Instrument Applications with the IoT Java SDK

The IoT Java SDK can be included in IoT Java applications running on an edge device like gateways, points of sale, car infotainment centers. This getting started will show you how to install the Java SDK and instrument your IoT application.

Follow the steps below to get your EUM App Key and instrument your IoT C/C++ apps.

1. Check the Requirements
2. Get the IoT Java SDK
3. Add the SDK Dependencies to the Gradle Configuration
4. Add the Instrumentation Code
5. Add and Send Events
6. Verify the Instrumentation in the Controller UI
7. Correlate Business Transactions with Network Requests (Optional)
8. Enable Logging for the SDK (Optional)
9. Customize the IoT Java Instrumentation (Optional)
10. Troubleshooting the IoT Java SDK

This Java SDK differs from the AppDynamics Java Agent in that it is a very lightweight library specially designed for lower-end devices. A lot of flexibility is also built into the way event information can be extended and the instrumentation code configured as well as controlled.

Check the Requirements

Before you get started, make sure you meet the following requirements:

- A device running one of the following version of Java Runtime:
  - Java SE 7
  - Java SE Embedded 7
  - Java SE 8
  - Java SE Embedded 8
- HTTPS interface to send beacons to the EUM Server
- EUM App Key

Get the IoT Java SDK

You can get the Java SDK by cloning or downloading the IoT Java SDK from GitHub. Follow the instructions given in Build the SDK to build the IoT Java SDK.

If you're using the IntelliJ IDE, add the file lib/appd-iot-sdk.jar to your project by following the instructions given in Working with module dependencies. Confirm that the JAR file shows up under External Projects in your IntelliJ project.
Upgrade the IoT Java SDK

From the root directory of your clone of the IoT Java SDK from GitHub:

- Update the repository: $ git pull origin master
- Follow the instructions given in Build the SDK to rebuild the IoT Java SDK.

Add the SDK Dependencies to the Gradle Configuration

Add the following to your `build.gradle` file:

```groovy
dependencies {
    runtime group: 'org.slf4j', name: 'slf4j-api', version: '1.7.25'
    runtime group: 'com.google.guava', name:'guava', version:'18.0'
    runtime group: 'com.google.code.gson', name: 'gson', version: '2.8.0'
}
```

Add the Instrumentation Code

Import the IoT SDK

In your application file, add the `import` statement that includes the Java IoT SDK:

```java
import com.appdynamics.iot.Instrumentation;
```

Configure the IoT Java Agent

Configure the instrumentation by providing the EUM App Key and the URL to the EUM Collector. If the EUM Collector URL is not specified, the default SaaS Collector URL is used.
Set Device Info

You are required to set the name and ID for devices. The name should consist of a short string that identifies the type and model of the device, such as "EV Model 3" or "Thermostat Model Star7". The device ID must be a unique identifier for the device, such as a UUID, the VIN number of a car, or the MAC address of the device.

The example code below sets the device ID to a random UUID and the name to "Smart Shelf".

```java
import java.util.UUID;
import com.appdynamics.iot.DeviceInfo;
...
DeviceInfo.Builder deviceInfoBuilder = DeviceInfo.builder("Smart Shelf P1", UUID.randomUUID().toString());
DeviceInfo deviceInfo = deviceInfoBuilder.withDeviceName("Smart Shelf").build();
```

Set Version Info

You can set the versions for the firmware, hardware, OS, and software as shown below.

```java
import com.appdynamics.iot.VersionInfo;
...
VersionInfo.Builder versionInfoBuilder = VersionInfo.builder();
VersionInfo versionInfo = versionInfoBuilder
    .withFirmwareVersion("2.3.4")
    .withHardwareVersion("1.6.7")
    .withOsVersion("8.9.9")
    .withSoftwareVersion("3.1.1").build();
```
Initialize the Agent

To initialize the agent, pass the `AgentConfiguration` object, the `DeviceInfo` object, and the `VersionInfo` object to the `start` method:

```java
Instrumentation.start(agentConfig, deviceInfo, versionInfo);
```

Build and Run the Application

Use your favorite Java IDE or CLI environment to build and run the application. Note that the AppDynamics IoT Java SDK needs to be in the build and runtime classpath.

For instructions on adding libraries to the classpath:
- IntelliJ IDE: Creating a library
- Eclipse: Classpath Variables
- Linux/Mac/Windows: PATH and CLASSPATH

For instructions to build and run the app:
- IntelliJ: Building and Running the Application
- Eclipse: Running your programs
- Gradle: Building Java Projects with Gradle

Add and Send Events

The following sections will show you how to create and send the supported events: Custom, Network Request, and Error.

Create a Basic Custom Event

A Custom event can be used to report any performance, device, or business logic data. It is the most general, configurable and flexible data type available.

The Custom event builder takes two required parameters.

- **Event Type**: A short human-readable description of the event, such as "FL Pressure Drop".
- **Description**: A string describing the event, such as "Front Left Tire Pressure Drop".

To make reporting this event meaningful, it is recommended that you provide a timestamp and at least one kind of datatype.

1. Create a basic custom event.
import com.appdynamics.iot.events.CustomEvent;
...
CustomEvent.Builder builder = CustomEvent.builder("FL Pressure Drop", "Front Left Tire Pressure Drop");
long eventStartTime = System.currentTimeMillis();
long duration = 6000;
builder.withTimestamp(eventStartTime).withDuration(duration);
builder.addLongProperty("PSI Drop", 37);
CustomEvent customEvent = builder.build();

Additional information can be added to the CustomEvent. For details, see the CustomEvent class in the latest Java IoT SDK documentation.

2. Add the custom event to the instrumentation (this adds it to the in-memory buffer).

    Instrumentation.addEvent(customEvent);

3. Send all the events to the EUM Server. This is a blocking call, so the application can send it on a separate thread as shown above.

    Instrumentation.sendAllEvents();

Send a Network Event

1. Report a Network Request Event using the HttpRequestTracker class. This call automatically adds an event to the in-memory buffer, so you need to explicitly import the class.

    import com.appdynamics.iot.HttpRequestTracker;
    ...
    String url = "http://ip.jsontest.com/?callback=showMyIP";
    // Add a Network Event
    try {
        URL thisUrl = new URL(url);
        // [AppDynamics Instrumentation] Get a Tracker
        HttpURLConnection con = (HttpURLConnection)
        thisUrl.openConnection();
        final HttpRequestTracker tracker = Instrumentation.beginHttpRequest(thisUrl);
con.setRequestMethod("POST");
con.setRequestProperty("Accept-Language", "en-US,en;q=0.5");

int responseCode = con.getResponseCode();
con.setDoInput(true);
con.setDoOutput(true);
DataOutputStream wr = new DataOutputStream(con.getOutputStream());
wr.flush();
wr.close();
System.out.println("Response Code :" + responseCode);

// [AppDynamics Instrumentation] Retrieve the headers from the response
Map<String, List<String>> headerFields = null;
System.out.println("Sending 'POST' request to URL :" + url);
BufferedReader in;
String inputLine;
new InputStreamReader(con.getErrorStream()));
if (responseCode >= 200 && responseCode < 300) {
in = new BufferedReader(new InputStreamReader(con.getInputStream()));
} else {
in = new BufferedReader(new
InputStreamReader(con.getInputStream()));
}
StringBuffer response = new StringBuffer();
if (headerFields != null && headerFields.size() > 0){
    while ((inputLine = in.readLine()) != null) {
        response.append(inputLine);
    }
in.close();
// [AppDynamics Instrumentation] Initiate adding NetworkRequestEvent
if (responseCode >= 200 && responseCode < 300) {
    tracker.withResponseCode(responseCode).withError(response.toString()).reportDone();
    .withResponseHeaderFields(headerFields)
    .reportDone();
} else {
    tracker.withResponseCode(responseCode).reportDone();
}
} else {
    tracker.withResponseCode(responseCode)
}
// End: Add for AppDynamics Instrumentation - Initiate adding NetworkRequestEvent
} catch (MalformedURLException e) {
    e.printStackTrace();
2. Send all the events to the EUM Server. This is a blocking call. The application can send it on a separate thread. The recommendation is to batch a number of events together before calling the `sendAllEvents` method.

```java
Instrumentation.sendAllEvents();
```

**Send an Error Event**

1. Report an Error Event using the API.

```java
try {
    //Force creating an exception
    float f = (5 / 0);
} catch (Throwable t) {
    Instrumentation.addErrorEvent(t,
    Instrumentation.Severity.ALERT);
}
```

2. Send all the events to the EUM Server. This is a blocking call. The application can send it on a separate thread.

```java
Instrumentation.sendAllEvents();
```

**Verify the Instrumentation in the Controller UI**

See [Confirm the IoT Application Reported Data to the Controller](#) to verify the instrumentation.

**Correlate Business Transactions with Network Requests (Optional)**
To correlate business transactions (BTs) with network requests, you need to have instrumented a business application and enabled business transactions in the Controller UI. See Correlate Business Transactions for IoT Monitoring to learn more.

The steps below show you how to get the BT response headers and use them to correlate the BT with an IoT Network Request event.

1. Make a network request that includes the AppDynamics HTTP request headers ADRUM and ADRUM_1 to one of your business applications:

```java
import com.appdynamics.iot.HttpRequestTracker;
...
// Create a network request to the business app.
String url = "<url_to_business_application>";
URL thisUrl = new URL(url);
[AppDynamics Instrumentation] Get a Tracker
HttpURLConnection con = (HttpURLConnection)
thisUrl.openConnection();
final HttpRequestTracker tracker =
Instrumentation.beginHttpRequest(thisUrl);
con.setRequestMethod("POST"); // Some HTTP method: GET, POST, PUT...

// Add the AppDynamics HTTP headers ADRUM and ADRUM_1 to the request.
con.setRequestProperty("ADRUM", "isAjax:true");
con.setRequestProperty("ADRUM_1", "isMobile:true");

// Make the request to your business app.
con.setDoInput(true);
```

2. The call will return response headers that contain information for correlating business transaction. If you were to print these BT response headers, you would see something like the following:

```json
{
  ADRUM_1=[globalAccountName:customer1_78203698-278e-428f-8726-bb381219c6cb],
  null=[HTTP/1.1 200 OK],
  ADRUM_0=[clientRequestGUID:2ff45113-6746-4c94-b6d0-4af26055613c],
  ADRUM_3=[btERT:269],
  ADRUM_2=[btId:4423],
  Server=[Jetty(9.4.z-SNAPSHOT)],
  ADRUM_5=[btDuration:327],
  ADRUM_4=[serverSnapshotType:f],
  Content-Length=[514],
}
```
3. Send a beacon containing the BT response headers to the EUM Server:

```java
// Fetch the response headers, which will include the BT headers (ADRUM_0, ADRUM_1, ...).
Map<String, List<String>> headerFields = con.getHeaderFields();

// Add the BT response headers to the request body of the Network Request event.  
// that you're reporting.
tracker.withResponseCode(responseCode).withError(response.toString()).reportDone();
  .withResponseHeaderFields(headerFields)
  .reportDone();

// Report the Network Request event to the EUM Server.
Instrumentation.sendAllEvents();
```

4. In the Controller UI, you should be able to view the correlated business transaction in the Device Details dialog.

---

**Enable Logging for the SDK (Optional)**

The IoT Java SDK uses the Simple Logging Facade for Java (SLF4J) as the logging framework. You can use your favorite logging engine that is compatible with SLF4J.

If no binding is found on the classpath, then SLF4J will default to a no-operation implementation and display console messages like the following:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation //
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

To use the `java.util.logging` engine, add the following line to the `build.gradle` file:
dependencies {
    // ...
    runtime group: 'org.slf4j', name: 'slf4j-jdk14', version: '1.7.25'
    // ...
}

To see all the debug messages from the library, append the following line to the bottom of the file /Library/Java/JavaVirtualMachines/<your-jdk-version>/Contents/Home/jre/lib/logging.properties:

```java
com.appdynamics.iot.level = FINEST
```

**Customize the IoT Java Instrumentation (Optional)**

You can further customize the IoT Java instrumentation using the IoT Java SDK. See the latest IoT Java SDK documentation or the previous versions listed below:

- [https://docs.appdynamics.com/javadocs/iot-java-sdk/4.5/4.5.0/](https://docs.appdynamics.com/javadocs/iot-java-sdk/4.5/4.5.0/)
- [https://docs.appdynamics.com/javadocs/iot-java-sdk/4.5/4.5.1/](https://docs.appdynamics.com/javadocs/iot-java-sdk/4.5/4.5.1/)
- [https://docs.appdynamics.com/javadocs/iot-java-sdk/4.5/4.5.2/](https://docs.appdynamics.com/javadocs/iot-java-sdk/4.5/4.5.2/)
- [https://docs.appdynamics.com/javadocs/iot-java-sdk/4.5/4.5.4/](https://docs.appdynamics.com/javadocs/iot-java-sdk/4.5/4.5.4/)

**Run the Sample Java Application**

The sample Java application sends sample data for Custom, Network Request, and Error events. The data mocks a smart car application, capturing usage information, network performance, and errors.

To run the sample app, follow the Getting Started instruction given in the [iot-java-sdk GitHub repository](https://github.com/appdynamics/iot-java-sdk).

**Troubleshooting the IoT Java SDK**

This section provides instructions for debugging common issues.

**Unable to Link the IoT Java Agent**

If you are getting the following error when trying to link the IoT Java Agent, it's because of a dependency on log4j.
loader constraint violation: when resolving method
"org.slf4j.impl.StaticLoggerBinder.getLoggerFactory()Lorg/slf4j/ILoggerFactory;"
the class loader (instance of com/intellij/ide/plugins/cl/PluginClassLoader) of the current class,
org/slf4j/LoggerFactory, and the class loader (instance of com/intellij/util/lang/UrlClassLoader) for the method's defining class,
org/slf4j/impl/StaticLoggerBinder, have different Class objects for the type org/slf4j/ILoggerFactory used in the signature
java.lang.LinkageError: loader constraint violation: when resolving method
"org.slf4j.impl.StaticLoggerBinder.getLoggerFactory()Lorg/slf4j/ILoggerFactory;"
the class loader (instance of com/intellij/ide/plugins/cl/PluginClassLoader) of the current class,
org/slf4j/LoggerFactory, and the class loader (instance of com/intellij/util/lang/UrlClassLoader) for the method's defining class,
org/slf4j/impl/StaticLoggerBinder, have different Class objects for the type org/slf4j/ILoggerFactory used in the signature
at org.slf4j.LoggerFactory.getILoggerFactory(LoggerFactory.java:273)
at org.slf4j.LoggerFactory.getLogger(LoggerFactory.java:241)
at org.slf4j.LoggerFactory.getLogger(LoggerFactory.java:254)
at com.appdynamics.iot.Instrumentation.<clinit>(Instrumentation.java:39)
...

To correct the problem, you need to remove the dependency that you added to enable logging. Thus, remove the line specifying the group org.slf4j shown below from dependencies:

dependencies {
    ....
    runtime group: 'org.slf4j', name: 'slf4j-jdk14', version: '1.7.25'
    ....
}