# Neato Web Service Interaction

Raja Software

Last updated: 04/01/2014

|  |  |  |
| --- | --- | --- |
| Date | Version | Author/Comments |
| 03/28/2014 | 0.1 | Initial draft – focus on smartphone web service interaction |
| 04/01/2014 | 0.2 | Added more details, discussed notifications |
| 04/03/2014 | 0.3 | Added details regarding specific web-services |

This document is intended for the all the developers on the Neato project to get the overall perspective of all the important web service calls made from SmartApps and User.

# Robot Profile Details

Before we get started on individual web services, let us look into how data is communicated between multiple SmartApps (logged in with same user) and the associated robot, and how the state of the robot is synchronized.

The web services involved are:

<http://neatostaging.rajatogo.com/api/rest/json?method=robot.set_profile_details3> and <http://neatostaging.rajatogo.com/api/rest/json?method=robot.get_profile_details2>

The robot profile, on the server, maintains a key-value pair along with a server timestamp of when the value associated with the key was changed.

The keys are defined for various properties which maintain a state of the robot as well as to have a command structure sent VIA server so that other SmartApps can be in sync.

Whenever a Robot updates a value for any key in the profile, a XMPP data changed notification is sent to all the online user-devices.

Whenever a SmartApp updates a value for any key in the profile, a XMPP data changed notification is sent to any other online user devices associated with the robot and the robot itself.(It is possible that one user is logged in to multiple Smartphone)

This notification is parsed and the receiver calls: <http://neatostaging.rajatogo.com/api/rest/json?method=robot.get_profile_details2> to update itself.

Every receiver maintains a “Key-Timestamp” pair in the local storage (SQLite, file) to know the version of the state received. It does not store the value of the key in the local storage.

The following diagram illustrates the mechanism through which a state is propagated between different systems:

Step3.

Get Robot Profile Details

Step1.

Set Robot Profile Details Call

Step2.

XMPP Notification

Robot

SmartApp

Step3.

Get Robot Profile Details

SmartApp

Step 1: The SmartApp user changes a robot profile key (name, robot command or schedule state etc)

Step 2: Server sends a XMPP message to the online users for the robot as well as the robot.

Step 3: The details are retrieved by the receiver and processed.

After retrieving the data from the server, the timestamps for each key is compared with the timestamps of the keys in the local storage and the “changed” keys are notified to the SmartApp user (Plugin notifies the JS layer about the changes).

Example:

If SmartApp UserA is using a robot RobotB:

1. UserA sends a Start Cleaning Command.
2. Set Robot Profile Details will be called with the key “cleaningCommand” for start cleaning along with the cleaning parameters as the value for the key(These parameters are discussed later)
3. The server will update the value as well as update the timestamp.
4. The RobotB will receive a data changed notification.
5. The RobotB will retrieve the data from the server and compare the local timestamps with the server details and process the “Changed” key(s) which in this case is the Cleaning command.

It will update the current state of the RobotB on the server which again is propagated to all online users of the RobotB.

There are multiple keys which are set using this design:

1. Schedule State of the robot
2. Name
3. Cleaning Command
4. Robot Current State
5. Intend To Drive
6. Online/Offline Status of the robot.
7. Notifications and Errors sent by the robot.

Whenever any of the keys is changed, it will fire a notification to the online systems so that the change will be propagated.

If a SmartApp comes online after some time, it can know the last state updated of the robot, the name (if changed), schedule changes as well as the schedule state of the robot; even if the robot is offline at that point.

The below list describes in short the details of key and values for the profile parameters. Some of the key value details need more description which will be done whenever the concerned section is discussed in this document:

|  |  |  |
| --- | --- | --- |
| Description | Key | Value |
| Start Cleaning/ Stop Cleaning/ Pause Cleaning/ Pause Cleaning/ Resume Cleaning/ Send To Base | cleaningCommand | XML command with all cleaning parameters. (Exact command structure is given in “Cleaning Section”)  Example Robot Command For Start Cleaning:  <?xml version='1.0' encoding='UTF-8' standalone='yes' ?><packet><header><version>1</version><signature>0xcafebabe</signature></header><payload><request><command>101</command><requestId>Random Request ID</requestId><timeStamp>Current Timestamp</timeStamp><retryCount>0</retryCount><responseRequired>false</responseRequired><distributionMode>2</distributionMode><replyTo>127</replyTo><params><cleaningModifier>1</cleaningModifier><cleaningMode>2</cleaningMode><cleaningCategory>2</cleaningCategory></params></request></payload></packet>  Note here the 101 is the command id for start cleaning. |
| Robot Current State | robotCurrentState | The integer signifying the current state of the robot (Exact state ids can be found in Cleaning Section) |
| Robot Name | name | The new robot name. If robot’s name is changed to e.g.; “My Robot” then the value will be a simple string i.e.; “My Robot”  (Its just a simple value and not the JSON) |
| Basic Schedule State | enable\_basic\_schedule | Boolean indicating whether the schedule is enabled or disabled  If the schedule is enabled for the robot then value will be “true"  (Its just a simple value and not the JSON) |
|  |  |  |
|  |  |  |
| Schedule Change | schedule\_updated | Placeholder Boolean for maintaining timestamp for when the schedule was updated.  If the timestamp changes, the schedule should be retrieved by the receiver.  If the schedule is updated for the robot then value will be “true”  (Its just a simple value and not the JSON) |
|  |  |  |
| Intend To Drive | intend\_to\_drive | {"device\_id":<device-id>, "wifi\_on\_time\_ms":<network\_on\_time>}. More details in manual mode cleaning section. |
| Available to Drive | available\_to\_drive | {"robotIpAddress":<ip address>,"driveAvailableStatus":<true/false>}. More details in manual mode cleaning section |
| Robot Notification | robotNotificationMsg | {“messageID”: <id for the message>}. More details in notifications section. |
| Robot Error | robotErrorMsg | {“messageID”: <id for the message>}. More details in error section. |
| Robot Online Status | robotOnlineStatus | 1 if robot is online, 0 if it is offline.  Example: Robot comes online then:  {robotOnlineStatus: “1”} |

## Plugin – UI Unsolicited Data Communication Framework:

As there can be unsolicited data communication to be given to the user, the JS layer registers for the data communication with the Plugin Native layer.

The plugin phonegap method used: RobotMgr.prototype.registerNotifications2

The data communication is of following format:

{robotDataKeyId:<robotDataKeyId>, robotId:<robotId>, robotData:<robotData>}

robotDataKeyId: This is the id of the data to be communicated

robotId: Robot id for which the data is being communicated

robotData: The actual JSON data.

The various examples of data and their id as well as example of robot data is as follows:

|  |  |  |
| --- | --- | --- |
| Description | Data Id | Data |
| Robot Current State Change | 4001 | {“robotCurrentState”: <state Id>} |
| Robot Virtual State (If there is a command queued, then it will signify that state, otherwise current state of the robot) | 4003 | {“robotStateUpdate”: <state Id>} |
| Robot Name Update | 4004 | {robotName: <new name>} |
| Robot Schedule State | 4005 | {scheduleState: <state 1 or 0>, scheduleType: 0}  0 signifies Basic schedule. |
| Schedule Updated | 4006 | None |
| Robot Direct Connection Established | 4007 | None |
| Robot Disconnected (TCP) | 4008 | None |
| Robot Error in Direct Connection | 4009 | {errorDriveResponseCode: <error code>}  If robot is does not update available to drive status: 10003  If IP address could not connected (because wifi different): 10005 |
| Robot Notification | 4013 | { robotNotification: {JSON notification string from robot}}  Currently:  { robotNotification: {“messageID”:<id for the message>}} |
| Robot Error | 4014 | {robotError: {JSON error string from robot}}  Currently:  { robotError: {“messageID”:<id for the message>}} |
| Robot Online Status Changed | 4015 | {“online”: <1 or 0 depending on online status>} |

The UI layer should listen to these data notifications and act on it.

# SmartApp – Web Service Interaction

# User Related Web Services

This part will describe in detail about what happens if a particular phone-gap plugin API is called and what all web services are called in the background:

## Create User

**Phonegap Plugin API**: UserMgr.prototype.createUser3

**Plugin**:

1. Web service: <http://neatostaging.rajatogo.com/api/rest/json?method=user.create3>

This web service creates the user on the server along with other necessary details. The

Web service will also return an authentication token for use.

1. After a web service success:

<http://neatostaging.rajatogo.com/api/rest/json?method=user.get_user_account_details>

is called to retrieve all the user details

<http://neatostaging.rajatogo.com/api/rest/json?method=user.set_attributes>

is called to set the user related device attributes on the server

Registration for push notifications is also done at this point.

## Login User

**Phonegap Plugin API**: UserMgr.prototype.loginUser

**Plugin**:

1. Web service called: <http://neatostaging.rajatogo.com/api/rest/json/?method=auth.get_user_auth_token>

With the success we get the authentication token for the user which will be saved locally.

1. If the server call is success, we get the user details and set attributes similar to create user:

<http://neatostaging.rajatogo.com/api/rest/json?method=user.get_user_account_details>

<http://neatostaging.rajatogo.com/api/rest/json?method=user.set_attributes>

Registration for push notifications is also done at this point.

## Resend Validation Mail

Phonegap Plugin API: UserMgr.prototype.resendValidationMail

Web service called: <http://neatostaging.rajatogo.com/api/rest/json?method=user.ResendValidationEmail>

Used to send the validation email to the user if not already validated

## Change Password

Phonegap Plugin API: UserMgr.prototype.changePassword

Web service called: <http://neatostaging.rajatogo.com/api/rest/json?method=user.change_password>

The web service is used to change the existing password of the user.

## Change Subscription / Country Code

Phonegap Plugin API: UserMgr.prototype .setUserAccountDetails

Web service called: <http://neatostaging.rajatogo.com/api/rest/json?method=user.set_account_details>

This web service is used to change the user details

## Forgot Password

Phonegap Plugin API: UserMgr.prototype.forgetPassword

Web service called: <http://neatostaging.rajatogo.com/api/rest/json/?method=user.forget_password>

The web service is used to send the next steps if the user forgot the password

## Link Robot

Phonegap Plugin API: UserMgr.prototype.linkRobot

Web service called: <http://neatostaging.rajatogo.com/api/rest/json?method=robot.link_to_robot>

The web service is used to link a robot with the user. This is done with linking code of the user.

## Get Associated Robots

Phonegap Plugin API: UserMgr.prototype.getAssociatedRobots

Web service called: <http://neatostaging.rajatogo.com/api/rest/json/?method=user.get_associated_robots>

This web service returns the list of associated robots with the user.

## Disassociate Robot

Phonegap Plugin API:

Web service called:http://neatostaging.rajatogo.com/api/rest/json?method=robot.clear\_robot\_association

This web service dis-associates the robot with the user.

## Get Push Notification Settings

Phonegap Plugin API: UserMgr.prototype.getNotificationSettings

Web service <http://neatostaging.rajatogo.com/api/rest/json?method=message.get_user_push_notification_options>

This web service returns the push notification options set by the user.

## Set Push Notification Settings

Phonegap Plugin API: UserMgr.prototype.turnNotificationOnoff

Web service: <http://neatostaging.rajatogo.com/api/rest/json?method=message.set_user_push_notification_options>

This web service sets the push notification options for the user. It can disable/enable any/all options.

# Cleaning Related Web Services

## Send Cleaning Commands (Non manual Mode): (Start/Stop/Pause/Resume/Send to base)

**Phonegap Plugins called -**

**RobotMgr.prototype.startCleaning**

**RobotMgr.prototype.stopCleaning**

**RobotMgr.prototype.pauseCleaning**

**RobotMgr.prototype.resumeCleaning**

**Plugin**

1. Web service: <http://neatostaging.rajatogo.com/api/rest/json?method=robot.set_profile_details3> is called to set the command on the server for key “cleaningCommand”.
2. A data changed XMPP message is sent to all the associated user-devices as well as the robot about “data changed”.

This notification also has the “causeAgentId” which reflects uniquely on who initiated the change.

1. When this notification is received on the SmartApp who initiated the cleaning command, the notification is ignored as the change was done by the Smartapp (comparing the causeAgentId).

IF Robot is ONLINE:

1. Robot receives the data changed notification and calls

<http://neatostaging.rajatogo.com/api/rest/json?method=robot.get_profile_details2>

1. After processing the set command, if the robot state changes due the set command, the new state is added on the server by calling <http://neatostaging.rajatogo.com/api/rest/json?method=robot.set_profile_details3> with the new state.
2. This will again fire a data changed notification from the server VIA XMPP which will cause the SmartApp to get the server details and get the current state of the robot.

If Robot is OFFLINE

The command timer on the SmartApp will fire a reset for the cleaning command so that the robot doesn’t get it when the robot comes online after a long time (which could be anytime). It calls <http://neatostaging.rajatogo.com/api/rest/json?method=robot.set_profile_details3> to reset the cleaning command. The timer expiry happens currently after 3 minutes and is only applicable for

cleaning commands.

**Key-Value details in webservice:**

The key set for adding a Cleaning command is:

|  |  |  |
| --- | --- | --- |
| Command | Key | Value |
| Start Cleaning | profile[cleaningCommand] | <?xml version='1.0' encoding='UTF-8' standalone='yes' ?><packet><header><version>1</version><signature>0xcafebabe</signature></header><payload><request><command>101</command><requestId>Random Request ID</requestId><timeStamp>Current Timestamp</timeStamp><retryCount>0</retryCount><responseRequired>false</responseRequired><distributionMode>2</distributionMode><replyTo>127</replyTo><params><cleaningModifier>1</cleaningModifier><cleaningMode>2</cleaningMode><cleaningCategory>2</cleaningCategory></params></request></payload></packet> |
| Stop Cleaning | profile[cleaningCommand] | <?xml version='1.0' encoding='UTF-8' standalone='yes' ?><packet><header><version>1</version><signature>0xcafebabe</signature></header><payload><request><command>102</command><requestId>aa8edd62-7eee-4cc0-9f5d-c34d0e0d6759</requestId><timeStamp>1373384167104</timeStamp><retryCount>0</retryCount><responseRequired>false</responseRequired><distributionMode>2</distributionMode><replyTo>127</replyTo><params /></request></payload></packet> |
| Pause Cleaning | profile[cleaningCommand] | <?xml version='1.0' encoding='UTF-8' standalone='yes' ?><packet><header><version>1</version><signature>0xcafebabe</signature></header><payload><request><command>107</command><requestId>14d2ad9f-5cf0-4638-8ea0-c636d408a4c9</requestId><timeStamp>1396513849564</timeStamp><retryCount>0</retryCount><responseRequired>false</responseRequired><distributionMode>2</distributionMode><replyTo>102</replyTo><params /></request></payload></packet> |
| Resume Cleaning | profile[cleaningCommand] | <?xml version='1.0' encoding='UTF-8' standalone='yes' ?><packet><header><version>1</version><signature>0xcafebabe</signature></header><payload><request><command>114</command><requestId>4995ce8c-fef9-4485-8dc7-93d08219a04f</requestId><timeStamp>1396513896593</timeStamp><retryCount>0</retryCount><responseRequired>false</responseRequired><distributionMode>2</distributionMode><replyTo>102</replyTo><params /></request></payload></packet> |
| Send To Base | profile[cleaningCommand] | <?xml version='1.0' encoding='UTF-8' standalone='yes' ?><packet><header>  <version>1</version><signature>0xcafebabe</signature></header>  <payload><request><command>104</command>  <requestId>7990013f-e2a1-4942-ab0f-edd0afeffb1a</requestId>  <timeStamp>1373384635417</timeStamp><retryCount>0</retryCount>  <responseRequired>false</responseRequired>  <distributionMode>2</distributionMode><replyTo>127</replyTo>  <params /></request></payload></packet> |

The value as you can see contains the exact command structure for the robot which contains command id and parameters for the command. It contains the signature and version as well.

Once this command is processed by the robot, it clears the **cleaningCommand** and also sets its own current state using set robot profile details webservice.

**The key-value pair being:**

Key: robotCurrentState

Value: ID for the state. This is just the current state id.

Following are the ids being used:

ROBOT\_STATE\_CLEANING = 10002

ROBOT\_STATE\_STOPPED = 10005

ROBOT\_STATE\_PAUSED = 10007

ROBOT\_STATE\_RESUMED = 10008

ROBOT\_STATE\_ON\_BASE = 10009

Note in Rosie Version: (0.6.5.00) the robot will also send the current state details to the server for more details for the smartapp user.

**Key value being used:**

Key: robotCurrentStateDetails

Value: "{"robotCurrentState:"<stateId>", "robotStateParams":{..}}"

In case it is cleaning: the robot state params will have following parameters:

{robotCleaningCategory:<category>, robotCleaningMode: <mode>, extraStateParams:<..>}

The extra state params will have extra parameters related to the cleaning state (spot length and spot height)

The whole response would be like for spot cleaning:

{"robotCurrentState: 10002, robotStateParams: {robotCleaningCategory:3, robotCleaningMode: 1, extraStateParams:{robotSpotCleaningArealength:5, robotSpotCleaningAreaheight:3}}

}

Example:

The user starts cleaning:

|  |  |
| --- | --- |
| cause\_agent\_id | 978b000f-13b0-4376-ab87-fec7e33a2027 |
| notification\_flag | 1 |
| profile[cleaningCommand] | <?xml version='1.0' encoding='UTF-8' standalone='yes' ?><packet><header><version>1</version><signature>0xcafebabe</signature></header><payload><request><command>101</command><requestId>Random Request ID</requestId><timeStamp>Current Timestamp</timeStamp><retryCount>0</retryCount><responseRequired>false</responseRequired><distributionMode>2</distributionMode><replyTo>127</replyTo><params><cleaningModifier>1</cleaningModifier><cleaningMode>2</cleaningMode><cleaningCategory>2</cleaningCategory></params></request></payload></packet> |
| source\_smartapp\_id | bo@yana.com |
| api\_key | 1e26686d806d82144a71ea9a99d1b3169adaad917 |
| serial\_number | yana |

The response is:

{

"status": 0,

"result": 1,

"extra\_params": {

"expected\_time": 1,

"timestamp": 1396524290

}

}

The robot when gets the message updates the cleaning state with which the get robot profile details is called:

Request:

api\_key 1e26686d806d82144a71ea9a99d1b3169adaad917

serial\_number yana

Response:

{

"status": 0,

"result": {

"success": true,

"profile\_details": {

"name": {

"value": "yna1234",

"timestamp": 0

},

"serial\_number": {

"value": "yana",

"timestamp": 0

},

"robotOnlineStatus": {

"value": "1",

"timestamp": "1396522540"

},

"robotCurrentState": {

"value": "10002",

"timestamp": "1396524293"

},

"enable\_basic\_schedule": {

"value": "1",

"timestamp": "1396523142"

},

}

}

Here the timestamps (in UTC) will be compared in the local DB and then the notifications will be sent to the User depending on the new updates.

## Get Current State:

Phonegap Plugin API: getRobotCleaningState

Webservice used: <http://neatostaging.rajatogo.com/api/rest/json?method=robot.get_profile_details2>

The current state is returned to the plugin UI layer as callback success:

{robotCurrentState:<robotCurrentState>, robotNewVirtualState: <robotNewVirtualState>, robotId:<robotId>}

Here the current state is the actual robot state updated by the robot.

The robot new virtual state is the command currently queued for processing for the robot.

## Start Cleaning Flow (Manual Mode)

### Intend to drive

Phonegap Plugin API: RobotMgr.prototype.intendToDrive

1. Web service:

<http://neatostaging.rajatogo.com/api/rest/json?method=robot.set_profile_details3>

This is the 1st step to start driving and this is a request to the robot to open the TCP server and provide an IP address to connect to.

The SmartApp will set the key-value as follows (while setting the profile details):

Key: intend\_to\_drive

Value: {"device\_id":<device-id>, "wifi\_on\_time\_ms":<network\_on\_time>}

The device id is to uniquely identity the device (which is also used as a cause agent id).

The wifi\_on\_time: This parameter was added to let the robot know the time for which it needs to stay online. This parameter was added for timed-mode support, as the robot would go back to offline mode which would not allow the manual cleaning process to work.

1. Robot if online will receive the data changed notification, and will retrieve the profile details. It will set whether it is available to drive (which is the case unless it is in driving mode already) along with its IP address using <http://neatostaging.rajatogo.com/api/rest/json?method=robot.set_profile_details3>

If the robot is available for drive mode, it will call Set robot profile details webservice with:

Key = available\_to\_drive

Value: Value = {"robotIpAddress":<ip address>,"driveAvailableStatus":<true>}

If it is unavailable, it will set value as:

Failure: {errorDriveReasonCode: <error id for no connection>, >,"driveAvailableStatus":<true/false>}

Example:

Value = {"robotIpAddress":"192.168.1.110","driveAvailableStatus":true}

Error status can be:

Robot Already Connected On TCP: 10001

Robot TCP connection timedout: 10002

The received data changed notification is tracked by the caller SmartApp to initiate a TCP

connection to the robot address.

1. Once the Smartapp receives the data changed notification it tries to connect to the given IP address.

If connection succeeds it gives notification to the JS layer using the notification framework for the plugin layer described in the top section

.

1. Appropriate callbacks are given to the UI JS layer about connection success or failure. These callbacks can be seen in the “Data Communication” framework between UI-Plugin explained in the top section.

### Start Manual Cleaning

Once the connection is formed, the Phonegap plugin API to be called are same as non-manual mode. They will work only when the connection is formed, otherwise will return an error to the plugin API now.

## Drive Robot:

The drive robot commands are sent VIA tcp connection formed. They won’t be sent VIA server.

### Cancel Intend To Drive

Phonegap API: RobotMgr.prototype.cancelIntendToDrive

This will clear the "intend to drive” request from the server unless the connection has already formed.

Web service called: <http://neatostaging.rajatogo.com/api/rest/json?method=robot.set_profile_details3>

The key: intend\_to\_drive is reset.

### Stop Robot Drive

Phonegap API: RobotMgr.prototype.stopRobotDrive

This disconnects the current connection to the robot if connected.

## Schedule Related Web Services

## Enable/Disable Schedule

Phonegap Plugin: RobotMgr.prototype.enableSchedule

Web service called: <http://neatostaging.rajatogo.com/api/rest/json?method=robot.set_profile_details3>

**Key – Value used:**

Key: enable\_basic\_schedule

Value: true if enabled else false.

This will set the state for the schedule which will fire a data changed notification to the associated users and the robot in order to retrieve the correct state.

## Is Schedule Enabled

Phonegap Plugin: RobotMgr.prototype.isScheduleEnabled

Web service called:<http://neatostaging.rajatogo.com/api/rest/json?method=robot.get_profile_details2>

Here the key - enable\_basic\_schedule is retrieved and the state is shown to the user.

This will get the profile details and retrieve the current schedule state.

## Schedule

A robot schedule is maintained on the Server along with a version id. Whenever the schedule changes, the version id is incremented by the server. To update a schedule, the caller should have the current

schedule id, otherwise the server will not allow the update.

Schedule or a Schedule Group of individual schedule events is saved as a JSON on the server. A schedule event is an individual cleaning event defined inside the schedule.

### Get Schedule For Robot

Phonegap Plugin APIs: RobotMgr.prototype.getScheduleEvents

This calls the web service: <http://neatostaging.rajatogo.com/api/rest/json?method=robotschedule.get_schedule_based_on_type>

It gets the current schedule from the server.

As an implementation detail, it maintains it in a local database as well. This was done because of the requirement of the UI layer team for the locally changing individual schedule events.

The Server schedule copy is treated as the master copy and robot as well as the user should update the schedule on the server.

Calls communicating with local database which interact with a schedule event:

RobotMgr.prototype.createSchedule

RobotMgr.prototype.getScheduleEventData

RobotMgr.prototype.addScheduleEvent

RobotMgr.prototype.updateScheduleEvent

RobotMgr.prototype.deleteScheduleEvent

These calls will basically interact with the local copy of the schedule. Each call would update/add/delete the individual schedule event in the schedule group. Only when the local copy is updated on the server, it becomes the master copy.

Note that the local maintenance of the schedule in the smartapp is just an implementation detail, and it should in no way treated as the master copy while showing the user. The plugin method “RobotMgr.prototype.getScheduleEvents” should be called by the UI layer when showing the schedule to the user which basically fetches it from the server.

### Update Schedule to Server

Phonegap Plugin: RobotMgr.prototype.updateSchedule

This calls the web service: <http://neatostaging.rajatogo.com/api/rest/json?method=robotschedule.update_data>

The schedule on the server is updated and the version number of the schedule is incremented.

## Schedule Sync:

Whenever the schedule is changed, the setter should also call set robot profile details so that other systems would know about the schedule change.

Currently the Smartapps updates the profile details with the key that the schedule is updated so that the other associated users as well as the robot knows about the profile change. For this it calls <http://neatostaging.rajatogo.com/api/rest/json?method=robot.set_profile_details3>.

Key-Value:

Key: schedule\_updated

Value: true.

Note that the value false is never set for this key. The key is added just for tracking the timestamp on which the schedule was changed.

When the user/robot system receives the notification, it will compare the timestamp as always and then only will retrieve the schedule.

Also note that the schedule has its own version number. So if a user who has an old copy of schedule (another user updated the schedule) tries to update it, it will get a version mismatch error. (The schedule version is a simple integer which is incremented on every schedule change). If there is a mismatch error, the latest schedule should be retrieved from the server and changes should be done on that schedule and then should be updated.

Currently if the SmartApp receives schedule change notification, it fires a notification to the UI layer. The UI layer is expected to make a call for getting the latest copy of schedule (if the user is in the schedule page).

Scenario: Schedule Updated when robot is offline:

The robot can call the <http://neatostaging.rajatogo.com/api/rest/json/?method=robotschedule.get_schedule_based_on_type>

to get the schedule version number for the current schedule. It can then compare the current schedule version with the local and update itself if there is a schedule version change.

Robot can also rely on the data changed notification fired when SmartApp when it changes the schedule (as explained above):

1. Whenever the robot schedule is changed, a set robot profile details call is made with key: “schedule\_updated”
2. The notification is sent to robot. When the robot is offline, it will be added to the offline queue in the xmpp.
3. Once the robot comes online, it will get the the notification. Getting the robot profile details and comparing the timestamp the robot should know that the schedule was updated at a later period.

Key: schedule\_updated

If the value is “true” and the timestamp is updated, the robot should call

<http://neatostaging.rajatogo.com/api/rest/json?method=robotschedule.get_schedule_based_on_type>

1. The robot would get the latest schedule.

It can also make sense for the robot to always get the latest schedule when it comes online (instead of waiting for the updation of the “schedule\_updated” key)

It can call: <http://neatostaging.rajatogo.com/api/rest/json?method=robotschedule.get_schedule_based_on_type>

to get the schedule details and compare the schedule version received and update its local schedule if it is changed.

Scenario: Robot changes the schedule

The robot when changing the schedule should update the key “schedule\_updated” using <http://neatostaging.rajatogo.com/api/rest/json?method=robot.set_profile_details3> webservice. The details are same as in the above scenario.

This will fire a data changed notification on the SmartApp (If it is online) and the Plugin layer will notify the JS layer about the schedule change. The JS layer should request for the latest schedule (may be if it is in the schedule page).

If the SmartApp is not online, it should always get the latest schedule from the server while showing the Schedule to the user.

The details of the schedule change notification are explained in the section “UI Unsolicited Data Communication Framework”.

## Set Robot Name

Phonegap plugin API: setRobotName2

Web service called: <http://neatostaging.rajatogo.com/api/rest/json?method=robot.set_profile_details3>

with the appropriate name for the robot. This will fire a notification to relevant entities.The key used is “name”.

## Get Robot Presense Status:

Phonegap plugin API: getRobotOnlineStatus

The webservice called is:

http://neatostaging.rajatogo.com/api/rest/json/?method=robot.get\_robot\_presence\_status

## Create Robot

The robot calls this method to create it’s instance on the server.

The webservice called is:

http://neatostaging.rajatogo.com/api/rest/json/?method=robot.create2

## Get Robot Details

The webservice used to get the robot details

http://neatostaging.rajatogo.com/api/rest/json/?method=robot.get\_details

## Request Linking Code

This webservice is called from the robot to generate a linking code:

Webservice called is:

http://neatostaging.rajatogo.com/api/rest/json/?method=robot.request\_link\_code

# Notifications

There are three types of notifications defined which are –

1. Push Notifications
2. Robot Notifications
3. Robot Errors

## Push Notifications

The push notifications are sent from the robot using:

<http://neatostaging.rajatogo.com/api/rest/json?method=message.send_notification_to_all_users_of_robot2>

This notification is sent to all the users of the robot.

Currently the supported push notifications are:

* Dirt bag full (20002)
* Robot is stuck (20001)
* Dirt Bin Missing (20006)
* Plug the cable (20004)
* Robot Cancel Error (20005)
* Cleaning is done (20003)

Each notification has a specific unique ID which is sent to the SmartApp (shown in the brackets above).

The SmartApp has the option to disable these notifications (explained earlier: Set Push Notification Settings).

These notifications are received even when the application is NOT alive.

## Robot Messages: Notifications / Error

The robot notifications and errors are the messages which are to be propagated to the SmartApps when the applications is running and is in foreground.

The robot messages have been distinguished into two types which are– “Notifications” and “Errors”.

## Notifications

There can be many notifications which the robot would want to send to the user for information.

This can be done by following steps:

**Webservice Method Name**: “robot.set\_profile\_details3”

[Note this method name is prefixed by the base json url for the server in use].

**Key-Value:**

**Key –** robotNotificationMsg

Value: <the json string for the message as described above>

This will trigger a data changed notifications which will ask the SmartApp to make the call to get the latest notification queued for it.

The actual notification is represented by a JSON String.

Currently the Simulator sends a notification json as follows:

**{“messageID”: <id for the message>}**

The **messageID** is the unique identifier for the notification which the SmartApp should interpret and show to the user. More attributes can be added to this structure if needed.

**The message ids for the notifications should be defined by the robot team for various notifications. The robot team can also add other attributes to this JSON object which the SmartApp UI should handle.**

**Example:**

If a robot “yana” wants to send a “Low Battery” notification with **messageID** being 10001.

|  |  |
| --- | --- |
| cause\_agent\_id | yana |
| profile[robotNotificationMsg] | {"messageID":10001} |
| notification\_flag | 1 |
| api\_key | 1e26686d806d82144a71ea9a99d1b3169adaad917 |
| source\_serial\_number | yana |
| serial\_number | yana |

This will pass the data changed notification to the smartapps which then will process the notification.

### Errors

There can be many errors which the robot would want to send to the user for information.

This can be done by following steps:

**Webservice Method Name**: “robot.set\_profile\_details3”

[Note this method name is prefixed by the base json url for the server in use].

**Key-Value:**

robotErrorMsg: <the json string for the message as described above>

This will trigger a data changed notifications which will make the SmartApp to make the call to get the latest error queued for it.

The actual error is represented by a JSON String.

Currently the robot sends an error JSON as follows (similar to the notification):

{“messageID”: <id for the message>}

The messageID is the unique identifier for the notification which the UI layer should interpret and show to the user. More attributes can be added to this structure if needed.

**The message ids for the errors should be defined by the robot team for various notifications. The robot team can also add other attributes to this JSON object which the SmartApp UI should handle.**

Example:

If the robot “yana” wants to send a “Cannot Find Base” error with messageID being 20001.

The webservice post parameters for setRobotProfileDetails3 would be:

|  |  |
| --- | --- |
| cause\_agent\_id | yana |
| notification\_flag | 1 |
| profile[robotErrorMsg] | {"messageID":20001} |
| api\_key | 1e26686d806d82144a71ea9a99d1b3169adaad917 |
| source\_serial\_number | yana |
| serial\_number | yana |

This will pass the data changed notification to the smartapps which then will process the error.

Webservice Method Name: <http://neatostaging.rajatogo.com/api/rest/json?method=robot.set_profile_details3>

There is a specified key for the error and the robot should update the value on the server to set the error object as the value for the key.

The actual error is represented by a JSON String.

Currently the Simulator sends an error json as follows (similar to the notification):

{“messageID”: <id for the message>}

The messageID is the unique identifier for the notification which the UI layer should interpret and show to the user. More attributes can be added to this structure if needed.

This will pass the data changed notification to the Smartapps which then will process the error.

The notification is passed to the UI layer by the Data Communication framework explained in the top section.

# UI: Handling Robot Notifications/Errors

The notification framework uses the “Data Communication” framework described in the top section.

A general notification received as callback is:

{robotDataKeyId:"robotDataKeyId", robotId:"robotId", robotData:"robotData"}

The keyID for a notification and error are:

var ROBOT\_MESSAGE\_NOTIFICATION = 4013;

var ROBOT\_MESSAGE\_ERROR = 4014;

which are defined in neatosmartapphelper.js.

If a notification is received for these Key Ids, then the robotData consists of the notification json sent by the robot.

The format for the robotData is:

Notification: { robotNotification: {JSON notification string from robot}}

Error: {robotError: {JSON error string from robot}}

The current notification/error format as described above is:

{“messageID”:<id for the message>}.

The id will be a unique identifier for the notification/error.

**The message ids for the errors/notifications should be defined by the robot team for various notifications. The robot team can also add other attributes to this JSON object which the SmartApp UI should handle.**

**Example**:

If the robot sends a low battery notification like explained in above section’s example, the UX will get a callback with the result json being:

{robotDataKeyId:4013, robotId:testrobot, robotData: { robotNotification: {messageID: 10001}}}

Here by the messageID the UX would show the appropriate notification to the UI.

End