Communicating between Lightning Components and Visualforce Pages

There are many places where you can use Visualforce pages in Lightning Experience. The Visualforce & Lightning Experience module in Trailhead covers these scenarios in detail.

feedb

In some of these places, a Visualforce page can coexist with Lightning Components on the same page. For example, in App Builder, you can use the Visualforce Standard Component to add a Visualforce page to a page. The Visualforce standard component is just a Lightning component wrapper around a Visualforce page. If your Visualforce page is self-contained and doesn't need to communicate with other Lightning components on the page, there is nothing else you need to do.

In this article, we will focus on scenarios where Visualforce pages and Lightning components do need to communicate, and we will describe a simple approach to implement those Lightning-to-Visualforce communication requirements.

Background

There are two important things you need to know about Visualforce pages running in Lightning Experience:

Different DOMs. A Visualforce page hosted in Lightning Experience is loaded in an **iframe**. In other words, it's loaded in its own window object which is different from the main window object where the Lightning Components are loaded.

Different Origins. Visualforce pages and Lightning Components are served from different domains. For example, if you are using a developer edition:

- Lightning Components are loaded from a domain that looks like this: **yourdomain-dev-ed.lightning.force.com**

- Visualforce pages are loaded from a domain that looks like this: **yourdomain-dev-ed–c.na35.visual.force.com**

The browser's same-origin policy prevents a page from accessing content or code in another page loaded from a different origin (protocol + port + host).

In our case, that means that a Visualforce page can't use the **parent** window reference to access content or execute code in the Lightning Component wrapper. Similarly, the Lightning component can't use the iframe's **contentWindow** reference to access content or execute code in the Visualforce page it wraps.

These restrictions are enforced for very good reasons. But fortunately, there is also an API (otherWindow.postMessage()) that provides a secure approach (when used properly) to exchange messages between different window objects with content loaded from different origins.

window.postMessage() is a standard

In the remainder of this article, we will look at different examples illustrating how **postMessage()** can be used to communicate between Lightning Components and Visualforce pages.

Lightning Component to Visualforce Page

In this first scenario, we have a Lightning Component that wraps a Visualforce page using the **iframe** tag, and we want the Lightning Component to send messages to the wrapped Visualforce page. A message could be a simple string indicating that something happened, a recordId, an object, a collection, etc.

Sending the Message in the Lightning Component

Component:

```
<aura:component implements="flexipage:availableForAllPageTypes"
                access="global">

    <aura:attribute name="message" type="String"/>
    <aura:attribute name="vfHost" type="String"
            default="yourdomain-dev-ed--c.na35.visual.force.com"/>

    <!-- Input field for message "data" -->
    <lightning:input type="text" label="Message:" value="{!v.message}"/>
    <lightning:button label="Send to VF" onclick="{!c.sendToVF}"/>

    <!-- The Visualforce page to send data to -->
    <iframe aura:id="vfFrame" src="{!'https://' + v.vfHost + '/apex/myvfpage'}"/>

</aura:component>
```

Code highlights:

- **vfHost** is the host Visualforce pages are loaded from in your environment. In a real-life application, you should obtain this value dynamically instead of hardcoding it.
- Avoiding Cross Site Scripting (XSS). The **vfHost** attribute provides a way to pass the host name (without protocol and port). The protocol (https://) is hardcoded in the iframe's **src** attribute. This is a simple way to sanitize part of the URL and avoid xss attacks where a script would be injected by passing a url starting with the **javascript://** protocol. Other approaches to sanitize URLs and avoid XSS attacks are described in the Lightning Security Guidelines document.
- **message** is a simple string message we will send to the Visualforce page

Controller:

```
({
    sendToVF : function(component, event, helper) {
        var message = component.get("v.message");
        var vfOrigin = "https://" + component.get("v.vfHost");
        var vfWindow = component.find("vfFrame").getElement().contentWindow;
        vfWindow.postMessage(message, vfOrigin);
    }
})
```

Code Highlights:

feedb

- The first argument of postMessage() is the data you want to pass to the other window. It can be a primitive data type or an object.
- The second argument of postMessage() is the origin (protocol + port + host) of the window you send the message to (vfWindow in this case). The event will not be sent if the page in vfWindow at the time postMessage() is called wasn't loaded from vfOrigin.

Receiving the Message in the Visualforce Page

To receive the messages in your Visualforce page, you simply set up a listener for **message** events:

```
<apex:page>

<script>
    var lexOrigin = "https://yourdomain-dev-ed.lightning.force.com";
    window.addEventListener("message", function(event) {
        if (event.origin !== lexOrigin) {
            // Not the expected origin: reject message!
            return;
        }
        // Handle message
        console.log(event.data);
    }, false);
</script>

</apex:page>
```

Code Highlights:

- **lexOrigin** is the origin (protocol + port + host) Lightning components are loaded from. This is where we expect the messages to come from.
- **event.origin** is the actual origin of the window that sent the message at the time postMessage() was called. **You should always verify that the actual origin and the expected origin match, and reject the message if they don't.**
- **event.data** is the message sent from the other window

With this infrastructure in place, the Lightning component can now listen to application events sent by other components on the page, and forward any relevant events to the Visualforce page.

Visualforce Page to Lightning Component

In this second scenario, we still have the same arrangement: a Visualforce page wrapped in a Lightning component. This time we need communication to happen in the opposite direction: The Visualforce page sends messages to the Lightning component.

Sending the Message in a Visualforce Page

feedb

This time, we invoke postMessage() on **parent**. This is a reference to the parent window, in other words the main window in Lightning Experience that hosts Lightning components.

```
<apex:page>
<input id="message" type="text"/>
<button onclick="sendToLC()">Send to LC</button>

<script>

    var lexOrigin = "https://yourdomain-dev-ed.lightning.force.com";

    function sendToLC() {
        var message = document.getElementById("message").value;
        parent.postMessage(message, lexOrigin);
    }

</script>

</apex:page>
```

Code highlights:

- **lexOrigin** is the origin (protocol + port + host) Lightning Components are loaded from
- The **message** field is used to capture the simple string message we will send to the Lightning Component
- The second argument of postMessage() is the origin of the **parent** window. Again, the event will not be sent if the content of the **parent** window at the time postMessage() is called wasn't loaded from **lexOrigin**.

Receiving the Message in a Lightning Component

To receive the messages in your Lightning Component, you set up a listener for **message** events:

Component:

```
<aura:component implements="flexipage:availableForAllPageTypes"
                access="global">

    <aura:attribute name="vfHost" type="String"
        default="yourdomain-dev-ed--c.na35.visual.force.com"/>

    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>

    <iframe aura:id="vfFrame" src="{!'https://' + v.vfHost + '/apex/myvfpage'}" />

</aura:component>
```

Controller:

```
({
    doInit : function(component) {
        var vfOrigin = "https://" + component.get("v.vfHost");
        window.addEventListener("message", function(event) {
            if (event.origin !== vfOrigin) {
                // Not the expected origin: Reject the message!
                return;
            }
            // Handle the message
            console.log(event.data);
        }, false);
    }

})
```

feedt

Code Highlights:

- **vfOrigin** is the origin (protocol + port + host) Visualforce pages are loaded from. This is where we expect the messages to come from.
- **event.origin** is the actual origin of the window that sent the message at the time postMessage() was called. **You should always verify that the actual origin and the expected origin match and reject the message if they don't.**
- **event.data** is the message sent from the other window

With this infrastructure in place, the Lightning component can now listen for events sent by the Visualforce page and forward any relevant events to other Lightning components on the page using standard Lightning application events.

One-to-One vs One-to-Many Messaging Scheme

When you send a message from a Lightning component to the iframe it wraps using **contentWindow.postMessage()**, there can only be one Visualforce page loaded in that **contentWindow**. In other words, that Visualforce page is the only place where you can set up a message event listener and get the messages sent by the Lightning component in that fashion. This is a **one-to-one** messaging scheme.

When you send a message from an iframed Visualforce page to its Lightning component wrapper using **parent.postMessage()**, **parent** is a reference to your main **window** in Lightning Experience where other Lightning components may be loaded. If other Lightning components loaded in the same window object set up a message event listener, they will receive the Visualforce messages as well. This is a **one-to-many** messaging scheme, and it's something to account for both when you send and receive messages. For example, you could name messages to allow Lightning components to filter incoming messages and only handle messages they are interested in.

As an example, let's modify the Visualforce-to-Lightning component example above. The Visualforce page now sends a named ("com.mycompany.chatmessage") message to its parent window:

```
<apex:page>
<input id="message" type="text"/>
<button onclick="sendToLC()">Send to LC</button>

<script>

    var lexOrigin = "https://yourdomain-dev-ed.lightning.force.com";

    function sendToLC() {
        var payload = document.getElementById("message").value;
        var message = {
            name: "com.mycompany.chatmessage",
            payload: payload
        };
        parent.postMessage(message, lexOrigin);
    }

</script>

</apex:page>
```

feedb

The Lightning component filters incoming messages and only handles "com.mycompany.chatmessage" messages:

```
({
    doInit : function(component) {
        var vfOrigin = "https://" + component.get("v.vfHost");
        window.addEventListener("message", function(event) {
            if (event.origin !== vfOrigin) {
                // Not the expected origin: Reject the message!
                return;
            }
            // Only handle messages we are interested in
            if (event.data.name === "com.mycompany.chatmessage") {
                // Handle the message
                console.log(event.data.payload);
            }
        }, false);
    }

})
```

Bidirectional Communication

You can easily combine the two examples above to establish bidirectional communication between the Lightning component and the Visualforce page. The video below shows a simple example of bidirectional communication based on a polished version of these examples.

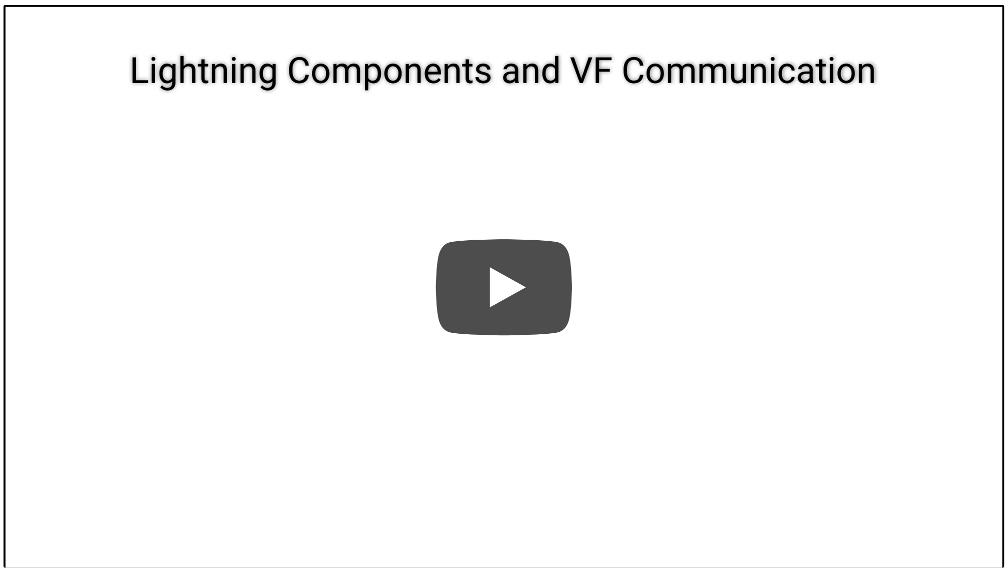
Watch the video:

The source code is available in this GitHub repository: VFWrapper component and WrappedVF Visualforce page.

Google Maps Example

Let's finish with a real life scenario based on the DreamHouse sample application. In this example, we have a master-details relationship. The master is a list of properties (houses for sales) and is built as a Lightning component. The details is another Lightning component that shows the selected property on a map. Since we can't use Google Maps directly in a Lightning component (libraries loaded from a CDN aren't currently supported), we wrap a Visualforce page where Google Maps is loaded. Whenever a new property is selected in the master component, the details component retrieves the property information and sends it to the Visualforce using postMessage().

Watch the Video:



The source code is available in this GitHub repository: GoogleMapWrapper component and GoogleMap Visualforce page.

feedb

Important Security Consideration

window.postMessage() is a standard web API that is not aware of the Lightning and Locker service namespace isolation level. As a result, there is no way to send a message to a specific namespace or to check which namespace a message is coming from. Therefore, messages sent using postMessage() should be limited to non sensitive data and should not include sensitive data such as user data or cryptographic secrets.

Summary

Using a simple Lightning Component wrapper and window.postMessage(), you can proxy Lightning events to a Visualforce page. And the other way around, you can send messages from a Visualforce page to its parent window. Using this secure and standard-based approach, you can tightly integrate Visualforce pages in Lightning Experience and support all your communication requirements: Lightning to Visualforce, Visualforce to Lightning, and even Visualforce to Visualforce inside Lightning Experience.

Additional Resources

- Trailhead: Visualforce & Lightning Experience module
- Trailhead: Lightning Experience Development
- window.postMessage() API documentation
- Sample code repository

Published