WEBASSEMBLY

By Aidan Wong and Tanzeem Hasan

Introduction

WebAssembly (Wasm) is a compact, binary instruction format that allows you to build web applications in languages other than JavaScript. It functions as a universal translator, allowing software written in languages ranging from Python to C++ to be delivered in a web browser with near-native performance. This enabled the development of more performance-intensive applications like 3D rendering and high-speed simulations that could be run without the installation of any additional software.

Wasm includes web assembly text (.WAT), a low-level human-readable language similar to Assembly, which is then converted to a binary format (.WASM) that runs on all modern browsers. However, this code isn't written directly and is used as a compilation target for programs written in other languages. For example, a game may be written in C#, but then compiled into WebAssembly to deliver in the browser.

Keep in mind that WebAssembly is not a replacement for JavaScript. The two languages work well together, some notable examples being Figma, an online design tool, and Google Earth.

KEY FEATURES

- **High Performance**: Wasm is designed for speed and performance, outclassing JavaScript by leveraging binary format and just-in-time (JIT) compilation
- **Portability**: Wasm's ability to execute on web browsers allows it to become platform-independent, running consistently on devices ranging from desktops to cell phones
- **Security**: Wasm runs in a sandboxed environment (only has access to what you specifically give it), which reduces security risks
- **Compatibility with JavaScript**: Allows integration with existing web applications and delegation of tasks (Ex. JS handles user input and DOM while Wasm handles the backend)
- **Compact**: Binary format is compact, which allows for quicker loading times

USE CASES

- **Compute-intensive applications**: Increases the speed of things like games, image/video editing, and CAD tools
- **Enterprise Fat Clients**: Increases the efficiency of business applications, such as databases
- **VPN**: Increases the security and efficiency of tunneling
- **Symmetric Computations Across Multiple Nodes**: Distributed computing and parallel processing benefit from Wasm's execution efficiency, allowing uses in machine learning and large-scale simulations
- **Edge Computing:** Makes Wasm suitable for low-latency computations, such as real-time analytics systems
- **Hybrid native apps on mobile devices:** Eliminates the need for platform-specific languages

CREATING .WASM FILES

For compiled languages like C++, C, and Rust, we would use compilers that accept WebAssembly as a target. For C and C++, we would use Emscripten. After the installation process, we can use the command:

```
emcc filename.filetype -o filename.html
```

This will create the corresponding HTML, JS, and WASM files that run the script within the specified filetype and display the results on the generated HTML.

Creating .wasm files from Python uses a different process due to its status as an interpreted language. Instead of compiling the Python file, the Python interpreter is converted into WebAssembly. Projects like Pyodide and PyScript accomplish this, allowing Python code to be run within a browser. Scripts are run within a Wasm-compiled runtime inside the browser instead of generating a separate .wasm file for each script.

USING .WASM FILES

After creating the .wasm file for compiled languages, we need to load the the file using built-in WebAssembly API into JavaScript. This gives JavaScript access to the functions in the .wasm file.

math.c

```
int add(int a, int b) {
    return a + b;
}
```

Simple Addition function in C

This command simply creates a .wasm file (without the HTML and JS) Emscripten-specific flag to export the specified functions to be accessible using JavaScript

```
emcc math.c -o math.wasm -s
EXPORTED_FUNCTIONS="['_add']"
```

math.html

```
fetch('math.wasm')
    .then(response => response.arrayBuffer())
    .then(bytes => WebAssembly.instantiate(bytes))
    .then(result => console.log(result.instance.exports.add(2, 3)));
```

Loading and using the add function with JavaScript (in the script tags of HTML, alternatively in an external JS file)

Python uses a different process. In Pyodide, for example, you would import it through a script tag:

```
<script src="https://cdn.jsdelivr.net/pyodide/v0.23.4/full/pyodide.js"></script>
```

This allows you to access Wasm through JavaScript, either through the script tags in the HTML or an external JavaScript file. For example, the following code executes the line "2 + 3" in Python and stores that result to a variable in the JavaScript file and prints it to the console. This is done without the creation of any additional files on your system.

```
async function runPython() {
    let pyodide = await loadPyodide();  // Loads the Pyodide runtime
    let result = pyodide.runPython("2 + 3");  // Executes the Python code "2 + 3"
    console.log(result);  // Logs the result of the Python expression
}
runPython();
```

Conclusion

Wasm has a vast array of uses. It applies to numerous fields, from blockchain and cryptography to running games and simulations. It has revolutionized the way in which programmers interact with their applications by providing a way for them to run their programs regardless of compatibility with the underlying software. But Wasm is still a very new tool, and it needs time to be developed to its full capacity. Browser compatibility is an issue for older browsers like Internet Explorer. Passing data between Wasm and JavaScript can be tedious or tricky. Furthermore, Wasm currently doesn't have a built-in garbage collector to take care of objects that are no longer referenced but still occupy a space in memory, unlike JavaScript. This makes memory management more prone to errors. As developers continue to advance Wasm, old problems will be resolved while others become less significant. Regardless, Wasm will continue to grow more prominent.

Resources

Official Documentation: https://webassembly.org/
Sample Codebase: https://github.com/awong50/TEDxSoftDev25