Wireless Debug API Specification

2017-05-26

Jonathan Sumner Evans
Amanda Giles
Reece Hughes
Daichi Jameson
Tiffany Kalin

Table of Contents

1 Overview

Wireless Debug is a combination of mobile APIs, a web application, and supporting libraries which allow remote debugging of mobile applications. Wireless Debug can be set up on a single server and allow multiple users with multiple devices each testing multiple applications to connect to the same server seamlessly. Wireless Debug can also be run by a single user on their local machine. This document describes the API between all of the components of Wireless Debug. See the Wireless Debug Design document for a higher level overview of the project.

2 Technology

Communication between the Mobile API and the Web App Backend will be done using WebSockets. Communication between the Web App Backend and the Web Interface will be done through a combination of WebSockets and AJAX. WebSockets will be used when the request does not expect a response. AJAX will be used when the request expects a response.

All WebSockets messages and AJAX requests will contain a payload as described in the sections below.

The Mobile API is a Java API for Android.

The Parsing Library is a Python module.

The Datastore Interface and the User Management Interface are Python classes for which none of the functions are implemented. Implementation classes can inherit from these classes and implement the functions.

3 URL Scheme

The WebSockets URL will be `ws(s)://<base_url>/ws`.

The URL for AJAX requests will be `http(s)://<base_url>/<action>`.

4 WebSocket message payload specification

4.1 Fields on all messages

Each message payload will have the following field:

- `messageType (string)`: the type of message being sent

4.2 Mobile API → Web App Backend

4.2.1 Start Session

Message Type: `startSession`

Payload Type: JSON

Purpose: when the app starts, a WebSockets connection to the Web App Backend will be established and a `startSession` message will immediately be sent to the Web App Backend. This message will tell the Web App Backend which device and app has connected.

Fields:

- `apiKey (string)`: the user's API Key that is shown to the user on the Web Interface.
- `osType (string)`: the type of OS from which the logs are being sent. Valid values are: "Android" and "iOS".
- `deviceName (string)`: the device name from which the logs are being sent.
- `appName (string)`: the name of the application from which logs are being sent.

Example:

```
{
    "messageType": "startSession",
    "apiKey": { ... },
    "osType": "Android",
    "deviceName": "Google Pixel",
    "appName": "Google Hangouts"
}
```

4.2.2 Log Dump

Message Type: `logDump`

Payload Type: JSON

Purpose: the Mobile API will send a `logDump` message to the Web App Backend with raw log data at an interval configured by the user of the Mobile API. Log entries in `rawLogData` are delimited by newline characters.

Fields:

- `rawLogData (string)`: the actual raw log data
- `osType (string)`: the type of OS from which the logs are being sent. Valid values are: "Android" and "iOS".

Example:

```
{
    "messageType": "logDump",
    “osType”: “Android”,
    "rawLogData": "05-22 11:44:31.180
7080-7080/com.google.wireless.debugging I/WiDB Example: aX: 3.0262709 aY:
2.0685902\n05-22 11:44:31.182 7080-7080/com.google.wireless.debugging
I/WiDB Example: aX: 3.193911 aY: 2.3091934"
}
```

```
{
    "messageType": "logDump",
    “osType”: “Android”,
    "rawLogData": "05-24 12:12:49.247 23930 23930 E AndroidRuntime: FATAL
EXCEPTION: main\n05-24 12:12:49.247 23930 23930 E AndroidRuntime: Process:
com.google.wireless.debugging, PID: 23930\n05-24 12:12:49.247 23930 23930 E
AndroidRuntime: \tat java.lang.reflect.Method.invokeNative(Native
Method)\n05-24 12:12:49.247 23930 23930 E AndroidRuntime: \tat
java.lang.reflect.Method.invoke(Method.java:515)\n05-24 12:12:49.247 23930
23930 E AndroidRuntime: \tat
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:
796)\n05-24 12:12:49.247 23930 23930 E AndroidRuntime: \tat
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:612)\n05-24
12:12:49.247 23930 23930 E AndroidRuntime: \tat
dalvik.system.NativeStart.main(Native Method)"
}
```

4.2.3 Device Metrics

Message Type: `deviceMetrics`

Payload Type: JSON

Purpose: the Mobile API will send a `deviceMetrics` message to the Web App Backend with statistics regarding the device health and performance. All data is specified in bytes.

Fields:

- `osType (string)`: the type of OS from which the logs are being sent. Valid values are: "Android" and "iOS".
- `cpuUsage (double)`: the actual raw log data
- `memUsage (int)`: current kilobytes used (active)
- `memTotal (int)`: total amount of memory available
- `netSentPerSec (double)`: bytes sent per second over the network
- `netReceivePerSec (double)`: bytes received per second over the network
- `timeStamp (int)`: the difference, measured in milliseconds, between the time of recording metrics, and midnight, January 1, 1970 UTC.

4.2.4 End Session

Message Type: `endSession`

Payload Type: JSON

Purpose: when the app closes and the Mobile API has finished sending all the logs to the Web App Backend, an `endSession` message will be sent which will tell the Web App Backend that the session is over.

Fields: No extra fields

Example:

```
{
    "messageType": "endSession"
}
```

4.3 Web App Backend → Web Interface

4.3.1 Log Entries

Message Type: `logData`

Payload Type: JSON

Purpose: when the Web App Backend receives a `logDump` from the Mobile API, it will parse the log via the parsing library and then it will send a `logData` message to all Web Interfaces that the user is connected to.

Fields:

- `osType (string)`: the type of OS from which the logs are being sent. Valid values are: "Android" and "iOS".
- `logEntries`: formatted HTML table rows to append to the log table. The table divisions will be ordered: time, tag, log type, log text.

Example:

```
{
    "messageType": "logData",
    "osType": "Android",
    "logEntries" : "
<tr class="exception">
    <td>2017-11-06 16:34:41.000</td>
    <td>TEST</td>
    <td>Exception</td>
    <td>FATAL EXCEPTION: main<br>
      Process: com.google.wireless.debugging, PID: 23930<br>
      java.lang.RuntimeException: Forced Crash<br>
      at
com.google.wireless.debugging.example.MainFragment$2.onClick(MainFragment.j
ava:73)<br>
      at android.view.View.performClick(View.java:4445)<br>
      at android.view.View$PerformClick.run(View.java:18446)<br>
      at android.os.Handler.handleCallback(Handler.java:733)<br>
      at android.os.Handler.dispatchMessage(Handler.java:95)<br>
      at android.os.Looper.loop(Looper.java:136)<br>
      at java.lang.reflect.Method.invoke(Method.java:515)<br>
      at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:
796)<br>
      at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:612)<br>
      at dalvik.system.NativeStart.main(Native Method)
    </td>
</tr>
<tr class="exception">
    <td>2017-11-06 16:34:41.001</td>
    <td>TEST</td>
    <td>Info</td>
    <td>test log</td>
</tr>"
}
```

4.3 Web Interface → Web App Backend

4.3.1 Associate User

Message Type: `associateUser`

Payload Type: JSON

Purpose: when a user logs in to the web interface, this message will be sent to the Web App Backend to tell the backend which logs to send to the Web Interface.

Fields:

- `apiKey (string)`: the API Key from the user defined User Management Interface.

Example:

```
{
    "messageType": "associateUser",
    "apiKey": "cb572446-21f2-44d0-8983-4cf30300b574"
}
```

5 AJAX Requests

5.1 Get User's Devices

Action Name: `getDeviceList`

Payload Type: JSON

Example URL: `https://<base_url>/devicelist`

HTTP Request Type: `GET`

Purpose: the Web Interface will send a request to the Web App Backend to retrieve a list of all the devices the current user has historical debugging sessions on.

Parameters:

- `apiKey (string)`: the API Key from the user-defined User Management Interface.

Return Value:

The response will contain the following fields:

- `devices (array)`: a list of devices aliases
    - `success (boolean)`: if the device list is empty or not

Example:

```
{
    “success”: True
    "devices": {‘Google Pixel’, ‘Samsung Galaxy S8’}
}
```

5.2 Get User’s Apps

Action Name: `getAppList`

Payload Type: JSON

Example URL: `https://<base_url>/appList`

HTTP Request Type: `GET`

Purpose: get the app aliases for a given user and device where historical log information exists.

Parameters:

- `apiKey (string)`: the ID Token from the user-defined User Management Interface.
- `device (string)`: the device name to load the session from
-

Return Value:

- `apps (array)`: list of the app aliases of the different sessions for the given device and apiKey.

Example:

```
["Flappy Birds", "Google Hangouts", "Google Hangouts", "Google Drive"]
```

5.3 Get User’s Sessions for Device/App

Action Name: `getAppListAppSessions`

Payload Type: JSON

Example URL: `https://<base_url>/sessionList`

HTTP Request Type: `GET`

Purpose: get the historical log sessions start times for the app.

Parameters:

- `apiKey (string)`: the ID Token from the user-defined User Management Interface.
- `device (string)`: the device name to load the session start times from
- `app (string)`: the application name to load the sessions start times from

Return Value:

- `starttimes (array)`: list of the app aliases start times of the different sessions for the given device and apiKey. and app

Example:

```
["2017-11-06T16:32:13.000Z", "2017-11-06T16:34:41.000Z",
"2017-11-06T16:43:54.000Z", "2017-11-06T16:55:26.000Z"]
```

5.4 Get Historical Log Session

Action Name: `getLogs`

Payload Type: HTML

Example URL: `https://<base_url>/logs`

HTTP Request Type: `GET`

Purpose: if a user wants to view a historical log session, the Web Interface will make a request to this URL.

Parameters:

- `apiKey (string)`: the ID Token from the user-defined User Management Interface.
- `device (string)`: the device name to load the session from
- `app (string)`: the application name
- `starttime (datetime)`: the start time of the historical log

Return Value: formatted HTML table rows to append to the log table. The table divisions will be ordered: time, tag, log type, log text.

See Section 4.3.1 for an example payload.

6 Other HTTP Requests

6.1 Webpage Requests

If a user requests the Wireless Debug instance's root URL (`http(s)://<base_url>`), the following will occur:

- If the user is logged in, they will be shown the Wireless Debug application page.
- If the user is not logged in, they will be redirected to the login page.

To determine whether the user is logged in, the User Management Interface (Section 10) will be called to determine whether the user is properly authenticated.

If the user requests the login page, the server will retrieve the login UI HTML from the User Management Interface and present it to the user (along with the app styles).

6.2 Login

Action Name: `login`

Payload Type: HTML Form Data

Example URL: `https://<base_url>/login`

HTTP Request Type: `POST`

Purpose: the Web Interface use this action to log the user in. Note that this login is not necessarily secure. It is meant to provide API to enable secure login, but it is not in and of itself secure.

Form Fields: Anything. The values of these fields (which are defined by the User Management Interface) will be passed to the User Management Interface to perform authentication.

Return Value: If the login is successful, the user will be redirected to the main application page. The User Management Interface can optionally set a cookie on the response containing some sort of identifier.

7 Parsing Library ↔ Web App Backend

The Web App Backend will send raw logs to the Parsing Library which will parse the raw logs into log entries. Log entries can be returned as Python dictionaries and/or formatted HTML.

The Parsing Library will expose a single class: `LogParser`. This class will have two public static functions:

`parse (raw_log_data, type_file)`

- **Arguments:**
  - `raw_log_data (string)`: the raw log data from the Mobile API.
  - `type_file: The type of return value. Default is 'dict'. Return types explained in detail below.`
- **Return Value:**
  - **`return_type = "dict":`** A Python list of Log Entry dictionaries, each of which have the following fields:
    - `time (datetime)`: the time the log occurred
    - `text (string)`: the text of the log
    - `tag (string)`: The tag on the log. For Android, this allows the user to specify an additional annotation to add to their log entry for organizational purposes.
    - `logType (string)`: The type of log (Warning, Error, Info, etc.)

`convert_to_html(parsed_log_dict)`

- **Arguments**
  - `parsed_log_dict (dict)`: A python list of Log Entry dictionaries, each of which have the following fields:
    - `time (datetime)`: the time the log occurred
    - `text (string)`: the text of the log
    - `tag (string)`: The tag on the log. For Android, this allows the user to specify an additional annotation to add to their log entry for organizational purposes.
  - `logType (string)`: The type of log (Warning, Error, Info, etc.)
- **Return Value**
  - `(string)`: A string containing the HTML table rows (not the entire table) which represent the log entries.

8 Android Client API

The Android Client API will provide the following static function in a class called `WirelessDebugger`:

- `start(String server, String apiKey, Context appContext, int interval = 200)` - initializes Wireless Debug and sends a batch of logs to the specified server using API Key authentication. The interval is an optional field to specify how frequently logs are sent to the server. The value is in milliseconds and the default is 200ms.

Example:

```
WirelessDebugger.start("yourserver.com",
      "f7f1a540-d652-46a9-b6b8-eab24d69eadc", getApplicationContext());
```

9 Datastore Interface

For a class to implement the Datastore Interface, it will need to define the following functions.

Any set up operations should occur in the constructor of the Datastore Interface.

Device names and app names can be passed as de-aliased device names and de-aliased app names, respectively.

`store_logs(api_key, device_name, app_name, start_time, os_type, log_entries)`

- **Purpose:** store a set of log entries to the datastore. This function may be called multiple times per session, so it must append the log entries in the storage mechanism.
- **Arguments:**
  - `api_key`: the API Key associated with the logs
  - `device_name`: the name of the device associated with the logs
  - `app_name`: the name of the app associated with the logs
  - `start_time`: the time that the session started
  - `os_type`: the OS type (iOS or Android)
  - `log_entries`: the log entries to store

`set_session_over(api_key, device_name, app_name, start_time)`

- **Purpose:** called to indicate to the datastore that the session is over. This can set a flag on the session in the datastore indicating that it should not be modified, for example.
- **Arguments:**
  - `api_key`: the API Key associated with the logs
  - `device_name`: the name of the device associated with the logs
  - `app_name`: the name of the app associated with the logs
  - `start_time`: the time that the session started

`retrieve_logs(api_key, device_name, app_name, start_time)`

- **Purpose:** retrieve the logs for a given session.
- **Arguments:**
  - `api_key`: the API Key to retrieve logs for
  - `device_name`: the name of the device to retrieve logs for
  - `app_name`: the name of the app to retrieve logs for
  - `start_time`: the time that the session started
- **Returns:** a dictionary with:
  - `osType`: the OS type

- `logEntries`: a list of log entries as Python dictionaries

`retrieve_devices(api_key)`

- **Purpose:** retrieve a list of devices associated with the given API Key.
- **Arguments:**
  - `api_key`: the API Key to retrieve devices for
- **Returns:** a list of devices associated with the given API Key. Each device is a dictionary with:
  - `array`: array of the names of the devices

`retrieve_apps(api_key, device_name)`

- **Purpose:** retrieve a list of apps associated with the given API Key and device.
- **Arguments:**
  - `api_key`: the API Key to retrieve apps for
  - `device_name:` the device name to retrieve apps for
- **Returns:**
  - `array` (string): a list of app names associated with the given device.

`retrieve_sessions(api_key, device_name, app)`

- **Purpose:** retrieve a list of sessions for a given API Key, device, and app.
- **Arguments:**
  - `api_key`: the API Key to retrieve sessions for
  - `device_name`: the name of the device to retrieve sessions for
  - `app_name`: the name of the app to retrieve sessions for
- **Returns:**
  - `array` (datetime objects): list of `datetime` objects, one for each of the session start times associated with the given API Key, device, and app.

`add_device_app(api_key, device_name, app)`

- **Purpose:** add a device/app combination to the device/app collection
- **Arguments:**
  - api_key: api_key of user
  - `device_name`: the name of the device
  - `app_name`: the name of the app

update_alias_device(api_key, device_raw_name, device_alias)

- **Purpose**: add a mapping from a device to a device alias.
- **Arguments**:
    - `api_key`: the API Key of the device's owner
    - `device_raw_name`: the raw device name
    - `device_alias`: the new alias of the device

update_alias_app(api_key, device_name, app_raw_name, app_alias)

- **Purpose**: add a mapping from a device to a device alias.
- **Arguments**:
    - `api_key`: the API Key of the device's owner
    - `device_name`: the raw device name
    - `app_raw_name`: name being aliased
    - `app_alias`: the new alias of the app

`get_raw_device_name_from_alias(api_key, device_alias)`

- **Purpose:** retrieve a raw device name given a device alias.
- **Arguments:**
    - `api_key`: the API Key of the device's owner
    - `device_alias`: the alias of the device
- **Returns:** `string`: the raw device name

`get_raw_app_name_from_alias(api_key, device_name, app_alias)`

- **Purpose:** retrieve a raw device name given a device alias.
- **Arguments:**
    - `api_key`: the API Key of the device's owner
    - `device_name`: device connected to the app
    - `app_alias`: the alias of the app
- **Returns:** `string`: the raw app name

clear_datastore()

- **Purpose:** clears datastore of records

10 User Management Interface

For a class to implement the User Management Interface, it will need to define the following functions:

`get_login_ui()`

- **Purpose:** generate an HTML UI to show on the login page.
- **Arguments:**
  - `None`
- **Returns**: a string with the HTML for the login UI. This HTML should include an HTML form which posts to the login URL (See Section 6.2).

`is_user_logged_in(request)`

- **Purpose:** determines if the user is already authenticated.
- **Arguments:**
  - `request`: the HTTP request context from Bottle
- **Returns:** true if the user is authenticated, false if not

`handle_login(form_data, request, response)`

- **Purpose:** perform server-side user authentication.
- **Arguments:**
  - `form_data`: the form data from the form generated by `get_login_ui`.
  - `request`: the HTTP request context from Bottle
  - `response`: the HTTP response context from Bottle
- **Returns:** A tuple containing:
  - login_successful: a boolean indicating if the login was successful or not.
  - error_message: a string containing the error message if the login failed. If login_successful is true, this can return either a blank string or log message to describe anything important.
- **Note:** this function can set cookies on the `response` object

`get_api_key_for_user(request)`

- **Purpose:** returns the API Key associated with the `request`.
- **Arguments:**
  - `request`: the HTTP request context from Bottle

- **Returns:** the API Key associated with the given request. The `request` parameter can be used to access any cookies that may have been set in `handle_login`.
- **Note:** this function can retrieve cookie values using the `request` object

`find_associated_websockets(api_key, websocket_connections)`

- **Purpose:** determines which WebSocket connections to send parsed logs to.
- **Arguments:**
  - `api_key`: the API Key for which to find associated sessions
  - `websocket_connections`: a list of WebSocket connections to Web UIs connected to the Web App Backend
- **Returns:** a list of WebSocket connections to send the logs to (this should be a subset of the `websockets_connections` list).