# LCET\_Events: Communication Between Lightning Components

One of the very important and useful principles of the Lightning Components Framework is that, as the name implies, you work with components. This a very important aspect of OOP that Salesforce has brought to building front-end applications (division of labor - one component should do one and only one thing, and do it well), and has the same benefits that OOP provides to the back end: Reusability, maintenance, etc.

But how can you have these independent components talk to each other? Good question, I’m glad you asked. Hopefully this blog helps you clear that question. Let’s start talking about events.



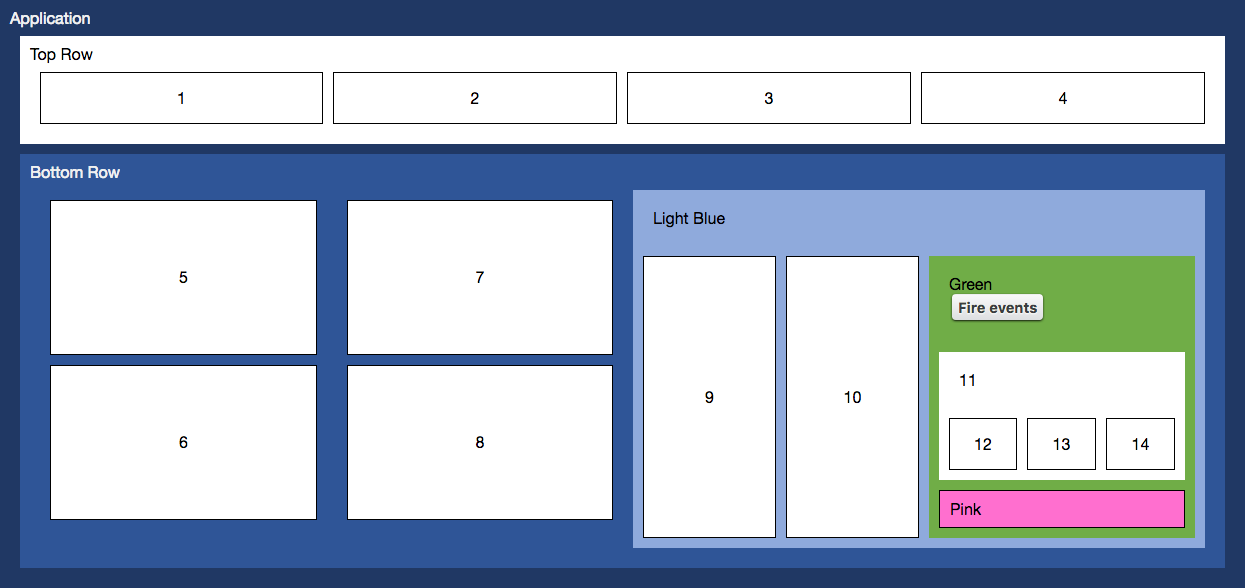
# Application Vs. Component Events

There are two types of events In the Lightning Components Framework: Application events and Component events.

They are somewhat similar in concept:

* You create an event bundle
* A component will fire an event
* Component(s) may handle it

The main difference is who can be notified of these events. I have built an application and posted it in Github ([You can access the sample files here](https://github.com/eltoroit/ETLC_Events)) that I’ll be using to demonstrate the different mechanisms used in Lightning Components and described in this article. The application looks like this:



As you can see we have few components in this application, each box represents a component. Let’s suppose the “Green Component” fires an event; the question then is whIch component could be notified? It depends on what type of component is fired!

## Component Events

If the “Green Component” fires a **component** event, then **only** the blue components will be notified (“Light Blue Component”, “Botom Row Component” and “Application”) – none of the other components will be notified. The “Top Row Component” component, the “Pink Component” and none of white boxes component (including the ones inside the “Green Component”) will NOT be notified.

The components that will be notified in order are: “Green Component”, “Light Blue Component”, “Botom Row Component” and “Application”. A component notifies it’s container, which will notifies it’s container, which notifIes it’s container, all the way until it reaches the outer most container which by the way happens to be an application bundle (don’t confuse an application bundle with an application event).

A blue component may choose to handle the event (they are not required to handle it), also the component could stop the event propagation so that their containers will not be notified. The components could also swap the event for something more meaningful to its containers, this is actually a best practice for events: separate low-level events from business logic events where a low-level event (onclick) is handled by a component and replaced with a business event (create account).

Note: A component can also capture a component event that it fires, so the “Green Component” could handle an event fired by itself. Although I have not find any good use case for this, it’s possible. Maybe it could probably be better to have the component call a function in its helper rather than going through events.

I have found this game usually helps my students understand the component events:

* I write in a piece of paper the word “jump” and pass to the student in the front row.
* The first student reads it and jumps. Then he passes the paper to the second student
* The second student does not read the paper and just passes it along to the third student.
* The third student reads the message and jumps, but adds to the paper “say Hi” and passes the paper to the fourth student
* The fourth student jumps and says hi, but destroys the piece of paper.
* Only 4 students received the paper and the rest of the class did not.

## Application Events

If the “Green Component” fires an **application** event, then **every** component (including the white, blue and pink components) will be notified.

The order in which your components are notified is “random” and therefore you can’t stop propagation of an application event. Each component may choose to handle the event but they are not required to do so.

The analogy I use for this type of events is to tell at loud to my students at once: “get up up, jump, and say hi!”

Now that we have defined the difference between component and application events, which one do you think is better? Like always, the answer is “it depends” because some times you will need to fire/handle application events, but you should always ask yourself if you really, really, really need an application event or if a component event is enough for your needs. Try to prefer component events.

# How To Implement Them?

Now that we understand the difference between Application and Component events, how do we build them? The general steps that you must follow are independent of the type of event, and are these:

1. Define the event
2. Register the event (Markup | Source component)
3. Initialize and fire the event (JavaScript | Source component)
4. Define handler for the event (Markup | Destination component)
5. Execute logic when event notification arrives (JavaScript | Destination component)

## Define The Event

In the developer console, Click on File | New | Lightning Event, give it a name and click submit. As I’ll show you in few minutes, you will use this name when you register and handle the event.

By default the developer console creates an APPLICATION event, but you should switch this to a COMPONENT event, unless you really, really, really want an application event… most likely you want a component event ;-)

You can also add attribute(s) for the data you need to pass between components.

<aura:event type="component">

<aura:attribute name="message" type="String"/>

</aura:event>

## Register The Event (Markup | Source Component)

The source component will notify other components that an action has occurred, by firing the event, but it needs to register the event in the markup before indicating that the source component can fire this event.

### Component Event

<aura:registerEvent name="evCmp" type="c:ETLC\_EV\_ComponentEvent" />

The name attribute is very important since a component could fire/handle several events of the same type (click events could be fired by different components with different meanings), and we need to differentiate the exact event. The type of the event is the same name that we used when the event was created, but preceded with “c:”

### Application Event

<aura:registerEvent name="evApp" type="c:ETLC\_EV\_ApplicationEvent" />

The name attribute is required but not used for application events. The name attribute is only relevant for component events. The type of the event is the same name that we used when the event was created, but preceded with “c:”

## Initialize And Fire The Event (JavaScript | Source Component)

When the source component decides to fire the event it has to perform these actions: Get an instance of the event, initialize the parameters, and fire the event.

In order to get an instance of the event, we need to know if we are working with a component event or with an application event.

### Component Event

var evCmp = component.getEvent("evCmp");

For component events, we use the name of the event.

### Application Event

var evApp = $A.get("e.c:ETLC\_EV\_ApplicationEvent");

For application events, we use the type of the event.

Once we have an instance of the event, we set the parameters and fire the event using this code. For this step, we do not care what type of event we are firing.

ev.setParams(params);

ev.fire();

Once the event has been fired, the other components will get notified. We already discussed at the top of this article which components can get notified…

## Define Handler For The Event (Markup | Destination Component)

The destination component will execute some client-side JavaScrip code when it gets notified that an event was fired by other components by handling the event, but before it can do that it has to declare in the markup that this destination component handles this event.

### Component Event

<aura:handler name=" evCmp" event="c:ETLC\_EV\_ComponentEvent" action="{!c.evAction}" />

The name attribute is very important since a component could fire/handle several events of the same type, and we need to differentiate the exact event. The type of the event is the same name that we used when the event was created, but preceded with “c:”

### Application Event

<aura:handler event="c:ETLC\_EV\_ApplicationEvent" action="{!c.evAction}" />

The name attribute is not included on the event handler definition for application events. The name attribute is only relevant for component events. The type of the event is the same name that we used when the event was created, but preceded with “c:”

## Execute Logic When Event Notification Arrives (Javascript | Destination Component)

When the destination component gets notified the *evAction* in the client-side controller will be automatically executed. The code does not depend of the type of event that was fired, and looks like this:

evAction : function(component, event, helper) {

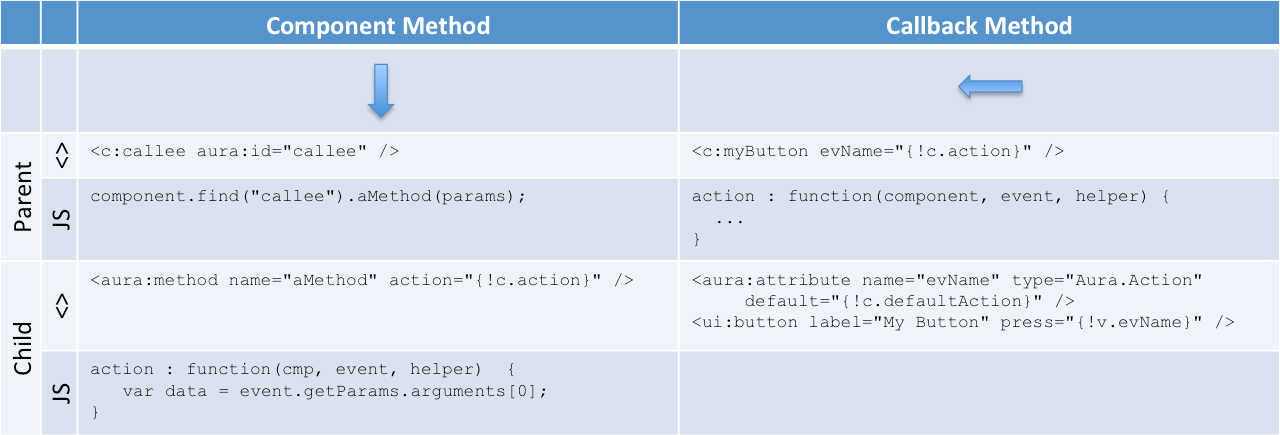
var params = event.getParams();

}

We have seen the two types of events in Lightning Components (Component and Applications) where a component fires an event, and hope that other components will catch that event and process it accordingly. But maybe no components will process it, and this could be fine in some situations.

But can you actually force a client-side controller function on a component when an action fires in another component? Good question, I’m glad you asked… The answer really depends, on where is the function to call:

* Component Methods fire on the child component when the parent component invokes it
* Callback Methods fire on the parent component when the child component invokes it



# Component Methods

Let’s first talk about component methods… There are several use cases for this type of inter-component communication, actually I have built a design pattern around this methods called “[ETLC\_ApexBridge](https://eltoroit.herokuapp.com/Blog.app?page=ETLC_ApexBridge)” which allows you to invoke a method that will communicate to Apex to query/DML data as needed.

## How To Implement A Component Method?

Following with the picture above, suppose that when we click the button on the “Green Component” we want to invoke a specific function in the “Pink Component”. Here we have two components:

* **Child**: Component that has a method that will be invoked (“Pink Component”)
* **Parent**: Component that will make the invocation (“Green Component”)

With this in mind, follow these steps to implement a component method:

1. Child Defines The Method In The Markup.
2. Child Defines Javascript Client-Side Controller Function.
3. Parent Gets Reference To The Child.
4. Parent Invokes The Method.

## 1. Child Defines The Method In The Markup

In the child’s markup, you must define the method like this:

<aura:method name="componentMethodName" action="{!c. componentMethod}" />

Note: Although we are going to send data to the child, we do not need to define the parameters in this definition. We are going to see how that is accomplished later.

## 2. Child Defines Javascript Client-Side Controller Function

The JavaScript client-side function in the child can be defined like this:

componentMethod : function(component, event, helper) {

...

}

As you will see when we call this method, we are going to pass only one parameter to this method. We can send complex data, because that single parameter is going to be a JavaScript object ;-)

## 3. Parent Gets Reference To The Child

The parent component needs to get a reference to the child component, for purposes of this samples we are going to basically embed the child inside the parent and by giving an aura:id attribute we can easily find the child component.

If we assume the parent embeds the child, like this:

<c:ETLC\_EV\_PinkBox aura:Id="pinkBox" />

Then the parent can easily get a reference to the child component using something like this:

var pinkBox = component.find("pinkBox");

Note: These component methods are not restricted to calling components that are nested inside the parent, but for purposes of this blog I have simplified the code. You can call any component’s method as long as you obtain the reference to the child component.

## 4. Parent Invokes The Method

The parent can invoke the child’s method by doing something like this:

pinkBox.componentMethodName();

## Passing Parameters

How does the parent pass parameters to the child? Great question, let me show you.

The parent will pass parameters by writing code like this

var dataFromGreen = {

var1: "data1",

var2: "data2",

var3: "data3"

};

pinkBox.componentMethodName(dataFromGreen);

The callee will receive the data like this:

var dataToPink = event.getParams().arguments[0];

This is a bit complex expression; let me explain it. The Component Methods can receive any number of parameters, but in order to make this a pattern that we can reuse anytime, I am going to always pass one and only one parameter, which will be a JavaScript object. This simulates a “method overload” calling the method with “different” parameters.

# Callback Methods

A very basic and common example is to pass a controller function to the press event of a ui:button. When the button is clicked, the function passed will be invoked. If you were building the ui:button component how will the button receive that attribute? Great question, the press attribute’s type is Aura.Action. Let me show you.

## How To Implement A Callback Method?

Following with the picture above, suppose that when we click the button on the “Green Component” we want to invoke a specific method in the “Light Blue Component”. Here we have two components:

* **Parent**: Component that has a method that will be invoked (“Light Blue Component”)
* **Child**: Component that will make the invocation (“Green Component”)

With this in mind, follow these steps to implement a callback method:

1. Child Defines An Attribute That Will Receive The Function To Invoke.
2. Child Calls The Parent’s Function.
3. Parent Defines The Javascript Controller Function To Be Invoked.
4. Parent Uses The Child Component, Setting The Attribute Value.

## 1. Child Defines An Attribute That Will Receive The Function To Invoke

In the child component, define the attribute that will receive the function to callback

<aura:attribute name="callbackMethodName" type="Aura.Action" default="{!c.defaultAction}" />

This defines the name of the attribute which will be used from the parent component, it also specifies this attribute has the special type “Aura.Action” and you may optionally a default action, which is highly recommended unless you define this attribute as required.

## 2. Child Calls The Parent’s Function

In the child component, perform the callback by getting a reference to it and enqueueing the action.

var action = component.get("v.callbackMethodName");

$A.enqueueAction(action);

Note: You could also call the callback method from the child’s markup like this:

<ui:button label="Callback" press="{!v.callbackMethodName}" />

## 3. Parent Defines The Javascript Controller Function To Be Invoked

In the parent’s JavaScript controller create a function that will be passed to the child component.

callbackMethod : function(component, event, helper) {

...

}

## 4. Parent Uses The Child Component, Setting The Attribute Value

In the parent’s JavaScript controller create a function that will be passed to the child component.

<c:ETLC\_EV\_GreenBox callbackMethodName="{!c.callbackMethod}" />

Hopefully this article has clarified the different ways components can talk to each other.