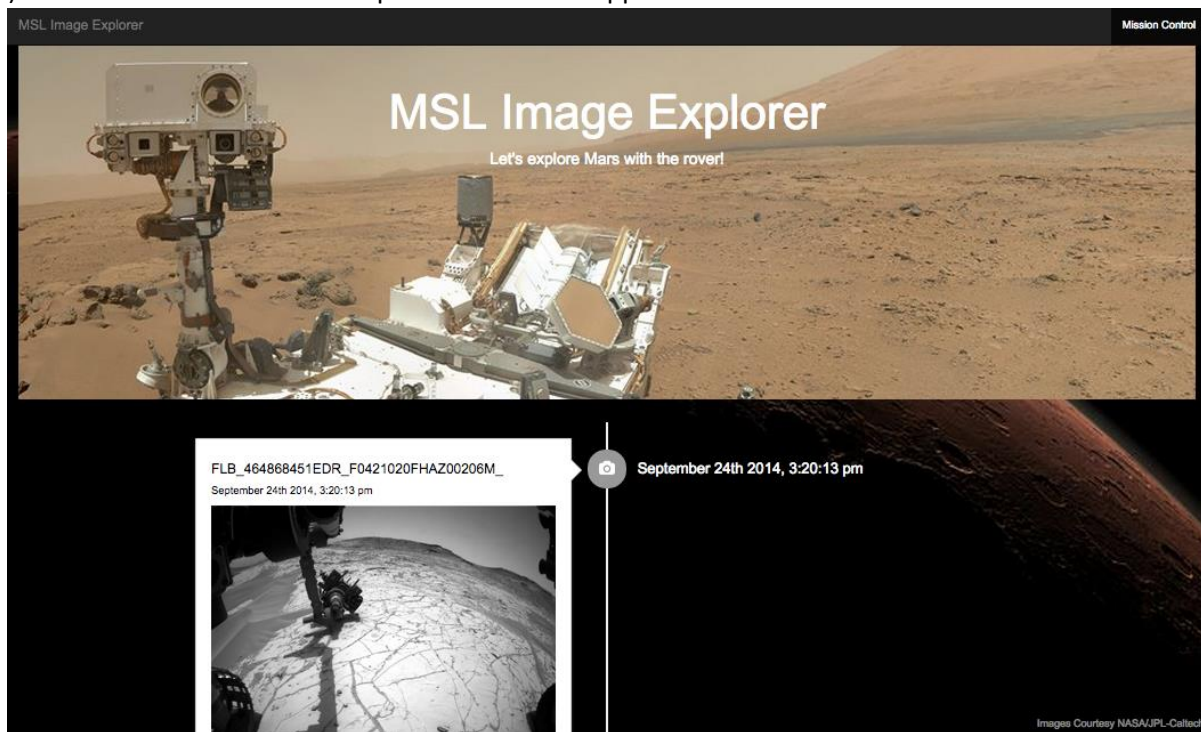
- a. `JSON.root` - this defines where the JSON will be read from. There are two predefined URLs included and you can select one by commenting out one and leaving the other one uncommented out.
- b. `dynamodb.endpoint` - This is the endpoint the DynamoDB client will connect to. The tables will be located in the region serviced by the endpoint. If the endpoint is set to `http://localhost:8000` it points to [DynamoDB Local](#), a free client-side version of DynamoDB. This can be changed to run the demo app on the cloud, if needed.
- c. `dynamodb.resource`, `dynamodb.image` - These are the table names for the resource and image DynamoDB tables. The image table is required, the resource table is required if the user enables the `ingester.track-resources` option

- d. `ingester.store-thumbnails` - This will store thumbnail data in DynamoDB if it is set to true. This is useful when you want to store larger items in your database.
- e. `ingester.manifest.threads`, `ingester.sol.threads`, `ingester.image.threads` - The number of threads that should be run for each task. These numbers can be tweaked based on the compute power and the provisioned throughput of the table to achieve a balance between cost and time.

5) We are now ready to launch our app. Let's go back to the directory that has our all our downloaded files. Let's launch the Mars Image Explorer app using the 'grunt' command:

```
8438354bae52:mars-json-demo $ cd viewer/
8438354bae52:viewer $ ls
Gruntfile.js      bower.json        node_modules
README.md          bower_components  package.json
app               lib               test
8438354bae52:viewer $ grunt serve
```

6) You should see the browser open this beautiful app:



7) You can browse the code snippets below in a file called `viewer/app/scripts/services/mars-photos.js`

8) You can now control what images you see from on the explorer application

- a. To select a different instrument go to "Mission control → Select Instrument"
- b. Behind the scenes, here is the code snippet that is doing the work:

```

/**
 * Queries photos table with the date index and give the result
 * to callback function.
 * @param {object} queryParams - The query parameters used in the query. The parameter
 * hashCode is required. The lastEvaluatedKey and rangeKey are optional.
 * @param {function} callback - The callback function used to return the result.
 */
queryWithDateIndex: function(queryParams, callback){
    assertHasKey(queryParams);
    assertFunction(callback);

    var params = {
        TableName: 'marsDemoImages',
        KeyConditions: [
            AWS.dynamoDB.Condition('Mission+InstrumentID', 'EQ', queryParams.hashCode)
        ],
        IndexName: 'date-gsi',
        Limit: 5,
        ScanIndexForward: false
    };
    if(hasLastEvaluatedKey(queryParams)) {
        params.ExclusiveStartKey = queryParams.lastEvaluatedKey;
    }
    if(hasRangeKey(queryParams)){
        params.KeyConditions.push (AWS.dynamoDB.Condition('TimeStamp', 'LE', queryParams.rangeKey));
    }
    logRequest('query', params);
    AWS.dynamoDB.query(params, callback);
}

```

- c. To see how updates work, you can vote on your favorite images.
- d. To vote for an image, click on the vote button.

September 24th 2014, 3:20:13 pm

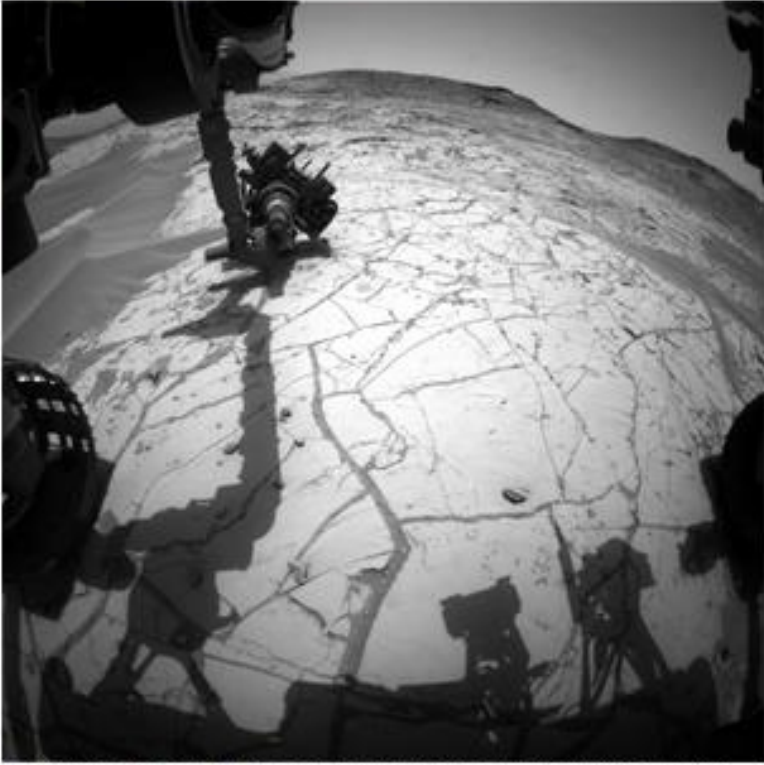



Image from the rover's Chemcam RMI during mission curiosity. It was taken at September 24th 2014, 3:20:13 pm and received on Earth a day later.

of votes: 644



- e. Each time you vote, an update code works behind the scenes. Here is what it looks like:

```
/**
 * Increments the voting count in the photo table. It gets the updated
 * votes count in return. The function is meant to be used privately
 * in MarsPhoto service.
 * @param {String} imageid - The ID of the photo to increment vote count.
 * @param {function} callback - The callback function used to return the result or error.
 */
var incrementVotesCount = function(imageid, callback) {
    var params = {
        TableName: 'marsDemoImages',
        Key: { imageid: imageid },
        UpdateExpression: 'add votes :v',
        ExpressionAttributeValues: { ':v': 1 },
        ReturnValues: 'UPDATED_NEW'
    };

    logRequest('updateItem', params);
    AWS.dynamodb.updateItem(params, callback);
};
```

- f. Conditional updates are simple to write as well. Let's say you want to prevent people from voting multiple times, you can do that by tracking user votes in a table and checking this table to see if a user has votes on a particular image. Here is the code snippet.

```
item.userid = userid;
var params = {
    TableName: 'userVotes',
    Item: item,
    Expected: [
        AWS.dynamodb.Condition('imageid', 'NULL')
    ]
};
logRequest('putItem', params);
AWS.dynamodb.putItem(params, function(error) {
    if (!error) {
        $log.debug('Liked image successfully');
        incrementVotesCount(photo.imageid, callback);
    } else {
        if (error.code === 'ConditionalCheckFailedException') {
            callback('You have already voted on this image');
        } else {
            $log.error(error);
        }
    }
});
```

We hope you had fun exploring the MSL Image Explorer app and digging into DynamoDB commands. For more DynamoDB help and instructions, visit our [developer guide](#).