Wireless Debug Design

2017-05-26

Jonathan Sumner Evans
Amanda Giles
Reece Hughes
Daichi Jameson
Tiffany Kalin

Table of Contents

1 Problem Statement

Debugging mobile applications can be a frustrating task at times, as mobile developers need to be connected by wire to their workstations in order to receive a real-time feed of debug information. If they 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 logs to find error messages, or check statuses while debugging. Additionally, the team will provide ways for users to to filter and search through logs in an easy-to-use interface.

2 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 server.
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 a login UI and determine which web UIs to stream logs to.
6. **Web UI:** a website that will display parsed logs to the user in a readable format. The user will also be able to filter and search through logs from the Web UI.

The components were split up in this manner because it compartmentalizes the solution into (mostly) independent components. The connections between the components are specified in the API Specification and as long as the correct API methods are implemented, the components should easily work together.

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 will use 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 constructor for the Mobile Library.

Once they have completed these steps, when the user 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.

In the future, additional device metrics such as memory, CPU, and GPU usage will be streamed by the Mobile Library and presented to the user on the Web UI.

Figure 1 gives a high level overview of the connections between all of these components. The various message types are described in sections 4 and 5 of the API Specification.

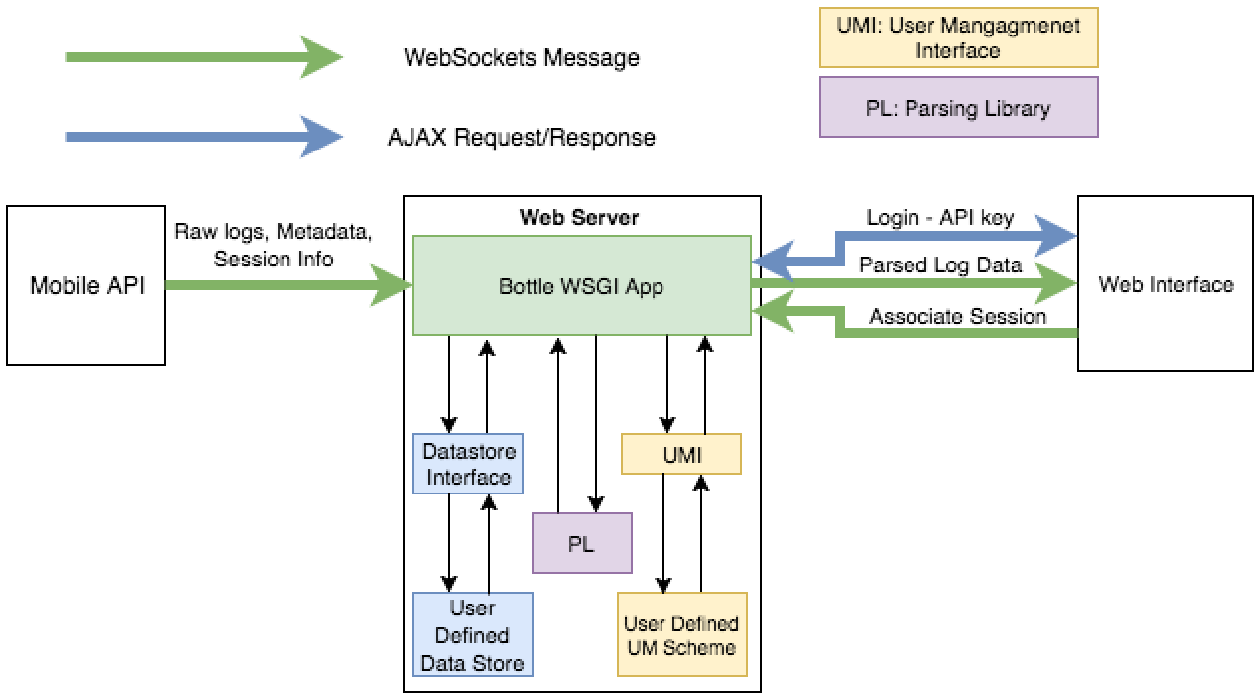


Figure 1: The connections between the main components of Wireless Debug

3 Technical Design

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 will send messages indicating the start and end of logging sessions, and send messages containing the logs from the application. See Section 4.2 in the API Specification for details on these messages. The Web App Backend will send nothing to the Mobile Library. See Section 3.1 for details.
- **Web App Backend:** connects all other components. Communicates with Mobile Library and Web UI using WebSockets and AJAX. See Section 3.2 for details.
- **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. See Section 3.3 for details.
- **Datastore Interface:** specifies a set of functions 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 whatever method they want. See Section 9 of the API Specification and Section 3.4 for details.
- **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. See Section 3.5 for details.

- **Web UI:** 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 searching of logs will be done from the Web UI. The Web UI also visualizes device metrics for the user. See Section 3.6 for details.

3.1 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 will establish a WebSocket connection to the Web App Backend and then stream the logs to the backend. After the application stops, the Mobile Library will send any remaining queued logs then exit.

To determine which user to associate the logs Mobile Library logs with, the constructor will accept an API Key. The API Key will be sent to the Web App Backend and then forwarded to the User Management Interface to determine which Web UIs to stream the logs to. See Section 3.5 for details on the User Management Interface and how API keys are generated.

3.2 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 a few of us have prior experience with it and because it is very lightweight. Deployment of the application is extremely flexible, as it can be run on a user's machine, a dedicated server, or on a cloud service, such as App Engine, AWS, or Heroku.

3.2.1 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 will be sent to the User Management Interface which will return a list of WebSocket connections to send logs to.
5. The log entry HTML will be sent to the WebSockets connections returned from (4).

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

1. The User Management Interface will be called and will return some login UI HTML.
2. The HTML from (1) will be displayed to the user.
3. The user will do whatever action is required by the User Management Interface, and then the authenticated user case (below) will occur.

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

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

3.2.2 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.

3.3 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 so that the library can be used in any application. Section 7 of the API Specification has more details about the inputs the parsing library expects and the outputs that will be generated.

For Wireless Debug, parsed HTML will be 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 will be 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 very nicely with certain forms of data storage, such as MongoDB.

3.4 Datastore Interface

The Datastore Interface specifies a set of functions required in order to implement historical log storage. These functions are elaborated on in Section 9 of the API Specification. Multiple example implementations will be provided with the Wireless Debug application including one which uses a MongoDB backend and one which does not store any log history.

3.5 User Management Interface

One of the major issues in creating this application is pairing mobile devices to Web UIs to stream logs. Without any sort of pairing, the Web App Backend would be unable to determine which logs need to be sent to which interfaces, and would only be able to broadcast logs to all connected Web UIs. In order to address this, the User Management Interface was created as a flexible solution to define how the user is associated with their devices.

By default, the User Management Interface will not perform any authentication, and the logs from the mobile device will be broadcasted to all Web UIs.

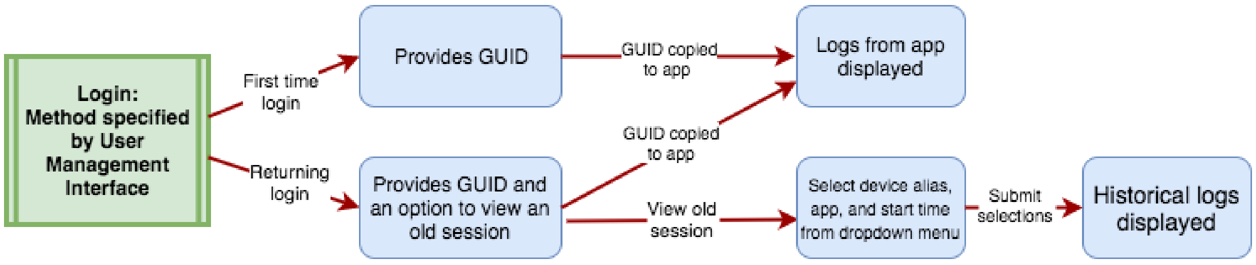
If the user wants user management, they will have to implement a Python class as specified in Section 10 of the API Specification. This class includes functions that return the Login UI, generate or look up the API Key for the user, and determine what WebSocket connections to send logs to. Any given implementation of a User Management Interface can internally implement a way to store a map of some identifier to an API Key, but the details of this can be determined by the implementation.

Multiple examples will be provided with the Wireless Debug application, including a simple form of authentication where the user logs in using their email and then the email to API Key relationships are stored in a text document, as well as a more complex form of authentication using Google Authentication and MongoDB to store user to API Key relationships.

3.6 Web UI

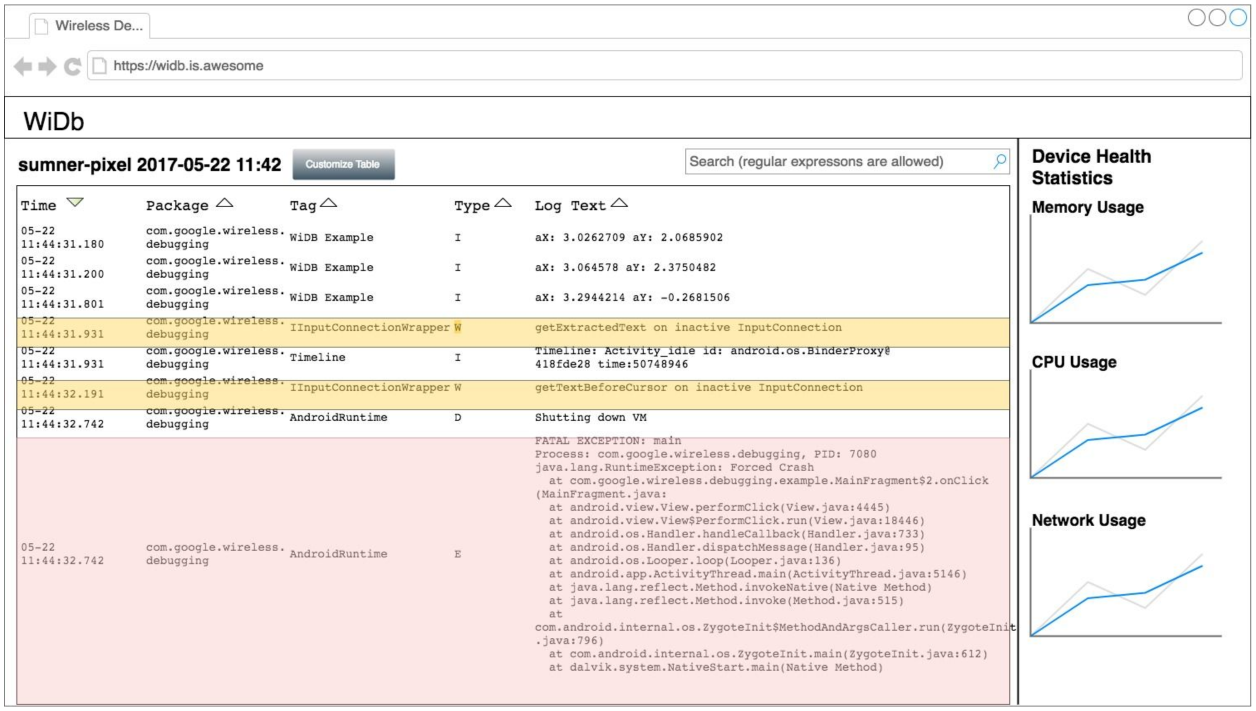
The main purpose of the Web UI is to display the parsed logs to the user and allow them to filter and sort the logs. The Web UI receives the parsed logs from the server through a WebSocket connection (See Section 3.2.2). These logs are displayed in an HTML table as the mobile app is running. The table consists of columns for the time, type, tag, log data, process ID, and thread ID. 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 will be displayed in the same table row and the multiple lines will be separated by newlines in the “Log Text” column. If two errors occur sequentially they will be displayed in two separate rows of the table.

The Web UI will have features for searching through the table, filtering the output, and displaying the user ID for the session. Device diagnostics will be displayed including memory usage, CPU usage, and network usage. There will be an option for customizing which fields are displayed in the table and which diagnostics are shown.



Login: Method specified by User Management Interface
First time login
Returning login
Provides GUID
Provides GUID and an option to view an old session
GUID copied to app
GUID copied to app
View old session
Logs from app displayed
Select device alias, app, and start time from dropdown menu
Submit selections
Historical logs displayed

Figure 2: Flowchart of user login experience



sumner-pixel 2017-05-22 11:42

Time	Package	Tag	Type	Log Text
05-22 11:44:31.180	com.google.wireless.debugging	WiDB Example	I	aX: 3.0262709 aY: 2.0685902
05-22 11:44:31.200	com.google.wireless.debugging	WiDB Example	I	aX: 3.064578 aY: 2.3750482
05-22 11:44:31.801	com.google.wireless.debugging	WiDB Example	I	aX: 3.2944214 aY: -0.2681506
05-22 11:44:31.931	com.google.wireless.debugging	IInputConnectionWrapper	W	getExtractedText on inactive InputConnection
05-22 11:44:31.931	com.google.wireless.debugging	Timeline	I	Timeline: Activity_idle id: android.os.BinderProxy@418fde28 time:50748946
05-22 11:44:32.191	com.google.wireless.debugging	IInputConnectionWrapper	W	getTextBeforeCursor on inactive InputConnection
05-22 11:44:32.742	com.google.wireless.debugging	AndroidRuntime	D	Shutting down VM
05-22 11:44:32.742	com.google.wireless.debugging	AndroidRuntime	E	FATAL EXCEPTION: main Process: com.google.wireless.debugging, PID: 7080 java.lang.RuntimeException: Forced Crash at com.google.wireless.debugging.example.MainFragment$2.onClick (MainFragment.java: at android.view.View.performClick(View.java:4445) at android.view.View$PerformClick.run(View.java:18446) at android.os.Handler.handleCallback(Handler.java:733) at android.os.Handler.dispatchMessage(Handler.java:95) at android.os.Looper.loop(Looper.java:136) at android.app.ActivityThread.main(ActivityThread.java:5146) at java.lang.reflect.Method.invokeNative(Native Method) at java.lang.reflect.Method.invoke(Method.java:515) at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:796) at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:612) at dalvik.system.NativeStart.main(Native Method)

Figure 3: Mockup of the Web UI

Note: The ‘Customize Table’ button provides options for including process ID and thread ID in the table, as well as which device diagnostics to display.

4 Technical Design Issues

Integrating App Engine into the project is one of the client’s specifications and has caused some issues so far. A local development App Engine instance has been set up, and we have been able to deploy a simple web application. However, we can not deploy our work for the first sprint on app engine. We will have to incrementally test that our product can deploy to app engine to determine what aspects of our code do not work with App Engine. Another issue with App Engine is its support for websockets, mostly with the free version of the service. It appears to support socket connections but it is limited on how many and their message frequency.

Logs will be stored using a database or datastore within the server. The challenge with the datastore is that it must be platform independent and abstracted from the problem we are trying to solve. MongoDB with PyMongo was chosen initially, but this does not abstract the storage of the logs. Understanding how to create a platform independent, abstracted database interface that can be integrated into the Web App Backend will be difficult.

Another issue is the basic client-server-web application has to be finished and tested before extra elements, such as searching, sorting, or session management, can be added to the app. This is an example of some of the dependency issues where some team members are unable to continue because other team members must finish modifying a library or component first.

Parsing logs is proving to be another challenge in the project. This issue is caused by a variability between logs. While logs generally have a similar format, logs have some amount of variation between each other. These variations range from a single character difference, such as a space or a hyphen, to timestamps being included or removed from lines of a stack trace. The parsing code is fairly delicate, and so these small variations can easily break it. Improving the regular expressions for extracting elements from the raw logs is a high priority; however, it is most likely impossible to account for every possible variation.

iOS integration is anticipated to be another challenge. No group members have experience with iOS development so a large amount of research is required to get started.