Wireless Debugging for Mobile Applications

2017-06-21

Team Members

Jonathan Sumner Evans
Amanda Giles
Reece Hughes
Daichi Jameson
Tiffany Kalin

Clients

Chris Navrides
Eugene Wang

Introduction

Description

Debugging mobile applications can be a frustrating task, especially because mobile developers need to be connected by wire to their workstations in order to receive a real-time feed of debugging information. If a developer would like to detach from their workstations, they must run their program and then collect log files from the mobile device after the application has run. This is a very inefficient process.

Google has asked a Colorado School of Mines field session team to build a solution that will streamline the mobile app development process by allowing users to view mobile application logs in real time over a wireless connection. This will greatly reduce the time needed to obtain logs, search through them to find error messages, or check statuses while debugging. Additionally, the team will provide ways for users to filter and sort through logs in an easy-to-use interface.

Definitions

- Logs: Messages seen on the Android Debug Bridge (Android) and NSLog (iOS)
- Mobile API: APIs which can be included in a mobile app to enable wireless logging
- Web UI: the interface where users can view the logs coming from the device
- Web Server Library: receives logs from the Mobile API. Parses, stores and sends logs to the Web Interface

Requirements

Functional Requirements

- Allow for a wireless connection from a mobile device to a server over which logs can be streamed
- Mobile API that can be attached to app projects to stream logs to a specified server when the application is started
- Web server library that receives, parses and stores the logs from a mobile device, then sends them to the web UI. Error information is also sent to web UI for Android
- Allow for session management (client side provides sessions by sending session ID, web server library keeps track of sessions, and web UI provides mechanism to select a session)
- Two test apps:
    - Android application to show use of the Android mobile API
    - Web Server instance to show use of the web server library
- Web UI allows the user to view logs in a readable format
    - View logs based on session ID
    - Real-time streaming of logs
    - Real-time filtering of logs
- Android support of mobile API
- iOS Support of mobile API
- Mobile API must work on Android API level 21 and above

Non-functional Requirements

- Identify logs by user, device, and session
- Platform independent - ability to run the library on any server
- Web Server Library can run locally on a single computer
- Web Server Library can run on a server and can support multiple users and devices
- Documentation of the project for other developers to use and extend the library
    - API calls
    - Walkthrough of implementation
    - Sample Deployment instance
- All code is written according to Google’s style guides

System Architecture & Design Decisions

Overview

Wireless Debug is a combination of mobile libraries, a web application, and supporting libraries which allow remote debugging of mobile applications. Wireless Debug has six major components:

1. **Mobile Library for Android and iOS:** streams logs from the mobile application to the web app backend.
2. **Web App Backend:** receives logs from the Mobile Library, sends them to the Log Parsing Library (3), saves them to the database (4), and sends the logs to the appropriate users (5) on the Web UI (6).
3. **Log Parsing Library:** parses a set of logs into either HTML or JSON.
4. **Datastore Interface:** an abstract API which allows users to create a connection to any database (or no database at all) so that they can store historical log sessions.
5. **User Management Interface:** an API that allows users to create an authentication structure and determine which web UIs to stream logs to.
6. **Web UI:** a website that displays parsed logs to the user in a readable format. The user is also able to filter and sort through logs from the Web UI.

The components were split up in this manner because it compartmentalizes the solution into (mostly) independent components.

By separating these components, we allow the user to change the functionality of Wireless Debug to fit their needs. For example, the Log Parsing Library can be included in any application, the Datastore Interface could be implemented to use any storage mechanism such as MongoDB, Bigtable, or tape reels, and the User Management Interface could be implemented to just require the user to enter an email or be more complex and require Google authentication. Wireless Debug can be set up on a server to allow multiple users with multiple devices each testing multiple applications to connect to the same server seamlessly. Additionally, setting up Wireless Debug on a server allows for debugging over a cellular connection. Wireless Debug can also be run by a single user on their local machine.

After setting up a server, when a user wants to use Wireless Debug, they need to:

1. Log in to the Web UI (this uses the login UI determined by the User Management Interface).
2. Add the Mobile Library to their app.
3. Paste an API key that they get from the Web UI into the Mobile Library.

Once the user has completed these steps and runs their app, the Mobile Library sends raw logs from the mobile device to the Web App Backend. The Web App Backend then passes the logs

to the Log Parsing Library where the logs are converted into log entries (JSON objects and/or HTML table rows which represent the log data from the Mobile Library). The log entries are then sent to the appropriate Web UI(s) (as determined by the User Management Interface) where they are displayed to the user. Additionally, the log entries can be saved using the Datastore Interface.

Device metrics for Android such as memory, CPU, and network usage are streamed by the Mobile Library and presented to the user on the Web UI. Future work will include developing the iOS library to do this.

Figure 1 gives a high-level overview of the connections between all of these components.

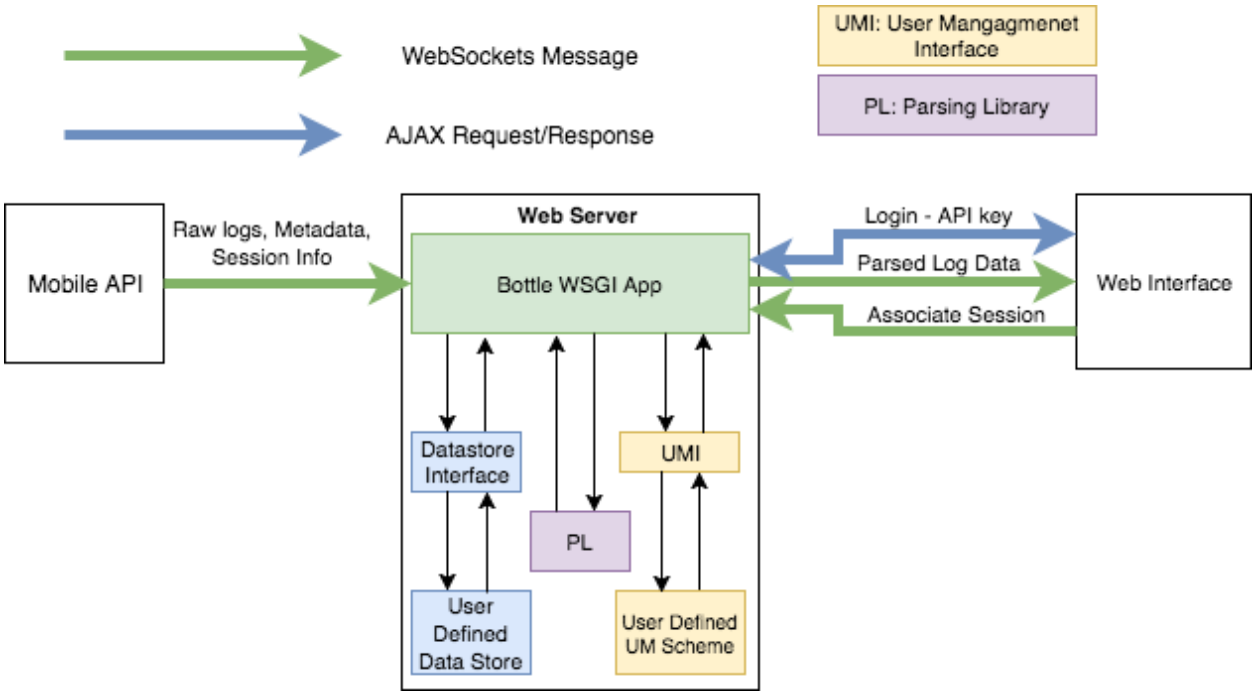

Figure 1: The connections between the main components of Wireless Debug

Technical Design and Decisions

The following list is an overview of the technical responsibilities of the various components that compose Wireless Debug.

- **Mobile Library:** communicates with the Web App Backend through a WebSocket connection. The Mobile Library sends messages indicating the start and end of logging sessions, and sends messages containing the logs from the application. The Web App Backend sends nothing back to the Mobile Library.
- **Web App Backend:** connects all other components. Communicates with Mobile Library and Web UI using WebSockets and AJAX.

- **Parsing Library:** parses logs into a readable format. It takes raw log data and outputs logs either as JSON and/or HTML. The library is designed so it can be included in any application.
- **Datastore Interface:** specifies a set of functions for users to implement to enable historical log storage. This interface is generic and allows users to use any database (or no database at all) and store the data using any method.
- **User Management Interface:** defines a set of functions to implement to enable user management. The interface is generic and allows users to use any login method.
- **Web UI:** serves as the user-facing part of the application. It receives parsed logs in HTML format over a WebSocket connection to the Web App Backend and shows them to the user. Filtering and sorting of logs is done from the Web UI. The Web UI also visualizes device metrics for the user.

Mobile Library

The Mobile Library is a debugging library that a mobile developer may easily add to their app. The library is added by including the module in their project and calling the Wireless Debug start method when their app is launched. The arguments of the Wireless Debug start method are where the user specifies the destination of the logs (IP/host name) and their API Key. Once the app is running, the Mobile Library establishes a WebSocket connection to the Web App Backend and streams the logs to the backend. After the application stops, the Mobile Library sends any remaining queued logs then exits.

To determine which user to associate the Mobile Library logs with, the constructor accepts an API Key. The API Key is sent to the Web App Backend and then forwarded to the User Management Interface to determine which Web UIs to stream the logs to.

Web App Backend

The Web App Backend connects all of the components together. It communicates with the Mobile Library, Parsing Library, Datastore Interface, User Management Interface, and the Web UI.

The Web App Backend is implemented in Python 3 using the micro web-framework Bottle. We chose this framework because some team members have prior experience with it and because it is very lightweight. Deployment of the application is extremely flexible. It can run on a user's machine, a dedicated server, or on a cloud service. Examples of deployment options are App Engine, AWS, or Heroku.

Data and User Flows

The general flow of log data through this component is:

1. Log data is received from the Mobile API.

2. Log data is sent to the parsing library which returns both HTML and JSON representations of the log entries.
3. The JSON representation of the log entry is sent to the Datastore Interface which handles storage of the data.
4. The API Key associated with the log entries is sent to the User Management Interface which returns a list of WebSocket connections to send logs to.
5. The log entry HTML is sent to the WebSockets connections returned from (4).

When an unauthenticated user requests the Wireless Debug website, the following occurs:

1. The User Management Interface is called and returns the login UI HTML.
2. The HTML from (1) is displayed to the user.
3. The user logs in, and then the authenticated user case occurs (below).

When an authenticated user requests the Wireless Debug website, the following occurs:

1. The User Management Interface is called and returns true, indicating that the user is authenticated.
2. The User Management Interface returns the user's API Key. (If this is the first time that the user logged in, this is automatically generated.)
3. The main app HTML page is returned to the user, giving them instructions on how to include the library in their application and showing them their API Key.

Communication Technology

Communication between the Web Interface and the Web App Backend occurs using WebSockets and AJAX. We are using WebSockets for any action that does not expect a response (for example, sending parsed logs). AJAX is being used for any action that expects a response (for example, retrieving a list of historical sessions). AJAX provides better transparency for determining which response corresponds to a request, thus we deemed it more appropriate for actions that require a response.

Parsing Library

The parsing library takes raw logs from the Mobile Library and parses them into log entries. These log entries are either JSON objects that contain information about the log or formatted HTML. We decided to allow the user of the library to specify the output format allowing the library to be used in any application.

For Wireless Debug, parsed HTML is passed from the backend to the Web UI. While larger and less machine-readable than JSON files, pre-parsing the HTML on the backend reduces the amount of processing and formatting that must occur in JavaScript on the Web UI. All the Web UI needs to do is append data to the list of log entries it has already received. JSON objects are passed from the backend to the Datastore Interface. JSON objects provide a significantly more

machine-readable format and are much simpler to work with in Python since they are effectively the same as Python dictionaries. Additionally, the JSON format works well with certain forms of data storage, such as MongoDB.

Datastore Interface

The Datastore Interface specifies a set of functions required to implement historical log storage. In addition to being streamed to the Web UI, logs are streamed into the Datastore Interface. If the Datastore Interface is implemented, logs will then be forwarded to some form of long term storage, such as a database. The Datastore Interface also determines how historical logs are retrieved from the datastore. Two example implementations are provided with the Wireless Debug application including one which uses a MongoDB backend and one which does not store any log history.

User Management Interface

The User Management Interface defines how users are authenticated, how users are identified, and who should receive logs. Implementation details are left up to the interface implementation chosen by the user. This interface describes the authentication process for a user, how user information is stored, and where logs from mobile devices are transmitted.

If the user wants user management, they will have to implement a Python class. This class includes functions that return the Login UI, generate or look up the API Key for the user, and determine what WebSocket connections to send logs to. Most implementations of a User Management Interface need to implement some method of storing a map of identifiers to API Keys.

Two example implementations of the User Management Interface are provided. The first example contains a simple form of authentication where the user logs in using their email and email to API Key relationships are stored in a text file. The second example does not perform any authentication, and the logs from the mobile device are broadcasted to all Web UIs.

Web UI

The main purpose of the Web UI is to display parsed logs to the user and allow them to filter and sort parsed logs. The Web UI receives the parsed logs from the server through a WebSocket connection, which are displayed in an HTML table. The table consists of columns for the time, type, tag, and log text. Each row generated is a new log entry from the app. The row is colored red if the log type is an error, yellow if the log type is a warning, and remains white otherwise. Multi-line error messages are displayed in the same table row and the multiple lines are separated by newlines in the “Log Text” column. If two errors occur sequentially, they

are displayed in two separate rows of the table. Logs can also be displayed by uploading them in a file or pasting in the raw logs.

The Web UI has features for filtering log messages and sorting the output. The Web UI also displays device diagnostics including memory usage, CPU usage, and network usage. The Web UI will also show the user's API key if they need to enter it into their mobile device.

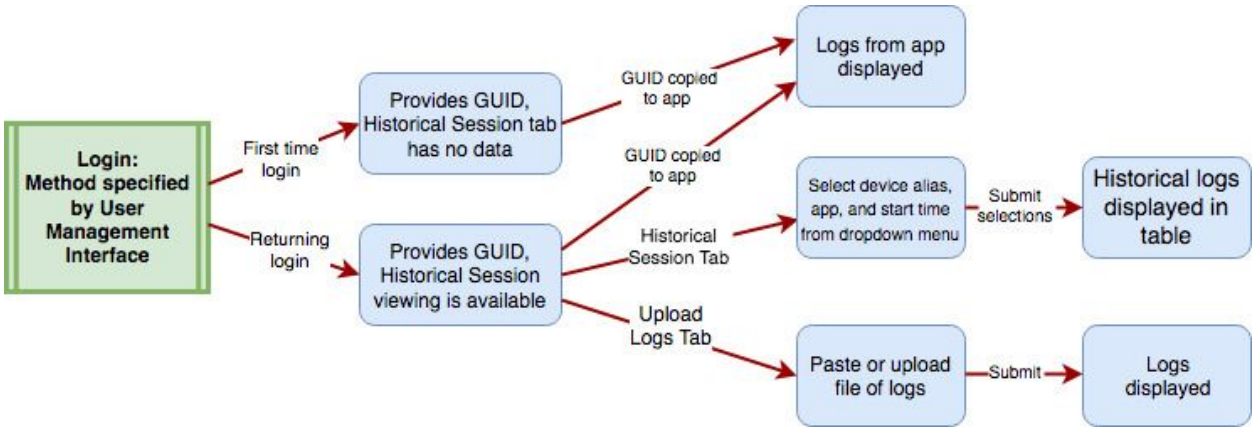

Figure 2: Flowchart of user login experience

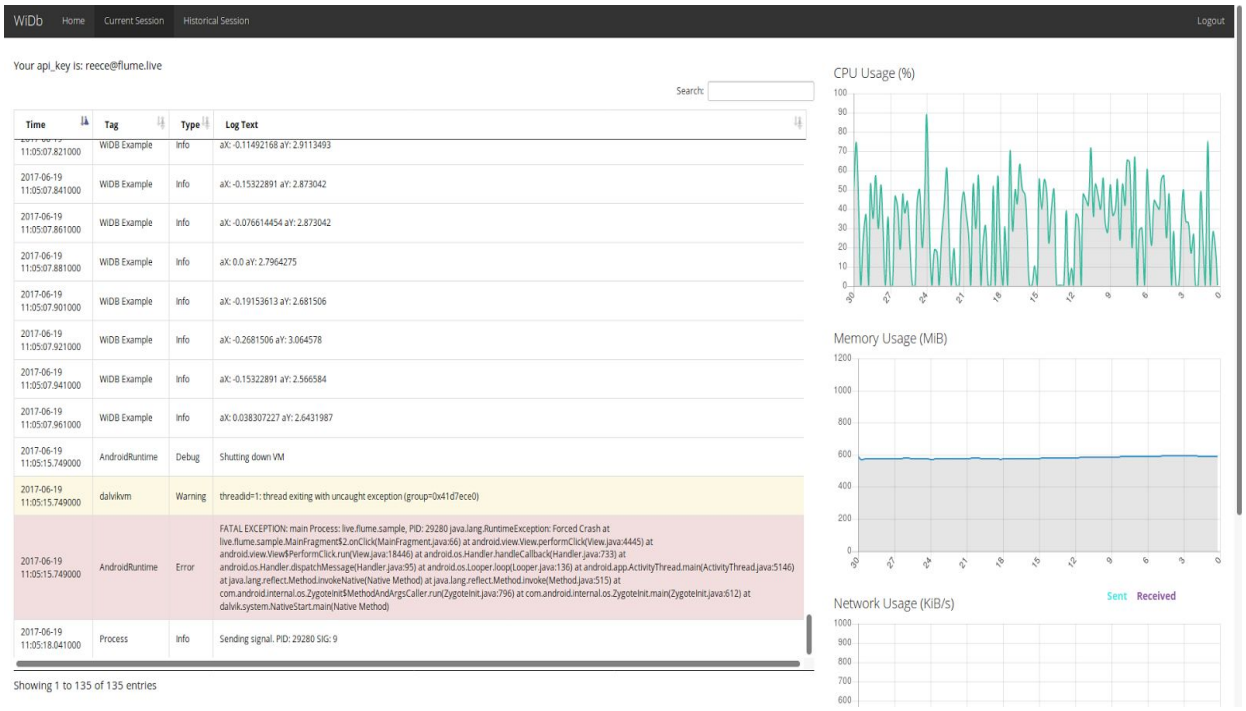

Figure 3: Web UI

Results

We successfully created a minimum viable project within the first two weeks, allowing us to spend the majority of our time working on additional features that our client found most valuable. We had many ideas for features we could add, but we were not able to implement all of them. For example, we are able to stream device metrics from Android devices to the server, however, we have not yet implemented this feature for iOS devices. Additionally, our program does not save the device metrics being streamed from Android devices. We were hoping to save these in order to allow users to view historical device metrics while they are viewing historical logs. Furthermore, we currently have authentication based solely on a user's email but we originally planned to also set up Google authentication as an alternative option. Finally, we did not implement filtering by type. Filtering by type would allow a user to use the current search command within fields, such as log type or tag,, such as log type or tag, as opposed to searching the table overall. The features that were implemented in the given time frame were the most important features for the functionality of the application.

In regards to agile programming, daily standups and weekly sprints greatly improved the efficiency and communication within our group. In retrospect, we realize that splitting large stories into smaller, more specific stories would have made it easier to accurately estimate the time needed for each story. Better time estimates would have improved our sprint planning and agile processes overall. Another lesson we learned is to consider more than one library before deciding which one to use. We wasted time trying to implement our device metric graphing with the D3 data visualization library and eventually switched to a library called Chart.js because it has built in features for live updating the graphs.

In the future, it would be great to see how users interact with our product to test its usability. We would also like to have a focus group to see which features are still needed for our project and give us the necessary feedback in order to improve our product and move it forward. For now, we have exceeded the basic requirements stated by our client and we are hopeful our product will be a useful tool for mobile app developers.

Appendix

GitHub

This project is open source under the Apache 2.0 license. The source code is available on GitHub at https://github.com/sumnerevans/wireless-debugging.

Installation

In order to use this product, a wireless debugging server will need to be set up, and the user will need to include the wireless debugging library into their mobile application. There are a variety of options for setting up the server, including using Docker, Google Cloud Compute, Amazon AWS, or a user's own machine.

Details on setting up a wireless debugging server can be found on the GitHub wiki at https://github.com/sumnerevans/wireless-debugging/wiki/Deployment.

Details on including the mobile library into an app may also be found on the GitHub wiki at https://github.com/sumnerevans/wireless-debugging/wiki/Usage.