Communication Within an Android App With EventBus

by Chike Mgbemena 25 Nov 2016

Length: Short Languages: English

Android | Android SDK | Mobile Development | Mobile App

A typical Android app tends to be composed of many layers, modules or structures such as Fragments, Activities, Presenters, and Services. Effective communication between these components can become difficult if they are tightly coupled together.

In the lower level of your app architecture, such as the database, when an action happens, you might want to send data to a higher level such as the view. To do this, you might want to create a listener interface, async tasks or callbacks. All of these will work, but they have some major drawbacks:

1. direct or tight coupling
2. registering and unregistering multiple dependencies individually
3. repetition of code
4. difficulty in testing
5. increased risk of bugs

Using publish/subscribe or message bus architecture prevents all the potential problems highlighted above. It is a very good way to implement effective

communications between components in an application without any of them needing to be aware of the others immediately. Using publish/subscribe in Android, any app component can publish events which it will hand over to the bus, and the relevant consumers can consume or subscribe to them.

To use greenrobot EventBus, you need to first add it to in the app module **build.gradle** file, include `compile 'org.greenrobot:eventbus:3.0.0'`, and then sync your project afterwards.

An Event Subscriber

A subscriber simply subscribes to an event by registering in the event bus and can also unregister that event. To be a subscriber, you have to do three main things:

1. Register the subscriber in the event bus with `register()`. This informs the event bus that you want to begin receiving events. In an activity, this is in the `onStart()` method, while in a fragment put this in the `onAttact(Activity activity)` method.

```
@Override
public void onStart() {
    super.onStart();
    EventBus.getDefault().register(this);
}
```

2. Unregister the subscriber, which means tell the event bus to stop sending me events. In an activity, this is in the `onStop()` method, while in a fragment put this in the `onDetach()` method.

```
@Override
public void onStop() {
    super.onStop();
    EventBus.getDefault().unregister(this);
}
```

3. Implement the `onEvent()` to indicate the type of event you want to receive and action to take when you receive the event. Notice the `@Subscribe` annotation at the top of this method. In this case, we want to subscribe to a normal event and not a sticky one—I'll explain the difference later.

```
@Subscribe
```

```
2  public void onEvent(MessageEvent event) {
3      Toast.makeText(this, "Hey, my message"+ event.getMessage(), Toast.
4  }
```

Defining Event Messages

The events in greenrobot EventBus are just objects that you define. You can have different event classes if you want. They do not inherit any base class or interface—they're just POJO (Plain Old Java Objects).

```
01  public class MessageEvent {
02
03      public String mMessage;
04
05      public MessageEvent(String message) {
06          mMessage = message;
07      }
08
09      public String getMessage() {
10          return mMessage;
11      }
12  }
```

Post Event and Post Sticky Event

The main difference between post event and post sticky event is the caching mechanism employed inside the event bus. When someone posts a sticky event, this event is stored in a cache. When a new activity or fragment subscribes to the event bus, it gets the latest sticky event from the cache instead of waiting for it to be fired again to the event bus—so this event stays in the cache even after a subscriber has gotten it.

Sticky events are posted with the `postSticky(MessageEvent)` method, and non-sticky events with the `post(MessageEvent)` method.

```
1  EventBus.getDefault().postSticky(new MessageEvent("Hey event subscriber!
2  EventBus.getDefault().post(new MessageEvent("Hey event subscriber!"));
```

For a regular, non-sticky event, if there no subscriber is found, the event will be thrown away. A sticky event will be cached, though, in case a subscriber comes along later.

So when do you decide to use post sticky event? You can do this if you are tracking down the user's location, or for simple caching of data, tracking battery levels, etc.

```
EventBus.getDefault().postSticky(new LocationReceivedEvent(6.4531, 3.3958
```

Subscribe to Post Sticky Event

```
// UI updates must run on MainThread
@Subscribe(sticky = true, threadMode = ThreadMode.MAIN)
public void onEvent(MessageEvent event) {
    textField.setText(event.getMessage());
}
```

To subscribe to a sticky event, you include `sticky = true` inside the `@Subscribe` annotation. This indicates that we want to receive a sticky event of type `MessageEvent` from the cache.

Removing Sticky Events

```
LocationReceivedEvent locationReceivedStickyEvent = EventBus.getDefaul
if(stickyEvent !=null) {
    EventBus.getDefault().removeStickyEvent(locationReceivedStickyEven
}
```

`removeStickyEvent(Event)`removes a sticky event from the cache, and `removeAllStickyEvents()`will remove all sticky events.

EventBus Thread Modes

There are four thread modes available for subscribers to choose from: posting, main, background, and async.

Posting

```
1  @Subscribe(threadMode = ThreadMode.POSTING)
```

This is the default. Subscribers will be called in the same thread as the thread where the event is posted. Including `ThreadMode.POSTING` in your `@Subscribe` annotation is optional.

Main

```
1  @Subscribe(threadMode = ThreadMode.MAIN)
```

In this thread mode, subscribers will receive events in the main UI thread, no matter where the event was posted. This is the thread mode to use if you

want to update UI elements as a result of the event.

Background

```
1  @Subscribe(threadMode = ThreadMode.BACKGROUND)
```

In this thread mode, subscribers will receive events in the same thread where they are posted, just like for `ThreadMode.POSTING`. The difference is that if the event is posted in the main thread, then subscribers will instead get them on a background thread. This makes sure that event handling doesn't block the app's UI. Still, don't run an operation that will take a long time on this thread.

Async

```
1  @Subscribe(threadMode = ThreadMode.ASYNC)
```

In this thread mode, subscribers will always receive events independently from the current thread and main thread. This enables the subscribers to run on a separate thread. This is useful for long-running operations such as network operations.

Subscriber Priorities

If you want to change the order in which subscribers get events, then you need to specify their priority levels during registration. Subscribers with a higher priority get the event before subscribers with a lower priority. This only affects subscribers in the same thread mode. Note that the default priority is 0.

```
@Subscribe(priority = 1);
public void onEvent(MessageEvent event) {
    textField.setText(event.getMessage());
}
```

Cancelling Events

If you want to stop an event from being delivered to other subscribers, call the `cancelEventDelivery(Object event)` method inside the subscriber's event handling method.

```
1  @Subscribe
2  public void onEvent(MessageEvent event){
```

```
3       EventBus.getDefault().cancelEventDelivery(event);
4   }
```

Conclusion

In this tutorial, you learned about:

- greenrobot EventBus and how it can improve your Android app
- the difference between regular and sticky events
- the different thread modes available and when to use each one
- subscriber priorities
- cancelling an event to stop receiving events

To learn more about greenrobot EventBus, I suggest you visit the official documentation.

Another library you can use to implement an event bus is RxAndroid. Check out our article on RxAndroid here on Envato Tuts+, or try some of our other Android courses or tutorials.

Getting Started With ReactiveX on Android

The codebase of complex apps with many network connections and user interactions are often littered with callbacks. Such code is not only lengthy and hard to

Ashraff Hathibelagal
10 Aug 2015

ANDRO

An Introduction to Loopj

In this tutorial, you'll learn to use Loopj, an eas
to-use library for HTTP requests in Android. To
help you learn, we're going to use Loopj to
create

Pedro Gonzalez Ferrandez
27 Jul 2016

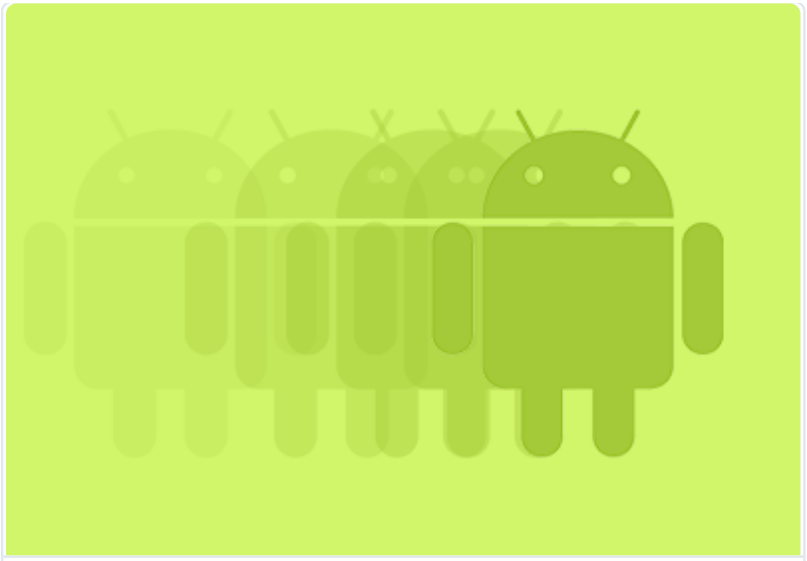
ANDROID STUD

Android From Scratch: Understanding Android Broadcasts

In this tutorial, you'll learn how to create, send and receive both local and system-wide broadcasts. You'll also learn how to use a popular third-party

Ashraff Hathibelagal
15 Aug 2016

ANDROID S

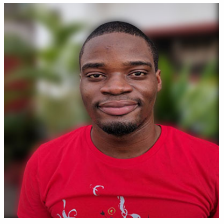


Animate Your Android App

Animations have become an important part of the Android user experience. Subtle, well-crafted animation is used throughout the Android OS and can make your

Ashraff Hathibelagal
10 Feb 2016

MOBILE DEVELOPME



Chike Mgbemena

Software Developer, Nigeria

Chike is a Computer Science graduate and Software Developer based in Lagos, Nigeria. Apart from Android, he also plays around with other

programming technologies such as PHP and JavaScript. He loves teaching, learning, listening to music and swimming.

chk01010

Weekly email summary

Subscribe below and we'll send you a weekly email summary of all new Code tutorials. Never miss out on learning about the next big thing.

Email Address

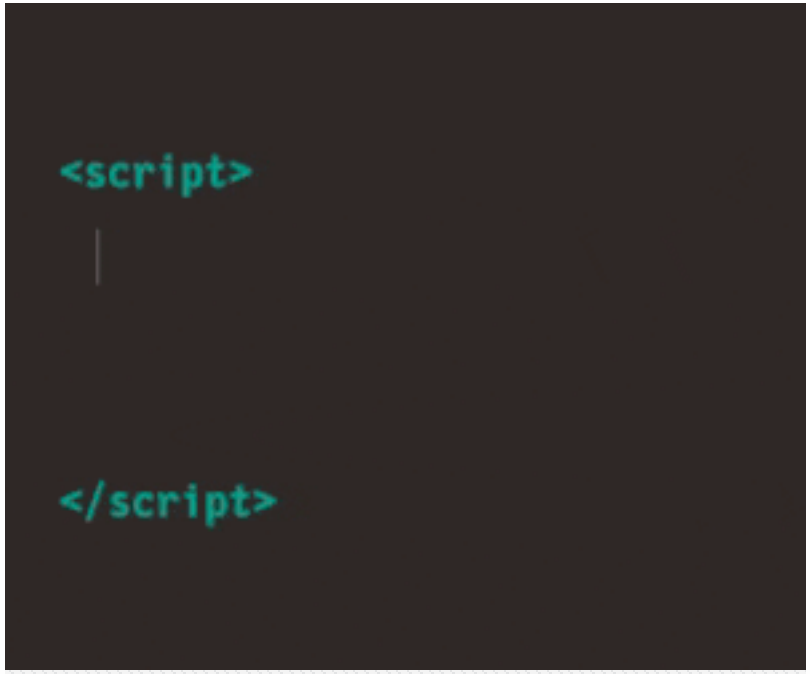
Update me weekly

Translations

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by native



1 Comment Tuts+ Hub

Recommend Share

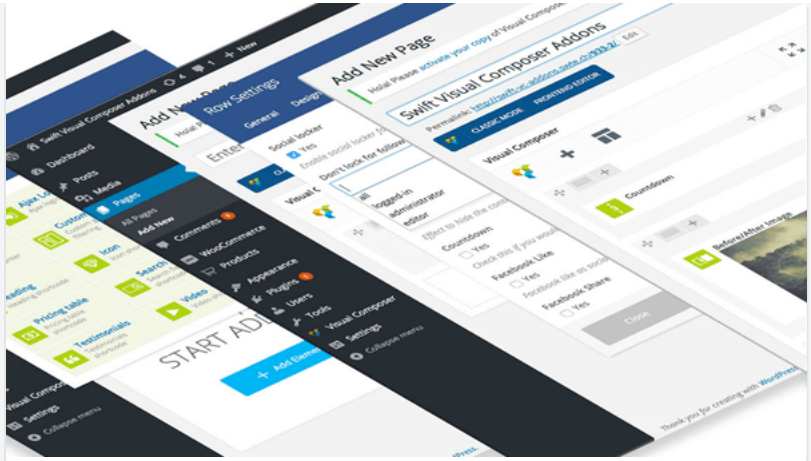
Join the discussion…

Wan • a month ago

This is really cool...am a big fan of Event bus..using it in nearly all my projects...but lately about it coz it crashes my app on Android devices <=API 19..Had no choice but to go bac codes using the stock broadcast receivers...I just hope the green robot guys look into this work around?

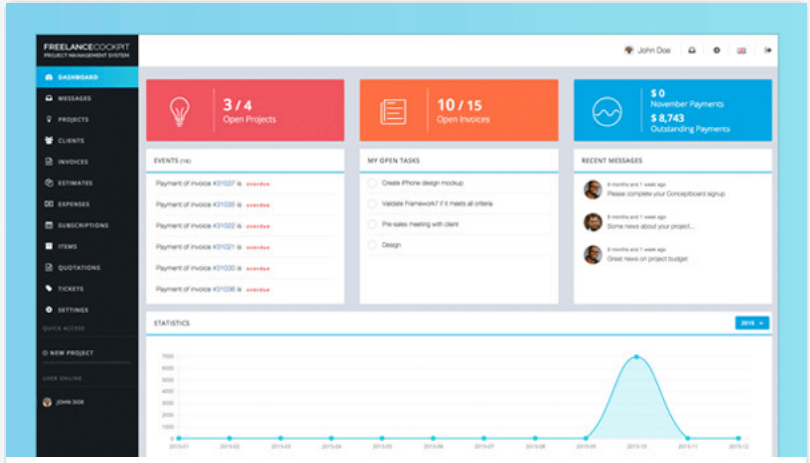
• Reply • Share ›

Subscribe Add Disqus to your site Privacy



WordPress Plugins

From $4



PHP Scripts

From $1

envatotuts+

Teaching skills to millions worldwide.

23,214 Tutorials **971** Video Courses **4,476** Translations

Meet Envato

About Envato
Explore our Ecosystem
Careers

Join our Community

Teach at Envato Tuts+
Translate for Envato Tuts+
Forums
Community Meetups

Help and Support

FAQ
Help Center
Terms of Use
About Envato Tuts+
Advertise

Email Newsletters

Get Envato Tuts+ updates, news, surveys & offers.

Email Address

Subscribe

Privacy Policy