Drag and Drop Form Builder

Data Verity #2
Erikka Baker
James Miller
Jordan Schmerge

June 21, 2016

Table of Contents

Introduction

Our client, Data Verity, uses their website to create custom tools and data visualization for business applications. This includes sales tracking, predictive sales analytics, and quality management. They mainly serve financial institutions, such as banks and credit unions. Because customization is such a large part of what Data Verity does, they needed an easier way to build these custom tools for their clients.

To meet this need, the client asked us to extend the functionality of their development suite by adding a drag and drop form builder. This form builder will eliminate the need for developers to hard code forms using HTML directly. It will also make the form creation process more efficient and much less error prone, because once components are created once they can be reused over and over.

This form builder also needed to be easily extendable so that they can quickly and easily add new form components to the available pool. Also, after creation, the components themselves need to be customizable so they can be adapted to whatever form a developer needs to build. Building these custom forms is a crucial aspect of what Data Verity does and this form builder will help streamline that process.

Figure 1 below shows a screenshot of part of the final product, and what it looks like when components are being dragged from the pool and onto the screen.

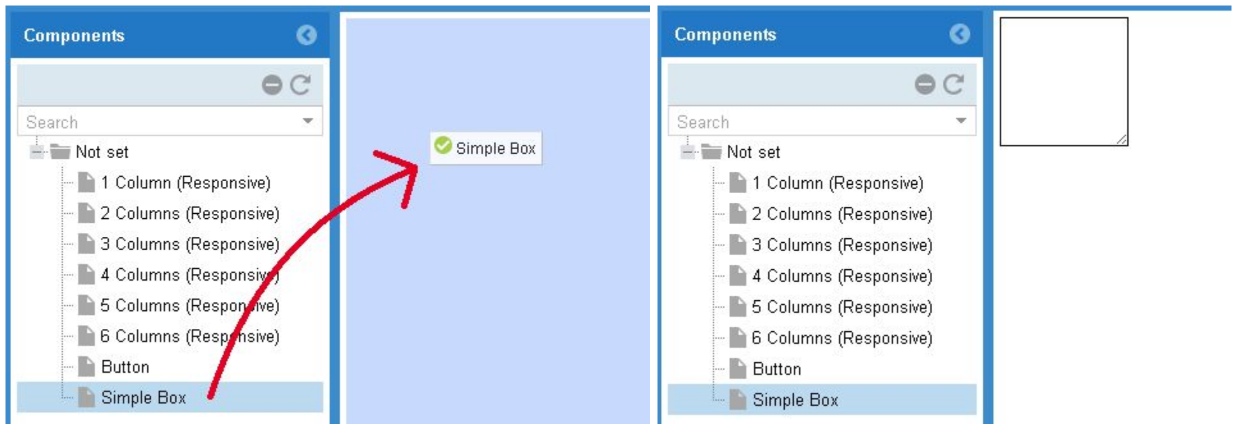

Figure 1: Before and After Dragging

Requirements

Functional Requirements

- A list of available components should be shown to the user in a panel on the side of the screen. These should be loaded from the client's database
- The builder should include listeners that allow the user to drop components within each other so they can become nested in the XML
- When a new component is created, the user will have to give the following specifications:
    - HTML representing what the component looks like
    - HTML representing what will be displayed as a component preview on the left side of the form builder window
    - Boolean representing whether or not other components can be nested within this component
    - Functions that represent how each component can be customized

Non-Functional Requirements

- The form builder should be created using Ext JS, the JavaScript framework that the rest of the client's site was created with
- The form builder should extend the Ext.panel.Panel prototype of Ext JS
- Code should be reasonably fast and responsive
- Any global data or functions should be namespaced to avoid conflicts with other apps within the client's system

System Architecture

Our final system will consist of two main parts: our form builder and the client's database. These two parts will interact with each other by communicating information about forms and form components. Our form builder will require information from the database to load forms and form components, while the database will require information from our app to save all that data once it has been edited. Figure 2 below summarizes this relationship.

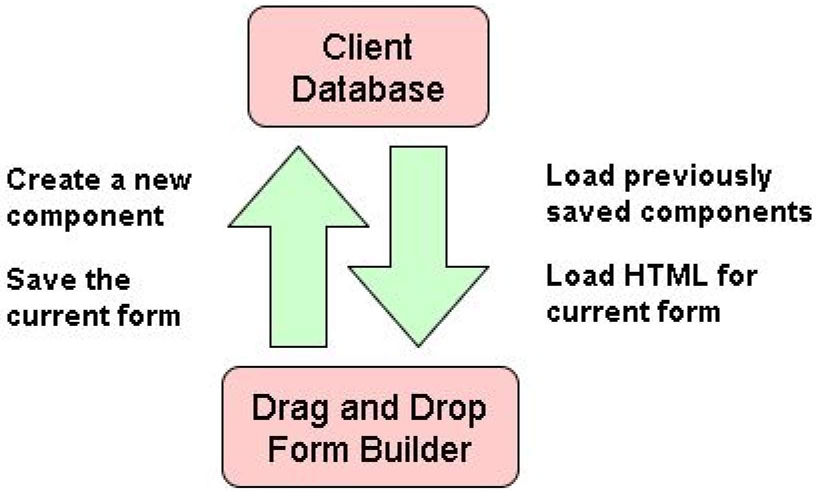
Client Database
Create a new component
Save the current form
Load previously saved components
Load HTML for current form
Drag and Drop Form Builder

Figure 2: Basic System Architecture

There are a few ways that our app can send information to the database. While new components can be inserted into the database directly, we have also included a form in our app that allows the user to create a new component from there, without having to open the database tools. We have included a “Create Component” button that allows the user to create a new component from scratch, as well as a “Save as New Component” button that allows the user to grab the current page contents to save as a new component. This way new components can be created without having to rewrite a lot of HTML code.

As for loading data from the database, this is achieved by passing in a config object to our form builder so that when a new form builder window is opened, the programmer can specify which table in the database stores all the components. Our app uses this to load all the components into a tree, and saves the connection information for later so it can be reloaded when the user presses “Refresh” on the tree.

Figure 3 below shows a more detailed description of all the connections between our app and the client’s database.

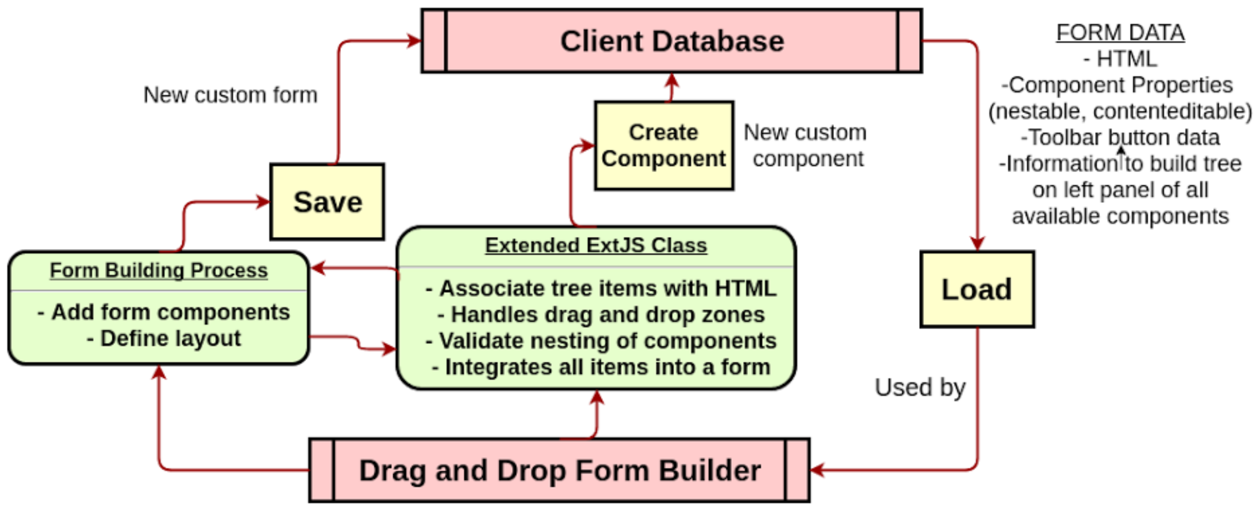

Figure 3: Detailed System Architecture

Technical Design

Component Highlighting

One feature that we added to our app is a highlighting effect, allowing users to see what element on the page they are currently dragging over. Because there can potentially be many overlapping areas on the page, we thought this would be a very useful feature.

To decide what parts of the page to highlight, we added *onmouseenter* and *onmouseout* events to all the HTML elements on the page. In the handler for *onmouseenter* we highlight the element the cursor is over, but only if the user is currently dragging something (which we determine by checking if the left mouse button is pressed). Also, before highlighting the target, we save the original background color so that it can be restored when the *onmouseout* event fires.

The flowchart in Figure 4 below shows the events we registered and what their handlers do.

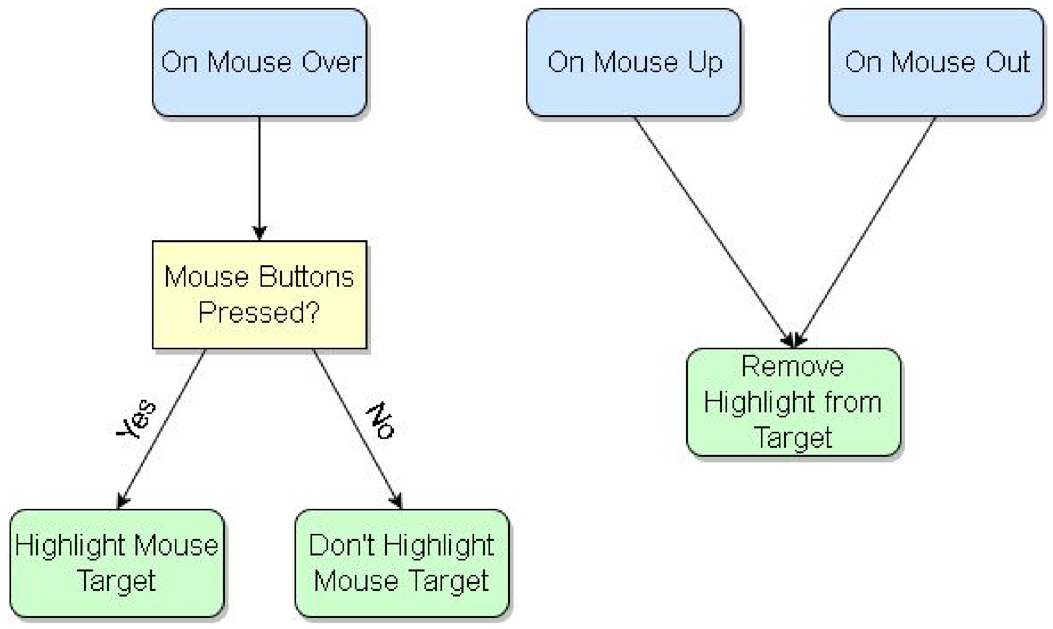

Figure 4: Highlighting Process

It was somewhat difficult to get this working properly, because not all HTML mouse events can be fired while a drag event is taking place. To solve this, we had to handle a couple cases individually, by calling our handler functions with manually constructed events. These events were simply objects with a *target* and *stopPropagation* property, since these were the only event attributes that our handlers needed to use.

Custom Tooltip Buttons

Another feature we included in our app is a tooltip that can be displayed for each component on the page when you right click on it. This will display some buttons underneath the component that allow you to delete it or edit its HTML directly. Due to the fact that each component might require different customization options, we decided to allow users to define some of the buttons themselves.

These buttons are defined by the user when they fill out the form to create a new component. They are given a field where they can list out any functions (in the form of a JSON object) that they may want to apply to their component later. This way the user can create a DIV component with a customizable background color, or a button with customizable padding, etc.

For this to work, the correct button data needs to be loaded from the database every time a tooltip is displayed. To know for which component the button data should be loaded, we store the component's ID in its wrapper DIV so that it can be retrieved from the right click event whenever the user wants to display the tooltip. We can then use this ID to identify the component in the database and load its button data.

In order to access the information from the database, we also added an *onTreeBuilt* event to our component holder, so that every time the component list is refreshed, the tree data is saved as a global variable that can be accessed by our tooltip function. This allows us to loop through all the tree nodes until the node with the correct ID is found.

Figure 5 below shows a summary of this process.

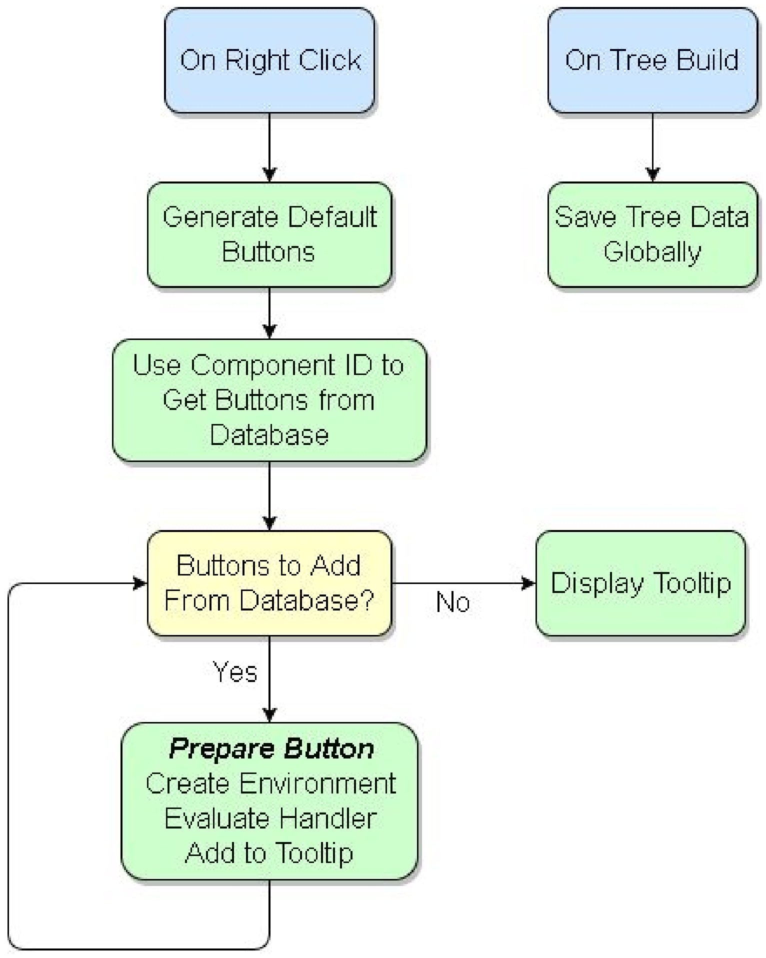

Figure 5: Custom Button Creation

Storing Form Builder Data

One complication that we encountered while working on our app was storing data for each form builder. Because it is possible that many form builder windows are open at one time, we needed to ensure that each window had its own unique data so that the windows would not interfere with each other. This could potentially happen, for example, when data is saved in the history array. Each window needs its own unique history array so that the Undo/Redo buttons apply to only one open window.

Normally, this could be solved by storing all the data within the form builder window object itself, but because our app contains many HTML events that need to access global Javascript functions, this would not work for us. The window data needed to be global as well.

To handle this, we created a global array that contains the data for all currently open form builder windows. We also assigned each window a unique ID that is used as its index into the data array. To prevent this array from getting too large when the site is running for a long time, we clear the data for a window when it is destroyed by setting its data to null. Then, when a new window is opened, its ID is generated by finding the first null position in the data array.

Figure 6 below shows a summary of all the global data we needed to store as well as a visualization of the data array.

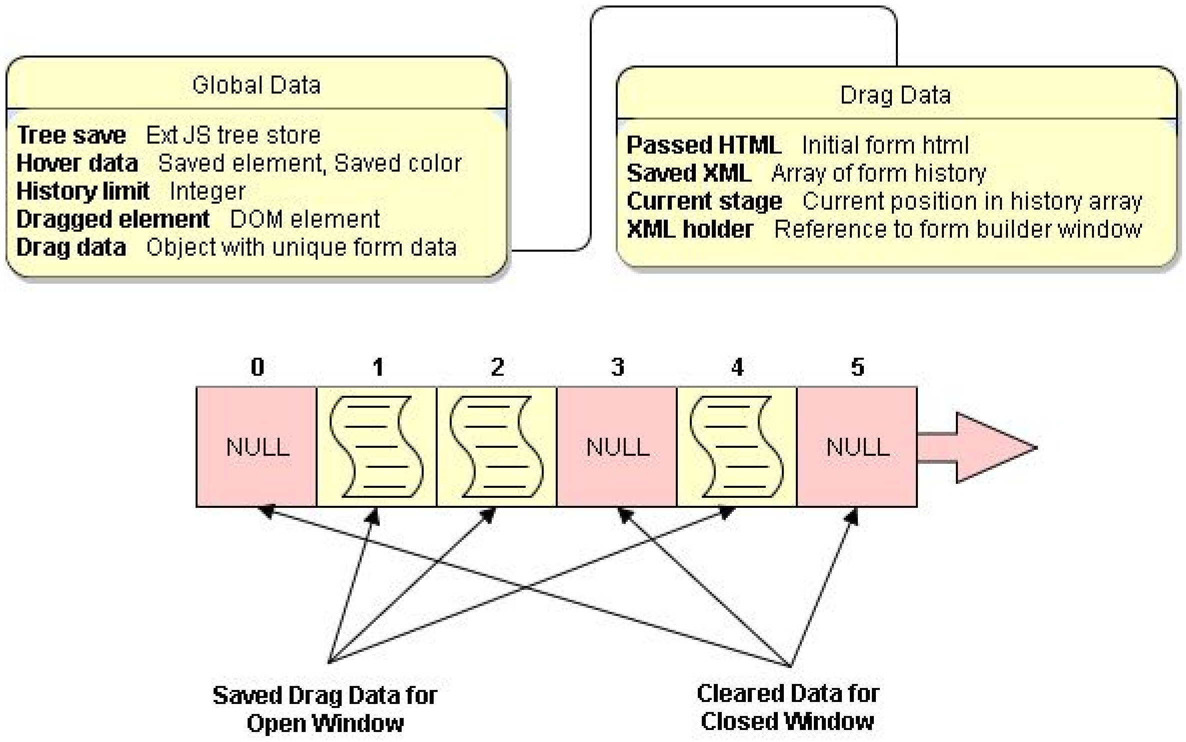
Global Data
Tree save Ext JS tree store
Hover data Saved element, Saved color
History limit Integer
Dragged element DOM element
Drag data Object with unique form data
Drag Data
Passed HTML Initial form html
Saved XML Array of form history
Current stage Current position in history array
XML holder Reference to form builder window
0
1
2
3
4
5
NULL
NULL
NULL
Saved Drag Data for Open Window
Cleared Data for Closed Window

Figure 6: Structure of Global Data

Design Decisions

Drag and Drop: Ext JS vs. HTML

To implement drag and drop in our app, we had to choose between using the drop listeners that are built into Ext JS, and the HTML properties *ondragstart* and *ondrop* to implement the functionality ourselves. We decided it would be best to use Ext JS when adding a new component to the drop region, because this allows us to store data for each component in an Ext JS object, including whether or not the object is nestable. However, once the component is added to the drop region, it is given the HTML properties instead. This allowed us to manipulate the DOM directly, providing more customization options.

Making Components Draggable vs. Using a Wrapper DIV

In our XML, we had to choose between making the components themselves draggable, and wrapping them in a DIV that has all the draggable properties. Using the second option adds a lot of unnecessary DIV components in our final XML, but it also allows us to easily add and remove the draggable properties as we import and export XML in our app. Because we can give all the wrapper DIVs their own class, it is very simple to locate and modify all of them at once. We also decided that this way would be better because it allows us to have more control over how groups of components can be nested.

Controlling Editable Components

As part of the requirements of our app, some components need to support editable text using the HTML *contenteditable* attribute. This means we need some text to be editable within our app, but not in the XML that our app is exporting. To handle this, we decided to create our own HTML attribute, called *dragproperties_editable*, that will automatically be replaced with *contenteditable* within our editor. The downside of doing this is that the final XML will contain some nonsense attributes, but as long as we are careful to make this attribute unique, it should not cause any conflicts.

Saving Data for Undo / Redo

We added an undo and redo button to our app so that the user can fix accidental drops in the XML. To do this, every time the user drops a component, we take the entire XML that they are currently working on and save it as a text string in an array. This can be traversed using the undo and redo buttons, but the downside of this method is that the array may become very large. Because our app is not likely to be used to create large or complicated XML documents, it is unlikely this will become an issue. Just in case it does, we've added a limit to the length of the history array.

Results

Unplanned Features

There were quite a few features that we did not originally plan on adding, but throughout development decided would be very useful. One example is the addition of an organized tree display of all the components, which is a lot cleaner than our original plan. Initially, we envisioned a sidebar that would have a small HTML preview of each element.

Another example is the Undo/Redo/Reset feature. This was not part of the initial requirements, but it was easy enough to add because it only involved keeping track of an array representing the history of the XML being created.

Future Work

In our app, it is currently only possible to add HTML to a page by nesting it within another element, or appending it. We would like to add more sophisticated drop listeners so that elements can be dragged in between each other, to allow easier rearrangement.

Something else we'd like to add is Bootstrap integration. We had trouble using Twitter Bootstrap because it interfered with some of the Ext JS styling that our client's site already used. Because Bootstrap would allow our client to create responsive components, this is something that we would like to explore further. For the time being, we have made some responsive components of our own for the client to use. They are not as advanced as everything that Bootstrap allows you to do, but it provided a useful alternative.

Summary of Testing

Testing our app was easy because we had regular meetings with our client. We were able to show him our work as it progressed, and he was able to tell us what things we should change and what he liked. This was nice because it allowed us to catch problems early and avoid miscommunication.

Also, to make sure that our app worked with a variety of HTML elements, we made lots of test components during development such as layout grids and resizable boxes. We tried to make test components that were similar to the components our client would actually create, to avoid any compatibility issues once the code is handed back to our client.

Lessons Learned

We got a somewhat slow start to our project, because we had to spend lots of time during the first two weeks learning how to use Ext JS. One mistake we made is that we tried to learn this new framework as we worked on our project, only looking up functionality as we needed it. We probably would have been able to produce cleaner code if we had learned Ext JS more thoroughly before starting to code.

We also did not do much planning regarding the structure of our code. The basic concept of our app was not too complicated, but we did have to do a little refactoring during week 4 once our code was starting to get unorganized.

Final Product

Overall, our project turned out to be a success. We were able to achieve most of the initial requirements we were given and, with all the extra features we added along the way, the software ended up being fairly simple and easy to use. In the end, we were able to create a form builder that Data Verity will be able to integrate into many parts of their website.