

[Build](#) > [Frontend](#) > [User interface](#) > [HTML Templates](#)

HTML Templates

WINSTON CHANG

DECEMBER 31, 2015

In most cases, the best way to create a Shiny application's user interface is to build it with R code, using functions like `fluidPage()`, `div()`, and so on. Sometimes, though, you may want to integrate Shiny with existing HTML, and starting with Shiny 0.13 (and `htmltools` 0.3), this can be done with the HTML templates. Templates can be used to generate complete web pages, and they can also be used to generate the HTML for components that are included in a Shiny app.

Complete web pages

To use an HTML template for the UI, first create an HTML file in your app directory, at the same level as the `ui.R`, `server.R`, or `app.R` files (not in a `www/` subdirectory). Here's an example template for a complete web page, `template.html`:

```
{% raw %}  
<!DOCTYPE html>  
<!-- template.html -->  
<html>  
  <head>  
    {{ headContent() }}  
  </head>  
  <body>  
    <div>  
      {{ button }}  
      {{ slider }}  
    </div>  
  </body>
```

```
</html>
{% enddraw %}
```

And here's a corresponding `ui.R` that uses the template:

```
## ui.R ##
htmlTemplate("template.html",
  button = actionButton("action", "Action"),
  slider = sliderInput("x", "X", 1, 100, 50)
)
```

Some things to notice:

The template is just plain HTML, except for the parts in `{{ }}` and `"{{"}}`. The parts in those curly braces are R code which is evaluated when the template is processed.

`headContent()` must be placed in the `<head>` section of the HTML, if this is a complete HTML page (as opposed to a component of a page, which we'll discuss later). This tells Shiny that the various Shiny header code should be included here.

In `ui.R`, `htmlTemplate()` is called with the named arguments `button` and `slider`. The values are used when evaluating the R code in the template.

Once processed, the HTML produced will look something like this. You can see where the Shiny head content was inserted, and similarly, the `actionButton` and `sliderInput` HTML code.

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <script type="application/shiny-singletons"></script>
    <script type="application/html-dependencies">
      json2[2014.02.04];
      jquery[1.11.3];
      shiny[0.13];
      ionrangeslider[2.0.12];
```

```

        strftime[0.9.2]
    </script>
    <script src="shared/json2-min.js"></script>
    <script src="shared/jquery.min.js"></script>
    <link href="shared/shiny.css" rel="stylesheet"/>
    <script src="shared/shiny.min.js"></script>
    <link href="shared/ionrangeslider/css/ion.rangeSlider.css"
        rel="stylesheet"/>
    <link href="shared/ionrangeslider/css/ion.rangeSlider.skinSh
        rel="stylesheet"/>
    <script src="shared/ionrangeslider/js/ion.rangeSlider.min.js"
    <script src="shared/strftime/strftime-min.js"></script>
</head>
<body>
    <div>
        <button id="action" type="button"
            class="btn btn-default action-button">Action</button>
        <div class="form-group shiny-input-container">
            <label class="control-label" for="x">X</label>
            <input class="js-range-slider"
                id="x"
                data-min="1"
                data-max="100"
                data-from="50" data-step="1" data-grid="true"
                data-grid-num="9.9" data-grid-snap="false"
                data-prettyfy-separator=","
                data-keyboard="true"
                data-keyboard-step="1.01010101010101"
                data-drag-interval="true"
                data-data-type="number"/>
        </div>
    </div>
</body>
</html>

```

Evaluating code and passing variables

In the previous example, the double-curly braces in the template simply contain the names of variables, `button` and `slider`. However, the R code blocks aren't limited to simple names; any R code can be placed inside a code block. In the example below, we'll put `actionButton()` and `sliderInput()` directly in the template, and we'll pass in an initial value to the `sliderInput()`:

```
{% raw %}  
<!DOCTYPE html>  
<!-- template.html -->  
<html>  
  <head>  
    {{ headContent() }}  
  </head>  
  <body>  
    <div>  
      {{ actionButton("action", "Action") }}  
      {{ sliderInput("x", "X", 1, 100, sliderValue) }}  
    </div>  
  </body>  
</html>  
{% endraw %}
```

```
## ui.R ##  
htmlTemplate("template.html",  
  sliderValue = 50  
)
```

To process the template's R code, the `htmlTemplate()` function first creates a child environment of the global environment, populates it with variables that were passed in as arguments to the function (like `sliderValue`), and then evaluates the template's R code. This means that the template's R code will also have access to any variables set in R's global environment – but it's not good practice to make use of global variables in the template; it's better to pass values explicitly in the call to `htmlTemplate()`.

NOTE: Only the last thing in a code block is included in the HTML output. This is because of how the code blocks are evaluated; only the last thing is returned. If you have a code block with multiple elements that you want in the HTML, you can split it into multiple code blocks, or put the elements together with `tagList()`.

Including other web dependencies

A common reason for using templates is to include custom JavaScript or CSS files. You can add these just as you would with ordinary HTML, with

`<script>` or `<link>` tags. For example, here's a template that uses a custom JavaScript library.

```
{% raw %}
<!DOCTYPE html>
<html>
  <head>
    <script src="customlib.js"></script>
    {{ headContent() }}
  </head>
  <body>
    ...
  </body>
</html>
{% endraw %}
```

In this example, the file `customlib.js` would be expected to be in the `www/` subdirectory of the app, so that it could be served to the client browser. You could also point to an absolute URL (starting with `//`, `http://`, or `https://`) served from another host. This is useful for using popular libraries served from a CDN.

When you include a web dependency this way, there is a possibility that some Shiny code will pull in the same dependency. Imagine that your web page uses D3, so you include a `<script>` tag for it, but then you also use an R component that pulls in a D3 dependency automatically, like `d3heatmap`.

```
{% raw %}
<!DOCTYPE html>
<html>
  <head>
    <script src="//d3js.org/d3.v3.min.js" charset="utf-8"></script>
    {{ headContent() }}
  </head>
  <body>
    <!-- JavaScript code that uses D3 here -->

    {{ d3heatmap::d3heatmap(mtcars, scale="column", colors="Blues") }}
  </body>
```

```
</html>
{% enddraw %}
```

In this case, the HTML after processing the template will include the D3 JavaScript library twice – once from the `<script>` tag that you added, and once from `d3heatmap()` automatically pulling it in:

```
{% raw %}
<!DOCTYPE html>
<html>
  <head>
    <script src="//d3js.org/d3.v3.min.js" charset="utf-8"></script>
    ...
    <script src="d3-3.5.3/./d3.min.js"></script>
    ...
  </head>
  ...
</html>
{% enddraw %}
```

This could cause problems in running the D3 code. To avoid having two copies of a library, use `suppressDependencies()`. This will ensure that Shiny components won't automatically pull in their own versions of web dependencies, and only the one that you manually added to the template will be used:

```
{% raw %}
<!DOCTYPE html>
<html>
  <head>
    <script src="//d3js.org/d3.v3.min.js" charset="utf-8"></script>
    {{ suppressDependencies("d3") }}
    {{ headContent() }}
  </head>
  ...
</html>
{% enddraw %}
```

Using Bootstrap components

Some web components in Shiny require the [Bootstrap](#) web framework in order to display correctly. These include, for example, `tabsetPanel()` and `actionButton()`.

Bootstrap is included automatically when a Shiny UI is created with `bootstrapPage()`, `basicPage()`, `fluidPage()`, `navbarPage()`, and others. However, if you use an HTML template to generate the web page, using `headContent()` will not include Bootstrap – it only includes Shiny's basic web dependencies. In order to include Bootstrap, use `bootstrapLib()`:

```
{% raw %}  
<!DOCTYPE html>  
<html>  
  <head>  
    {{ headContent() }}  
    {{ bootstrapLib() }}  
  </head>  
  <body>  
    {{ actionButton("action", name) }}  
  </body>  
</html>  
{% endraw %}
```

Templates for components

In the examples above, HTML templates were used to generate an entire web page. They can also be used for components that are included in a larger application. For example, you could have this `component.html` and `ui.R`:

```
{% raw %}  
<!-- component.html -->  
<div>  
  This is an HTML template named <code>{{ name }}</code>.  
</div>  
{% endraw %}
```

```
## ui.R ##  
bootstrapPage(  
  h2("HTML template example"),  
  htmlTemplate("component.html", name = "component1")  
)
```

When using a template as a component, don't include `headContent()`, or a `<html>` or `<head>` tag. Just include the HTML which you want to be inserted in the web page.

Using templates in packages

Packages can use HTML templates for components. If you have a package named `mypackage` and have a template file in the package sources at `inst/templates/component.html`, you can access that file with:


```
system.file("templates", "component.html", package = "mypackage")
```

The package could contain a function that makes use of the template:

```
myComponent <- function(name = "component") {  
  htmlTemplate(  
    system.file("templates", "component.html", package = "mypackage"),  
    name = name  
  )  
}
```

Someone using your package could call `myComponent("exampleName")` to put the component in their UI.

The example above use a template to generate a component, but the same strategy can be used to generate a complete web page.

Proudly supported by  **posit**

