@wasmer/sdk

The Wasmer JavaScript SDK

Javascript library for running Wasmer packages at ease, including WASI and WASIX modules.

Getting Started

Install from NPM

For installing `@wasmer/sdk`, run this command in your shell:

```
npm install --save @wasmer/sdk
```

You can now run packages from the Wasmer registry:

```
import { init, Wasmer } from "@wasmer/sdk";

await init();

const pkg = await Wasmer.fromRegistry("python/python");
const instance = await pkg.entrypoint.run({
    args: ["-c", "print('Hello, World!')"],
});

const { code, stdout } = await instance.wait();
console.log(`Python exited with ${code}: ${stdout}`);
```

Use with a `<script>` tag (without bundler)

It is possible to avoid needing to use a bundler by importing `@wasmer/sdk` from your script tag in unpkg.

```
<script defer type="module">
    import { init, Wasmer } from
"https://unpkg.com/@wasmer/sdk@latest/dist/index.mjs";

    async function runPython() {
        await init();

        const packageName = "python/python";
        const pkg = await Wasmer.fromRegistry(packageName);
        const instance = await pkg.entrypoint.run({
            args: ["-c", "print('Hello, World!')"],
        });

        const { code, stdout } = await instance.wait();

        console.log(`Python exited with ${code}: ${stdout}`);
    }

    runPython();
</script>
```

Using a custom Wasm file

By default, `init` will load the Wasmer SDK WebAssembly file from the package. If you want to customize this behavior you can pass a custom url to the init, so the the wasm file of the Wasmer SDK can ve served by your HTTP server instead:

```
import { init, Wasmer } from "@wasmer/sdk";
import wasmerSDKModule from "@wasmer/sdk/wasm?url";

await init({ module: wasmUrl }); // This inits the SDK with a custom URL
```

Using a JS with the Wasm bundled

You can also load Wasmer-JS with a js file with the Wasmer SDK WebAssembly file bundled into it (using base64 encoding), so no extra requests are required. If that's your use case, you can simply import `@wasmer/sdk/wasm-inline`:

```
import { init, Wasmer } from "@wasmer/sdk";
import wasmerSDKModule from "@wasmer/sdk/wasm-inline";

await init({ module: wasmerSDKModule }); // This uses the inline wasmer SDK
version
```

Cross-Origin Isolation

Browsers have implemented security measures to mitigate the Spectre and Meltdown vulnerabilities.

These measures restrict the sharing of `SharedArrayBuffer` objects with Web Workers unless the execution context is deemed secure.

The `@wasmer/sdk` package uses a threadpool built on Web Workers and requires sharing the same `SharedArrayBuffer` across multiple workers to enable WASIX threads to access the same address space. This requirement is crucial even for running single-threaded WASIX programs because the SDK internals rely on `SharedArrayBuffer` for communication with Web Workers.

To avoid Cross-Origin Isolation issues, make sure any web pages using `@wasmer/sdk` are served over HTTPS and have the following headers set:

```
"Cross-Origin-Opener-Policy": "same-origin"
"Cross-Origin-Embedder-Policy": "require-corp"
```

See the `SharedArrayBuffer` and Cross-Origin Isolation section under the *Troubleshooting Common Problems* docs for more.

Creating packages

Users can create packages providing a manifest and using the `Wasmer.createPackage()` function:

```
import { init, Wasmer } from "@wasmer/sdk";

await init({ token: "YOUR_TOKEN" });

const manifest = {
    command: [
        {
            module: "wasmer/python:python",
            name: "hello",
            runner: "wasi",
            annotations: {
                wasi: {
                    "main-args": [
                        "-c",
                        "print('Hello, js!'); ",
                    ],
                },
            },
        },
    ],
    dependencies: {
        "wasmer/python": "3.12.9+build.9",
    }
};

let pkg = await Wasmer.createPackage(manifest);
let instance = await pkg.commands["hello"].run();

const output = await instance.wait();
console.log(output)
```

Publishing packages

User can publish packages following the same flow used to create a package and then calling the `Wasmer.publishPackage()` function:

```
import { init, Wasmer } from "@wasmer/sdk";

await init({ token: "YOUR_TOKEN" });

const manifest = {
    package: {
        name: "<YOUR_NAME>/<YOUR_PACKAGE_NAME>"
    }
    command: [
        {
            module: "wasmer/python:python",
            name: "hello",
            runner: "wasi",
            annotations: {
                wasi: {
                    "main-args": [
                        "-c",
                        "print('Hello, js!'); ",
                    ],
                },
            },
        },
    ],
    dependencies: {
        "wasmer/python": "3.12.9+build.9",
    }
};

let pkg = await Wasmer.createPackage(manifest);
await Wasmer.publishPackage(pkg);
```

Trying to publish packages without a `package.name` property in the manifest will result in a failure.

Deploying apps

User can deploy apps by providing an app configuration and calling the `Wasmer.deployApp()` function:

```
import { init, Wasmer } from "@wasmer/sdk";

// Get your token here: https://wasmer.io/settings/access-tokens
await init({ token: "YOUR_TOKEN" });

let appConfig = {
  name: "<YOUR_APP_NAME>",
  owner: "<YOUR_NAME>",
  package: "wasmer/hello"
  default: true,
};

await Wasmer.deployApp(appConfig);
```

Users can also publish apps with their own packages simply providing the package in the config:

```
import wasmUrl from "@wasmer/sdk";

// Get your token here: https://wasmer.io/settings/access-tokens
await init({token: "YOUR_TOKEN"});

const echo_server_index = `
    async function handler(request) {
      const out = JSON.stringify({
        env: process.env,
      });
      return new Response(out, {
        headers: { "content-type": "application/json" },
      });
    }

    addEventListener("fetch", (fetchEvent) => {
      fetchEvent.respondWith(handler(fetchEvent.request));
    });
    `;


const manifest = {
    "command": [
        {
            "module": "wasmer/winterjs:winterjs",
            "name": "script",
            "runner": "https://webc.org/runner/wasi",
            "annotations": {
                "wasi": {
                    "env": [
                        "JS_PATH=/src/index.js"
                    ],
                    "main-args": [
                        "/src/index.js"
                    ]
                }
            }
        }
    ],
    "dependencies": {
        "wasmer/winterjs": "1.2.0"
    },
    "fs": {
        "/src": {
            "index.js": echo_server_index
        }
    },
};

let wasmerPackage = await Wasmer.createPackage(manifest);

let appConfig = {
    name: "my-echo-env-app",
    owner: "edoardo",
    package: wasmerPackage,
    default: true,
};

let res = await Wasmer.deployApp(appConfig);
console.log(res.url)
```

Features

The Wasmer SDK Javascript Package supports:

- ✅ WASI support
  - ✅ Environment variables
  - ✅ FileSystem access
  - ✅ Command-line arguments
  - ✅ Stdio
- ✅ WASIX support
  - ✅ Multi-threading
  - ✅ Spawning sub-processes
  - ❌ Networking (on the works)
- ❌ Mounting directories inside the WASIX instance
- ✅ Running packages from the Wasmer Registry
- ✅ Platforms
  - ✅ Browser
  - ✅ NodeJS
  - ✅ Deno
- ✅ Registry API
  - ✅ Create a package
  - ✅ Publish a package
  - ✅ Deploy an application

License

The entire project is under the MIT License. Please read the `LICENSE` file.