UNDERSTANDING EVENT BUS : EVENTS FOR ANDROID

Amrita Chaturvedi[1], Palak Keshwani[2]

[1]B.E. Student, CSE Dept, Kruti Inst. of Tech. & Engg, Raipur(C.G)

[2]Asst. Professor , CSE Dept, Kruti Inst. of Tech. & Engg, Raipur(C.G)

E-mail- amritachaturvedi97@gmail.com, palakeshwani@gmail.com

ABSTRACT

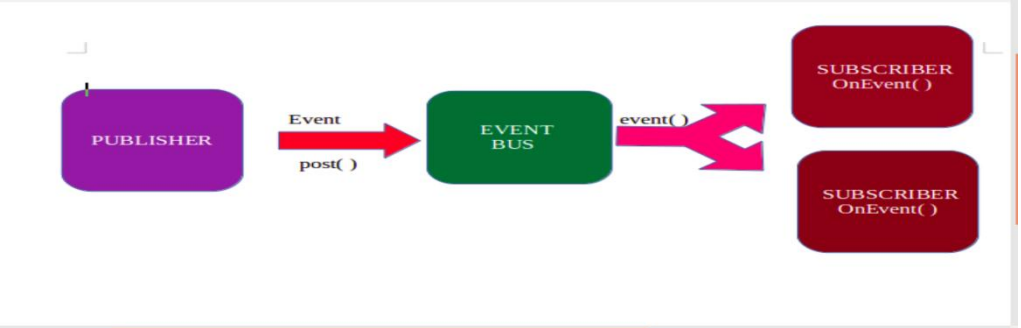
In order to reduce dependencies, event based techniques have been used in software system since a long period of time. EventBus provides an open-source library for Android. EventBus enables low coupling and high cohesion, by providing a centralized communication between classes. By imparting EventBus to our code we can simplify the code, remove dependencies and speed up app development.

Keywords- *Event, Activities, Classes, Objects*

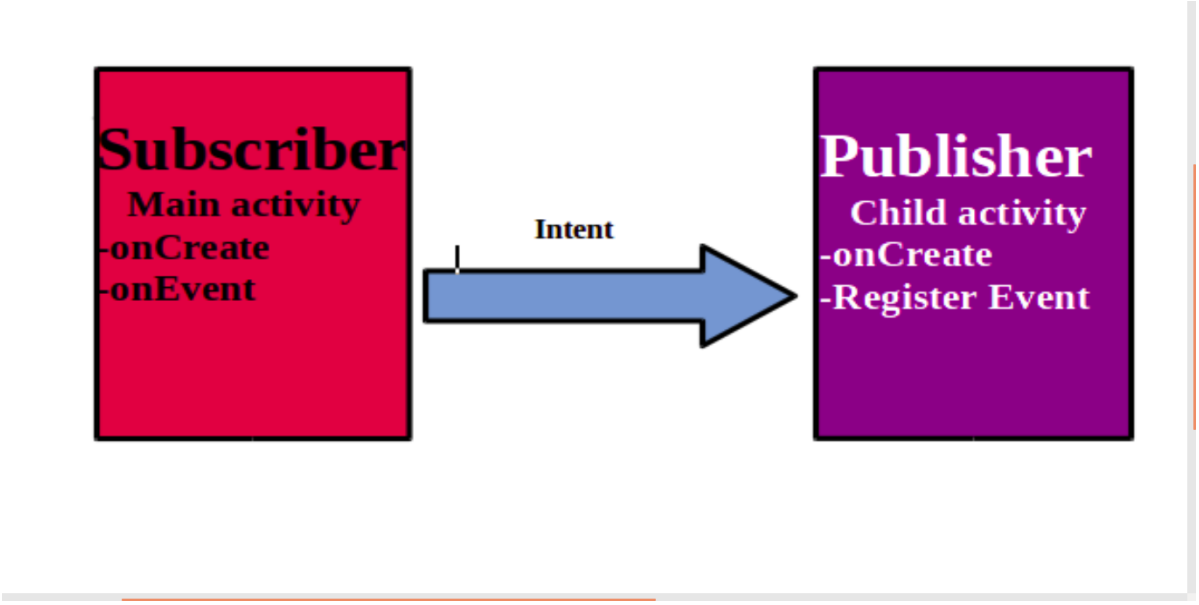
1. Introduction

Event based software is basically used to avoid any dependency between the software component that triggers an operation and the component that performs it. The observer pattern is a software design pattern. In this, the observer (also known as subject) maintains a list of dependents. This list of dependents is called observer. In case of any state changes the subject automatically notifies the observer. This is usually done by calling one of their methods. Observer pattern allows us to accomplish low coupling and high cohesion which is something extraordinary in software architecture. By using EventBus, we are able to maintain a system of objects that communicates through an Event Bus system, rather than connecting all of our class instances together through references. In simple words, instead of passing the reference of objects back and forth, which allows us to make a callback on the parent activity we can use the Event Bus. EventBus performs well with UI artifacts such as Activities and Fragments and background threads. It is optimized for higher performance and it also has advanced features like delivery threads, subscriber priorities. It needs zero configuration, one can instantly start using the default EventBus instance throughout the code. EventBus always runs on the Main UI thread.

2. Description



EventBus- In reference to the diagram above, the publisher posts the Event to the EventBus. The Event Bus then passes the event to the Subscriber. These events are just plain java objects. Consider two activities, one is the MainActivity and another is the ChildActivity. The ChildActivity will register the Event and the EventBus will send the event to the respective references.



EventBus offers low coupling refers to limiting the number of connections between classes and high cohesion refers to the degree to which functionality belongs to. EventBus works by registering as a subscriber to an event. The MainActivity will subscribe to the event posted by the Child Activity which would be the data that will generate in the Child Activity will create an event and pass that to the EventBus. Hence our Child Activity known as the publisher generates the event and if there exists any object that cares about that event it will get notified, the other events won't know about the event happening.

When there exists multiple objects that care about the Child Activity, with EventBus we can make all the related object subscribe to the an event when the data is created in Child Activity.

Implementing Event Bus-

The process of implementing Event Bus can be broken into 5 steps:

- Add compile 'org.greenrobot:eventbus:3.0.0' to the gradle file of the project.
- A separate class needs to be created for handling information that is to be passed (say the name of class is CustomMessage can have just the basic getter and setter methods.
- Register the class to listen to the events (generally done in the MainActivity) This is done by passing the statement EventBus.getDefault().register(this)
- Next, Subscribe to those events

For this simply use the annotation @Subscribe and make an onEvent method which takes the object of CustomMessage as its parameters and performs what is to be don

- At last we need to post the event by creating an instance of the CustomMessage class (say event) and posting it through
  EventBus.getDefault().post(event)

Features of EventBus-

- It simplifies the communication between the components.
- While interacting with the User Interface, EventBus delivers the event in the main thread, irrespective of how the event was posted.
- While the Subscriber is performing any long running task, EventBus can use the background thread to prevent UI blocking.
- In EventBus, the event and subscriber classes follow the object oriented programming pattern.
- It eradicates error-prone dependencies and life cycle issues

3. Conclusion

The basic goal of this paper this paper was to define event-based software system, which allows us to maintain a system of the object references that communicate through an EventBus system. It makes the overall application have higher reliability, robustness, reusability and understandability. EventBus helps us achieve this by making a class instance register with EventBus to start receiving in the app another object say Child Activity then creates an event and sends that to EventBus if it's the same object that the first object cares about it will get notified, never does the second object needs to know about the first object or its interface.

4. Acknowledgement

This is paper is a result of support of many people. I thank Mrs. Palak Keshwani, Assistant Professor, Kruti Institute of Technology and Engineering for comments that greatly improved the manuscript and for being a motivator. I would also like to show my gratitude to Mr. Meghal Agrawal, for sharing his knowledge during the course of making this paper. I am also thankful to Mrs. Geeta Chaturvedi for being a constant support and motivation, although any errors are my own and should not tarnish the reputation of these esteemed persons.

5. References

[1]. Enhancing Event Driven Architectures With Event-Bus Types Martin S. Kew, Timothy Ferris Georg Grossmann, Markus Stumptner.

[2]. B. P. Douglass, Real-Time Design Patterns, Addison-Wesley, 2003.

[3]. E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[4]. B. Meyer, The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design, Festschrift in Honor of Ole-Johan Dahl, eds. Olaf Owe et al., Lecture Notes in Computer Science Vol 2635, Springer-Verlag, 2003.

