Notions avancées

- Dynamic Linking
- Introduction au WAT
- Docker et Wasm
- Exécutions parallèles avec SIMD

Dynamic Linking

Le Dynamic Linking n’est pour le moment supporté que par LLVM, ainsi ce chapitre est réalisé en langage C++ avec le compilateur Emscripten.

Qu'est-ce que le Linking

Avant d'exploiter le Dynamic Linking avec le WebAssembly, nous allons tout d'abord commencer par comprendre ce qu’est le Linking de librairie, ainsi que les avantages et inconvénients de ces méthodes.

Le Static Linking consiste à empaqueter les différentes dépendances dont votre application a besoin dans un binaire unique, qui n’aura ainsi besoin d’aucune dépendance préalablement installer sur un système pour fonctionner. Ce genre de binaire est souvent appelé un binaire portable étant donné qu’il ne nécessite aucun fichier externe afin de démarrer. En contrepartie, on se retrouve très généralement avec un binaire qui pèse plutôt lourd et qui doit être remplacé lorsqu’on souhaite le mettre à jour.

Contrairement au Static Linking le Dynamic Linking permet quant à lui d’externaliser les différentes dépendances de l’application et de les joindre à l’application lors de son exécution. Dans de gros projets, on peut retrouver des fichiers avec l'extension .DLL (Dynamic Link Library), ce sont justement les librairies que l’application va intégrer au lancement. L’avantage principal de cette approche est la possibilité de remplacer unitairement les dépendances de l’application lorsqu’elles sont mises à jour. Mais cela est couteux en termes de performances et de temps de chargement lors de l’utilisation de l’application.

Et pour finir le Dynamic Loading qui permet de la même manière que le Dynamic Linking de séparer en plusieurs binaires notre application, mais il va nous permettre tout simplement de les charger au besoin. C’est potentiellement la méthode qui nous intéressera le plus étant donné qu’il permet d’alléger la charge réseau sur notre navigateur. Et le plus gros avantage est que l’application est prête à l’emploi avant même que nous ayons chargé toutes les dépendances. Ce qui nous permettra ainsi de d’abord charger le module principal, puis de charger de manière asynchrone les modules secondaire.

Dans le cas d’utilisation du WebAssembly, étant donné que le binaire doit être téléchargé depuis internet, le poids du binaire devient ainsi une préoccupation auxquelles, le Dynamic Linking peut être une solution.

Le Linking en WebAssembly

Maintenant qu'on a vu ce qu'est le Linking de librairie, voyons comment il s'applique au WebAssembly. Nous allons nous baser sur une application afin de voir ensemble comment mettre en place les différentes méthodes.

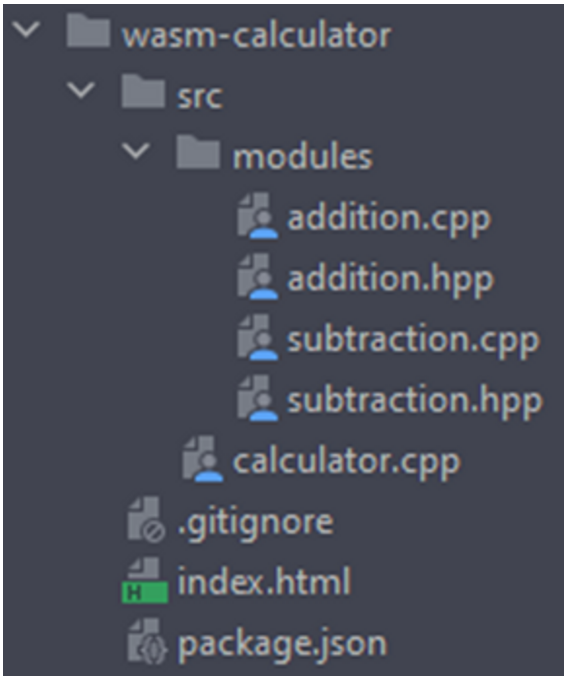
L'application sur laquelle nous allons nous baser consiste en un `Main` qui recense les différentes opérations de calcul que peut effectuer un utilisateur ainsi que les différents modules qui comporte la logique de ces opérations.

Static Linking

Étant donné que ce n'est pas l'objectif de ce chapitre, nous allons simplement voir dans cette section comment est constituée notre application de référence et faire un rappel sur Emscripten.

Composition du projet

Tout d'abord, voyons de quoi est composé le projet. On remarque deux choses, d'abord qu'il y a des fichiers `.cpp` qui servent pour notre application en WebAssembly.



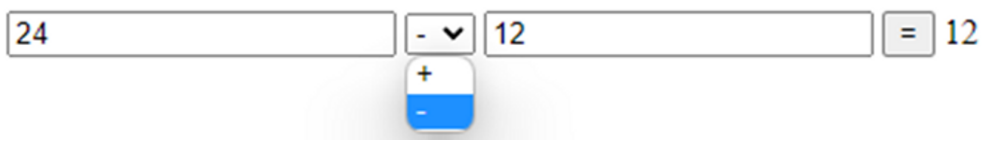
On retrouve également un fichier « `package.json` » qui correspond au fichier de configuration d'un projet Node, celui-ci va nous aider à lancer la compilation de notre application.

```
"scripts": {
  "serve": "serve",
  "build": "em++ --bind -s EXPORT_ES6=1 -o calculator.out.js src/modules/subtraction.cpp ./src/modules/addition.cpp src/calculator.cpp"
},
```

Si on regarde dans le `package.json`, on retrouve la commande de compilation de notre WebAssembly qui va permettre après une réadaptation de changer notre méthode de Linking. On

trouve aussi la commande `serve` qui sert à lancer un serveur HTTP permettant de servir les différents fichiers, ce qui peut être nécessaire étant donné que certains navigateurs ne peuvent accéder à des fichiers statiques qu'au travers d'un serveur.

Dans le fichier index.html, on retrouve notre affichage qui se présente sous la forme de deux entrées de texte, une dropdown pour choisir l'opérateur, un bouton égal et une zone d'affichage pour notre résultat.



Le fichier index.html est composé d'un module JavaScript qui se charge d'importer le fichier généré par le compilateur et de mettre à disposition du front, une fois notre module charger, l'instance de notre module ainsi que la fonction permettant de faire appel à notre module.

```
import calculateModule from "./calculator.out.js";

calculateModule().then(instance => {
    console.log("WASM module loaded");
    window.calculate = calculate;
    window.instance = instance;
});
```

La fonction `calculate` est très simple, elle récupère les entrées utilisateur depuis le DOM, fait appel à notre fonction de calcul écrite en WebAssembly et affiche la valeur de retour à l'utilisateur.

```
function calculate() {
    let aInput = document.getElementById('a').value;
    let bInput = document.getElementById('b').value;
    let operator = document.getElementById('operator').value;
    let result = document.getElementById('result');
    result.innerHTML = `<span>${instance.calc(parseInt(aInput), parseInt(bInput), operator.charCodeAt(0))}<span>`;
}
```

Rappel Emscripten

Faisons un bref rappel de comment on utilise Emscripten avec la commande que l'on utilise pour notre application.

```
em++ --bind -s EXPORT_ES6=1 -o calculator.out.js src/modules/subtraction.cpp
./src/modules/addition.cpp src/calculator.cpp
```

Tout d'abord, il faut voir l'outil em++ comme un équivalent de g++ mais qui compile exclusivement nos fichiers sources pour du WASM. Donc pour compiler un binaire cela se passe de la même manière, on spécifie les fichiers qui doivent être compilé et on spécifie le fichier de sortie.

- *-o FICHIER{.wasm, .js, .html}* : ce qui différencie cet argument avec celui de g++ c'est qu'au lieu de spécifier un fichier object, on spécifie soit un fichier .wasm, .js ou .html. Ce qui génère tous les fichiers nécessaires (-o main.html génère le fichier html ainsi que .js et .wasm), ainsi dans notre cas, on spécifie qu'on souhaite générer les fichiers js et wasm.
- *--bind :* permet de dire au compilateur qu'on a des fonctions que l'on aimerait rendre disponible pour une utilisation dans notre code JavaScript.
- *-s EXPORT_ES6* : permet de générer les fichiers .js en ES6.

Il ne faut pas oublier de spécifier quelles fonctions on veut exporter ainsi que le nom par lequel on pourra les appeler dans notre code JavaScript. D'abord, on importe la librairie d'emscripten `<emscripten/bind.h>` puis comme sur l'image qui suit on spécifie la fonction que l'on veut exposer.

```
EMSCRIPTEN_BINDINGS(calculator) {
    function("calc", &calculate);
}
```

Dynamic Linking

Afin de lier nos modules à notre application, on va devoir faire appel aux arguments fournis par Emscripten, Si tout se passe bien, on ne devrait avoir à faire aucune modification dans notre code pour transformer en module WASM nos fichiers source.

Avant de commencer, veillez à garder la commande de compilation originale, car on va devoir la retravailler pour créer notre module principal.

Pour ce faire, on va d'abord commencer par déclarer les modules secondaires en séparant en plusieurs commandes pour chaque module que l'on va créer et en ajoutant l'argument qui désigne notre binaire WASM comme étant un module secondaire.

`em++ -s SIDE_MODULE=1 ./src/modules/subtraction.cpp -o subtraction.out.wasm`

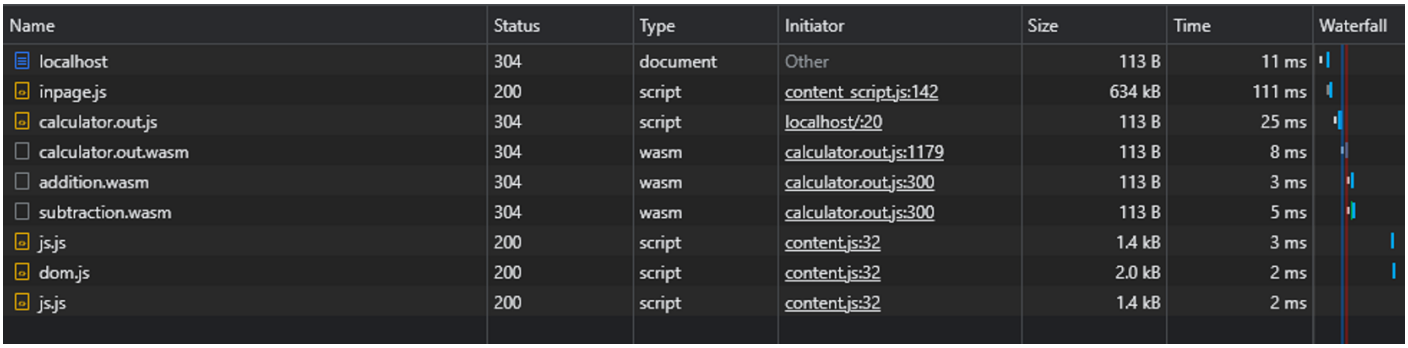
`em++ -s SIDE_MODULE=1 ./src/modules/addition.cpp -o addition.out.wasm`

une fois les modules compilés avec les commandes précédentes, on va reprendre la commande originale et on va simplement déclarer que c'est devenu un module principale grâce à l'argument `-s MAIN_MODULE=1` puis pour que le module sache quelle sont les librairies qu'il doit intégrer lors de l'initialisation, on doit définir les fichiers .wasm précédemment compiler dans notre commande.

On devrait se retrouver avec une commande tel que la suivante.

`em++ --bind -s EXPORT_ES6=1 -s MAIN_MODULE=1 -o calculator.out.js src/calculator.cpp addition.wasm subtraction.wasm`

Maintenant compilons le module principal et voyons ce qui se passe dans le navigateur.



Name	Status	Type	Initiator	Size	Time	Waterfall
localhost	304	document	Other	113 B	11 ms	
inpage.js	200	script	content_script.js:142	634 kB	111 ms	
calculator.out.js	304	script	localhost/:20	113 B	25 ms	
calculator.out.wasm	304	wasm	calculator.out.js:1179	113 B	8 ms	
addition.wasm	304	wasm	calculator.out.js:300	113 B	3 ms	
subtraction.wasm	304	wasm	calculator.out.js:300	113 B	5 ms	
js.js	200	script	content.js:32	1.4 kB	3 ms	
dom.js	200	script	content.js:32	2.0 kB	2 ms	
js.js	200	script	content.js:32	1.4 kB	2 ms	

On voit à présent que lors du chargement de la page, que tous nos fichiers WASM ont été téléchargés afin de lancer notre application.

Dynamic Loading

Nous allons désormais nous intéressez à la partie qui va nous permettre de réduire l'utilisation du réseau en téléchargeant nos modules au besoin.

Dans nos modules, nous allons déclarer les fonctions que l'on veut exporter avec l'instruction `extern`.

```
extern "C" int subtraction(int a, int b);
```

Contrairement au Dynamic Linking, on va devoir apporter des modifications à notre module principal. On va notamment faire usage de la fonction **`dlopen()`** cette fonction va nous permettre de remplacer nos instructions *`include`* et ainsi de chargé une fois appelé le module spécifié dans la fonction.

Étant donné que l'on doit retirer nos `*include*`, les signatures des fonctions doivent être définies dans notre module principal, autrement, on ne sait pas dans notre module principal comment nos fonctions doivent être appelé.

```
7   typedef int (*add_function)(int, int);
8   typedef int (*sub_function)(int, int);
```

Dans le code suivant, nous faisons un appel à la fonction **`dlopen()`** et nous récupérons un pointeur sur un `handle` vers la librairie que nous venons de charger. Enfin, on récupère grâce à la fonction `**dlsym()**` l'adresse de notre fonction que l'on va `cast` avec la signature définie précédemment.

```
10  int addition(int a, int b) {
11      void *handle = dlopen("assets/modules/addition.out.so", RTLD_NOW);
12      sub_function add = (add_function)dlsym(handle, "addition");
13      return add(a, b);
14  }
```

Une fois les modifications faites dans notre code, les commandes doivent être réadaptées pour le Dynamic Loading.

Dans nos commandes permettant de compiler les modules secondaires, il n'y a pas beaucoup de changements, seul l'argument `-s EXPORT_ALL=1` est ajouté afin de permettre l'accès aux fonctions de ce module.

Pour ce qui est du module principal, on doit lui retirer les noms de nos modules qui permettaient d'indiquer, ils devaient être importés au lancement. Puis, nous allons ajouter à la place l'argument `--preload-file` qui prend soit un fichier ou bien un répertoire.

Il faut savoir que l'argument `--preload-file` permet d'empaqueter dans un fichier .data nos différents modules, ce qui les rend disponibles une fois le chargement du paquet terminé. Mais rien ne nous empêche d'implémenter notre propre moyen de chargement des modules en JavaScript.

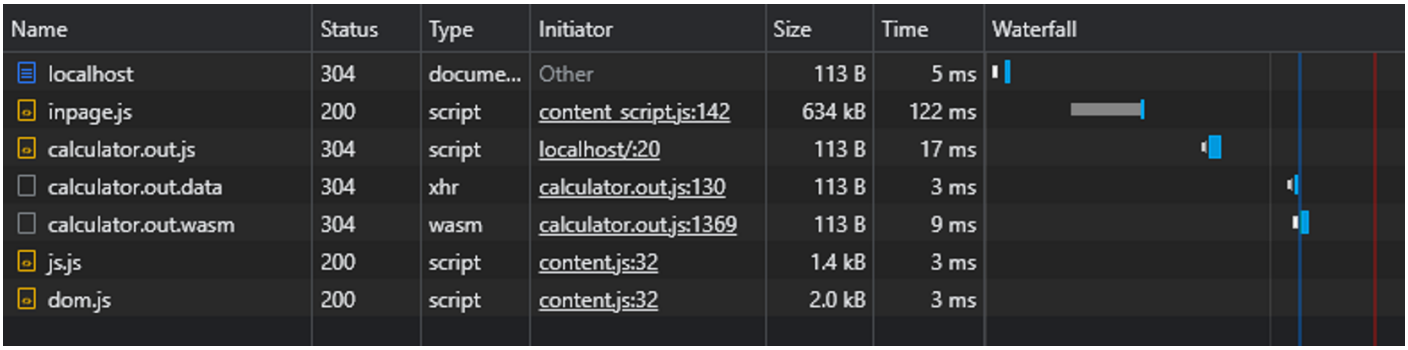
Nous retrouvons donc la commande suivante, qui permet de compiler le module principal et de fournir tous les modules supplémentaires nécessaires pour le bon fonctionnement de l'application. En effet, si nous ne spécifions pas les modules à lier au lancement, la commande va les chercher et les charger au fur et à mesure des opérations. Cela nous permet d'avoir une application plus fluide et plus efficace.

```
em++ --bind -s EXPORT_ES6=1 -s MAIN_MODULE=1 -o calculator.out.js src/calculator.cpp --preload-file assets/modules
```

Ainsi que cette commande qui permet de compiler un de nos modules.

```
em++ -s SIDE_MODULE=1 -s EXPORT_ALL=1 ./src/modules/subtraction.cpp -o assets/modules/subtraction.out.so
```

Maintenant que notre projet est prêt, compilons le et voyons ce qui se passe :



Name	Status	Type	Initiator	Size	Time	Waterfall
localhost	304	docume...	Other	113 B	5 ms	
inpage.js	200	script	content_script.js:142	634 kB	122 ms	
calculator.out.js	304	script	localhost/:20	113 B	17 ms	
calculator.out.data	304	xhr	calculator.out.js:130	113 B	3 ms	
calculator.out.wasm	304	wasm	calculator.out.js:1369	113 B	9 ms	
js.js	200	script	content.js:32	1.4 kB	3 ms	
dom.js	200	script	content.js:32	2.0 kB	3 ms	

On voit sur la capture d'écran que désormais les modules ne sont pas plus directement chargés et ont été remplacés par le fichier **calculator.out.data**

Ressources

wasm-calculator-dynamic-linking.rar

wasm-calculator-dynamic-loading.rar

wasm-calculator-static.rar

https://medium.com/@arora.aashish/webassembly-dynamic-linking-1644c9f40c8c

https://www.dynamsoft.com/codepool/webassembly-standalone-dynamic-linking-wasm.html

Introduction au WAT

Le WAT (WebAssembly Text Format) est un format de fichier qui permet de représenter du code WebAssembly sous forme de texte humainement lisible. Sa syntaxe est issue des assembleurs classiques et utilise des mnémoniques pour représenter les instructions de WebAssembly. (func, module, import ...)

Le WAT est principalement utilisé pour le développement et le débogage de code WebAssembly car il permet de lire et de modifier facilement le code source.

Il est aussi possible de se servir de WAT comme une destination de compilation pour pouvoir analyser les instructions utilisé.

Par exemple :

```
wasm-bindgen <source>.rs --out-dir <output_directory> --out-name <output_name> --target web
--no-modules
```

- `<source>.rs` est le fichier source Rust que vous souhaitez compiler.
- `<output_directory>` est le répertoire de sortie où le fichier WAT sera généré.
- `<output_name>` est le nom du fichier WAT généré.

La commande a été faite avec WASM bindgen mais il est tout à fait possible de le faire avec d'autres outils.

Le WebAssembly Text Format (WAT) est un format ouvert et indépendant de la plateforme qui peut être compilé en binaire WebAssembly (.wasm) pour être exécuté dans un navigateur ou un environnement d'exécution WebAssembly.

Il est important de noter que le WAT n'est pas un langage de programmation à part entière, mais plutôt un format de fichier pour représenter le code WebAssembly. Cependant, certaines librairies ou outils peuvent vous permettre d'écrire directement en WAT.

Le WAT présente des avantages pour le développement et le débogage, ainsi que pour l'apprentissage et la pédagogie. En utilisant le WAT, les étudiants et les développeurs débutants peuvent mieux comprendre comment fonctionne WebAssembly en étudiant le code source sous forme de texte.

Le WAT permet par d'ailleurs une meilleure compréhension des instructions de WebAssembly et de leur utilisation, ce qui peut aider à améliorer la compréhension des concepts de bas niveau tels que

la manipulation de la mémoire et les opérations de bas niveau.

Docker et Wasm

Nous avions vu précédemment qu’aujourd’hui, l’utilisation du WebAssembly ne se limite plus qu’aux navigateurs et profite d’une utilisation également côté serveur, où dans le cloud…

Docker VS WebAssembly

Les débats autour de Docker et WebAssembly sont devenus de plus en plus intenses ces derniers temps. Certains pensent que le WebAssembly va remplacer Docker, ce qui n’est pas le cas.

Le WebAssembly et Docker sont des outils complémentaires, et non des outils concurrents. En effet, le WebAssembly ne remplace pas Docker, mais peut en fait être intégré à Docker pour rendre ses applications plus rapides et plus sûres.

Le WebAssembly peut offrir une meilleure sécurité et une vitesse d'exécution plus rapide et une image Docker plus légère. Cependant, le WebAssembly est encore une technologie relativement récente, et il y a encore des contraintes de sécurité et des difficultés de débogage à prendre en compte.

À l'avenir, nous pourrions voir une plus grande adoption des applications WebAssembly dans Docker, ainsi que des mises à jour permettant plus de sécurité et de facilité de débogage. Il est possible que nous commencions à voir des applications Docker-Compose multi-services basés sur le WebAssembly, ce qui pourrait offrir une solution à certains des problèmes de scalabilité et de sécurité rencontrés aujourd'hui.

Docker With WebAssembly

Aujourd'hui, nous ne devons plus penser que le WebAssembly peut remplacer Docker, mais au contraire, s'intégrer avec ce dernier.

Docker & Linux

Aujourd'hui, toutes les images Docker partent d'une base Linux. Cette dernière peut être plus ou moins légère, mais nous avons toujours cette couche supplémentaire d'un système d'exploitation, aussi léger soit-il, qui ralentit (un peu) le démarrage d'un conteneur et sa vitesse d'exécution.

Une nouvelle très bienvenue

Fin octobre 2022, une annonce officielle tombe. Celle de la prise en charge native du WASM dans Docker. Cette fonctionnalité n’est qu’au stade de BETA, mais cette annonce va amener des changements majeurs pour le futur de Docker.

Cette prise en charge native va permettre une vitesse d'exécution beaucoup plus rapide et une image Docker beaucoup plus légère. De plus, le WASM permet de s'affranchir de l'architecture serveur et offre une meilleure sécurité.

Docker WASM

Intérêt

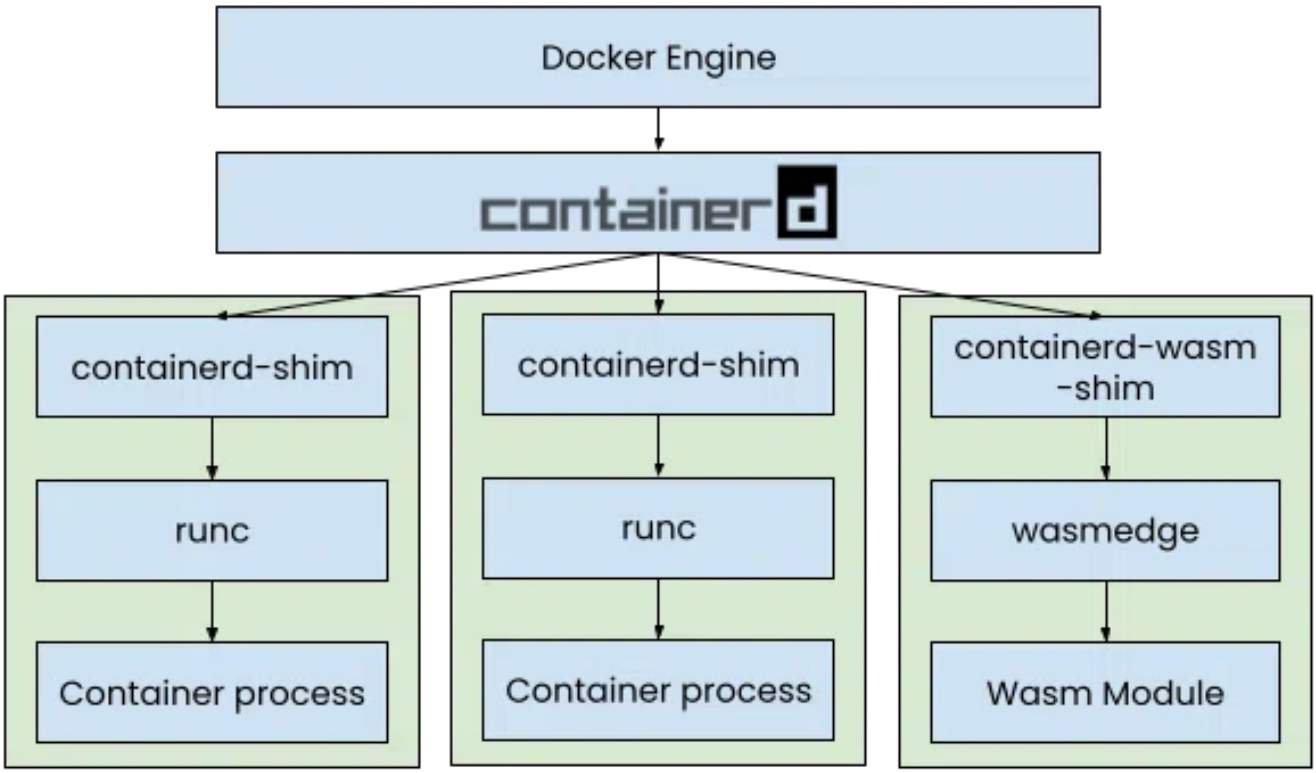
Qu’est-ce que cela va apporter à Docker ?

- Plus de sécurité
- Une vitesse de démarrage / arrêt beaucoup plus grande
- Des images beaucoup plus légères
- Libéré de l’architecture du serveur

Très utile donc dans un environnement Cloud qui nécessite du scaling très rapide, de la légèreté et vitesse de démarrage (coucou les lambda)

Cependant, l'utilisation de WebAssembly dans un environnement Docker est encore très limitée, car le WebAssembly est encore une technologie relativement récente. Il n'y a pas encore beaucoup de bibliothèques disponibles pour le WebAssembly, et les contraintes de sécurité sont toujours une préoccupation. De plus, il est encore très difficile de déboguer du code WebAssembly, ce qui est une tâche courante dans le développement logiciel.

Runner containerd



Le WebAssembly fonctionne dans Docker grâce au moteur d'exécution Runner containerd. Ce moteur est responsable de l'exécution des conteneurs et des microservices gérés par Docker. Il prend en charge plusieurs formats d'images, dont l'image WebAssembly, qui est compilée à partir du code source.

Une fois compilée, l'image WebAssembly est envoyée au moteur d'exécution Runner containerd, qui la charge et l'exécute.

Le runner WebAssembly actuel de Docker supporte wasm-edge, et pourrait supporter d'autres formats à l'avenir.

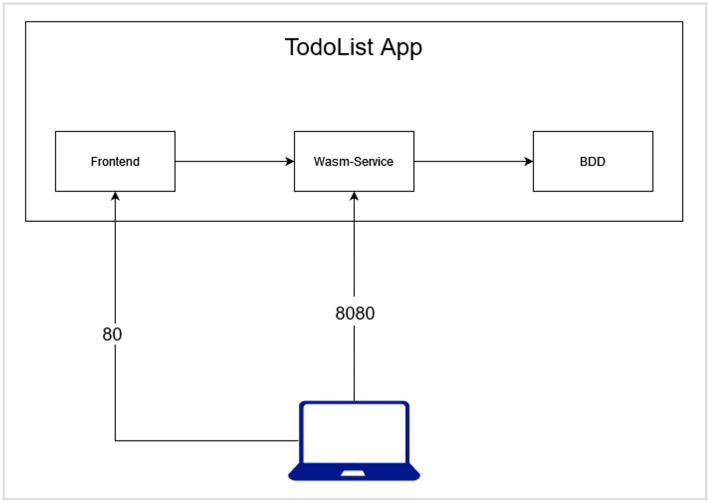
TD Énoncé

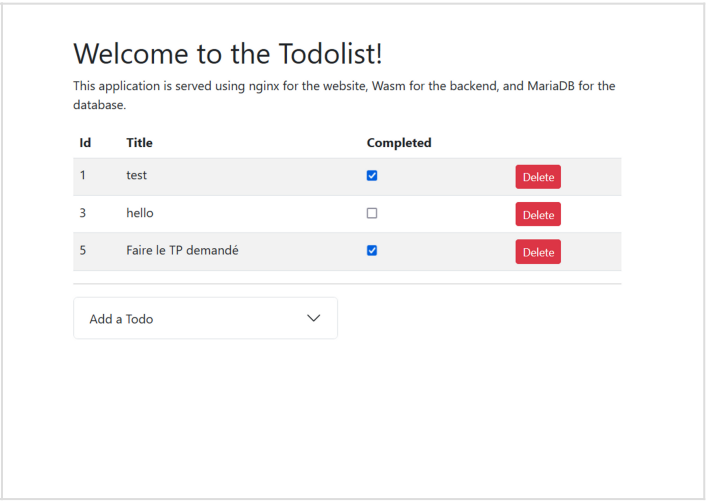
Ressources

“https://docs.docker.com/desktop/wasm/”

“https://github.com/second-state/wasmedge-containers-examples/blob/main/http_server_wasi_app.md”

“https://nigelpoulton.com/getting-started-with-docker-and-wasm/”





TD - Correction

TD - Performances & Comparaison

Conclusion

Le WebAssembly offre un grand nombre d’avantages pour l’utilisation de Docker. Il permet une vitesse d'exécution plus rapide, des images Docker plus légères et de s'affranchir de l'architecture serveur. Néanmoins, le WebAssembly est encore une technologie relativement récente, et il y a encore des contraintes de sécurité et des difficultés de débogage à prendre en compte.

À l'avenir, nous pourrions voir une plus grande adoption des applications WebAssembly dans Docker, ainsi que des mises à jour permettant plus de sécurité et de facilité de débogage. Il est possible que nous commencions à voir des applications Docker-Compose multi-services basés sur le WebAssembly, ce qui pourrait offrir une solution à certains des problèmes de scalabilité et de sécurité rencontrés aujourd'hui.

Ressources

todolist-rust-mysql-linux.zip

todolist-rust-mysql-wasm.zip

todolist-rust-mysql performances.zip

https://thenewstack.io/when-webassembly-replaces-docker/

https://www.programmez.com/actualites/webassembly-integre-dans-docker-preversion-34570

https://www.youtube.com/watch?v=9JVV2qrp080

https://techcrunch.com/2022/10/24/docker-launches-a-first-preview-of-its-webassembly-support/?guce_referrer=YW5kcm9pZC1hcHA6Ly9jb20uZ29vZ2xlLmFuZHJvaWQuZ29vZ2xlcXVpY2tzZWFyY2hib3gv&guce_referrer_sig=AQAAACCiX71P1TpECiG4S7K4MZJhVEzhfEzieZqCXtiIepBnMPEPumRydxLFPmA-N0bXv7iersP3wh28LIU3VB1Qn9oXd8d5B6FH1GJTrXUhTjIRZDl0tW3a80BvHd4TcFGT0DFkDAmt5Iq4dxRJYknQe9lYCwZDHIbaxRyj5P4nR7G3&guccounter=2

https://docs.docker.com/desktop/wasm/

https://github.com/second-state/wasmedge-containers-examples/blob/main/http_server_wasi_app.md

https://nigelpoulton.com/getting-started-with-docker-and-wasm/

https://docs.docker.com/desktop/wasm/

Exécutions parallèles avec SIMD

Le SIMD (Single Instruction Multiple Data), qui fait partie de la topologie de Flynn, permet d'exécuter des instructions sur plusieurs données simultanément, ce qui améliore considérablement les performances des traitements. Il est très utilisé pour les traitements multimédias, comme la modification des pixels d'images ou le traitement de flux audio, et est particulièrement efficace pour les traitements nécessitant un grand nombre de données à traiter en même temps.

WebAssembly prend en charge les instructions SIMD sur 128 bits, mais rencontre des difficultés lorsqu'il s'agit de s'adapter à différentes architectures de processeur. Par exemple, sur l'architecture ARM, l'optimisation ARM NEON permet d'utiliser des registres de taille plus petite, ce qui accélère considérablement les accès. Cependant, avec l'arrivée de webGPU, ces problèmes sont moins préoccupants, car les traitements multimédias lourds peuvent être déplacés vers la carte graphique, ce qui permet d'obtenir des performances optimales.

En conclusion, le SIMD offre une puissance de traitement considérable pour les traitements multimédias, et WebAssembly permet de le faire fonctionner sur de nombreuses architectures, avec les optimisations nécessaires. Avec l'arrivée d'API telles que webGPU, la puissance de traitement peut encore être améliorée, ce qui permet d'obtenir des performances optimales pour les traitements lourds.