

API

Julia to Javascript
Javascript to Julia
Back-and-forth
Tasks

» Communication

C Edit on GitHub

# Communication between Julia and Javascript

After creating a Window and loading HTML and JS, you may want to interact with julia code (e.g. by clicking a button in HTML, or displaying a plot from julia).

This section covers this two-way communication.

# Julia to Javascript

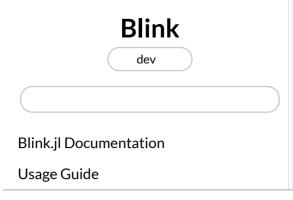
The easiest way to communicate to javascript from julia is with the @js and @js\_ macros. These macros allow you to execute arbitrary javascript code in a given Window.

```
julia> @js win x = 5;
julia> @js win x
5
```

The @js\_ macro executes its code asynchronously, but doesn't return its result:

```
julia> @time @js win begin # Block
    for i in 0:1000000 end # v
        i # return i
    end
    0.439024 seconds (173.59 k allocat
1000001

julia> @time @js_ win begin # Retu
    for i in 0:1000000 end # v
        i # This is ignored
    end
    0.012111 seconds (5.50 k allocatic
```



API

Julia to Javascript
Javascript to Julia
Back-and-forth
Tasks

```
Page(1, WebSocket(server, CONNECTED)
```

If your javascript expression is complex, or you want to copy-paste existing javascript, it can be easier to represent it as a pure javascript string. For that, you can call the <code>js</code> function with a <code>JSString</code>:

```
julia> body!(win, """<div id="box" {
julia> div_id = "box";
julia> js(win, Blink.JSString("""doc
"red"
```

Note that the code passed to these macros runs in its own scope, so any javascript variables you create with var (or the @var equivalent for @js) will be inaccessible after returning:

```
julia> @js win (@var x_var = 5; x_va
5

julia> @js win x_var
ERROR: Javascript error ReferenceErr
```

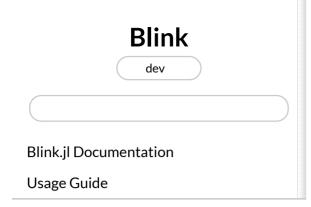
## Javascript to Julia

Communication from javascript to julia currently works via a message passing interface.

To invoke julia code from javascript, you specify a julia callback via handle:

```
julia> handle(w, "press") do args
     @show args
     end
```

This callback can then be triggered from javscript via Blink.msg():



Julia to Javascript
Javascript to Julia
Back-and-forth
Tasks

API

```
julia> @js w Blink.msg("press", "He]
args = "Hello from JS"
```

Note that the javascript function Blink.msg takes *exactly* 1 argument. To pass more or fewer arguments, pass your arguments as an array:

```
julia> handle(w, "event") do count,
       end
#3 (generic function with 1 method)
julia> @js w Blink.msg("event", [1,
MethodError: no method matching (::c
Closest candidates are:
  #3(::Any, !Matched::Any, !Matched:
Stacktrace:
 [1] #invokelatest#1 at ./essentials
 [2] invokelatest at ./essentials.j]
 [3] handle_message(::Page, ::Dict{S
 [4] macro expansion at /home/travis
 [5] ws_handler(::Dict{Any,Any}) at
 [6] splitquery(::typeof(Blink.ws_ha
 [7] #1 at /home/travis/.julia/packa
 [8] wcatch(::getfield(Mux, Symbol('
 [9] todict at /home/travis/.julia/r
 [10] #3 at /home/travis/.julia/pack
 [11] (::getfield(Mux, Symbol("##1#2
 [12] (::getfield(Mux, Symbol("##9#1
 [13] upgrade(::getfield(Mux, Symbol
 [14] (::getfield(WebSockets, Symbol
 [15] macro expansion at /home/travi
 [16] (::getfield(HTTP.Servers, Symt
```

Finally, here is an example that uses a button to call back to julia:



Julia to Javascript
Javascript to Julia
Back-and-forth
Tasks
API

Now, clicking the button will print HELLO to julia's STDOUT.

## **Back-and-forth**

Note that you cannot make a synchronous call to javascript from within a julia callback, or you'll cause julia to hang:

## BAD:

**GOOD**: Instead, if you need to access the value of x, you should simply provide it when invoking the press handler:



Julia to Javascript

Javascript to Julia

Back-and-forth

**Tasks** 

API

## **Tasks**

The julia webserver is implemented via Julia Tasks. This means that julia code invoked from javascript will run *sort of* in parallel to your main julia code.

## In particular:

- Tasks are *coroutines*, *not threads*, so they aren't truly running in parallel.
- Instead, execution can switch between your code and the coroutine's code whenever a piece of computation is interruptible.

So, if your Blink callback handler performs uninterruptible work, it will fully occupy your CPU, preventing any other computation from occuring, and can potentially hang your computation.

## **Examples:**

**BAD**: If your callback runs a long loop, it won't be uninterruptible while it's running:

**BAD**: The same is true if your *main* julia computation is hogging the CPU, then your callback can't run:

```
julia> handle(w, "press") do args...
```



Julia to Javascript
Javascript to Julia
Back-and-forth
Tasks
API

```
println("Start")
    sleep(5) # This will happil
    println("End")
    end
#41 (generic function with 1 method)

julia> body!(w, """<button onclick='

julia> while true end # Infinite lc
# NOW, CLICK THE go BUTTON, AND NOTH
```

**GOOD**: So to allow for happy communication, all your computations should be interruptible, which you can achieve with calls such as yield, or sleep:

```
julia> handle(w, "press") do args...
    println("Start")
    sleep(5) # This will happil
    println("End")
    end
#39 (generic function with 1 method)

julia> body!(w, """<button onclick='

julia> while true # Still an infini
    yield() # This will yield
    end

# NOW, CLICKING THE go BUTTON WILL V
Start
End
```

Previous: Usage Guide Next: API