

Kotlin to WebAssembly

We'll cover the following

- Setting up Node.js application
- Using jsinterop

WebAssembly (WASM) is poised to be the next major disruptive technology. In the past, JavaScript was the only predominant viable option for executing code within browsers. With WebAssembly, you can compile code written in many different languages—including C, C++, Rust, Go, C#, Python, Java, and Kotlin—to the [WASM](#) binary instruction format, for a stack-based virtual machine that runs within modern browsers. WASM brings many benefits, including speed of development using higher-level languages, high runtime performance, enhanced safety, and code that's easier to test and debug and that interoperates with code written in different languages.

Kotlin to WebAssembly is currently in initial stages at the time of this writing. Treat the material in this appendix as experimental.

To target Kotlin to WASM, use the Kotlin/Native compiler with the `-target wasm32` command-line option. If you've not had a chance to install the Kotlin/Native distribution in [Kotlin/Native](#), do so before you continue.

Setting up `Node.js` application

In this appendix, we'll create a small example to draw on an HTML5 canvas, using Kotlin code to illustrate Kotlin to WASM targeting. We'll need an HTML file to fire up a page in the browser. We'll also need Kotlin source code to draw on the canvas that will be defined within the HTML file. In addition to using Kotlin/Native, we also need a minimal lightweight web server to serve the files into a browser. Let's get these things set up, one step at a time.

In an empty directory named `wasm`, run the command `npm init` and accept the defaults—for this step you'll need to have [Node.js](#) installed on your system. When

done, install the lite-server using the command `npm install lite-server --save`.

After the installation completes, open the file `package.json` and edit it to add a start command to start the lite-server. The file should look like the following when you're done with these steps—the version of lite-server that you see may be different.

```
// package.json
{
  "name": "wasm",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "lite-server"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "lite-server": "^2.4.0"
  }
}
```


Create an `index.html` file with the following content:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Kotlin to WebAssembly</title>
  </head>
  <body>
    <h1>Drawing into HTML 5 Canvas using Kotlin</h1>
    <canvas id="display" width="300" height="300"></canvas>
  </body>
  <script wasm="check.wasm" src="check.wasm.js"></script>
</html>
```

index.html

In the HTML file, we've defined an HTML5 canvas into which we'll render from within the Kotlin code we'll write soon. We've also included, using the `script` tag, a `check.wasm` file that uses the `wasm` attribute and a `check.wasm.js` file that uses the `src` attribute. Modern browsers that support WASM will recognize the `wasm` attribute and load the code within the embedded WASM virtual machine.

As a next step, let's create the Kotlin code to render into the canvas:



```
import kotlinx.interop.wasm.dom.*

fun main() {
    val canvas = document.getElementById("display").asCanvas

    val context = canvas.getContext("2d")

    val rect = canvas.getBoundingClientRect()
    val offsetX = (rect.right - rect.left) / 4
    val offsetY = (rect.bottom - rect.top) / 4

    context.apply {
        fillStyle = "green"
        strokeStyle = "white"
        lineWidth = 10

        fillRect(rect.left, rect.top, rect.right, rect.bottom)
        beginPath()
        moveTo(rect.left + offsetX / 2, rect.bottom - 3 * offsetY)
        lineTo(rect.left + offsetX, rect.bottom - 2 * offsetY)
        lineTo(rect.right - offsetX, rect.top + offsetY)
        stroke()
    }
}
```

check.kt

We first import the contents of the `kotlinx.interop.wasm.dom` package—we'll see soon where this dependency comes from. In the `main()` function we get a reference to the canvas object from the HTML file's DOM `document` object. Then we obtain a reference to the “2D” context object of the canvas. Then using the canvas API, we fill the rectangle of the canvas with green color and draw white color lines to represent a check mark. Even though we're writing this code in Kotlin, the API is pretty much the HTML5 canvas API—refer to the W3C standard [API documentation](#).

Using `jsinterop`

It's time to compile the code and see it in action. First we need to create the JavaScript interop functions to talk to the canvas API from our Kotlin code. In that vein, the `jsinterop` tool is like the `cinterop` tool we saw in [Kotlin/Native](#). At the time of this writing, the `jsinterop` tool is limited to creating interop functions only for two libraries, and it also has limitations to the functions it maps to Kotlin signature. Over time, as development continues, these limitations will disappear, and the tool will become versatile so we can easily interact from Kotlin with different libraries written in JavaScript and other languages.

Run the `jsinterop` tool and specify the necessary `target` and the `pkg` command-line options:

```
jsinterop -target wasm32 -pkg kotlinx.interop.wasm.dom
```

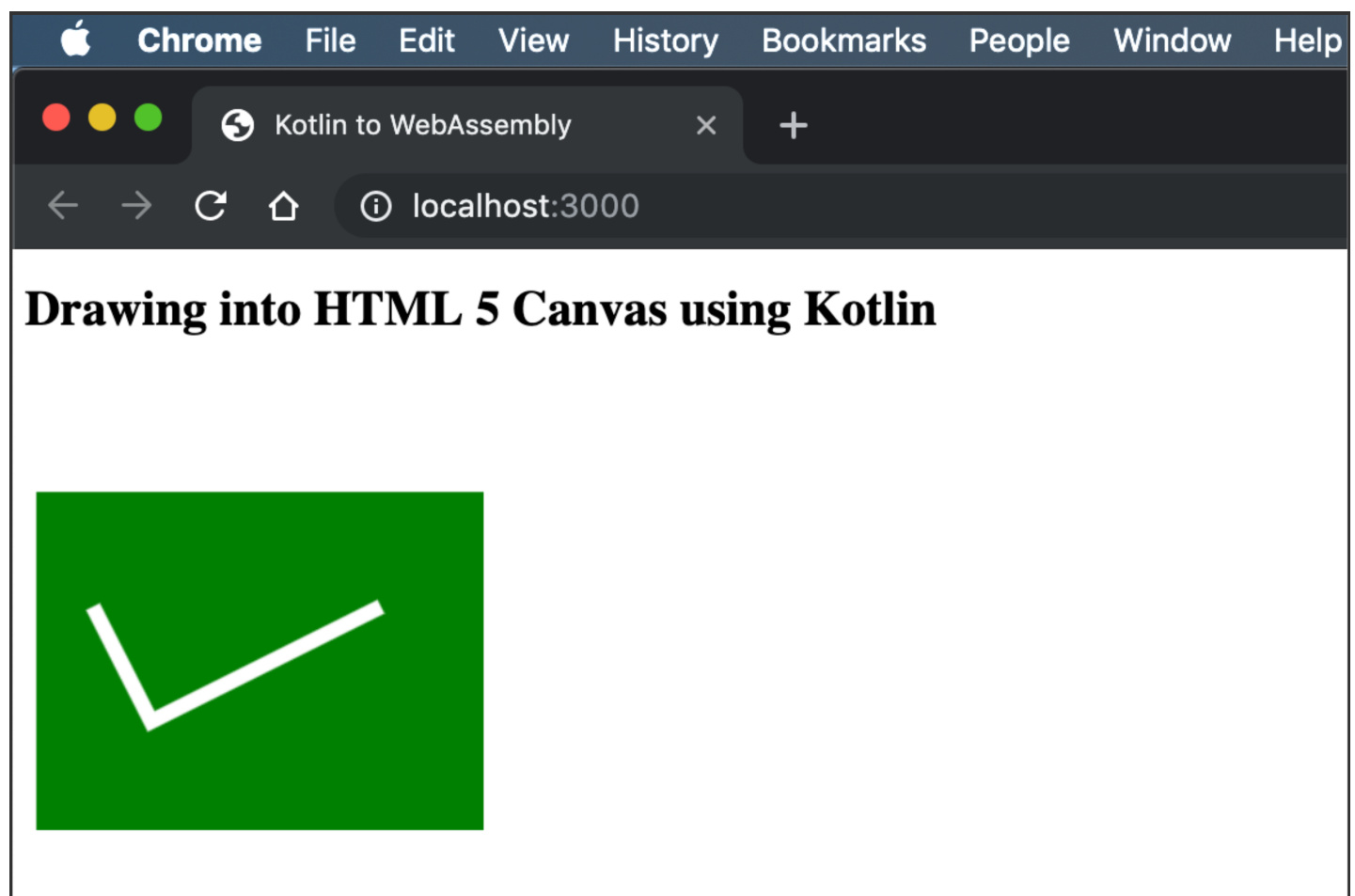
This command will generate the Kotlin signatures for the DOM API and thus the files necessary to satisfy the `import` we saw at the top of our Kotlin code. Take note of the generated files and the `nativelib` library—we'll use this library in the next step.

Next, run the `kotlinc-native` compiler to compile the Kotlin source code to WASM:

```
kotlinc-native -target wasm32 check.kt -library nativelib -o check
```

This command will result in the creation of the `check.wasm` and `check.wasm.js` files. We've already referenced these two files in the `index.html` file, and we're ready to take the WASM code compiled from Kotlin for a ride.

Run the command `npm start`—this will start the lite-server and fire up your default browser to load the `index.html` file. You should then see something like:



This figure shows the Chrome browser displaying the `index.html` file with the canvas, which has the check mark that was drawn by our Kotlin code, running within the WASM virtual machine.
