Talking to the Server Using The afterMove Event

The Reorderer enables users to directly move around and re-arrange content on a page. Application developers can easily integrate the Reorderer into their application, connecting it to their server in a variety of ways. The `afterMove` event is used for this purpose. For more information about how to listen for a Fluid event, see the Events for Component Users page.

Whenever an item has been moved by the user, your `afterMove` listener is invoked and provided with three arguments:

1. The item that was just moved by the user, as a `jQuery` instance
2. Information about the new position of the item, as a `position` object
3. A list of all the orderable elements, in the new order, as a `jQuery` instance

The `position` argument is a mini-object that provides specific information about the item's destination. It looks like this:

Field	Type	Description
`element`	DOM element	The item the dragged item was dropped next to (i.e. the drop target)
`position`	integer	A constant representing the relation of the drop position to the drop target - one of the constants `fluid.position.BEFORE`, `fluid.position.AFTER`, `fluid.position.INSIDE`, or `fluid.position.REPLACE`

On This Page

Still need help?

Join the infusion-users mailing list and ask your questions there.

Communicating the Order Back to the Server

The Reorderer is a markup-driven component. It manages the interaction of dragging and dropping, ensuring that each element is positioned correctly in the DOM. To communicate this order back to the server, you'll probably want to process these DOM elements in some way. Here are a few of the scenarios you might consider for saving the order back to the server:

1. Sending a JSON-formatted array of items back to the server
2. Using hidden form fields to store the order of items, POSTing the form back to the server when changes are made

The JSON-based approach is simple to implement, while the form-based approach is more amenable to supporting Progressive Enhancement for users who don't have JavaScript enabled.

The JSON Approach

The JSON-based approach involves sending the item order back to the server as in the JSON (JavaScript Object Notation) format, a common data exchange format used in Ajax-driven applications.

To start, we need to provide some kind of unique identifier for each DOM element that the server will recognize. You'll probably render this out using whatever server-side presentation framework you typically use for creating markup. For example, your HTML might look like this:

```
<ol id="fruits">
  <li class="flc-reorderer-movable" id="apple">Apple</li>
  <li class="flc-reorderer-movable" id="cherry">Cherry</li>
  <li class="flc-reorderer-movable" id="kiwi">Kiwi</li>
</ol>
```

Any time an item is moved, the `afterMove` event will fire and your listener will be invoked. If you want to save the new order to the server immediately, your `afterMove` listener will need to do the following:

1. Walk through the list of orderables (the third argument passed by the Reorderer your listener)
2. Create an array containing each unique ID

3. POST the ordered array back using Ajax.

In this example, the `sendOrderToServer` function is responsible for sending the data back the server using jQuery's ajax() function. When the application on the server receives the data, it can store the new information in whatever way it sees fit. Here's what that looks like:

```
var sendOrderToServer = function (theFruitThatMoved, position, allFruits) {
    // Loop through each item in the ordered list and update its hidden form field.
    var fruitIDs = fluid.transform(allFruits, function (fruit) {
        return $(fruit).attr("id");
    });


    // Send the item order array back to the server via an AJAX POST request.
    $.ajax({
        type: "POST",
        dataType: "json",
        url: "http://myserver.org/fruit/order",
        data: fruitIDs,
        success: function (data, ajaxStatus) {
            // Do something upon success.
        },
        error: function () {
            // Do something else if an error occurs.
        }
    });
};
```

Weaving this all together, here's how you'd instantiate your Reorderer and register your `afterMove` event listener:

```
fluid.reorderer("#container", {
    listeners: {
       afterMove: sendOrderToServer
    }
});
```

The Form-Based Approach

A form submission can be used to send the updated order to the server. One way to accomplish this might be to associate a hidden `<input>` element with each orderable item - and to nest the element inside the markup for the item so that it is carried along with it when reordered. Give each form field a unique `name` attribute that identifies the item to the server, and store the order of item in the `value` attribute. For example:

```
<form id="fruitOrder" action="...">

  <div id="container">
    <div  class="flc-reorderer-movable">
      Apple
      <input name="apple" value="0" type="hidden"/>
    </div>

    <div class="flc-reorderer-movable">
      Cherry
      <input name="cherry" value="1" type="hidden"/>
    </div>

    <div class="flc-reorderer-movable">
      Kiwi
      <input name="kiwi" value="2" type="hidden"/>
    </div>
  </div>
</form>
```

This technique is particularly effective if you want to make your application also work without JavaScript enabled. Include a submit button and hide the form elements with JavaScript. Users who don't have JavaScript enabled can still use the standard form controls to change the item order and submit it to the server in the old fashioned way.

As mentioned previously, the `afterMove` event is fired each time the user moves an item. In the form-based approach, your listener will need to update the `value` of each hidden input with its new index, and then submit the form. Here's what that looks like:

```
var afterMoveListener = function (theFruitThatMoved, position, allFruits) {
    // Loop through each item in the ordered list and update its hidden form field.
    allFruits.each(function (idx, fruit) {
        $(fruit).children("input").val(idx);
    });

    // POST it back to the server.
    postOrder();
};
```

In this example, the `postOrder` function is responsible for sending the form back to the server using AJAX. Assuming your form has an `id` of `fruitOrder` , here's how it is implemented:

```
// Serialize the form and post it back to the server.
var postOrder = function () {
    var form = $("#fruitOrder"); // Get the form out of the DOM
    var fruitOrderRequest = $(form).serialize(); // Use jQuery to serialize it into a standard form request.

    // Send it back to the server via an AJAX POST request.
    $.ajax({
        type: "POST",
        url: form.action,
        data: fruitOrderRequest,
        complete: function (data, ajaxStatus) {
            // Handle success or failure by being nice to the user.
        }
    });
};
```

Once again, instantiating your Reorderer and registering your event listener can be done all at once:

```
fluid.reorderer("#container", {
    listeners: {
       afterMove: afterMoveListener
    }
});
```