Actions from Data Views: Adding images to the Media Library

<!-- @begin notes →

Notes Part 1

https://docs.google.com/document/d/1BwsiVbo1T46KYpb0N-dCV0y3REC0_YFKNH3D0yPo8Cs/edit

https://developer.wordpress.org/news/2024/08/27/using-data-views-to-display-and-interact-with-data-in-plugins/

<!-- @end notes →

It's been almost a month since Using Data Views to display and interact with data in plugins was published. In that post, you discovered how to create a plugin that displays a React app in the WordPress admin to list a dataset of pictures using Data Views. With the `DataViews` component, you were able to display a dataset of images and provide your users with a nice UI to display, sort, search, and filter the list of photos.

In the previous post, you learned how to:

- Add a custom React app to the admin screens.
- Leverage the DataViews component to display datasets.

With the User Interface ready for the user to display, sort, search, and filter a list of pictures, it's time to take the Data Views actions further and provide users with tools to, among other things, add any images listed directly to the Media Library.

In this article, you'll build:

- Actions that enable users to upload selected images directly to the Media Library.

- Actions that open modal windows containing intermediate dialogs to launch specific operations.
- A user-friendly interface that offers real-time feedback, keeping users informed about the processes being executed.

<note callout>
After laying the groundwork for using Data Views, this post explores the actions you can take with them. These include interacting with WordPress media through the REST API, creating notification boxes, displaying modal windows, and using other UI components,
</note callout>

Here's a video showing what the features you'll add to the app in this article will look like:

<insert video>

Before you start

In this article, you'll build on the project started in the previous article, continuing from where we left off.

To get the project at the starting point of the project for this article, you can do the following:

```
None
git clone
git@github.com:wptrainingteam/devblog-dataviews-plugin.git
git checkout part1
```

<tip callout>
The final code of the project explained in this article is available on GitHub. Throughout the article, you'll find links to specific commits corresponding to the changes being explained, to help you track the project's progress.
</tip callout>

Action to add images to the Media Library

At its current state, the project has only one action to display the images in full size. Let's now create a new action to upload images to the Media Library.

Add the following code to the `actions` array that is passed to the `actions` prop of the `Dataviews` component:

[src/App.js]

JavaScript

```
const actions = [
	{
		id: 'upload-media',
		label: __( 'Upload Media' ),
		isPrimary: true,
		icon: 'upload',
		supportsBulk: true,
		callback: ( images ) => {
			images.forEach( ( image ) => {
				console.log( `Image to upload: ${
	image.slug }` );
			} );
		},
	},
	...
]
```

This new action will enable multi-selection in the Dataviews UI. For now, when the action is triggered, it will log a message to the console for every image selected.

<tip callout>
Remember to have `npm start` (which is the alias of "npm run start") running to automatically generate the project's build version when a file changes.
</tip callout>

The only action currently defined for this project (`see-original`) was defined through the properties 'id,' 'label,' and 'callback'. As you can see above, the definition of this new 'upload-media', action uses some additional properties such as:

- `isPrimary` - By setting this property to `true` this action will become the default action for each item.
- `supportsBulk` - You can enable multi-selection of items by setting this property to `true`. With this setting enabled you can perform the action on several items at the same time.
- `icon` - icon to show for primary actions.

Notice how the callback function of this action is now prepared to act on one or more items by using `forEach` to perform some logic on each one of the items selected.

<info callout>
Check here the complete list of properties that can be used to define actions for the Dataviews items.
</info callout>

Now that the 'upload-media' action is created, let's take some time to consider the steps needed to upload an image to the Media Library given a URL.

1- Download the image from the provided URL and convert it to a blob
2- Create a FormData object with the image blob that can be sent as the body of the POST request
3- Send a POST request with apiFetch to the proper endpoint of the WordPress REST API to add the image to the Media Library

Basically, you'll need to make a POST request to a specific REST API URL, including data with the image in binary format. To do this, some preliminary steps are required to prepare the data for the REST API request.

Add these steps as comments of the `forEach` callback to have them as a reference as we add the logic for each step:

[src/App.js]

```JavaScript
const actions = [
	{
		id: 'upload-media',
		...,
		callback: ( images ) => {
			images.forEach(( image ) => {
				// 1- Download the image and convert it to a blob
				// 2- Create FormData with the image blob
				// 3- Send the request to the WP REST API with apiFetch
			} );
		},
	},
	...
]
```

Download the image and convert it to a blob

The image URL for each item is available at `image.urls.raw`, so you could use the browser's native `fetch` method to download each image and convert the response to a blob.

<note callout>
To send an image in a POST request, you need to convert it into a format that can be transmitted over the network. A common format for this is a binary large object (Blob).
</note callout>

Add this code to the `forEach's` callback function:

[src/App.js]

JavaScript

```
images.forEach( async ( image ) => {
      // 1- Download the image and convert it to a blob
      const responseRequestImage = await fetch( image.urls.raw
);
      const blobImage = await responseRequestImage.blob();
      ...
})
```

Since `fetch` is an asynchronous method that returns a promise, you can declare the `forEach` callback as an `async` function . This allows you to use the `await` operator and avoid promise chains to handle the promise responses.

The successful response from `fetch` is encapsulated in a Response instance, which includes a `blob()` method. This method allows you to directly generate a blob from the response.

Create FormData with the image blob

With the image in blob format, the next step is to prepare the data to be sent in an HTTP request to the REST API. One way to send data in a POST request is to use a `FormData` object as the request body.

<info callout>
Methods like fetch (or WordPress' `apiFetch`) can use a `FormData` object as the request body. This data is encoded and transmitted with the `Content-Type` set to `multipart/form-data`
</info callout>

Add this code to the `forEach`'s callback function:

[src/App.js]

```JavaScript
images.forEach( async ( image ) => {
	...
	// 2- Create FormData with the image blob
	const formDataWithImage = new FormData();
	formDataWithImage.append(
		'file',
		blobImage,
		`${ image.slug }.jpg`
	);
	...
})
```

The code above creates a new `FormData` object, appending a `file` field containing the image in blob format.

<info callout>
A `FormData` object represents HTML form data. From the server's perspective, it would be as if you submit the data from an HTML form.
</info callout>

Send the request to the WP REST API

At this point, everything is ready to do the REST API request with the `apiFetch` method.

<info callout>
`apiFetch` is a wrapper around `window.fetch` that offers several advantages when making requests to the WP REST API, such as automatically completing the Base URL for the REST API endpoint and including the `nonce` in the request headers.
</info callout>

Go to the beginning of `App.js` and import `apiFetch` method from `@wordpress/api-fetch`:

[src/App.js]

```JavaScript
import apiFetch from "@wordpress/api-fetch";
```

Now add this code to the `forEach`'s callback function:

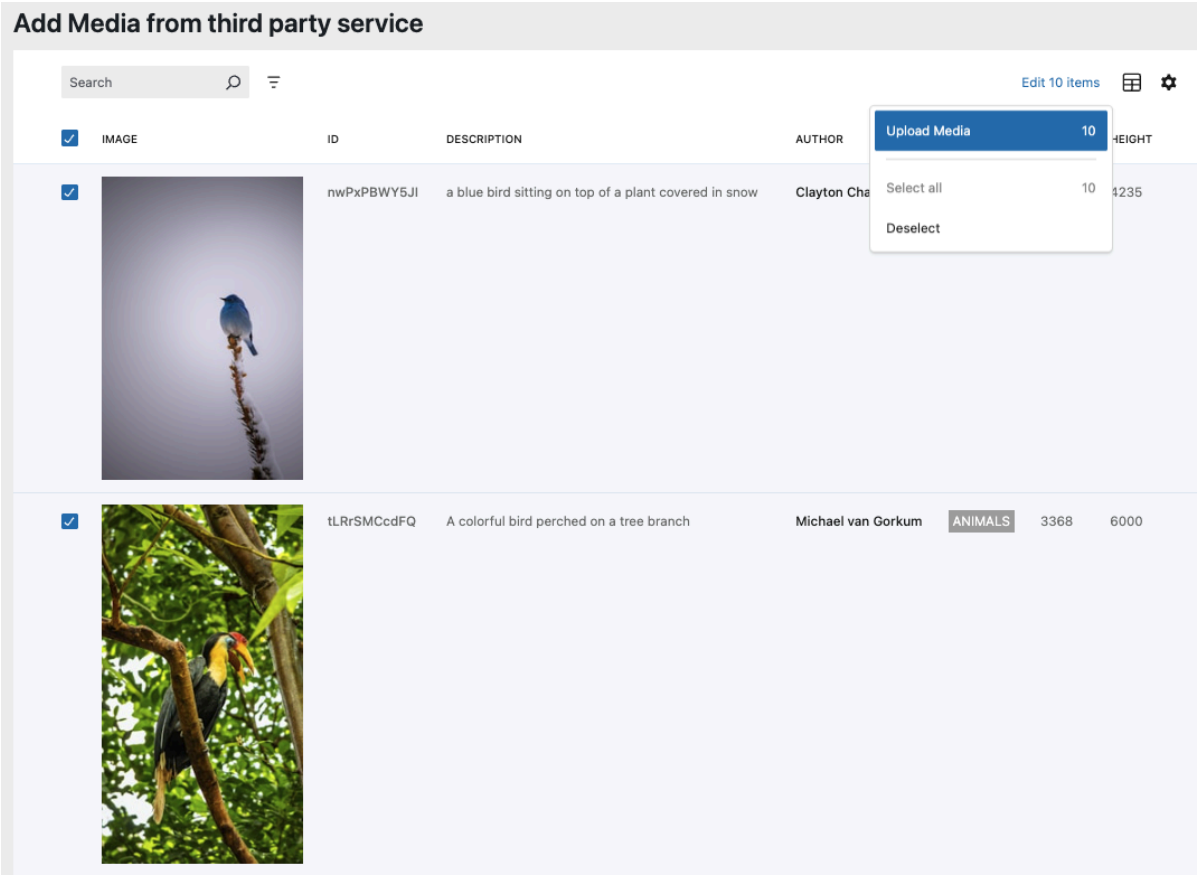
[src/App.js]

```JavaScript
images.forEach( async ( image ) => {
	...
	// 3- Send the request to the WP REST API with apiFetch
	await apiFetch({
		path: "/wp/v2/media",
		method: "POST",
		body: formDataWithImage,
	})
	.then( console.log )
	...
})
```

The code above uses `apiFetch` to do a (POST) request to the ` wp/v2/media` endpoint. As documented in the REST API Handbook, you can create a Media Item in the WordPress installation via POST requests to the `/wp/v2/media` endpoint.

<info callout>
Public REST API has been part of the core since WP 4.7
</info callout>

Your Data Views now includes a new feature that allows you to upload displayed photos directly to the Media Library. You can upload images individually or select multiple images for bulk uploading.



However, the feedback the user is getting about the upload process isn't great, as it only provides a message in the console. Let's work on improving this user experience.

Notification boxes with the outcome of the upload processes

WordPress provides a notification system that can be managed via the Notices Store. A good approach to interact with the Notices UI, is to wrap your React component with the `withNotices` Higher-Order Component.

By wrapping your `App` component with `withNotices`, the `App` component will receive the additional props `noticeOperations` and `noticeUI`:

- `noticeOperations` is an object that contains the `createNotice` method (among others), which you can use to generate a specific notification box.

- ` noticeUI` is the Notice React component that will be displayed when a notification box is created.

Go to the beginning of `App.js` and import `withNotices` method from "@wordpress/components":

[src/App.js]

```
JavaScript
import { withNotices } from "@wordpress/components";
```

Go to the line that starts the definition of your `App.js` and replace it with the following code:

[src/App.js]

```
JavaScript
const App = withNotices(({ noticeOperations, noticeUI }) => {
      const { createNotice } = noticeOperations;
      ...
});
```

You can use the `createNotice ` method destructured from `noticeOperations` to create notification boxes that will appear wherever you place the `noticeUI` component on your React App's screen. These notification boxes can be used to provide information to the user about the success or failure of images uploaded to the Media Library.

Chain the following `then` and `catch` methods to the `apiFetch` call to handle the success or error of the API request, using the yet-to-be-defined `onSuccessMediaUpload` and `onErrorMediaUpload` functions:

[src/App.js]

```
JavaScript

  await apiFetch({
```

```
            path: "/wp/v2/media",
            method: "POST",
            body: formDataWithImage,
          })
            // 5- Capture the success and error responses of
the REST API request
            .then(onSuccessMediaUpload)
            .catch(onErrorMediaUpload);
        });
```

Now, add the following code to the App component (and before the call of these functions) with the `onSuccessMediaUpload` and `onErrorMediaUpload' functions definitions:

[src/App.js]

JavaScript

```
const onSuccessMediaUpload = (oImageUploaded) => {
    const title = oImageUploaded.title.rendered;
    createNotice({
      status: "success",
      content: __(`${title}.jpg successfully uploaded to Media
Library!`),
      isDismissible: true,
    });
  };

  const onErrorMediaUpload = (error) => {
    console.log(error);
    createNotice({
      status: "error",
      content: __("An error occurred!"),
      isDismissible: true,
    });
  };
```

The `onSuccessMediaUpload` receives the object returned by the "/wp/v2/media" endpoint with the details of the image successfully uploaded. With this info you can generate a `success` box informing the user about the success of the image upload.

If the upload operation fails, the `onErrorMediaUpload` will receive the specific error that prompted the failure of the upload operation. Inside this function you can also call a `error` box to notify the user that something went wrong.

To display the notification boxes triggered by the `createNotice` methods, add the `noticeUI` component to the return of the `App` component:

[src/App.js]

```JavaScript
return (
   <>
     {noticeUI}
     <DataViews
        ...
     />
   </>
 );
```

<info callout>
React components can only return one parent element, so you can use the `Fragment` component to group your elements together.
</info callout>

Since notification boxes will appear at the top of the screen, you can add the following code at the start of the 'upload-media' callback to scroll to the top each time this action is triggered, ensuring users don't miss any notifications:

[src/App.js]

```JavaScript
window.scrollTo( 0, 0 );
```

A Spinner to indicate uploading processes in progress

A good UI element that could be added is an indicator of uploading processes in progress. The `Spinner` component from WordPress components is perfect for this.

But before using the Spinner component, some logic needs to be added to ensure it is displayed only while upload processes are running. To monitor the ongoing upload processes, you can use a state variable that holds an array of all the uploads in progress.

Keeping track of the upload processes using state variables

Include the following code at the start of your App component to add a state variable using `useState`:

[src/App.js]

```JavaScript
const [isUploadingItems, setIsUploadingItems] = useState([]);
```

The `isUploadingItems` state variable will keep track of the upload operations in progress. Every time a new image starts its upload process, its slug will be added to the `isUploadingItems` array. And every time a new image is successfully uploaded, its slug will be removed from the `isUploadingItems` array.

Add the following code at the beginning of the `forEach` callback (inside the `upload-media` callback):

[src/App.js]

```
JavaScript

setIsUploadingItems((prevIsUploadingItems) => [
  ...prevIsUploadingItems,
  image.slug,
]);
```

The code above adds every image's slug being selected for upload to the `isUploadingItems` state variable array.

To remove an image's slug from the `isUploadingItems` state variable array when it has been successfully uploaded or emptying it completely when an error occurs you can add the following pieces of code to the `onSuccessMediaUpload` and `onErrorMediaUpload` functions.

[src/App.js]

```
JavaScript

const onSuccessMediaUpload = (oImageUploaded) => {
    ...
    setIsUploadingItems((prevIsUploadingItems) =>
      prevIsUploadingItems.filter((slugLoading) => slugLoading
!== title)
    );

    ...
  };

const onErrorMediaUpload = (error) => {
    setIsUploadingItems([]);
    ...
  };
```

Displaying the Spinner component

While the `isUploadingItems` state variable array is not empty, the Spinner component should be displayed to indicate to the user that there's some uploading process in progress.

Navigate to the top of App.js and import the Spinner component from @wordpress/components:

[src/App.js]

```
JavaScript
import { Spinner } from "@wordpress/components";
```

Finally, to display the `Spinner` component based on the the existence of any image being uploaded add the following code to the return of the `App` component:

[src/App.js]

```
JavaScript
return (
   <>
     {!!isUploadingItems.length && <Spinner />}
     ...
     <DataViews
        ...
     />
   </>
 );
```

Action with Modal window

Remember the action you created in the previous article to display an image in full size? What about improving this action to show a modal so users can choose the size of the image they want to open in a new window?

Data Views actions provide a mechanism to display a Modal window when triggered. Through the `RenderModal` property you can define a React component that will be displayed upon the action call as a Modal window. When the `RenderModal` property is provided, the `callback` property is ignored.

A `modalHeader` property can also be defined to set a header text for the Modal window.

You could replace the 'see-original' action definition with the following code:

[src/App.js]

```JavaScript
{
	id: 'see-original',
	label: __( 'See Original' ),
	modalHeader: __( 'See Original Image', 'action label' ),
	RenderModal: ( { items: [ item ], closeModal } ) => {
		return (
			<div>
				<button
					onClick={ () => {
						closeModal();
						window.open( item.urls.raw,
'_blank' );
					} }
				>
					Open original image in new window
				</button>
			</div>
		);
	},
}
```

The code above would open a modal when the action is triggered over an image and provide a button to open the original image in a new window. This

is the same behavior defined in the previous version via `callback` but with a Modal window in between.

Let's make this Modal window nicer and more interesting by providing a Dropdown so users can select the size of the image to be opened in a new window.

First, import some components to be used in the modal window:

[src/App.js]

```
JavaScript
import {
	SelectControl,
	Button,
	__experimentalText as Text,
	__experimentalHStack as HStack,
	__experimentalVStack as VStack,
	...
} from '@wordpress/components';
```

<info callout>
The SelectControl, Button, Text, HStack and VStack are WordPress components available for WordPress development with React. You can find live examples of these components and others in the Gutenberg Storybook.
</info callout>

Next, use the following code to define the 'see-original' action:

[src/App.js]

```
JavaScript
{
	id: 'see-original',
	label: __( 'See Original' ),
	modalHeader: __( 'See Original Image', 'action label' ),
	RenderModal: ( { items: [ item ], closeModal } ) => {
		const [ size, setSize ] = useState( 'raw' );
```

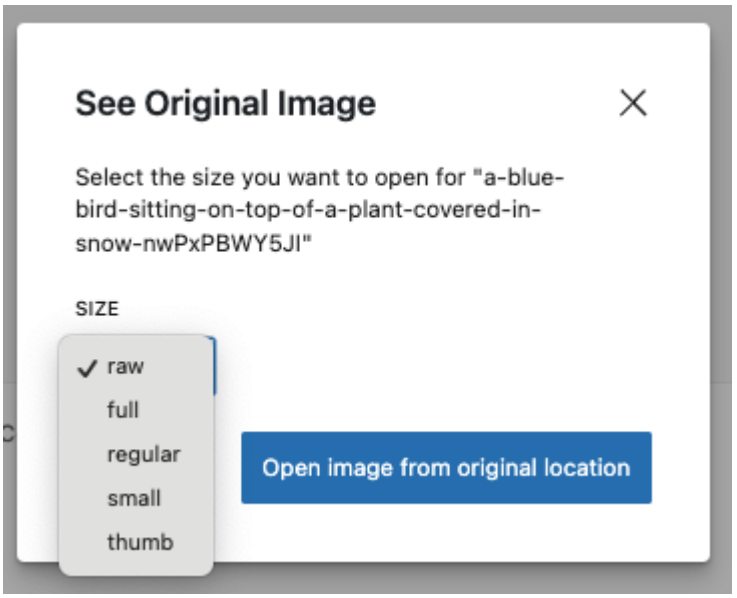
```
            return (
                  <VStack spacing="5">
                        <Text>
                              { `Select the size you want to open
for "${ item.slug }"` }
                        </Text>
                        <HStack justify="left">
                              <SelectControl
                                    __nextHasNoMarginBottom
                                    label="Size"
                                    value={ size }
                                    options={ Object.keys(
item.urls )
                                          .filter( ( url ) => url
!== 'small_s3' )
                                          .map( ( url ) => ( {
                                                label: url,
                                                value: url,
                                          } ) ) }
                                    onChange={ setSize }
                              />
                        </HStack>
                        <HStack justify="right">
                              <Button
                                    __next40pxDefaultSize
                                    variant="primary"
                                    onClick={ () => {
                                          closeModal();
                                          window.open( item.urls[
size ], '_blank' );
                                    } }
                              >
                                    Open image from original
location
                              </Button>
                        </HStack>
                  </VStack>
            );
      },
```

```
},
```

The code above uses the `SelectControl` component to display the sizes available for each image from the object with the info for each photo. The image size selected is stored in the `size` state variable through the `setSize` function.

The Button component's `onClick` calls the closeModal (available via props) and then opens the selected image (with the selected `size`) in a new window.

Now, when you click on the 'see-original' action on an image you should a modal window like the following one:



Full implementation and output

At this point, the final `src/App.js` file of the project should look like this:

[src/App.js]

```JavaScript
import { DataViews, filterSortAndPaginate } from
'@wordpress/dataviews';
import { getTopicsElementsFormat } from './utils';
import { useState, useMemo } from '@wordpress/element';
import {
	SelectControl,
	Button,
	__experimentalText as Text,
	__experimentalHStack as HStack,
	__experimentalVStack as VStack,
	Spinner,
	withNotices,
} from '@wordpress/components';

import { __ } from '@wordpress/i18n';
import apiFetch from '@wordpress/api-fetch';

import './style.scss';

// source "data" definition
import { dataPhotos } from './data';

// "defaultLayouts" definition
const primaryField = 'id';
const mediaField = 'img_src';

const defaultLayouts = {
	table: {
		layout: {
			primaryField,
		},
	},
	grid: {
		layout: {
			primaryField,
			mediaField,
		},
	},
```

```
};

// "fields" definition
const fields = [
	{
		id: 'img_src',
		label: __( 'Image' ),
		render: ( { item } ) => (
			<img alt={ item.alt_description } src={
item.urls.thumb } />
		),
		enableSorting: false,
	},
	{
		id: 'id',
		label: __( 'ID' ),
		enableGlobalSearch: true,
	},
	{
		id: 'author',
		label: __( 'Author' ),
		getValue: ( { item } ) =>
			`${ item.user.first_name } ${
item.user.last_name }`,
		render: ( { item } ) => (
			<a target="_blank" href={ item.user.url }
rel="noreferrer">
				{ item.user.first_name } {
item.user.last_name }
			</a>
		),
		enableGlobalSearch: true,
	},
	{
		id: 'alt_description',
		label: __( 'Description' ),
		enableGlobalSearch: true,
	},
	{
```

```
            id: 'topics',
            label: __( 'Topics' ),
            elements: getTopicsElementsFormat( dataPhotos ),
            render: ( { item } ) => {
                  return (
                        <div className="topic_photos">
                              { item.topics.map( ( topic ) => (
                                    <span key={ topic }
className="topic_photo_item">
                                          { topic.toUpperCase() }
                                    </span>
                              ) ) }
                        </div>
                  );
            },
            filterBy: {
                  operators: [ 'isAny', 'isNone', 'isAll',
'isNotAll' ],
            },
            enableSorting: false,
      },
      {
            id: 'width',
            label: __( 'Width' ),
            getValue: ( { item } ) => parseInt( item.width ),
            enableSorting: true,
      },
      {
            id: 'height',
            label: __( 'Height' ),
            getValue: ( { item } ) => parseInt( item.height ),
            enableSorting: true,
      },
];
const App = withNotices( ( { noticeOperations, noticeUI } ) =>
{
      const { createNotice } = noticeOperations;
```

```
	const [ isUploadingItems, setIsUploadingItems ] =
useState( [] );

	// "view" and "setView" definition
	const [ view, setView ] = useState( {
		type: 'table',
		perPage: 10,
		layout: defaultLayouts.table.layout,
		fields: [
			'img_src',
			'id',
			'alt_description',
			'author',
			'topics',
			'width',
			'height',
		],
	} );

	// "processedData" and "paginationInfo" definition
	const { data: processedData, paginationInfo } = useMemo(
() => {
		return filterSortAndPaginate( dataPhotos, view,
fields );
	}, [ view ] );

	const onSuccessMediaUpload = ( oImageUploaded ) => {
		const title = oImageUploaded.title.rendered;
		setIsUploadingItems( ( prevIsUploadingItems ) =>
			prevIsUploadingItems.filter(
				( slugLoading ) => slugLoading !== title
			)
		);

		createNotice( {
			status: 'success',
			// translators: %s is the image title
			content:
				`${ title }.jpg ` +
```

```
                    __( 'successfully uploaded to Media
Library' ),
                isDismissible: true,
            } );
      };

      const onErrorMediaUpload = ( error ) => {
            setIsUploadingItems( [] );
            console.log( error );
            createNotice( {
                  status: 'error',
                  content: __( 'An error occurred!' ),
                  isDismissible: true,
            } );
      };

      // "actions" definition
      const actions = [
            {
                  id: 'upload-media',
                  label: __( 'Upload Media' ),
                  isPrimary: true,
                  icon: 'upload',
                  supportsBulk: true,
                  callback: ( images ) => {
                        images.forEach( async ( image ) => {
                              // 1- Download the image and
convert it to a blob
                              const responseRequestImage = await
fetch( image.urls.raw );
                              const blobImage = await
responseRequestImage.blob();

                              // 2- Create FormData with the
image blob
                              const formDataWithImage = new
FormData();
                              formDataWithImage.append(
                                    'file',
```

```
                                        blobImage,
                                        `${ image.slug }.jpg`
                                   );

                                   // 3- Send the request to the WP
REST API with apiFetch
                                   await apiFetch( {
                                        path: '/wp/v2/media',
                                        method: 'POST',
                                        body: formDataWithImage,
                                   } ).then( console.log );
                              } );
                         },
                    },
                    {
                         id: 'see-original',
                         label: __( 'See Original' ),
                         modalHeader: __( 'See Original Image', 'action
label' ),
                         RenderModal: ( { items: [ item ], closeModal }
) => {
                              const [ size, setSize ] = useState( 'raw'
);
                              return (
                                   <VStack spacing="5">
                                        <Text>
                                             { `Select the size you
want to open for "${ item.slug }"` }
                                        </Text>
                                        <HStack justify="left">
                                             <SelectControl

__nextHasNoMarginBottom
                                                  label="Size"
                                                  value={ size }
                                                  options={

Object.keys( item.urls )

                                                       .filter( (

url ) => url !== 'small_s3' )
```

```
                                                  .map( ( url )
=> ( {
                                                       label:
url,
                                                       value:
url,
                                                  } ) ) }
                                             onChange={ setSize
}
                                        />
                                   </HStack>
                                   <HStack justify="right">
                                        <Button

__next40pxDefaultSize
                                             variant="primary"
                                             onClick={ () => {
                                                  closeModal();
                                                  window.open(
item.urls[ size ], '_blank' );
                                             } }
                                        >
                                             Open image from
original location
                                        </Button>
                                   </HStack>
                              </VStack>
                         );
                    },
               },
          ];
          return (
               <>
                    { !! isUploadingItems.length && <Spinner /> }
                    { noticeUI }
                    <DataViews
                         data={ processedData }
                         fields={ fields }
                         view={ view }
```

```
                        onChangeView={ setView }
                        defaultLayouts={ defaultLayouts }
                        actions={ actions }
                        paginationInfo={ paginationInfo }
                    />
                </>
        );
} );

export default App;
```

The DataViews project is now complete!

Go to your Admin panel and open the 'Add Media from Third Party Service' subpage under the 'Media' Settings. You should observe the following behavior:

- In table mode, each image has a primary 'Upload Media' action, displayed as an icon when hovered.
- Clicking the three-dot button on an image reveals two actions: 'Upload Media' and 'See Original.'
- You can select multiple images and perform the 'Upload Media' action on all of them simultaneously.
- Notification boxes appear once an image upload process is complete.
- A spinner icon is shown while images are being uploaded.
- The “See Original” action displays a Modal Windows allowing the user to choose the size of the image to be opened in a new window.

<info callout>
The full code of this project is available here. There’s also a live demo of the project powered by Playground.
</info callout>

Wrapping up

This article concludes a two-part series exploring the potential of the DataViews component:

- Using Data Views to display and interact with data in plugins covered the foundational concepts for getting started with Data Views.
- Actions from Data Views: Adding images to the Media Library (this article) got deeper into Data Views actions and the types of tasks that can be triggered through them.

If you're interested in following the progress of this feature, you can check the issues with "[Feature] Data Views" label in the gutenberg repo. This component also has biweekly updates, which are shared on https://make.wordpress.org/design/tag/dataviews/.

The @wordpress/dataviews package is a new tool that opens possibilities for plugin developers, and WordPress Core developers working on this feature would love to hear from you. Have you used it and found it interesting? Or ran into something you weren't able to do? Please share your thoughts in the comments or the Gutenberg repo as Issues.