Building Inputs

Shiny comes equipped with a variety of useful input components, but as you build more ambitious applications, you may find yourself needing input widgets that we don't include. Fortunately, Shiny is designed to let you create your own custom input components. If you can implement it using HTML, CSS, and JavaScript, you can use it as a Shiny input!

(If you're only familiar with R and not with HTML/CSS/JavaScript, then you will likely find it tough to create all but the simplest custom input components on your own. However, other people can – and hopefully will – bundle up their custom Shiny input components as R packages and make them available to the rest of the community.)

Design the Component

The first steps in creating a custom input component is no different than in any other form of web development. You write HTML markup that lays out the component, CSS rules to style it, and use JavaScript (mostly event handlers) to give it behavior, if necessary.

Shiny input components should try to adhere to the following principles, if possible:

- **Designed to be used from HTML and R:** Shiny user interfaces can either be written using R code (that generates HTML), or by writing the HTML directly. A well-designed Shiny input component will take both styles into account: offer an R function for creating the component, but also have thoughtfully designed and documented HTML markup.
- **Configurable using HTML attributes:** Avoid requiring the user to make JavaScript calls to configure the component. Instead, it's better to use HTML attributes. In your component's JavaScript logic, you can easily access these values using jQuery (or simply by reading the DOM attribute directly).

When used in a Shiny application, your component's HTML markup will be repeated once for each instance of the component on the page, but the CSS and JavaScript will generally only need to appear once, most likely in the `<head>`. For R-based interface code, you can use the functions `singleton` and `tags$head` together to ensure these tags appear once and only once, in the head. (See the full example below.)

Write an Input Binding

Each custom input component also needs an *input binding*, an object you create that tells Shiny how to identify instances of your component and how to interact with them. (Note that each *instance* of the input component doesn't need its own input binding object; rather, all instances of a particular type of input component share a single input binding object.)

An input binding object needs to have the following methods:

`find(scope)`

Given an HTML document or element (`scope`), find any descendant elements that are an instance of your component and return them as an array (or array-like object). The other input binding methods all take an `el` argument; that value will always be an element that was returned from `find`.

A very common implementation is to use jQuery's `find` method to identify elements with a specific class, for example:

```
exampleInputBinding.find = function(scope) {
  return $(scope).find(".exampleComponentClass");
};
```

`getId(el)`

Return the Shiny input ID for the element `el`, or `null` if the element doesn't have an ID and should therefore be ignored. The default implementation in `Shiny.InputBinding` reads the `data-input-id` attribute and falls back to the element's `id` if not present.

`getValue(el)`

Return the Shiny value for the element `el`. This can be any JSON-compatible value.

`setValue(el, value)`

Set the element to the specified value. (This is not currently used, but in the future we anticipate adding features that will require the server to push input values to the client.)

`subscribe(el, callback)`

Subscribe to DOM events on the element `el` that indicate the value has changed. When the DOM events fire, call `callback` (a function) which will tell Shiny to retrieve the value.

We recommend using jQuery's event namespacing feature when subscribing, as unsubscribing becomes very easy (see `unsubscribe`, below). In this example, `exampleComponentName` is used as a namespace:

```
exampleInputBinding.subscribe = function(el, callback) {
  $(el).on("keyup.exampleComponentName", function(event) {
    callback(true);
  });
  $(el).on("change.exampleComponentName", function(event) {
    callback();
  });
};
```

Later on, we can unsubscribe `".exampleComponentName"` which will remove all of our handlers without touching anyone else's.

The `callback` function optionally takes an argument: a boolean value that indicates whether the component's rate policy should apply (`true` means the rate policy should apply). See `getRatePolicy` below for more details.

`unsubscribe(el)`

Unsubscribe DOM event listeners that were bound in `subscribe`.

Example:

```
exampleInputBinding.unsubscribe = function(el) {
  $(el).off(".exampleComponentName");
};
```

`getRatePolicy()`

Return an object that describes the rate policy of this component (or `null` for default).

Rate policies are helpful for slowing down the rate at which input events get sent to the server. For example, as the user drags a slider from value A to value B, dozens of change events may occur. It would be wasteful to send all of those events to the server, where each event would potentially cause expensive computations to occur.

A rate policy slows down the rate of events using one of two algorithms (so far). **Throttling** means no more than one event will be sent per X milliseconds. **Debouncing** means all of the events will be ignored until no events have been received for X milliseconds, at which time the most recent event will be sent. This blog post goes into more detail about the difference between throttle and debounce.

A rate policy object has two members:

- `policy` - Valid values are the strings `"direct"`, `"debounce"`, and `"throttle"`. `"direct"` means that all events are sent immediately.
- `delay` - Number indicating the number of milliseconds that should be used when debouncing or throttling. Has no effect if the policy is `direct`.

Rate policies are only applied when the `callback` function in `subscribe` is called with `true` as the first parameter. It's important that input components be able to control which events are rate-limited and which are not, as different events may have different expectations to the user. For example, for a textbox, it would make sense to rate-limit events while the user is typing, but if the user hits Enter or focus leaves the textbox, then the input should always be sent immediately.

Register Input Binding

Once you've created an input binding object, you need to tell Shiny to use it:

```
Shiny.inputBindings.register(exampleInputBinding, "yourname.exampleInputBinding");
```

The second argument is a name the user can use to change the priority of the binding. On the off chance that the user has multiple bindings that all want to claim the same HTML element as their own, this call can be used to control the priority of the bindings:

```
Shiny.inputBindings.setPriority("yourname.exampleInputBinding", 10);
```

Higher numbers indicate a higher priority; the default priority is 0. All of Shiny's built-in input component bindings default to a priority of 0.

If two bindings have the same priority value, then the more recently registered binding has the higher priority.

Example

For this example, we'll create a button that displays a number, whose value increases by one each time the button is clicked. Here's what the end result will look like:

0

To start, let's design the HTML markup for this component:

```
<button id="inputId" class="increment btn" type="button">0</button>
```

The CSS class `increment` is what will differentiate our buttons from any other kind of buttons. (The `btn` class is just to make the button look decent in Twitter Bootstrap.)

Now we'll write the JavaScript that drives the button's basic behavior:

```
$(document).on("click", "button.increment", function(evt) {

  // evt.target is the button that was clicked
  var el = $(evt.target);

  // Set the button's text to its current value plus 1
  el.text(parseInt(el.text()) + 1);

  // Raise an event to signal that the value changed
  el.trigger("change");
});
```

This code uses jQuery's delegated events feature to bind all increment buttons at once.

Now we'll create the Shiny binding object for our component, and register it:

```
var incrementBinding = new Shiny.InputBinding();
$.extend(incrementBinding, {
  find: function(scope) {
    return $(scope).find(".increment");
  },
  getValue: function(el) {
    return parseInt($(el).text());
  },
  setValue: function(el, value) {
    $(el).text(value);
  },
  subscribe: function(el, callback) {
    $(el).on("change.incrementBinding", function(e) {
      callback();
    });
  },
  unsubscribe: function(el) {
    $(el).off(".incrementBinding");
  }
});

Shiny.inputBindings.register(incrementBinding);
```

Both the behavioral JavaScript code and the Shiny binding code should generally be run when the page loads. (It's important that they run before Shiny initialization, which occurs after all the document ready event handlers are executed.)

The cleanest way to do this is to put both chunks of JavaScript into a file. In this case, we'll use the path `./www/js/increment.js`, which we can then access as `http://localhost:8100/js/increment.js`.

If you're using an `index.html` style user interface, you'll just need to add this line to your `<head>` (make sure it comes after the script tag that loads `shiny.js`):

```
<script src="js/increment.js"></script>
```

On the other hand, if you're using `ui.R`, then you can define this function before the call to `shinyUI`:

```
incrementButton <- function(inputId, value = 0) {
  tagList(
    singleton(tags$head(tags$script(src = "js/increment.js"))),
    tags$button(id = inputId,
                class = "increment btn",
                type = "button",
                as.character(value))
  )
}
```

Then in your `shinyUI` page definition you can call `incrementButton` wherever you want an increment button rendered. Notice the line that begins with `singleton` will ensure that the `increment.js` file will be included just one time, in the `<head>`, no matter how many buttons you insert into the page or where you place them.

← Previous Next →