Compiling the jq command line tool

So far, we've covered compiling relatively small codebases. Let's turn our attention to `jq`, a much larger project that consists of tens of thousands of lines of code. If you're not familiar with `jq`, it's a command line tool for parsing, creating and modifying JSON strings directly on the command line. It's a must-have tool for wrangling JSON on the command line, so if you've not used it before, I highly recommend you check it out.

First, let's clone the repo and enter the source code folder:

```
git clone https://github.com/stedolan/jq.git
cd jq
```

Before we compile `jq`, we need to fetch `oniguruma`, an open-source library that `jq` uses to support regular expressions; that library is a submodule in the git repo:

```
git submodule update --init
```

> **Note**
>
> The general approach to compiling codebases to WebAssembly starts by making sure you're able to compile them to binary (which is sometimes not straightforward!), then mapping those steps to WebAssembly using `emconfigure`, `emmake`, and `emcc`.

You'll find the instructions for compiling jq to binary in `jq`'s README file—shown below in the left panel. And in the right panel, you'll find the corresponding instructions to compile jq to WebAssembly (with differences highlighted in red):

```
# Generate ./configure file
autoreconf -fi

# Run ./configure
./configure \
    --with-oniguruma=builtin \
    --disable-maintainer-mode

# Build jq executable
make LDFLAGS=-all-static
```

```
# Generate ./configure file
autoreconf -fi

# Run ./configure
emconfigure ./configure \
    --with-oniguruma=builtin \
    --disable-maintainer-mode

# Build jq executable
emmake make LDFLAGS=-all-static
```

So far so good! The only difference is that we're using Emscripten's `emconfigure` and `emmake` tools to wrap around the existing `configure` and `make` utilities, such that they use `emcc` and `em++` instead of `gcc` and `g++`.

However, we do have a few more steps to go. The `emmake` command above created the WebAssembly executable `jq`. Since we want the convenience of the .js glue code that Emscripten generates for us, let's create it:

```
# But first, rename the file to .o; otherwise,
# emcc complains that the "file has an unknown suffix"
mv jq jq.o

# Generate .js and .wasm files
emcc jq.o -o jq.js \
  -s ERROR_ON_UNDEFINED_SYMBOLS=0
```

Note that we use the flag `ERROR_ON_UNDEFINED_SYMBOLS` to ignore warnings such as `"undefined symbol: llvm_fma_f64"` (if you try compiling with just `emcc jq.o -o jq.js`, you'll see the error).

As you can see, the differences during the build process are surprisingly minimal given that we're compiling tens of thousands of lines of code from C to being able to run them in the browser! Can you imagine re-writing `jq` from scratch, in JavaScript? 😱

To make sure it works, let's try some examples from the jq tutorial directly on the command line:

```
# Output the description of the latest commit on the jq repo
$ curl -s "https://api.github.com/repos/stedolan/jq/commits?per_page=5" | \
    node jq.js '.[0].commit.message'
"Improve linking time by marking subtrees with unbound symbols"

# Output an array of commit messages and committer's name
$ curl -s "https://api.github.com/repos/stedolan/jq/commits?per_page=5" | \
    node jq.js '.[] | {message: .commit.message, name:
.commit.committer.name}'
{
  "message": "Improve linking time by marking subtrees with unbound
symbols",
  "name": "Nico Williams"
}
[...]
```

Building an interactive jq app

Now let's take it a step further and use the compiled WebAssembly in an app that lets us interactively make jq queries right in the browser. There are apps that already exist to do that—namely jqplay.org—though those tools tend to rely on sending the request to the backend, where it is evaluated. This means the potential for slower response time, and could be a security concern (we need to make sure users can't execute arbitrary commands on the server).

The app we're building here uses almost exactly the same logic that I used to build jqkungfu.com!

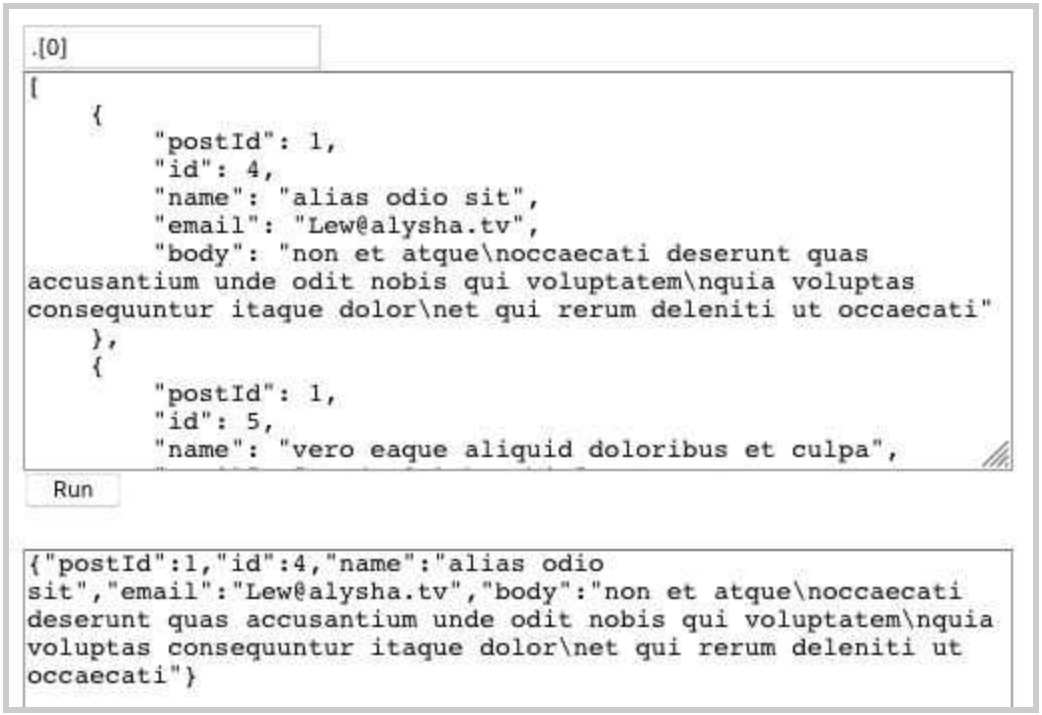
The HTML/JS code below is all you need to enable that (with a minimal UI that is!). The code is commented so you can follow along:

```
<!-- Just a bit of styling -->
<style type="text/css">
#input { width: 50%; height: 200px; }
#output { width: 50%; height: 200px; }
</style>

<!-- The main logic-->
<script type="text/javascript">
// WebAssembly module config
var output = [];
var Module = {
      // Don't automatically run main() function on page load
      noInitialRun: true,

      // Print functions (capture jq output)
      print: stdout => output = stdout,
      printErr: stderr => console.warn(stderr),

      // When the module is ready, enable the "Run" button
      onRuntimeInitialized: function() {
            document.getElementById("btnRun").disabled = false;
      }
};

// Utility function that runs jq, given a JSON string and a query
function jq(jsonStr, query)
```

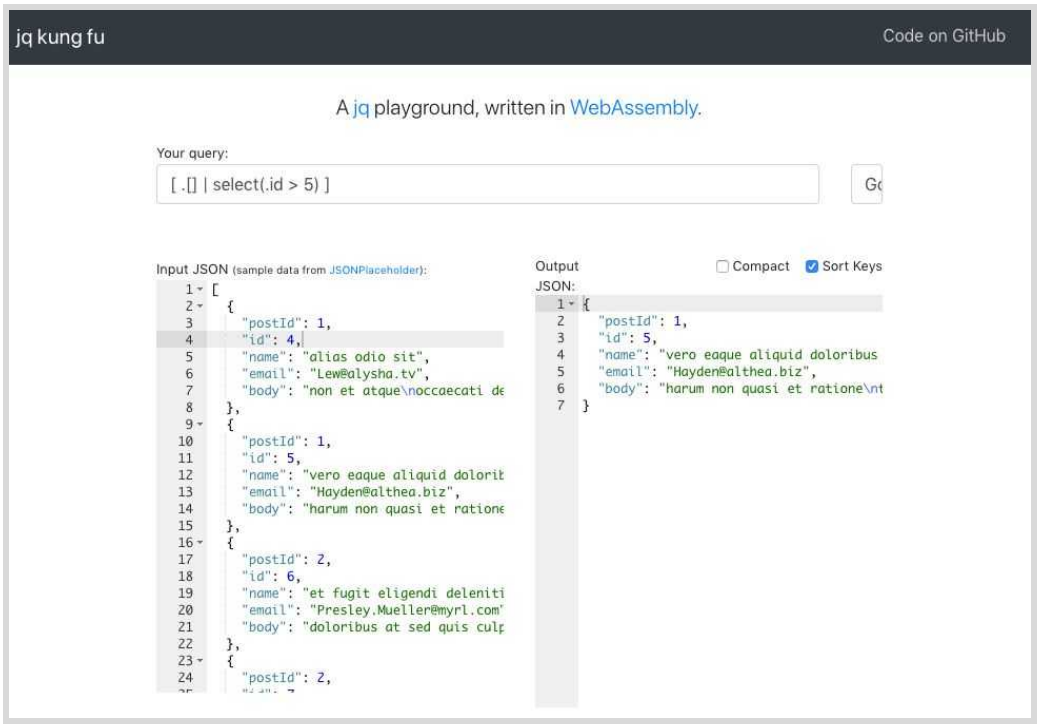
```
{
	var fileName = "/tmp/data.json";

	// Create file from JSON string
	FS.writeFile(fileName, jsonStr);

	// Launch jq's main() function on that file
	//   -M to disable colors
	//   -r to output in raw format
	//   -c to output in compressed format
	output = [];
	Module.callMain([ "-M", "-r", "-c", query, fileName ]);

	// Re-open stdout/stderr since jq closes them when it exits
	// Otherwise we get an error about having a "Bad file descriptor"
	// if we run jq twice in a row
	FS.streams[1] = FS.open("/dev/stdout", "w");
	FS.streams[2] = FS.open("/dev/stderr", "w");

	return output;
}

// On page load
document.addEventListener("DOMContentLoaded", function()
{
	// On button click
	document.getElementById("btnRun").addEventListener("click",
function()
	{
		// Run jq and update output textarea
		document.getElementById("output").value = jq(
			document.getElementById("input").value,
			document.getElementById("query").value
		);
	});
});
</script>

<!-- Load jq WebAssembly module -->
<script src="jq.js"></script>

<!-- Basic UI for jq -->
```

```
<input id="query" type="text" value=".b"><br />
<textarea id="input">
{
      "a": 123,
      "b": {
            "bb": 12,
            "cc": 45
      },
      "c": 789
}
</textarea><br />

<input id="btnRun" type="button" value="Run" disabled><br /><br />
<textarea id="output"></textarea>
```

Here's a screenshot of the code above in action:



Admittedly, this doesn't look like much, but with a bit more CSS, it can look like this:



See GitHub for jqkungfu's source code.

To me, this is where WebAssembly's strength shines through: the ability to take off-the-shelf command-line utilities, and turn them into libraries that are useful on the web, whether to speed up existing code or to enable a feature in the first place, is extremely powerful. `jq` in particular is a great example of that; it's a powerful command-line tool, yet can also be very useful on the web. Here we built a tool for playing around with `jq` outside the command line, but you can easily imagine how it could also be used in data-heavy web applications, to enable users to query JSON data using the familiar language that `jq` provides.