Breaking down barriers using the DataTransfer API

- Transferring data
  - Transfer using copy and paste
- Custom data formats
- Connecting the web
- Concerns
  - Browser compatibility
  - Discoverability
  - Accessibility
  - Security and privacy
- Getting started with the Transmat helper library
- Acknowledgements

You might have heard about the DataTransfer API, which is part of the HTML5 Drag and Drop API and Clipboard events. It can be used to transfer data between source and receiving targets.

The drag-drop and copy-paste interactions are often used for interactions within a page to transfer simple text from A to B. But what is oftentimes overlooked is the ability to use these same interactions to go beyond the browser window.

Both the browser's built-in drag-and-drop and the copy-paste interactions can communicate with other applications, web or otherwise, and are not tied to any origin. The API supports multiple data entries with different behaviors based on where data is transferred to. Your web application can send and receive the transferred data when listening to incoming events.

This capability can change the way we think about sharing and interoperability in web applications on desktop. Transferring data between applications doesn't need to rely on tightly coupled integrations anymore. Instead you can give users full control to transfer data to wherever they like.

https://www.youtube.com/watch?v=EYMgUhn_Zdo

An example of interactions that are possible with the DataTransfer API. (Video does not include sound.)

Transferring data

To get started, you'll need to implement drag-drop or copy-paste. The examples below show drag-drop interactions, but the process for copy-paste is similar. If you are unfamiliar with the Drag and Drop API, there's a great article explaining HTML5 Drag and Drop, which explains the ins and outs.

By providing MIME-type keyed data, you are able to freely interact with external applications. Most WYSIWYG editors, text editors, and browsers respond to the "primitive" mime-types used in the example below.

```
document.querySelector('#dragSource')
.addEventListener('dragstart', (event) => {
  event.dataTransfer.setData('text/plain', 'Foo bar');
  event.dataTransfer.setData('text/html', '<h1>Foo bar</h1>');
  event.dataTransfer.setData('text/uri-list', 'https://example.com');
});
```

Notice the `event.dataTransfer` property. This returns an instance of `DataTransfer`. As you'll see, this object is sometimes returned by properties with other names.

Receiving the data transfer works almost the same as providing it. Listen to the receiving events (`drop`, or `paste`) and read the keys. When dragging over an element, the browser only has access to the `type` keys of the data. The data itself can only be accessed after a drop.

```
document.querySelector('#dropTarget')
.addEventListener('dragover', (event) => {
  console.log(event.dataTransfer.types);
  // Without this, the drop event won't fire.
  event.preventDefault();
});

document.querySelector('#dropTarget')
.addEventListener('drop', (event) => {
  // Log all the transferred data items to the console.
  for (let type of event.dataTransfer.types) {
```

```
    console.log({ type, data: event.dataTransfer.getData(type) });
  }
  event.preventDefault();
});
```

Three MIME-types are widely supported across applications:

- **`text/html`:** Renders the HTML payload in `contentEditable` elements and rich text (WYSIWYG) editors like Google Docs, Microsoft Word, and others.
- **`text/plain:`** Sets the value of input elements, content of code editors, and the fallback from `text/html`.
- **`text/uri-list`:** Navigates to the URL when dropping on the URL bar or browser page. A URL shortcut will be created when dropping on a directory or the desktop.

The widespread adoption of `text/html` by WYSIWYG editors makes it very useful. As in HTML documents, you can embed resources by using Data URLs or publicly accessible URLs. This works well with exporting visuals (for example from a canvas) to editors like Google Docs.

```
const redPixel = '';
const html = '<img src="' + redPixel + '" width="100" height="100" alt="" />';
event.dataTransfer.setData('text/html', html);
```

Transfer using copy and paste

Using the DataTransfer API with copy-paste interactions is shown below. Notice that the `DataTransfer` object is returned by a property called `clipboardData` for clipboard events.

```
// Listen to copy-paste events on the document.
document.addEventListener('copy', (event) => {
  const copySource = document.querySelector('#copySource');
  // Only copy when the `activeElement` (i.e., focused element) is,
  // or is within, the `copySource` element.
  if (copySource.contains(document.activeElement)) {
    event.clipboardData.setData('text/plain', 'Foo bar');
    event.preventDefault();
  }
});

document.addEventListener('paste', (event) => {
  const pasteTarget = document.querySelector('#pasteTarget');
```

```
  if (pasteTarget.contains(document.activeElement)) {
    const data = event.clipboardData.getData('text/plain');
    console.log(data);
  }
});
```

Custom data formats

You are not limited to the primitive MIME types, but can use any key to identify the transferred data. This can be useful for cross-browser interactions within your application. As shown below, you can transfer more complex data using the `JSON.stringify()` and `JSON.parse()` functions.

```
document.querySelector('#dragSource')
.addEventListener('dragstart', (event) => {
  const data = { foo: 'bar' };
  event.dataTransfer.setData('my-custom-type', JSON.stringify(data));
});

document.querySelector('#dropTarget')
.addEventListener('dragover', (event) => {
  // Only allow dropping when our custom data is available.
  if (event.dataTransfer.types.includes('my-custom-type')) {
    event.preventDefault();
  }
});

document.querySelector('#dropTarget')
.addEventListener('drop', (event) => {
  if (event.dataTransfer.types.includes('my-custom-type')) {
    event.preventDefault();
    const dataString = event.dataTransfer.getData('my-custom-type');
    const data = JSON.parse(dataString);
    console.log(data);
  }
});
```

Connecting the web

While custom formats are great for communication between applications you have in your control, it also limits the user when transferring data to applications that aren't using your format. If you want to connect with third-party applications across the web, you need a universal data format.

The JSON-LD (Linked Data) standard is a great candidate for this. It is lightweight and easy to read from and write to in JavaScript. Schema.org contains many predefined types that can be used, and custom schema definitions are an option as well.

```
const data = {
  '@context': 'https://schema.org',
  '@type': 'ImageObject',
  contentLocation: 'Venice, Italy',
  contentUrl: 'venice.jpg',
  datePublished: '2010-08-08',
  description: 'I took this picture during our honey moon.',
  name: 'Canal in Venice',
};
event.dataTransfer.setData('application/ld+json', JSON.stringify(data));
```

When using the Schema.org types, you can start with the generic Thing type, or use something closer to your use case like Event, Person, MediaObject, Place, or even highly-specific types like MedicalEntity if need be. When you use TypeScript, you can use the interface definitions from the schema-dts type definitions.

By transmitting and receiving JSON-LD data, you will support a more connected and open web. With applications speaking the same language, you can create deep integrations with external applications. There's no need for complicated API integrations; all the information that's needed is included in the transferred data.

Think of all the possibilities for transferring data between any (web) application with no restrictions: sharing events from a calendar to your favorite ToDo app, attaching virtual files to emails, sharing contacts. That would be great, right? This starts with you! □□

Concerns

While the DataTransfer API is available today, there are some things to be aware of before integrating.

Browser compatibility

Desktop browsers all have great support for the technique described above, while mobile devices do not. The technique has been tested on all major browsers (Chrome, Edge, Firefox, Safari) and operating systems (Android, ChromeOS, iOS, macOS, Ubuntu Linux, and Windows), but unfortunately Android and iOS didn't pass the test. While browsers continue to develop, for now the technique is limited to desktop browsers only.

Discoverability

Drag-drop and copy-paste are system-level interactions when working on a desktop computer, with roots back to the first GUIs more than 40 years ago. Think about how many times you have used these interactions for organizing files. This isn't yet very common on the web.

You will need to educate users about this new interaction, and come up with UX patterns to make this recognizable, especially for people whose experience with computers so far has been confined to mobile devices.

Accessibility

Drag-drop is not a very accessible interaction, but the DataTransfer API works with copy-paste, too. Make sure you listen to copy-paste events. It doesn't take much extra work, and your users will be grateful to you for adding it.

Security and privacy

There are some security and privacy considerations you should be aware of when using the technique.

- Clipboard data is available to other applications on the user's device.
- Web applications you are dragging over have access to the type keys, not the data. The data only becomes available on drop or paste.
- The received data should be treated like any other user input; sanitize and validate before using.

Getting started with the Transmat helper library

Are you excited about using the DataTransfer API in your application? Consider taking a look at the Transmat library on GitHub. This open-source library aligns browser differences, provides JSON-LD

utilities, contains an observer to respond to transfer events for highlighting drop-areas, and lets you integrate the data transfer operations among existing drag and drop implementations.

```
import { Transmat, TransmatObserver, addListeners } from 'transmat';

// Send data on drag/copy.
addListeners(myElement, 'transmit', (event) => {
  const transmat = new Transmat(event);
  transmat.setData({
    'text/plain': 'Foobar',
    'application/json': { foo: 'bar' },
  });
});

// Receive data on drop/paste.
addListeners(myElement, 'receive', (event) => {
  const transmat = new Transmat(event);
  if (transmat.hasType('application/json') && transmat.accept()) {
    const data = JSON.parse(transmat.getData('application/json'));
  }
});

// Observe transfer events and highlight drop areas.
const obs = new TransmatObserver((entries) => {
  for (const entry of entries) {
    const transmat = new Transmat(entry.event);
    if (transmat.hasMimeType('application/json')) {
      entry.target.classList.toggle('drag-over', entry.isTarget);
      entry.target.classList.toggle('drag-active', entry.isActive);
    }
  }
});
obs.observe(myElement);
```

Acknowledgements

Hero image by Luba Ertel on Unsplash.

□□□ #3
□ □□□ □□□ 2026-10-31 21:48:45 UTC

 cocobook-sync □□□ 2026-11-04 17:37:00 UTC