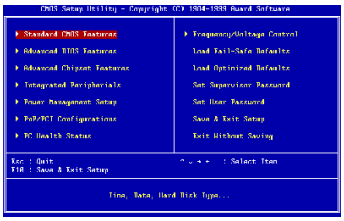
Sommaire

Les runtimes modernes

- Retour sur le concept de runtime, son positionnement
- Les contextes d'usage et les services rendus
- Runtimes et langages
- Les machines virtuelles
- Les containers
- Exemples de Java et .NET : le code intermédiaire
- Focus sur LLVM
- Les langages script et les transpilers
- Le moteur v8 JavaScript de Google
- L'architecture Node et les serveurs non bloquants
- WebAssembly
- Le cas de PHP

La course à efficacité JavaScript à la convergence des besoins et problèmes

- Le monde applicatif bascule : la grande majorité des applications ont et auront un client Web
- L'interface utilisateur :
  - Avant les années 2000 : Bios
  - 2000 à 2015 : OS
  - Depuis 2015 : HTML et JavaScript
- Ce qui compte aujourd'hui :
  - Moteur JavaScript performant (V8…)
  - API de haut niveau : Angular, React
  - Architectures distribuées : MVC, MSA…
  - Sécurité maîtrisée (JavaScript)
  - **Performances élevées**
  - Langages serveurs efficaces : PHP, ASP, Java, Node…



Ce qui compte aujourd'hui

Universalité : il faut s'éloigner de la spécificité des solutions et être le plus générique possible dans la conception des outils

Simplicité : les solutions ne doivent pas être des usines à gaz, incompréhensibles, impossibles à maintenir et garantes de futurs problèmes

Compatibilité : les solutions ne doivent pas remettre en cause l'existant, mais s'y adapter. Avec le moins de ruptures possible. Un moyen terme est à trouver entre compatibilité et dégradation de l'existant : performances, fonctions...

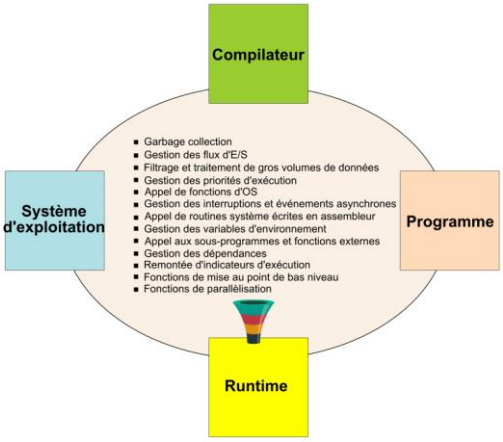
Performances : les volumes de données, le temps réel, l'immédiateté des résultats, induisent d'élaborer des solutions très performantes

Transparence : l'amélioration des comportements doit être transparente et ne pas nécessiter d'introspections lourdes de la part des usagers.

Universalité, Performances, Simplicité, Compatibilité, Transparence

Le concept de runtime et son positionnement

- Sujet dont on ne parle pas de manière générique, mais toujours en le reliant à un produit
- Les performances des systèmes modernes s'expliquent pourtant largement par les runtimes et leurs architectures d'exécution
- Le runtime est une plate-forme matérielle et/ou logicielle dédiée à un langage, à un SGBD, à une API, etc, qui lui apporte des services
- Il est sollicité uniquement pendant la phase d'exécution
- Pour un langage, le runtime est constitué de l'ensemble des "comportements" qui ne sont pas pris en charge par le programme lui-même
- Mais il y a aussi les engins liés à une API (architecture), aux bases de données (SGBD), aux compilateurs SQL (PL/SQL), aux convertisseurs de procédures Map Reduce, etc
- Nous allons prendre deux familles d'exemples :
  - Java, .NET et Python, pour leur machine virtuelle
  - Les langages script : JavaScript, PHP…



Les fondements de Java

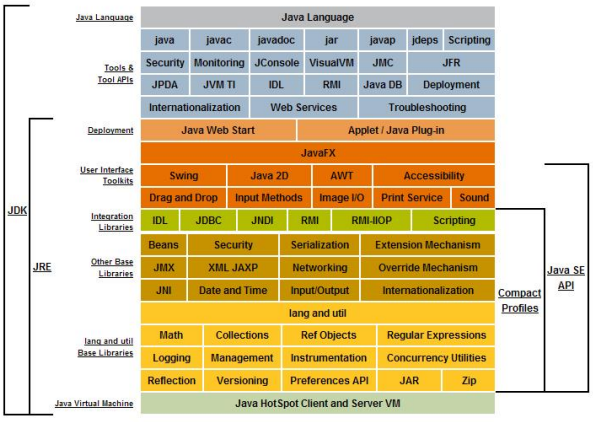
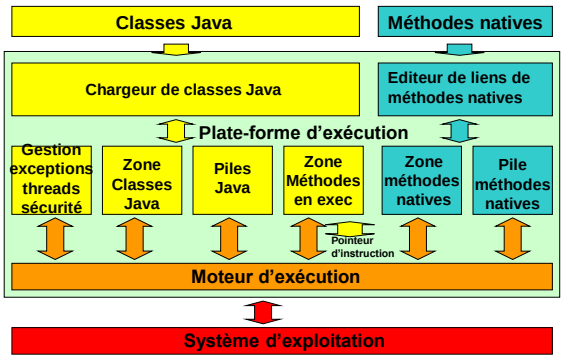


Illustration Oracle

Le découpage des fonctions assurées par l'API Java SE, le JDK et le run time JRE. Ce n'est pas toujours évident de s'y retrouver. C'est l'arrière-cuisine des développeurs…

JVM : l'ordinateur logiciel

- La JVM : Java Virtual Machine, est un espace d'exécution des programmes Java, indépendant du système d'exploitation et du matériel sous-jacent
  - Elle interprète le byte code, issu de la compilation d'un source Java (.java)
  - Elle interagit avec l'OS
  - Elle gère sa propre mémoire
  - Elle dispose de son système de sécurité
- La JVM comporte son propre jeu d'instructions, mais pas de registres utilisateurs (il n'y a que des registres système, tels que le Stack Pointer)



- Le moteur d'exécution est le cœur de la JVM : c'est lui qui traduit les bytecodes Java en instructions exécutables par le processeur physique
- Le chargeur de classes copie en mémoire les classes qui lui sont demandées à l'exécution (zone classes Java). C'est le garbage collector qui se charge de l'attribution mémoire aux objets, puis de leur récupération.
- Le traitement des données est effectué dans la zone méthodes et dans les piles. La zone méthode contient le code exécutable, les variables et les constantes.
- A chaque fois que la JVM charge une classe, elle crée un thread dédié, chacun d'eux disposant de sa propre pile (le jeu d'instructions comporte des instructions de manipulation de piles).

Le byte code Java

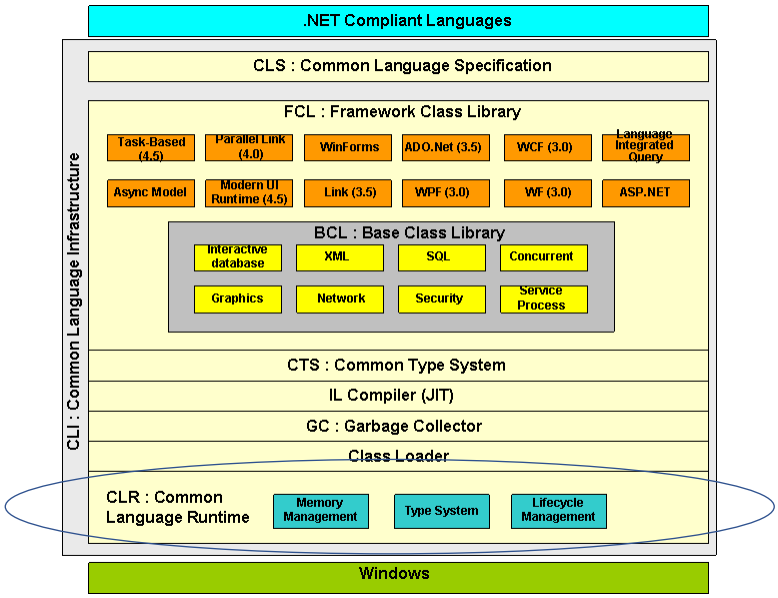
- Le byte code Java est un pseudo-code, issu de la compilation d'un programme Java (par javac par exemple)
- Il est constitué d'un peu plus de 200 instructions, qui sont regroupées en 7 grands chapitres :
  - Load et store : aload_0, istore
  - Arithmétique et logique : ladd, fcmpl
  - Conversion de types : i2b, d2i
  - Création et manipulation d'objets : new, putfield
  - Gestion de la pile opératoire : swap, dup2
  - Transfert du contrôle : ifeq, goto
  - Appel de méthode et retour : invokespecial, areturn
- La plupart des instructions ont un préfixe ou un suffixe, qui correspondent au type de données manipulées :
  - i pour integer, l pour long, s pour short, b pour byte, c pour character, f pour float, d pour double et a pour référence
  - iadd fait la somme de deux entiers
- Il n'est pas plus nécessaire de connaître la syntaxe du byte code, que de connaître l'assembleur…mais cela peut être très utile !!!

```
outer:
for (int i = 2; i < 1000; i++){
        for (int j = 2; j < i; j++) {
                if (i % j == 0)
                        continue outer;
        }
        System.out.println (i);
}
```

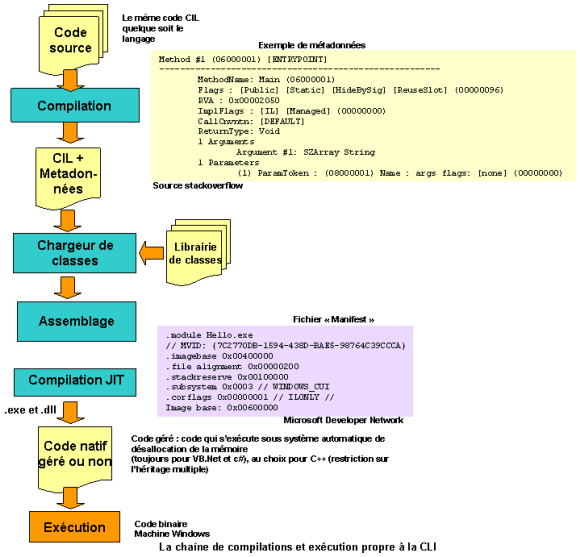
Un exemple de code java, compilé en byte code

```
0: iconst_2 //charge l'entier 2 dans la pile
1: istore_1 //charge une valeur entière dans la variable 1
2: iload_1 //charge une valeur entière depuis la variable locale 1
3: sipush 1000 //pousse le short 1000 dans la pile
6: if_icmpge 44 //test pour branchement
9: iconst_2 //charge l'entier 2  dans la pile
10: istore_2 //charge une valeur entière dans la variable 2
11: iload_2 //charge une valeur entière depuis la variable locale 2
12: iload_1 //charge une valeur entière depuis la variable locale 1
13: if_icmpge 31 //test pour branchement
16: iload_1 // charge une valeur entière depuis la variable locale 1
17: iload_2 //charge une valeur entière depuis la variable locale 2
18: irem //
19: ifne 25 //test et branchement
22: goto 38 //branchement
25: iinc 2, 1 //incrément de 1 pour la variable 2
28: goto 11 //branchement
31: getstatic #84; //Field java/lang/System.out:Ljava/io/PrintStream;
34: iload_1 //charge une valeur entière depuis la variable locale 1
35: invokevirtual #85; //Method java/io/PrintStream.println:(I)V
38: iinc 1, 1 //incrément de 1 de la variable 1
41: goto 2 //branchement
44: return //retour
```

CLI et le runtime CLR de Microsoft



La chaîne CLR de Microsoft



La chaine de compilations et exécution propre à la CLI

La machine virtuelle de Python

- Créé par Guido Van Rossum au début des années 90, l'élément clé du développement Big Data
- Lisible (pas d'accolades), Python est un langage orienté objet, interprété, multi-plates-formes, doté d'un typage dynamique fort, d'une gestion automatique des objets par ramasse-miettes et d'un excellent système de gestion des exceptions.
- Utilisé dans de nombreux frameworks et sur toutes les plates-formes.
- Extensible et adaptable par modules.
- Structures de données de haut niveau, librairies
  - Des listes : lst1 = ["claude", 255, 3.141592] ou lst2 =[["a","b","c"],[1,2,3,4]]
  - Ou des dictionnaires : dct1 = {1:"blanc", 2:"bleu", 3:"rouge}" ou dct2={4:["a","b"], 6:[1,2,3,4]}
- Le compilateur Python génère un byte code qui est interprété dans une MV
- Tout est transparent, l'usager n'a pas à compiler de manière explicite, le "bytecode" est un artefact de confort
- La PVM ("Python Virtual Machine") est un programme qui fournit un environnement paramétrable.

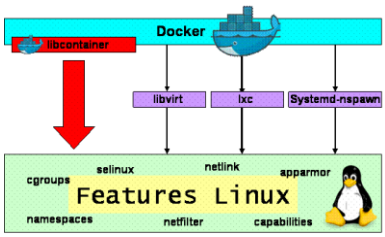

```
>>> import dis
>>> def example(x):
        for i in range (x):
            print(2*i)
>>> dis.dis(example)
  2           0 SETUP_LOOP              28 (to 30)
              2 LOAD_GLOBAL              0 (range)
              4 LOAD_FAST                0 (x)
              6 CALL_FUNCTION            1
              8 GET_ITER
        >>   10 FOR_ITER                16 (to 28)
             12 STORE_FAST               1 (i)

  3          14 LOAD_GLOBAL              1 (print)
             16 LOAD_CONST               1 (2)
             18 LOAD_FAST                1 (i)
             20 BINARY_MULTIPLY
             22 CALL_FUNCTION            1
             24 POP_TOP
             26 JUMP_ABSOLUTE           10
        >> 28 POP_BLOCK
        >> 30 LOAD_CONST                0 (None)
             32 RETURN_VALUE
>>>
```

Le module **dis** de python est le désassembleur standard qui montre la structure du bytecode Python.

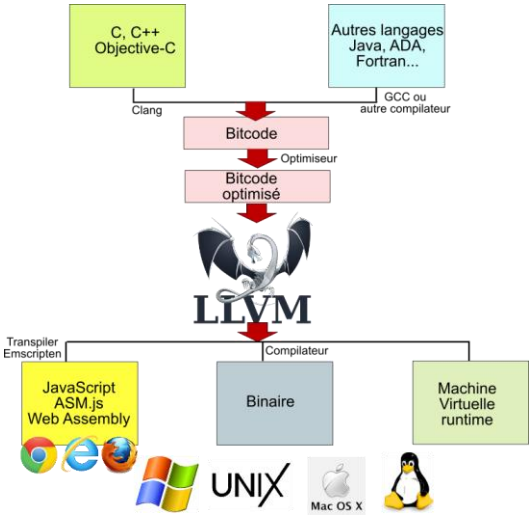
Les containers

- Diffusion d'applications "containerisées" qui vont trouver localement les ressources dont elles ont besoin
- Les ressources locales, Linux essentiellement, constituent l'engin d'exécution
- Le principe du container est de construire une boîte applicative, dans laquelle il y aura l'application, mais aussi un minimum de « features » système dont elle va avoir besoin pour s'exécuter, sans avoir à transférer le système complet lui-même.
  - Objectif : diffuser les applications par une simple commande, sur d'autres machines, sans être contraints par la lourdeur habituelle de la virtualisation
  - Victoire du « couper/coller » container contre « installer/configurer/exécuter » de la virtualisation
  - L'idée est de construire un repositiory d'images (les build exécutables), que l'on pourra transférer à la demande sur des machines distantes, dans un Cloud, sans avoir à imaginer des procédures de migrations, complexes et coûteuses à gérer.



LLVM la "boîte à outils" universelle

- LLVM (Low Level Virtual Machine) est un ensemble d'outils écrits en C++ qui permettent d'écrire du code portable et proche du langage machine à la compilation.
- C'est une « respiration nouvelle » pour tous les langages statiquement typés dont les programmes peuvent être recompilés et devenir plus performants.
- LLVM a été imaginé pour produire des outils de compilation originaux et performants, susceptibles de générer du code intermédiaire, dit « bitcode », à partir de n'importe quel langage.
- LLVM est donc au départ un projet de représentation d'un code de bas niveau, proche des machines, comme peut l'être un byte-code Java.



- LLVM facilite l'écriture des compilateurs grâce à différents modules : traducteur de code source en bitcode, compilateur de bitcode en langage machine, l'équivalent des JIT (Just-In Time), éditeur de liens (linker), un désassembleur, un outil de mise au point LLDB, un optimiseur de code, des machines virtuelles JIT, un interpréteur.
- Toute une galaxie d'outils fondés sur ce code intermédiaire de bas niveau, qui continue d'évoluer avec une communauté qui s'enrichit et constitue un véritable écosystème, avec de nombreux projets de recherche, dont une centaine estampillés « Open Source », mais aussi des réalisations concrètes chez des éditeurs…plus propriétaires.

LLVM la "boîte à outils" universelle

- Conçu pour la portabilité
- LLVM comporte des primitives indépendantes de l'architecture des machines, qui cache la diversité et complexité des plates-formes : des entiers de longueur quelconque, des moyens pour générer un code compatible avec n'importe quel jeu d'instructions
- L'usage de LLVM par les langages
  - La plupart utilisent LLVM en tant que compilateur amont (AOT : "Aead-Of-Time"), ex de Clang
  - D'autres pour le "just-in-time", compilation à l'exécution : Julia qui a besoin d'un code rapide pour interagir avec le mode REPL : "Read-Eval-Print-Loop" (prompt interactif), pour la compilation de requêtes SQL avec PostgreSQL (cinq fois plus rapides)
  - LLVM peut être utilisé pour l'optimisation du code : fonctions d'inlining, suppression de code mort, traitement des boucles ("unrolling loops")
  - Transpiler JavaScript avec le projet Emscripten (conversion de bitcode en JavaScript)
  - Pour étendre le périmètre d'un langage, sans le remettre en cause : ex du Nvidia CUDA Compiler, qui ajoute le support CUDA à d'autres langages

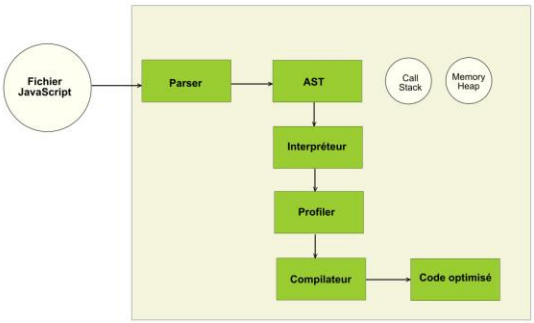
```
int arith(int x, int y, int z){
     return (x*y + z);
}
```
Code C ou Java

```
define i32@arith(i32 %x, i32 %y, i32 %z){
     entry
     %tmp = mul i32 %x, %y
     %tmp2 = add i32 %tmp, %z
     ret i32 %tmp2
}
```
Bitcode LLVM

```
public class demo.Demo {
     public static int arith(int, int, int);
          Code:
               0: iload_0
               1: iload_1
               2: imul
               3: iload_2
               4: iadd
               5: ireturn
}
```
Bytecode Java

Les engins JavaScript

- Le programme JavaScript est un flux, qui est traduit en "tokens" : mots réservés, identificateurs, littéraux, ponctuation, tout sauf les espaces
- Ce sont ces tokens qui sont envoyés au parser pour analyse et fabrication d'un arbre syntaxique (AST), dont les nœuds sont les tokens.
- L'interpréteur navigue dans l'AST et génère un byte code, code intermédiaire, qui remplace l'AST
- Le profiler surveille le code et l'optimise
- Le compilateur génère du code de plus bas niveau, compréhensible par la machine
- Optimisation

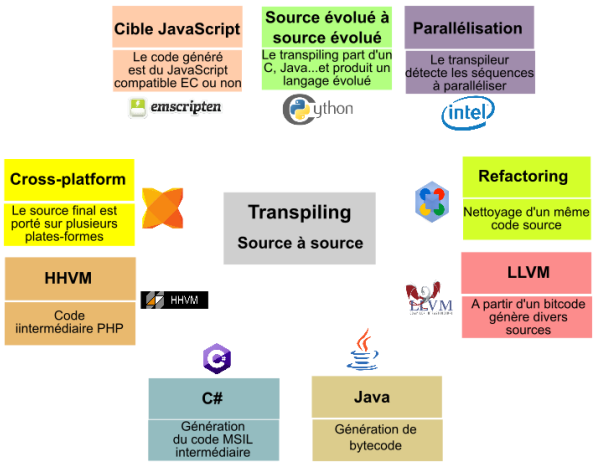


Le moteur v8 JavaScript de Google

- V8 fonctionne en mode JIT (Just In Time) et compile le code .js, en l'optimisant grâce aux techniques de « code inlining » ou de « copy elision » (technique d'optimisation qui évite de copier les objets inutiles).
- Développé pour Chrome et Chromium, pour améliorer les performances JavaScript (Lars Bak en 2008)
- Ce que fait (bien) V8 :
  - Compile le code JavaScript
  - Gère les call stacks (pour garder la position des appels de fonctions en cas d'appels multiples)
  - Gère la mémoire assignée aux programmes ("heap memory")
  - Garbage collection
  - Compatible avec tous les types de données
- Elément essentiel de V8 : "ingnition interpreter", qui compile le code JS en code machine non optimisé
- A l'exécution, le code machine est analysé et recompilé pour obtenir les meilleures performances, via deux composants clés : TurboFan et Carnkshaft
- En 2018 : V8 Lite, dont les éléments de simplification sont portés dans V8
- En moyenne : 18 % d'amélioration pour les sites Web classiques

Inlining : remplacement par le compilateur du code d'un appel de fonction par le code de la fonction appelée. Ce qui fait disparaître les appels de fonctions.
Le compilateur doit évaluer si cela présente un intérêt : la taille du code peut augmenter sensiblement, le code peut être plus lent et consommer plus de ressources.
Tout dépendra des fonctions. L'inlining n'est pas qu'une technique d'optimisation, c'est un encouragement à écrire du code optimisé, bien découpé.

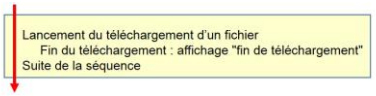
Le « transpiling » ou transpilage



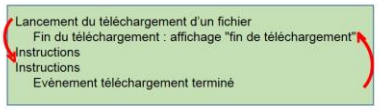
Compilation de source à source

- Remplacer un langage script par un langage de haut niveau
- Ajouter des nouvelles fonctions qui n'existent pas dans la cible
- Améliorer les performances
- Imposer de bonnes pratiques d'écriture
- Faciliter la migration d'un langage à l'autre
- Développer des langages cross-platform

Node.js : JavaScript sur les serveurs



Serveur bloquant
La suite de la séquence ne s'exécuter que lorsque le téléchargement du fichier est terminé

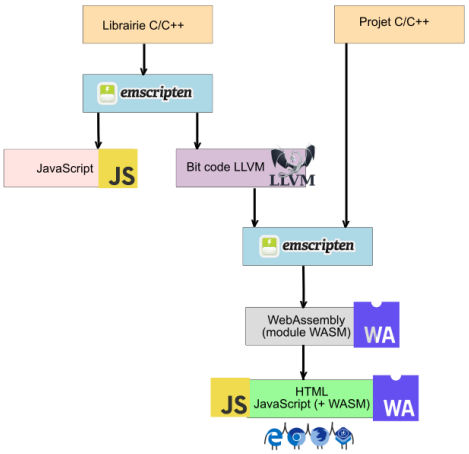


Serveur non bloquant
La suite de la séquence commence à s'exécuter, sans attendre que la requête de téléchargement soit terminée.
On peut ainsi décorréler l'exécution de la séquence avec divers requêtes qui vont continuer à s'exécuter indépendamment.

- Environnement pour développer du code JavaScript sur un serveur.
- C'est aussi une API de bas niveau, ce qui explique ses performances, mais oblige le développeur à traiter des aspects de ses programmes qu'il avait l'habitude de sous-traiter à d'autres composants de son architecture, tels que Apache.
- Node est très rapide et produit des applications Web performantes et peu encombrantes
- Deux raisons à cette performance :
  - l'usage du moteur V8 JavaScript.
  - sa faible consommation en ressources.
- Node.js ne passe pas par un serveur Web HTTP, pour traiter les demandes (requêtes) des utilisateurs.
- Avec un autre langage, le serveur HTTP traite chaque requête individuellement et crée un « thread » (processus d'exécution) pour chaque demande.
- Avec Node.js, c'est l'inverse. Il faut programmer soi-même le ballet des « request/response », mais il n'y a plus qu'un seul thread pour traiter l'ensemble des demandes.
- Ce que l'on gagne en performances, on peut le perdre en complexité, puisqu'il faut tout développer.
- Il existe d'autres API serveurs JavaScript : Jaxer (Axana), EJScript, RingoJS, AppendgineJS

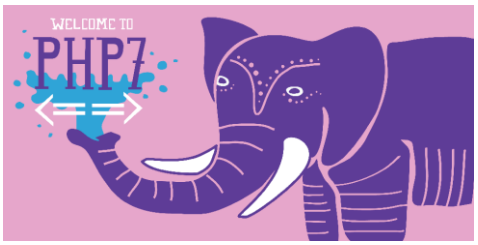
Web Assembly

- Le WebAssembly, est un formatage de bas niveau, dont l'objet est d'exécuter, à grande vitesse, des applications dans un navigateur Web
- Résultat de la compilation d'un code écrit avec un langage déjà proche de la machine, comme C, C++ ou Rust. La raison est la gestion mémoire : en JavaScript, Java et d'autres langages de « haut niveau », la gestion mémoire s'effectue via un « garbage collector » ou équivalent, qui prend à sa charge le « nettoyage » de l'espace mémoire, alors qu'avec C, C++ ou Rust, il faut tout gérer manuellement. WebAssembly fonctionne de cette façon, ce qui le rend plus facile à produire en compilation avec C, C++ ou Rust, qu'avec un autre langage.
- Très proche de JavaScript, on peut appeler des modules WebAssembly à partir d'un code JavaScript, ils sont complémentaires.
- S'appuie sur deux éléments : une machine virtuelle spécifique, capable d'exécuter du WebAssembly ET du JavaScript et une API, des fonctions que le navigateur va appeler pour faire fonctionner l'application.
- L'un des avantages du WebAssembly est qu'on peut le générer à partir de plusieurs langages de haut niveau : les langages proches, mais aussi d'autres qui font l'objet de nombreux projets.
- On peut par exemple écrire du code C, C++ ou Rust, que l'on va compiler avec Emscripten ou d'autres plates-formes LLVM.
- Le transpiler TypeScript, peut aussi produire du WebAssembly.
- En Java, il faut s'intéresser à Bytecoder et surtout à TeaVM, qui traduit du bytecode Java en WebAssembly, de même qu'à partir du code intermédiaire Kotlin.
- Chez Google, Go est compatible.



PHP chez Facebook, une affaire de run time

- PHP : 81 % des sites Internet
- Langage script côté serveur pour le développement de sites Web dynamiques
- Des milliers de classes disponibles
- Deux grands acteurs : Zend Technologies (racheté par Rogue Wave) et Facebook
- Le problème de PHP, comme Java : chaque requête génère un "thread" : consommation élevée de CPU
- Facebook HiHop en 2008, un transpiler PHP 5.X qui génère du binaire C++ compilé
  - Performances x 2, mais temps de compilation élevés et différences entre les plates-formes
- Puis en 2011 : HHVM ("HipHop Virtual Machine)
  - Transpiler PHP en JavaScript, exécutable dans un moteur v8.
  - Le couple PHP/JavaScript est plus performant que C++…
  - Compilation JIT : performances améliorées de 100 %.
  - Problème : les techniques de "garbage collector" de PHP v8 ne sont pas les mêmes.
  - Conséquence, Facebook écrit son propre JIT, ce qui donne naissance à HHVM, une machine virtuelle, le code PHP étant transformé en HHBC : HipHop Byte Code, stocké dans une cache, avec une structure SQLite.
  - Processus JIT, quand c'est nécessaire : 100 % d'amélioration des performances
  - Evolution : PHP est remplacé par Hack, dérivé de PHP, avec typage statique, expressions lambda, des collections de types standard :Vector, Map, Set, Pair, gestion des valeurs "Null"…
  - Coexiste avec PHP dans la machine virtuelle



Le moteur Zend de PHP

- Zend : première version du Zend Engine en 1999 avec PHP 4 : écrit en C, très performant, gestion des ressources dont mémoire
- Deuxième version avec PHP 5, Zend Engine II
- 2014, Andi Gutmans, l'un des deux fondateurs de Zend, avec Zeev Surasky, lance son propre JIT PHP, accompagné d'un gros nettoyage des API « core » et d'une forte amélioration de l'usage de la mémoire, via des modifications sur les structures internes des données.
- Baptisé PHPNG (pour « New Generation »), il sert de base aujourd'hui à PHP 7
- Le Zend Engine actuel, Zend Engine III, est à la fois un compilateur et un runtime.
- Les scripts PHP sont chargés en mémoire et compilés en opcodes (255 fonctions PHP pré-définies)