An Introduction to WebAssembly

In order to understand the necessity for WebAssembly (WASM), one must first understand that JavaScript, which is the current default for web app development, uses a Just in Time Compiler. This means that, as web applications grow more complex, it takes JavaScript longer to compile them, because when each instruction is called, it has to be individually compiled into something the browser can understand (Low Level Learning 2022).

WebAssembly was created as a solution to the overhead problem of JavaScript by compiling applications to byte code (Hayes 2022). This means that the original program can now run on any browser that supports WASM without the developer having to ensure backwards compatibility or try to translate their OOP program into JavaScript, which means that they can write far more complex applications for the web (Royse 2020). **(131 words)**

An AssemblyScript Tutorial

This AssemblyScript tutorial will take you through the following key steps:

1. Installing and Setting Up Node.js
2. Getting AssemblyScript Working
3. Creating a simple application using the software you've installed

Installing and Setting Up Node.js

1. If you haven't installed Node.js, go to nodejs.org and download the setup wizard.
2. Open Node.js command prompt
3. Navigate to the folder where you plan on working

```
C:\Users\andrea> cd Documents
```

4. Create a new directory for your application and initialize npm:

```
C:\Users\andrea\Documents> mkdir assemblyscript-app

C:\Users\andrea\Documents> cd assemblyscript-app

C:\Users\andrea\Documents\assemblyscript-app> npm init
```

5. Press enter in response to all of the prompts

Getting AssemblyScript Working

1. AssemblyScript is like any other npm package. This means that any dependencies that you need to install can be done using `npm install`.

```
C:\Users\andrea\Documents\assemblyscript-app> npm install --save
@assemblyscript/loader
```

2. If you look at package.json (either by opening it in a text editor, or running `more package.json` in the node.js terminal), you should see something like this:

```
{
    "name": "assemblyscript-app",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC",
    "dependencies": {
        "@assemblyscript/loader": "^0.21.3"
    }
}
```

3. Next, we install AssemblyScript itself:

```
C:\Users\andrea\Documents\assemblyscript-app> npm install --save-dev
assemblyscript
```

4. The AssemblyScript compiler provides a handy scaffolding utility to quickly set up a new project in the current directory:

```
C:\Users\andrea\Documents\assemblyscript-app> npx asinit .
```

5. This will create several directories and files. At this point (if you haven't yet) you should open the assemblyscript-app folder in a text editor of your choice. Make sure you see a folder called assembly and open it. Within it, you should see index.ts, which is where we'll be writing our code!

Creating a simple application using the software you've installed

```
/*
* If your text editor has some form of built-in TypeScript checking
tsconfig.json,
* which is also in the assembly folder, will look like it has errors. You
can ignore them.
*/
```

1. index.ts starts with a simple addition function:

```
export function add (a: i32, b: i32): i32 {
    return a + b;
}
```

It's made to add two numbers, and if you want to test it (just to generally see how AssemblyScript works), you can do that by running these commands in the Node.js command prompt:

```
C:\Users\andrea\Documents\assemblyscript-app> npm run asbuild //this
```

```
compiles the code in index.ts to WebAssembly
C:\Users\andrea\Documents\assemblyscript-app> npm test //this executes
the test case in tests/index.js
C:\Users\andrea\Documents\assemblyscript-app> npm start //this starts a
web server, which will serve index.html
```

Node.js will copy the address of the webpage where index.html is being served to your keyboard. If you paste it into a browser, it should be displaying a number on the page– the sum of the two numbers given as input in tests/index.js. To play around with it, try change the numbers and run these commands again.
To stop the web server, use CTRL + C twice.

2. Now we're going use AssemblyScript to create a word game called Mojpal. You can start by removing the `add()` function from assembly/index.ts.
3. The premise of Mojpal is fairly simple:

How many five-letter words can a player come up with without resuing the placement of the letters?

And we're going to create it in index.html and assembly/index.ts

4. mojpal is based on three key functions:
    - Function #1: `isWord(guess)`, which takes a string and returns `true` or `false` based on whether the string matches any of the strings in a dictionary array
    - Function #2: `letterUsed(guess)`, which checks each letter of the string against an array of the characters previously used at that position, and returns `true` if it finds a match and `false` if the letter placement is unique
    - Function #3: `check(word)`, which runs the other functions and returns a string result
    - Function #4: `reset()`, which clears all the saved information and

restarts the game

5. The first step of creating Mojpal is creating the variables we'll use in assembly/index.ts:

```
const place1: string[]  = []
const place2: string[]  = []
const place3: string[]  = []
const place4: string[]  = []
const place5: string[]  = []
let number = 0;
for(let i = 0; i < 26; i++){
  place1.push("0");
  place2.push("0");
  place3.push("0");
  place4.push("0");
  place5.push("0");
}
```

6. Next, we have to add a dictionary to check which words are real. We can do this by copying the `WORDS` array available here. When minimized, it should look something like this:

```
const WORDS = [· · ·
]
```

7. Now we create `isWord(guess)`, which will loop through every word in `WORDS` and compare it to the given string:

```
export function isWord(guess: String): boolean{
    for (let i = 0; i < WORDS.length; i++) {
      if (guess.toLowerCase() === WORDS[i]){
```

```
        return true;
      }
    }
    return false;
  }
```

8. Now we create `letterUsed(guess)`, which will loop through every letter stored for each position and return `true` if the letter has already been used. If the owrd is unique, it will add the new letters to the storage arrays.

```
export function letterUsed(guess: String): boolean{
   let letters = guess.split('');
    for (let i = 0; i < 5; i++) {
     switch (i){
       case 0:
         for (let j = 0; j < 26; j++){
           if (place1[j] != "0" &&  place1[j] === letters[i]){
             return true;
           }
         }
       break;
       case 1:
         for (let j = 0; j < 26; j++){
           if (place2[j] != "0" &&  place2[j] === letters[i]){
             return true;
           }
         }
       break;
       case 2:
         for (let j = 0; j < 26; j++){
```

```
            if (place3[j] != "0" &&  place3[j] === letters[i]){
              return true;
            }
          }
        break;
        case 3:
          for (let j = 0; j < 26; j++){
            if (place4[j] != "0" &&  place4[j] === letters[i]){
              return true;
            }
          }
        break;
        case 4:
          for (let j = 0; j < 26; j++){
            if (place5[j] != "0" &&  place5[j] === letters[i]){
              return true;
            }
          }
        break;
      }
    }
    place1[number] = letters[0];
    place2[number] = letters[1];
    place3[number] = letters[2];
    place4[number] = letters[3];
    place5[number] = letters[4];
    number++;
    return false;
  }
```

9. Most importantly for index.ts, we create `check()`, which will:

- Test if the user's guess is five letters long
- Then test if the user's guess is a real word
- *Then* test that none if the letters of that word have already been used in the same position
- And finally return a result in the form of a string

```
export function check(word: string): string{
    if (word.length != 5) {
      return "Not five letters.";
    }
    else if (!isWord(word)){
      return "Not in my dictionary.";
    }
    else if (!letterUsed(word)) {
      let str = word + " - ";
      return str;
    }
    else {
      return "you've already used a letter";
    }
  }
```

10. And lastly we create a simple function that will reset number and each of the arrays of stored letters:

```
export function reset(): void {
    let number = 0;
    for(let i = 0; i < 26; i++){
      place1.push("0");
      place2.push("0");
      place3.push("0");
```

```
      place4.push("0");
      place5.push("0");
    }
}
```

11. The last step is to update index.html so that it can use these functions we've created. Remove everything that the file generates by default and replace it with this:

```
<!DOCTYPE html>
    <html lang="en">
    <head>
     <!--this changes the name of the webpage in your tabs-->
        <title>mojpal</title>
        <script type="module">
            //import { methodname } is how we call the function
            //from where it was built as Web Assembly
            import { check } from "./build/release.js";
            import { reset } from "./build/release.js";
            //this is how we make it availableto the whole page,
            //so that we can use it later
            window.check = check;
            window.reset = reset;
        </script>
    </head>
    <body>
         <!--this is where the player will type out their guess-->
        <input type="text" id="guess" name="guess">

         <!--this button calls a local JavaScript -->
        <!--function which will then call check() from index.ts-->
```

```
        <button onclick="checkWord()" id="guessbutton">Guess</button>

         <!--this button resets the game by calling a a local JavaScript-->
        <!-- function which will then call reset() from index.ts-->
        <button onclick="rs()" id="restart">Restart?</button>

         <!--this paragraph displays the list of words the player-->
        <!--has guessed so far, or why they lost-->
        <p id="history"></p>

        <script type="text/javascript" >
        var history = document.getElementById("history");
            function checkWord() {
                var guess = document.getElementById("guess").value;
                var checkResult = check(guess);
                if(checkResult.length > 8){ //if there is an error message
                history.innerHTML = checkResult;
                }
                else {
                    history.innerHTML = history.innerHTML + checkResult;
                }
                //resets the inpurt field for player convenience
                document.getElementById("guess").value = "";
            }

            function rs(){
                reset();
                history.innerHTML = "";
            }
        </script>
    </body>
```

```
    </html>
```

Playing Your Game

To start playing, first go to the node.js terminal and ru nthe following commands:

```
C:\Users\andrea\Documents\assemblyscript-app> npm run asbuild
C:\Users\andrea\Documents\assemblyscript-app> npm start
```

The response should say that it's copied the local address to your clipboard. Go to your browser, paste the address, and start playing!

Well done! You've learned how to setup node.js and WebAssembly, and code a simple application using Assembly Script.

References

Danilo, A. and Gandluri, D. 2017. Real World WebAssembly (Chrome Dev Summit 2017).

Hayes, B. 2022. Keynote: View from Above: Birds-eye View of the Wasm Landscape and Where It's Heading.

Low Level Learning 2022. *What is WebAssembly and Is It Better Than JavaScript? | WASM Breakdown for Beginners and Web Devs*. Available at: https://www.youtube.com/watch?v=sR22HtWztrY [Accessed: 11 September 2022].

Royse, G. 2020. An introduction to WebAssembly | DevNation Day 2020.