

Building Blocks

A Modern Approach to Building Media Workflows

How Media Shuttle, Slack and a little javascript shaved critical time for a distributed production team

*This article is part of an upcoming series on **Implementing Modern Media Workflows with SaaS Building Blocks**. Over the course of this series, Signiant will offer real-world technical examples demonstrating the power of connecting off-the-shelf functional modules via APIs.*

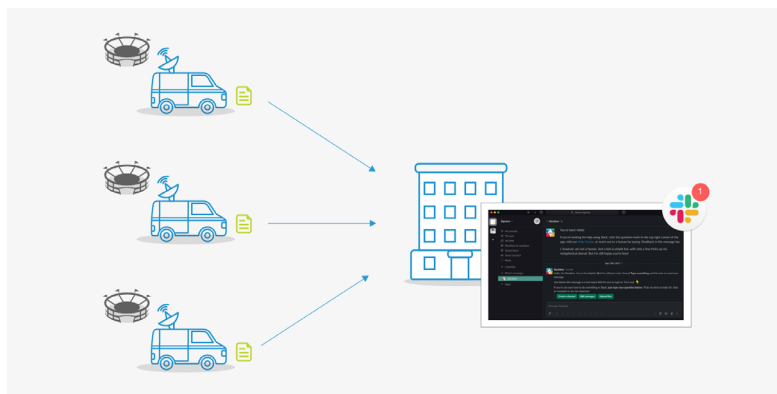
In this installment, Signiant demonstrates connecting Media Shuttle and Slack in a simple integration used recently in a live sports production.

Build vs. buy has always been a continuum rather than a single binary decision. Every media enterprise buys technology at some level and builds systems by hooking the various pieces together. In the old days, the build process involved connecting hardware products with coax and BNC connectors. In the cloud era, the analogous process is connecting various web services with APIs and lightweight business logic. Because there are endless possibilities within the giant sandbox of web services, careful management of the build vs. buy dimension is essential for media companies.

With that, it's not a question of build vs. buy but rather which size blocks to build with. Signiant believes that the sweet spot is somewhere near the middle of the continuum from microservices through end-to-end systems. Our technology is packed into full-stack products, providing media companies with modules that can be readily deployed as components of larger workflows. With web services and good APIs, these products can easily be connected together to address specific business requirements and can be readily swapped out as technology evolves. With this approach, the customer is rewarded with much faster time-to-value while maintaining agility and favorable economics.

Real-World Application

Let's take a look at a real production example where this approach was put into practice with great success and in a very short amount of time. A large sports broadcaster was simultaneously covering three live sporting events in three different cities across the U.S. In addition to the live action, camera crews at each location were shooting interviews and capturing B-roll to intercut into highlight reels. The broadcaster needed to move camera card files from the remote production location to the main production facility quickly, and a distributed production team needed to be informed in the most efficient way as new clips became available.



As always, things are moving very fast. The sports director wants it now, now, now, and the editors also have other packages to concentrate on. Rather than waiting and watching for the delivery in multiple messaging services, like email, the broadcaster wanted those notifications to come to their existing Slack service used for internal production communication — a single communication source.

Coupling Signiant Media Shuttle and Slack using some simple scripting lets them do exactly that and stay in the platform they already use. The workflow went from idea to production in a couple of days, using familiar tools, and helped get quick edits on the air, easily. Since both Media Shuttle and Slack are SaaS offerings, minimal effort was required to deploy the two major building blocks themselves.

This is a very simple example of an actual Signiant use case that illustrates the power of the building block approach; it happened to be in the sports industry but the same concept could be applied to many situations. Here is the actual code to make it happen.



Connecting Media Shuttle and Slack

Signiant has a rich set of API's allowing integration with Signiant products. There are a variety of REST API's to Transfer, Manage, and Audit in and around the Media Shuttle product. In this build example, we focus on a small and specific subset of the Media Shuttle REST API — just enough to get the job done.

- **Media Shuttle REST API** supports a Webhook concept – essentially a user-defined HTTP callback that allows for a web backend to call out to an HTTP service based on some event or system-related action. In this case, we focus on “File Transfer” events. It's possible for errors to occur during a transfer and we will present a simple mechanism for handling an error condition, but the emphasis here will be on sending Slack notifications upon a successful transfer.
- **Slack** also has a very rich Web (REST) API to interact with. Again, here we'll focus on just a small subset of the Slack Web API.
- **The code example(s)** will be in the form of JavaScript, specifically in the context of a node.js application.

Setting Up A Slack Application

Start by obtaining a Slack API token. There are a few different types of Slack tokens. In this example we use a “[bot user token](#)” since we will effectively be operating as a bot to send notifications based on an upload complete webhook callback from Signiant Media Shuttle.

1. To obtain the bot user token, create (define) a new [Slack application](#) and click on **CREATE NEW APP**.

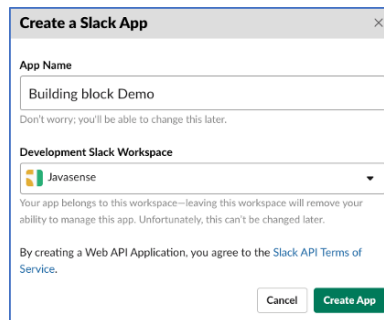
Build something amazing.

Use our APIs to build an app that makes people's working lives better. You can create an app that's just for your workspace or create a public Slack App to list in the App Directory, where anyone on Slack can discover it.

Create an App

2. Next, fill in the **APP NAME** field and choose a **DEVELOPMENT SLACK WORKSPACE** from the dropdown list. Note that you may need to sign in to Slack to get to the proper Workspace.

Click the **CREATE APP**.



Create a Slack App

App Name
Building block Demo
Don't worry, you'll be able to change this later.

Development Slack Workspace
Javasense
Your app belongs to this workspace—leaving this workspace will remove your ability to manage this app. Unfortunately, this can't be changed later.

By creating a Web API Application, you agree to the [Slack API Terms of Service](#).

Cancel Create App

3. In the **BUILDING APPS FOR SLACK** window, click the **BOTS** box.

Building Apps for Slack

Create an app that's just for your workspace (or build one that can be used by any workspace) by following the steps below.

Add features and functionality

Choose and configure the tools you'll need to create your app (or review all our [documentation](#)).

Building an internal app locally or behind a firewall?

To receive your app's payloads over a WebSockets connection, enable [Socket Mode](#) for your app.

Incoming Webhooks

Post messages from external sources into Slack.

Interactive Components

Add components like buttons and select menus to your app's interface, and create an interactive experience for users.

Slash Commands

Allow users to perform app actions by typing commands in Slack.

Event Subscriptions

Make it easy for your app to respond to activity in Slack.

Bots

Allow users to interact with your app through channels and conversations. 🤖

Permissions

Configure permissions to allow your app to interact with the Slack API.

4. The next window is to assign scopes to the bot token. Click **REVIEW SCOPES TO ADD**.

First, assign a scope to your bot token


A [bot token](#) makes it possible for users to interact with your app. A bot token lets users at-mention it, and add it to channels and conversations. It also allows you to turn on tabs in your app's home.

[Scopes](#) govern an app's capabilities and permissions. You'll need to add at least one to your bot token to show the Home or Message tab in your app home. You can review and add the appropriate scopes under **Scopes** section in [OAuth & Permissions](#).

[Review Scopes to Add](#)

5. Select the following and click **ADD AN OAUTH SCOPE** after selecting each one.

- chat:write:customize
- chat:write
- chat:write:public




 You can now show tabs on App Home
Manage which tabs your user sees in your app's home. [Go to App Home](#)

Scopes

A Slack app's capabilities and permissions are governed by the [scopes](#) it requests.

Bot Token Scopes

Scopes that govern what your app can access.

OAuth Scope	Description	
chat:write.customize	Send messages as Building block Demo with a customized username and avatar	
chat:write	Send messages as Building block Demo	
chat:write.public	Send messages to channels Building block Demo isn't a member of	

[Add an OAuth Scope](#)

6. Under **OAuth Tokens & Redirect URLs** at the top of the page, click **INSTALL TO WORKSPACE**.

OAuth Tokens & Redirect URLs

These [OAuth Tokens](#) will be automatically generated when you finish connecting the app to your workspace. You'll use these tokens to authenticate your app.

[Install to Workspace](#)

Redirect URLs

You will need to configure redirect URLs in order to automatically generate the Add to Slack button or to distribute your app. If you pass a URL in an OAuth request, it must (partially) match one of the URLs you enter here. [Learn more](#).

Redirect URLs

You haven't added any Redirect URLs

[Add New Redirect URL](#)

[Save URLs](#)

7. You may be prompted for access permissions. Click the **ALLOW** button.



Building block Demo is requesting permission to access the Javasense Slack workspace

What will Building block Demo be able to do?

Perform actions in channels & conversations

[Cancel](#)

[Allow](#)

8. Copy the **BOT USER OAuth ACCESS TOKEN**. This access token is used later to send a message from the node.js application.

OAuth Tokens & Redirect URLs

Tokens for Your Workspace

These tokens were automatically generated when you installed the app to your team. You can use these to authenticate your app. [Learn more](#).

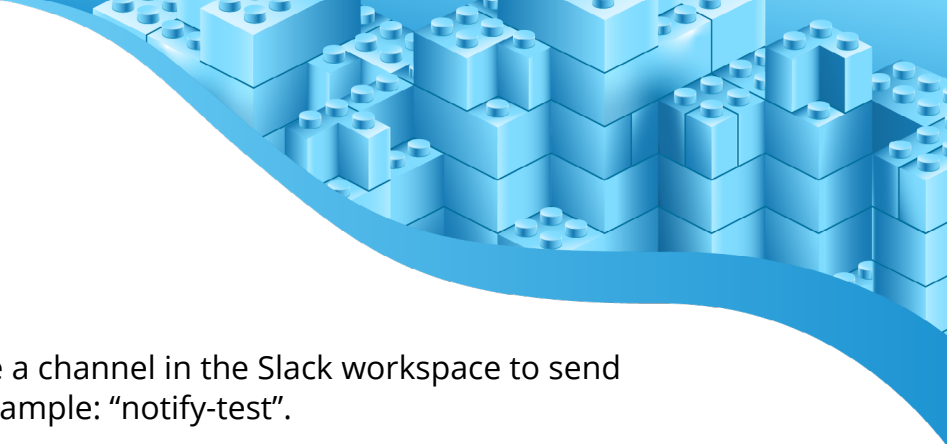
OAuth Tokens for Your Team

Bot User OAuth Access Token

xoxb-633316546305-1783454502805-2KqLHPdohXgHaTuVNYJ6xMk [Copy](#)

Access Level: Workspace

[Reinstall to Workspace](#)

- 
9. If you haven't done so, create a channel in the Slack workspace to send notification messages. For example: "notify-test".

Now let's turn our attention to Setting up a Media Shuttle Webhook.

Setting up a Media Shuttle Webhook

In this section, we use the Signiant Media Shuttle API to obtain a list of available portals and their detailed information and install a webhook (subscription) to monitor transfers related to that portal.

NOTE: This tutorial assumes an already configured Media Shuttle account and at least one portal for use here. Reference the [Get Started with Media Shuttle API](#) instructions for additional information.

1. To authenticate with the Signiant Media Shuttle API, an API key is needed from the Media Shuttle administrative interface. Refer to the [Get Started with Media Shuttle API](#) instructions on how to obtain an API key.
2. Make a copy of your API key. The key is needed in subsequent steps. You will only need to reference what is in the "Authentication" section in the [Get Started with Media Shuttle API](#).

The Media Shuttle API key authenticates when making API calls against the Media Shuttle account.

It is important to note the API key is used as the value for the HTTP Authorization Header in REST API calls made.

For example with curl:

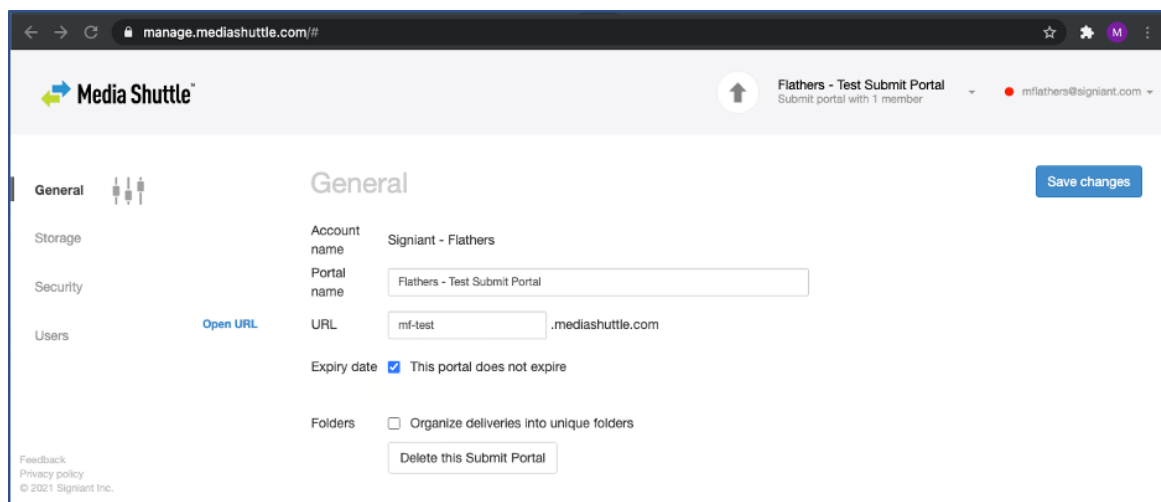
```
curl ... -H "Authorization: <YOUR_API_KEY>"
```

3. Here is a complete curl example demonstrating exactly how to use the API key.

The screenshot below is from a test account showing details for a portal by the name of "mf-test" in the Media Shuttle UI.

Note the URL for the portal, **mf-test.mediashuttle.com** in this case.

The Media Shuttle API uses internal portal identifiers (id's) to reference a particular portal. Instructions below show how to obtain a particular portal's id and use it in subsequent API calls.



4. Now let's issue our first Media Shuttle API call to obtain detailed portal information with curl:

curl -X GET

"https://api.mediashuttle.com/v1/portals?url=mf-subtest.mediashuttle.com"

-H "accept: application/json" -H "Authorization: <YOUR_API_KEY>"

5. Replace **<YOUR_API_KEY>** with the API key made above.
6. Replace **mf-subtest.mediashuttle.com** with the URL for your Media Shuttle Portal as indicated above.

7. The curl command should return something similar to the following:

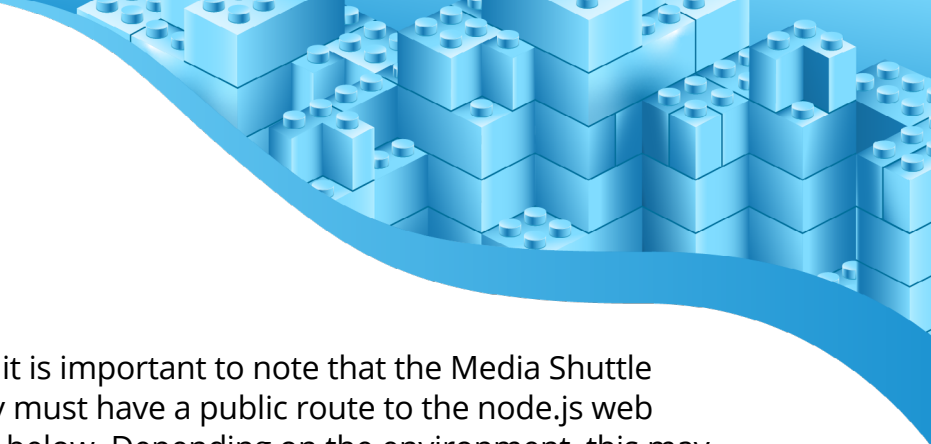
```
{
  "items":[
    {
      "id":"0824e-d1ea-4fb4-a122-221055506621e",
      "name":"Flathers - Metadata Test Submit Portal",
      "url":"mf-subtest.mediashuttle.com",
      "type":"Submit",
      "createdOn":"2021-01-28T01:49:02.924Z",
      "lastModifiedOn":"2021-01-28T01:50:00.739Z",
      "authentication":{
        "mediaShuttle":true,
        "saml":false
      },
      "linkAuthentication":{

      },
      "notifyMembers":true
    }
  ]
}
```

Note the information returned regarding the portal of interest:

```
"id":"0824e-d1ea-4fb4-a122-221055506621e",
"name":"Flathers - Metadata Test Submit Portal",
"url":"mf-subtest.mediashuttle.com",
```

The value of the id field (0824e-d1ea-4fb4-a122-221055506621e) is how we will reference this portal in order to install a transfer monitoring webhook (subscription).

- 
- Before installing the webhook, it is important to note that the Media Shuttle control plane hosted externally must have a public route to the node.js web application that will be created below. Depending on the environment, this may include the need for port mapping which is beyond the scope of this document.

In the example below, the application hosting the webhook will be **my-test.signiant.com on port 8081** (my-test.signiant.com:8081). Update the hostname and port to match your own environment.

In the node.js code used, we define a service endpoint of /TransferUpdate resulting in the following URL endpoint to use in the call to install the webhook:

http://my-test.signiant.com:8081/TransferUpdate

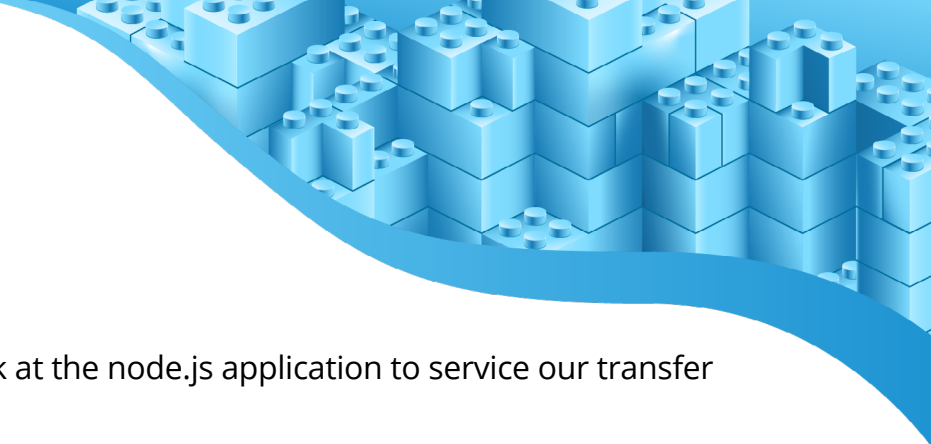
- Issue the following curl command to install the webhook.

```
curl -X POST
"https://api.mediashuttle.com/v1/portals/<YOUR_PORTAL_ID>/subscriptions" -H "accept: application/json" -H "Authorization: <YOUR_API_KEY>" -H
"Content-Type: application/json" -d
"{\"type\": \"webhook\", \"details\": {\"url\": \"http://my-test.signiant.com:8081/TransferUpdate\"}}"
```

Note: Substitute <YOUR_PORTAL_ID> with your portal id and <YOUR_API_KEY> with your API key and my-test.signiant.com with the public IP address of the machine which will be hosting the node.js application discussed above.

- If successful, you should get a response similar to the following:

```
{
  "id": "728d13e2-1ae9-4ced-a64d-064363985866",
  "type": "webhook",
  "details": {
    "url": "http://my-test.signiant.com:8081/TransferUpdate"
  }
}
```



Now we're ready to have a look at the node.js application to service our transfer operation webhook.

11. Ensure a proper version of the Node.js runtime and NPM is already installed.
12. Download the source code and dependencies for the building block Webhook Server from the following location:

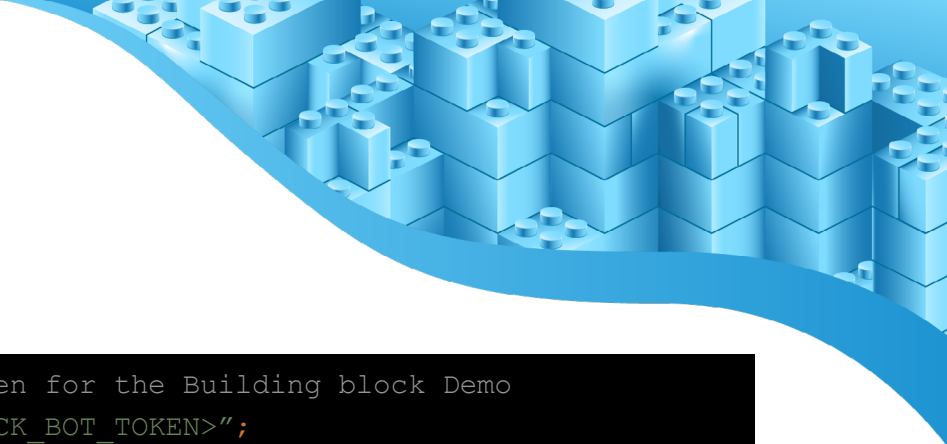
<INSERT DOWNLOAD LOCATION HERE>

This node.js application uses the following external packages: Express, Body-Parser, and @slack/web-api.

13. Make adjustments to the Javascript code to match your environment. If you encounter runtime reference errors, you may need to run the following commands to ensure the external packages listed above are installed properly:

```
npm install express
npm install body-parser
npm install @slack/web-api
```

14. Note: Although the Slack REST API could be used directly, for this example we will be using the [Node Slack SDK](#) (@slack/web-api).
 - The code you'll need to modify is in a single JavaScript source file: **Slackutils.js**.
 - All other supporting source files should not need to be modified unless choosing to do so. For example, to change routing.
 - Additionally, although nothing needs to be changed in the server.js file, a few code snippets contained therein are referenced.
15. SlackUtils.js. There are only two lines of code to change here.
16. First, change the "token" constant to the Slack Bot token value obtained from the Slack configuration at the beginning of this guide.
17. Second, change the "channel" constant to the Slack channel's name the notifications should be sent to. Don't forget to add the # at the beginning of the channel name. See the code snippets below:



```
//Below is the Slack token for the Building block Demo
const token = "<YOUR_SLACK_BOT_TOKEN>";

// ChannelId from Building Block Demo
const channel = '<YOUR_SLACK_CHANNEL>';
```

The SlackUtils.js file really has only a single function in it which simply calls the Slack API postMessage function to send the message passed in as a parameter:

```
sendMessage: function (message) {
  (async () => {
    // See: https://api.slack.com/methods/chat.postMessage
```

```
const res = await web.chat.postMessage({ channel: channel,
text: message });

// `res` contains information about the posted message
console.log('Message sent: ', res.ts);
}) (); }
```

18. Now let's have a look at the server.js file. This is where the meat of the webhook processing is done.
19. First, we use Express to provide the web server functionality and listens on port 8081:



```
var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host,
port)
})
```

In this setup, when a file action such as a transfer occurs on the monitored portal, an HTTP POST is issued to the url specified when the webhook was installed IE:

`http://my-test.signiant.com:8081/TransferUpdate`

The details behind how this HTTP POST is handled may be seen in the code section that starts with:

```
app.post('/TransferUpdate', ...
```

20. You can inspect the details here. As you can see, we evaluate the payload to formulate a message to send to Slack, mostly focusing on Upload Complete, Download Complete, and Deleted event types.

Also note the code handles the case where there are multiple files involved.

The variable "message" contains the final result of the Slack message that will be sent.

For example the following is a code snippet that shows a portion of how the final message is created:

```
if (transferObj.eventType === "package.upload.complete" ||
transferObj.eventType === "package.download.complete") {
```

```

basePath =
transferObj.payload.portalDetails.storage[0].configuration repositoryPath;
    let firstOne = true;
    transferObj.payload.packageDetails.files.forEach(filename =>
    {
        if (firstOne) {
            if (basePath) {
                filenames += basePath + "/" + filename.path + "
(" + convertBytes(filename.size) + ")";
            }
            else {
                filenames += filename.path + " (" +
convertBytes(filename.size) + ")";
            }
            firstOne = false;
        }
        else {
            if (basePath) {
                filenames += "\n" + basePath + "/" +
filename.path + " (" + convertBytes(filename.size) + ")";
            }
            else {
                filenames += "\n" + filename.path + " (" +
convertBytes(filename.size) + ")";
            }
        }
    });

    let direction = "";

    if (transferObj.eventType === "package.upload.complete") {
        direction = "uploaded";
    }

```

```

    else if (transferObj.eventType ===
"package.download.complete") {
        direction = "downloaded";
    }

    let message = "The following files have been " + direction
+ " by " + user + ":\n" + filenames;

```

21. Finally the `SlackUtils.sendMessage()` is called to send the Slack message:

```

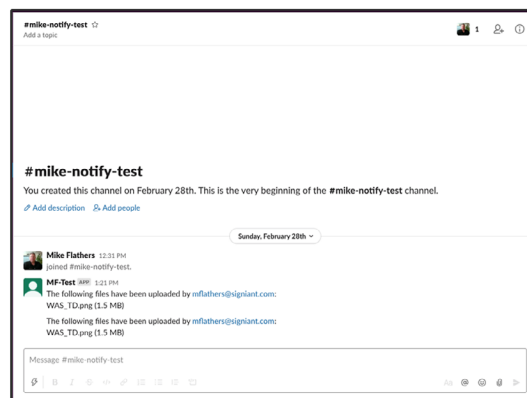
slackUtils.sendMessage(message);

```

22. After making the changes mentioned above, we are ready to run the application. This may be done with the following command:

Node server.js

Properly done, a Slack message is sent when a Media Shuttle file action is performed.





Building with the Right Size Blocks

This example illustrates the power of connecting functional modules via APIs to quickly solve pressing operational problems. It's also a good reminder that connecting a general-purpose tool (Slack) with a media-centric one (Media Shuttle) is often the best way to meet media industry needs. Both Slack and Media Shuttle are out-of-the-box functional building blocks that are sufficiently general-purpose to warrant the supplier's investment in a multi-tenant, SaaS product offering. And because most modern SaaS products offer robust APIs, it is straightforward to connect these products into larger workflows.

Each building block does a specific job. By using off-the-shelf, best-of-breed products, combined with modern APIs and scripting, media companies can create solutions that can be easily adapted as situations change. Scripts can be updated and any block can easily be swapped out for another if necessary. This approach enables media companies to build new workflows faster, using familiar tools, and dramatically shrink the time-to-value.

*Signiant was an early leader in cloud-native SaaS, and developed one of the first SaaS solutions for use directly in the media supply chain. Media Shuttle was released in 2012, providing media companies of all sizes with access to advanced transport technology that enables fast, securable, reliable transfer of large files. Since then, Signiant has continued to innovate, introducing two additional SaaS products, *Flight* and *Jet*, built on the same SaaS platform and with the same cloud-native approach. While these full-stack products are widely used for stand-alone use cases, the products themselves and components within them are often components of much larger and more complex workflows thanks to an investment in robust, modern APIs. This approach is commonplace in the broader technology industry and is now emerging in the media technology world as the industry moves from an on-premise world towards a more cloud-centric world.*

About Signiant



Signiant's enterprise software provides the world's top content creators and distributors with fast, reliable, secure access to large media files, regardless of physical storage type or location. By enabling authorized people and processes to seamlessly exchange valuable content — within and between enterprises — Signiant connects the global media supply chain. **Find out more at www.signiant.com.**