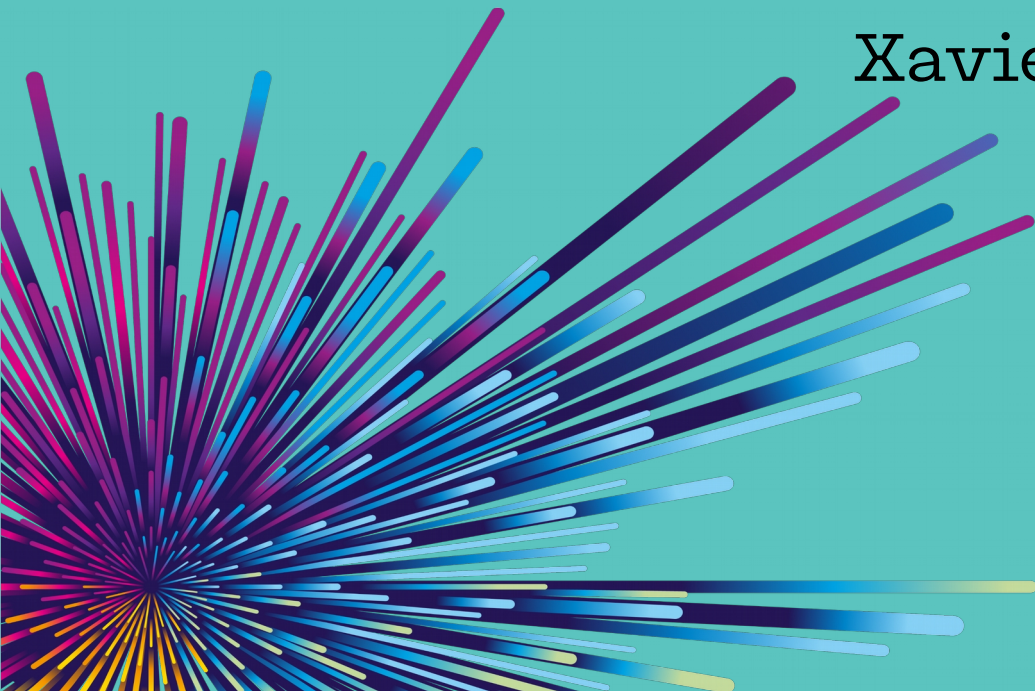


Les Unikernels , un des futurs potentiels de la conteneurisation

Xavier Van de Woestyne

Margo Bank





Margo Bank
work.margo.com



- Xavier Van de Woestyne
- Bruxelles, Lille, Paris
- Lambda Lille

J'aime bien la programmation, idéalement avec un langage **fonctionnel** et impératif, **strict**, avec un système de types expressif, des **objets** et du **sous-typage structurel**, un **système de modules évolué** et un mécanisme d'**inférence** «puissant»*. (Et beaucoup d'autres choses.)

* Il n'existe probablement qu'un seul langage adapté à cette description...



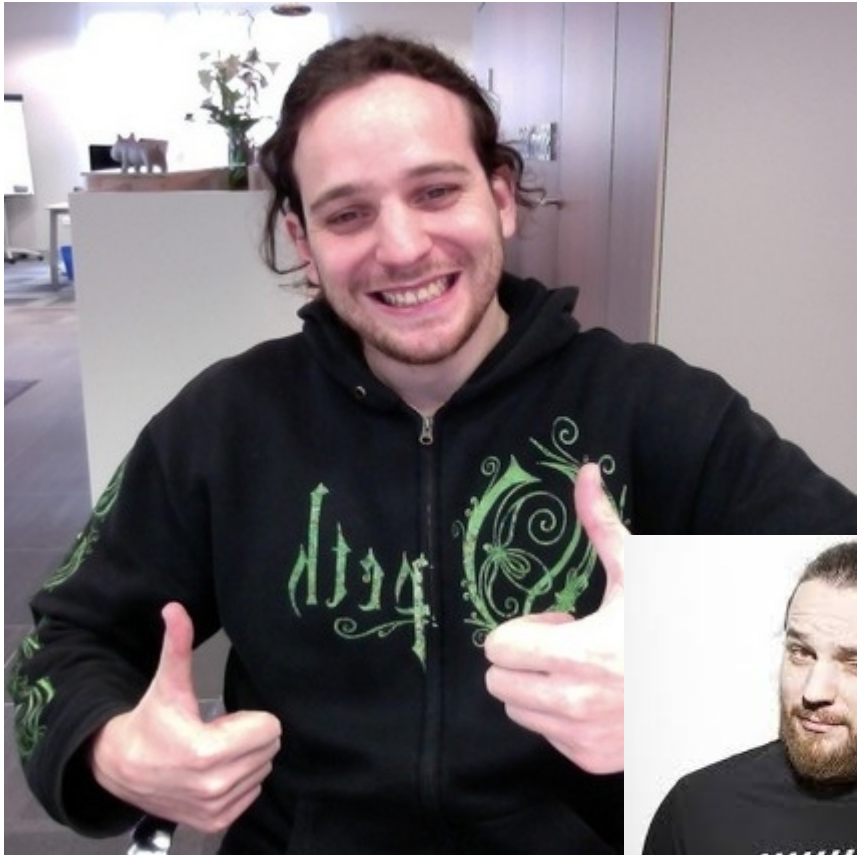
@vdwxv - <https://xvw.github.io> - merveilles.town/@xvw

Disclaimers

- Je suis une quille en Devops/SRE
- Cette présentation est en dehors de mon périmètre habituel
- Je n'ai travaillé avec des Unikernels que durant mon temps libre, pour des projets personnels

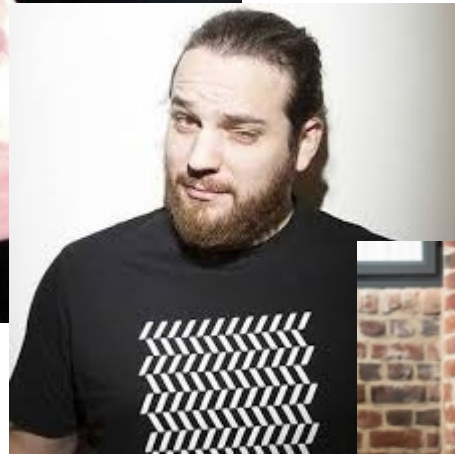
~5 years ago

- Premier talk à **Takeoff Talks!** (Comment les formalistes russes et la programmation fonctionnelle peuvent nous aider à construire des histoires « potentiellement » infinie)
- La boucle est bouclée !



Merci bro'

De me permettre de faire la promotion, une fois de plus, de OCaml <3



Sommaire

- Développer une application au 21ème siècle !
- Des VM's aux unikernels
- Un exemple : MirageOS
- Conclusion

Une application web (en PHP) au 21ème siècle : La tour de Babel

métaphore empruntée à **Romain Calascibetta** à **Lambda World 2018**



Développeur

PHP

Symfony

Composer

OpenSSL Apache

Aptitude LibC

ZLib SSH Vi Emacs

ALSA SystemD

Le Kernel Linux



- Les logiciels sont construits **localement**
- Mais déployé à **distance**

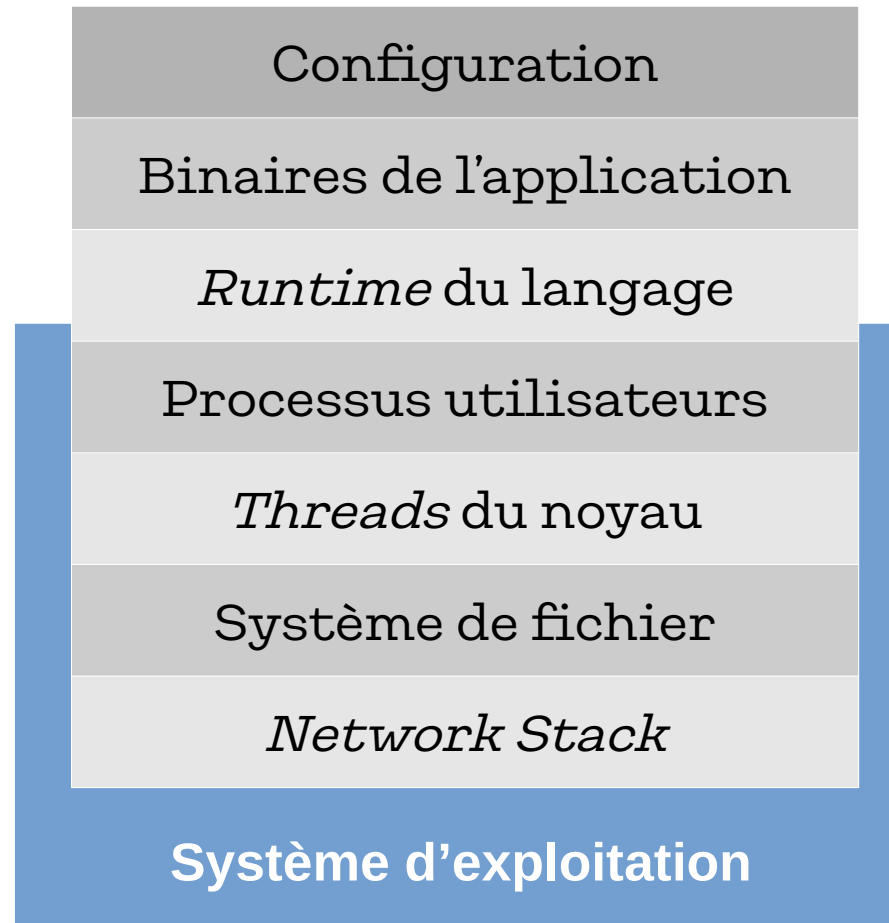
- La stack complète devient très complexe à comprendre
- Celui qui comprend tout est un véritable **Fullstack Developer** (non euclidien)

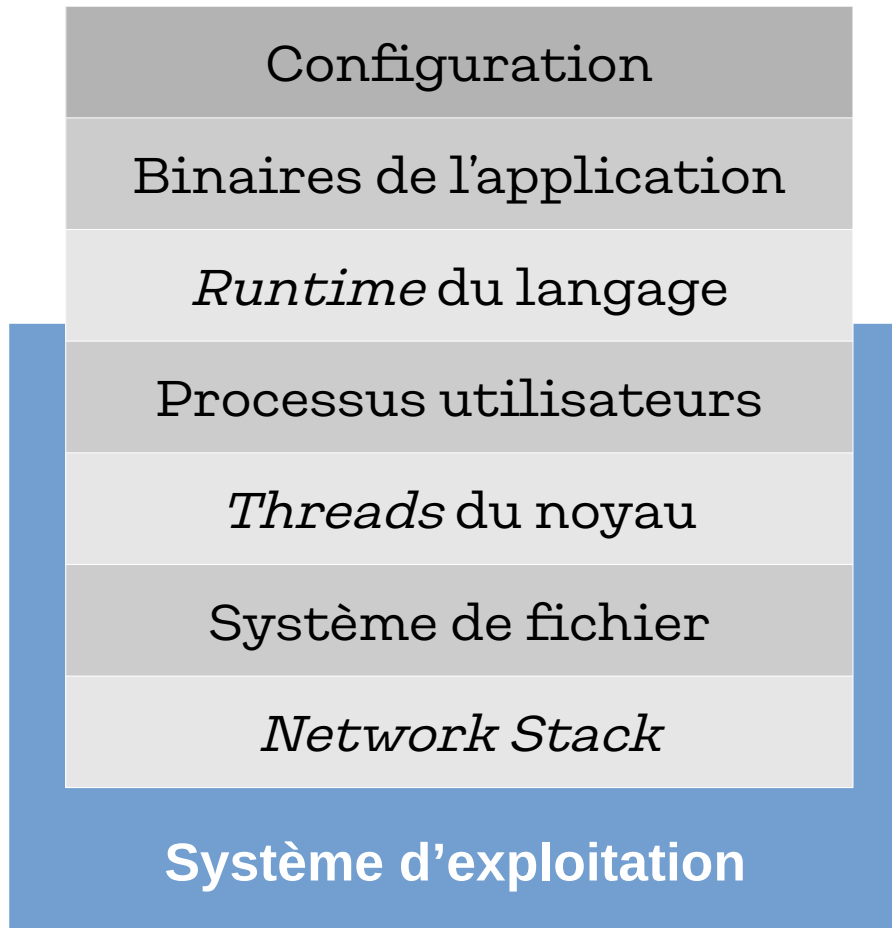
Une solution ?




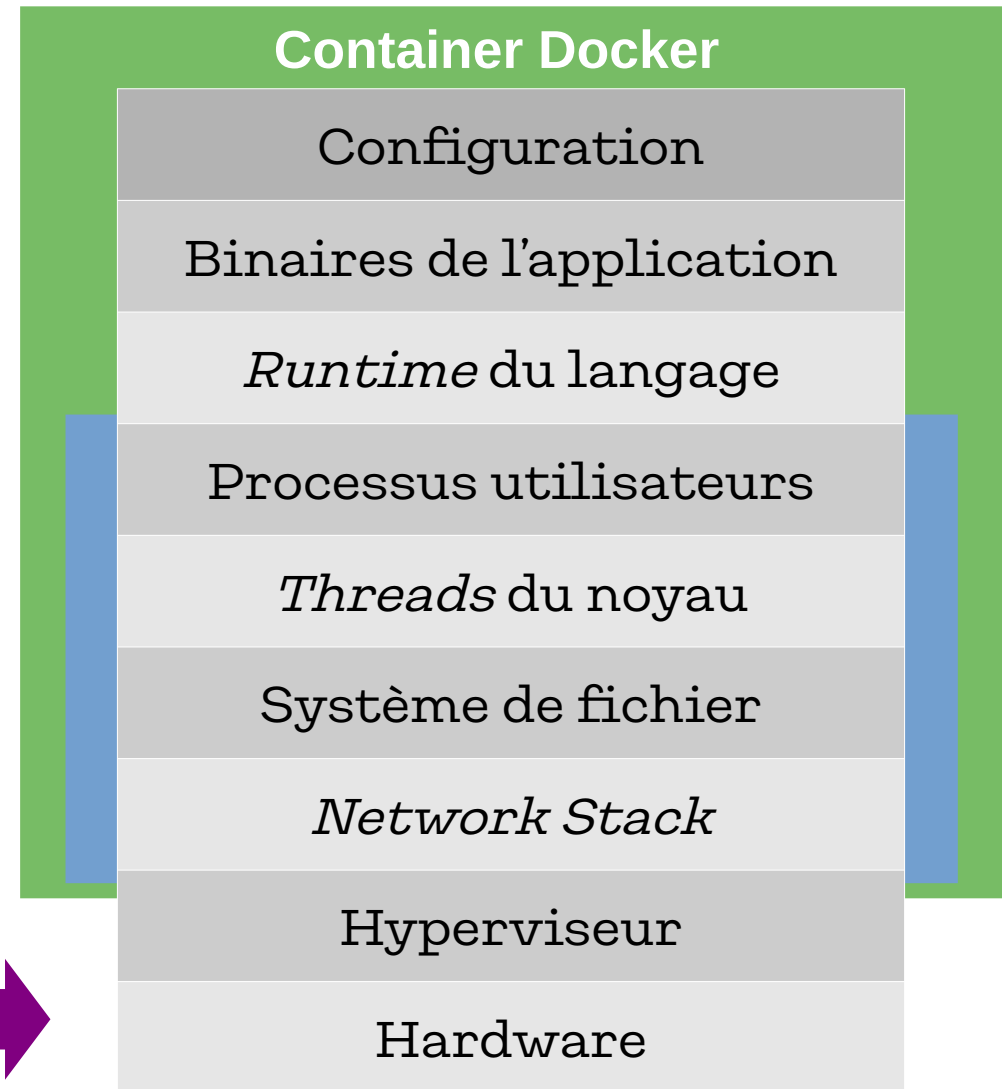


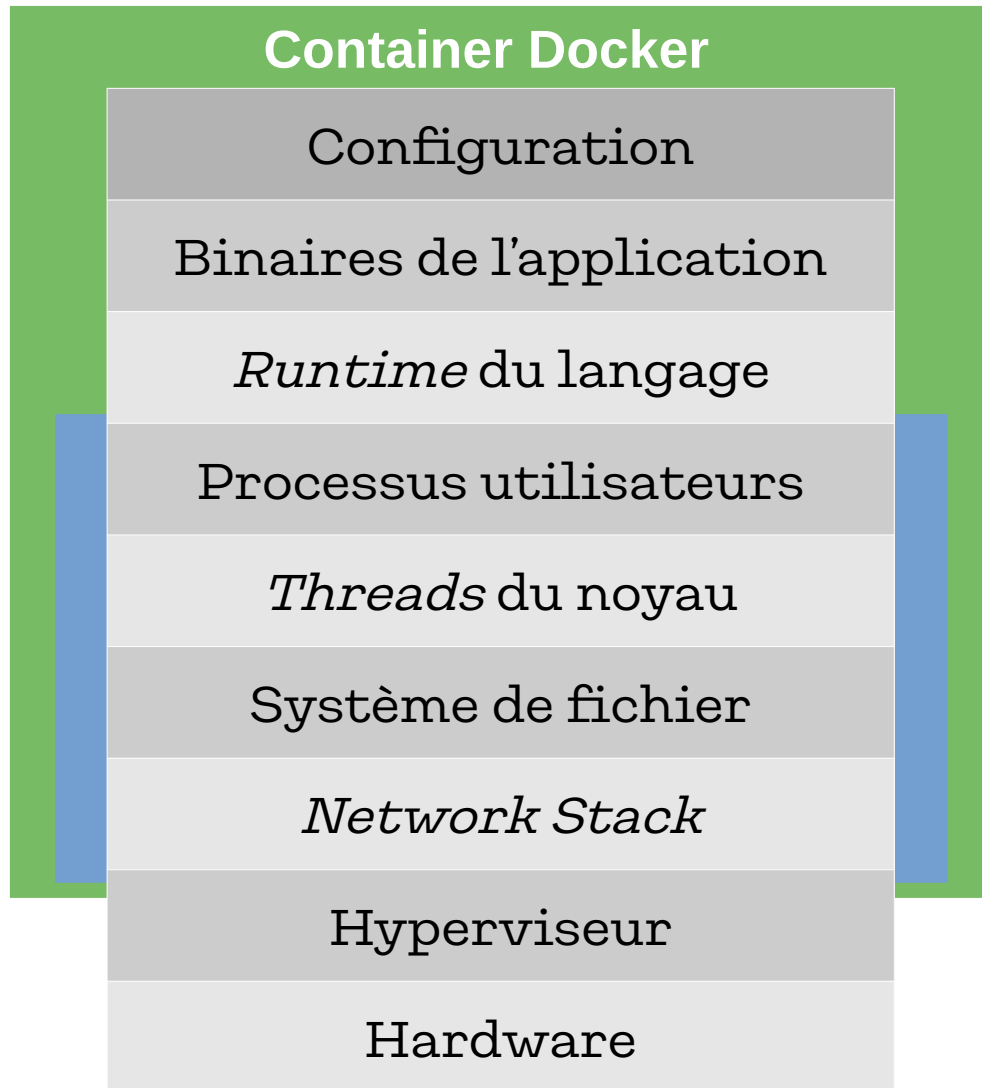
It works on my machine...



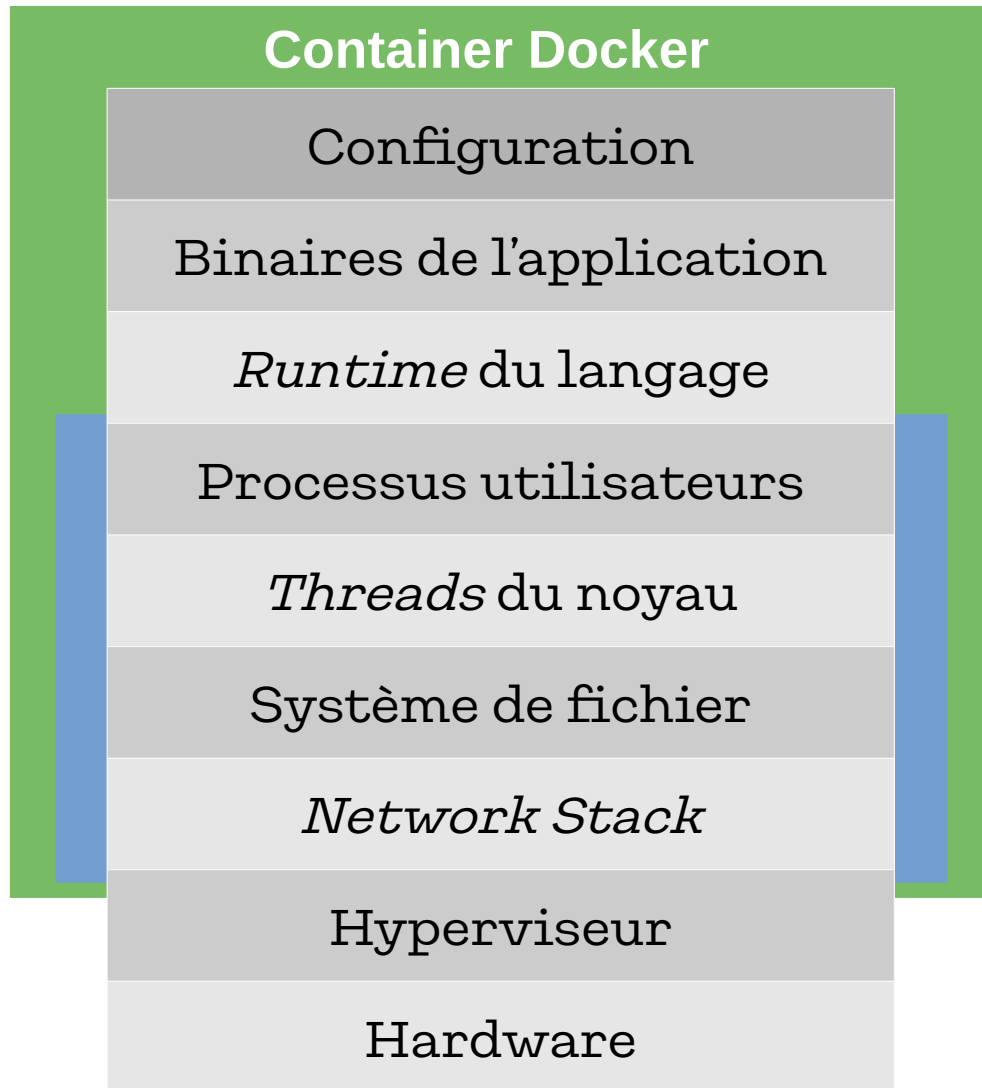



Docker





- Isolation via des espaces de noms
- Simili virtualisation via **Runc, LibCRT, LXC** ou **SystemD-nspawn**

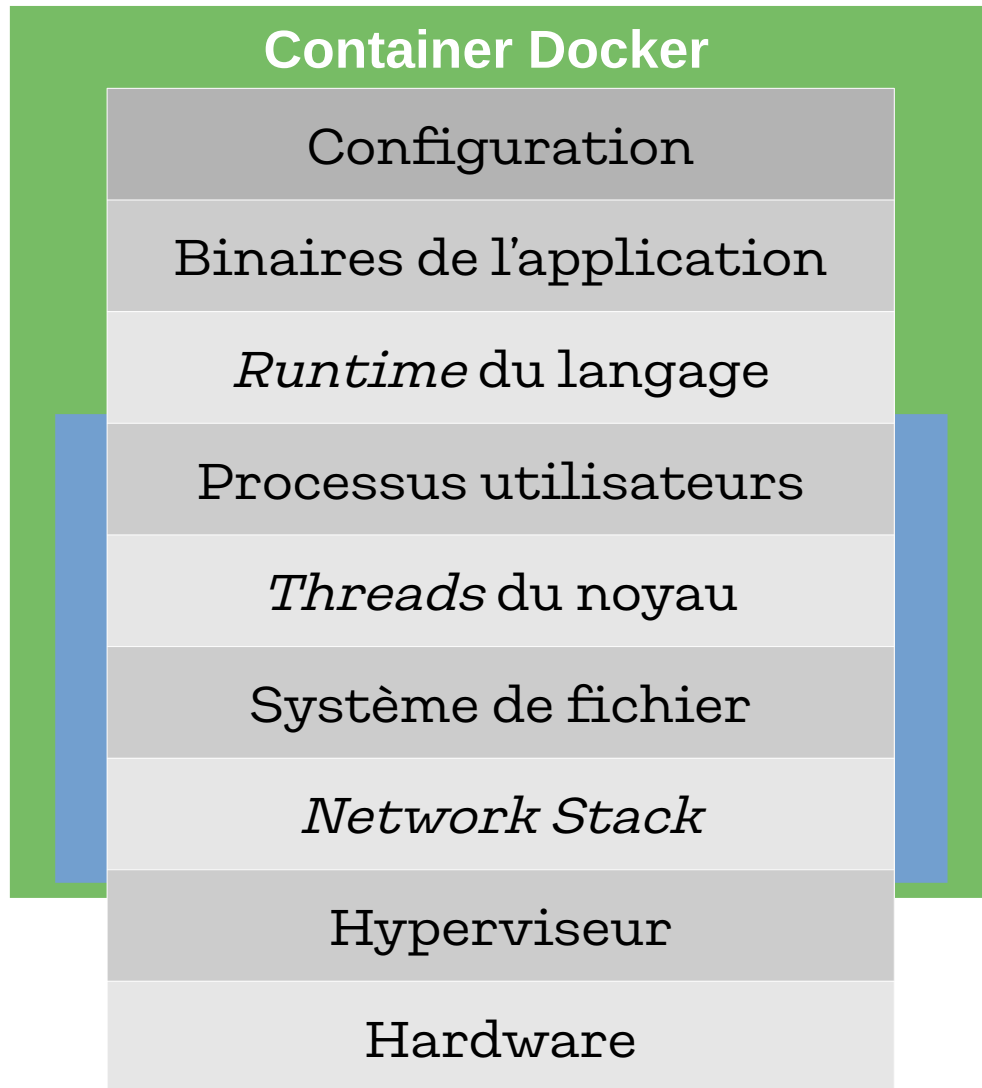


- Isolation via des espaces de noms
- Simili virtualisation via **Runc, LibCRT, LXC** ou **SystemD-nspawn**

+ de portabilité

+ de flexibilité

+ de légèreté qu'une VM



Construire (et déployer) un logiciel est complexe !

Pourtant, une majeure partie des logiciels que l'on déploie est ***single-purpose***

Dans les grandes lignes

- On *conteneurise* une application

Dans les grandes lignes

- On *conteneurise* une application
- On virtualise le *container* via l'hôte

Dans les grandes lignes

- On *conteneurise* une application
- On virtualise le *container* via l'hôte
- On l'exécute dans l'hôte



Forces

- Contexte d'exécution en développement isomorphe au contexte d'exécution final
- Rapide a *booter* (bien plus qu'une VM classique)
- Bien documenté, orchestrable etc.

Faiblesses

- POSIX est un système général (qui fait « trop »)
- Le code du *Kernel* est gigantesque et populaire (donc doté d'une surface d'attaque gigantesque)
- Duplication de ce que nous offre Linux
- Trop de partages (Kernel, Mémoire, *FileSystem*, Matériel)
- Pour les quilles, comme moi, l'écriture de Dockerfile...

Où peut être exécuté Linux ?

Partout.

Que peut exécuter Linux ?

Tout.

Linux est un OS « ultra compatible », même si l'on veut exécuter une seule application, avec un seul utilisateur...

Compatibilité > Efficacité



VM's classiques

- Plus sécurisées (parce que moins de partages)
- OS hétérogènes

Containers

- Moins lourd (donc plus facile à booter)

Une perspective de solution

- Simplification des couches de l'application
- Réduction de la surface d'attaque en limitant les capacités du contexte d'exécution, en élaguant ce qui n'est pas utile :
 - Certains *drivers*
 - Les utilisateurs OS
 - La vérification constante des permissions (par exemple)

Le meilleur des deux mondes

<< potentiellement >>

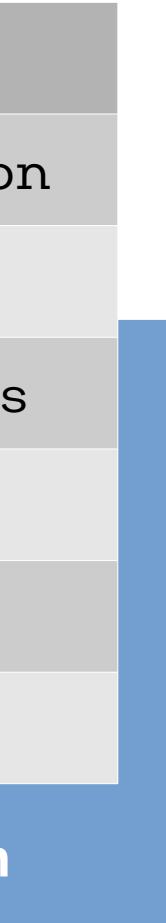
Les Unikernels



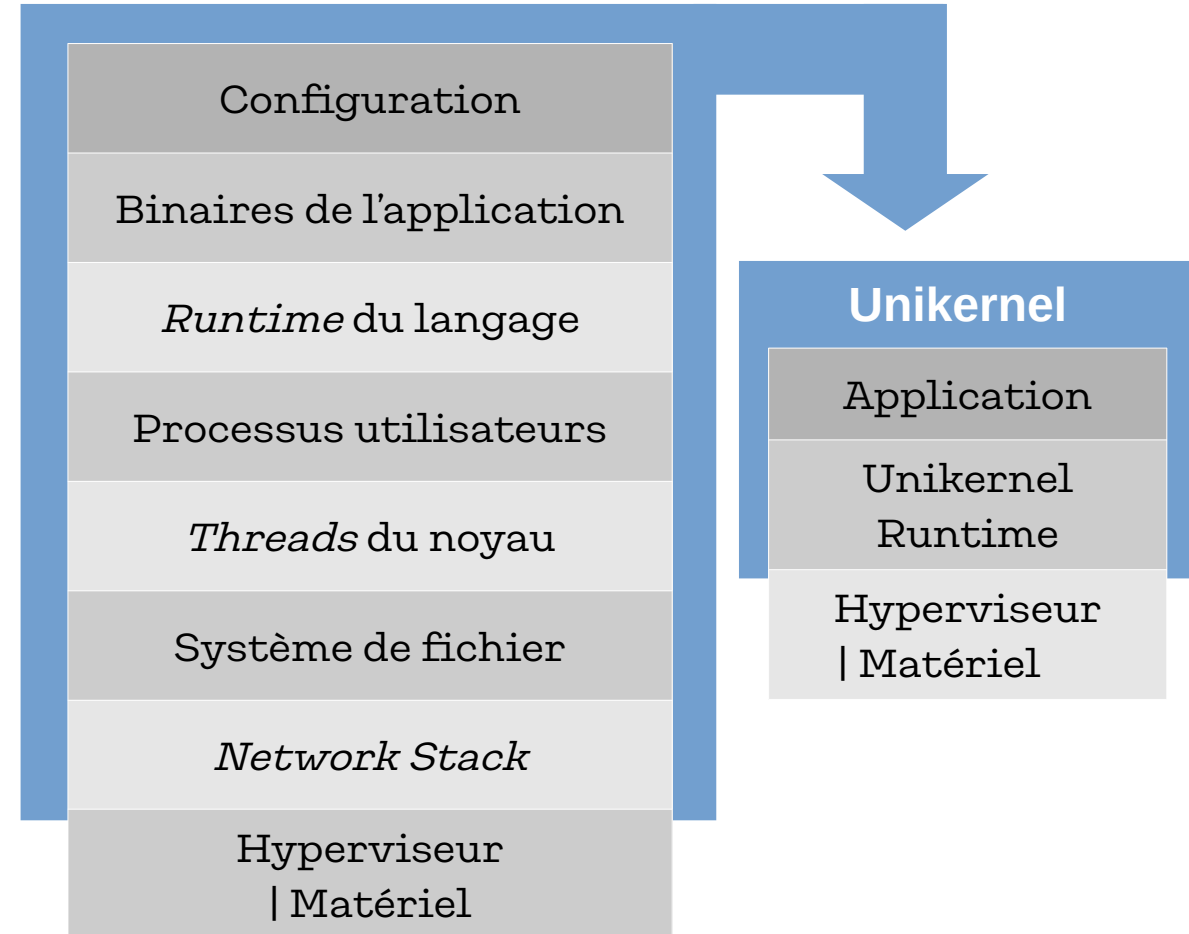
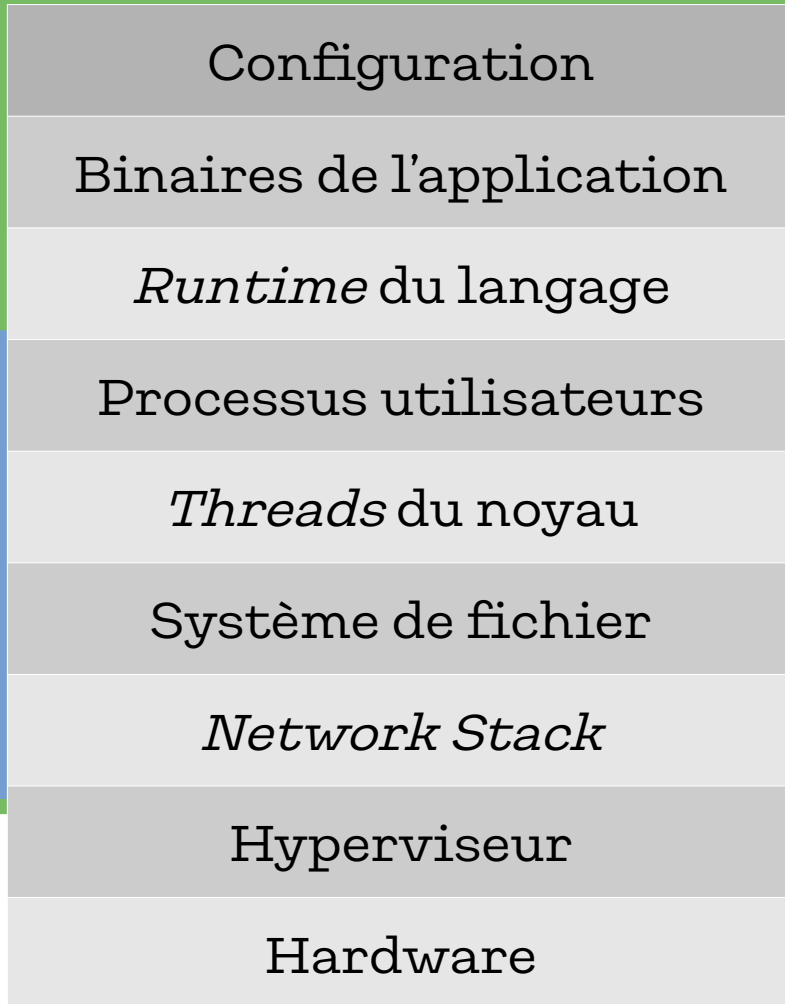
Un Unikernel

Une image spécialisée **construite pour les besoins de l'application** qui l'habite. Fonctionnant directement sur un **hyperviseur** ou sur du **matériel** !

- Le développeur choisit un ensemble de composant modulaire correspondant aux services d'un système d'exploitation
- L'application est compilée avec ses composants et ses fichiers de configuration
- Production d'une image légère n'exposant que ce qui a été demandé



Container Docker



2015 : Unikernel System

Lancement d'une Startup pour la promotion des Unikernels



- **ClickOS** : Unikernel très efficace (VM : ~5MB, Boot < 20ms etc.)
- **Clive** : Unikernel pour le cloud et les système distribué.. en Go
- **Drawbridge** : Project de microsoft pour construire des Sandboxes
- **HaLVM** : De Haskell à l'hyperviseur de Xen
- **IncludeOS** : C++ pour des hardwares virtuels
- **LING** : Unikernel Erlang qui comprend les fichiers .beam (seulement 3 bibliothèques externes)
- **MirageOS** : Unikernel en Ocaml
- Et d'autres : **Osv, Rumprun, Runtime.JS, Unik, HermitCore**

2016 : Rachat par Docker

~~Fermeture d'une Startup pour la promotion des Unikernels~~

MirageOS

Vous en doutez ?



MirageOS

- Un outillage pour construire des Unikernels incluant :
 - Un compilateur d'image
 - Plus d'une centaine de **bibliothèques agnostiques**
 - Une documentation riche
- Ecrit en OCaml, (avec un minimum de Stubs C)
- Backend pour Xen, KVM, Unix

MirageOS :

quelques bibliothèques notables

- ocaml-tls
- ocaml-git
- mirage-tcpip
- cohttp
- Decompress
- Irmin
- Lwt (liée à Ocsigen)

MirageOS : Fonctionnement

- Configuration + écriture de l'unikernel (en OCaml)
- Compilation de l'image
- Compilation vers une cible :
 - `mirage configure -t unix && mirage build`
 - → `./my_unikernel`
 - `mirage configure -t xen && mirage build`
 - → `xl create my_unikernel.cfg`
 - `mirage configure -t kvm && mirage build`
 - → `./solo5-hvt my_unikernel.hvt`

Pourquoi OCaml

- Langage de programmation fonctionnel, statiquement typé, strict et expressif
- Dôté d'un système de types riche et expressif avec un mécanisme d'inférence très évolué
- **Dôté d'un langage de modules très très riche**
- (et de mécanismes impératifs, OO etc.)

Bibliothèques Mirage

- Agnostique de la plateforme (AMD64, IA32, PowerPC, ARM ... JavaScript) et la cible (UNIX, POSIX, Windows, Mac, MirageOS)
 - **OCaml-tls** utilisé dans IncludeOS
 - **Mirage-tcpip** utilisé dans Docker For Mac
 - **OCaml-git** utilisé dans un browser
- Fortement paramétrable grâce au **langage de module**

Le langage de modules

- Permet la compilation séparée
- Sépare l'implémentation de l'interface. Une interface pouvant servir plusieurs implémentations
- Un langage fonctionnel indépendant, dans le langage

Un exemple, le filesystem

- Une interface : **FS**, exprimant les primitives d'un système de fichier (comme étant la composition d'un système **ReadOnly** et d'un système **WriteOnly**)
- Plusieurs implémentations :
 - **Ext3 : FS**
 - **Ext4 : FS**
 - **ZFS : FS**

Rappel sur les différents niveaux de valeurs dans les langages de programmation :

Value level

val **x** = 25

λ (Value level)

```
val x = f(25)
```

Type level

λ (Value level)

```
val x : Int = f(25)
```

λ (Type level)

λ (Value level)

data T1 f a = T1 (f a) - T1 :: (* -> *) -> * -> *

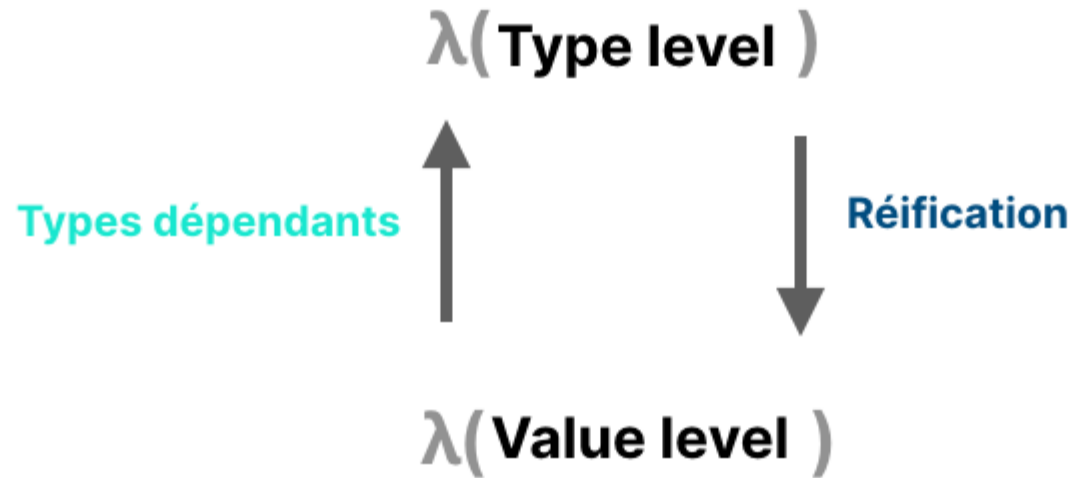
λ (Type level)



Réification

λ (Value level)

```
inline fun <reified T, A> f(x: A) = x is T
```



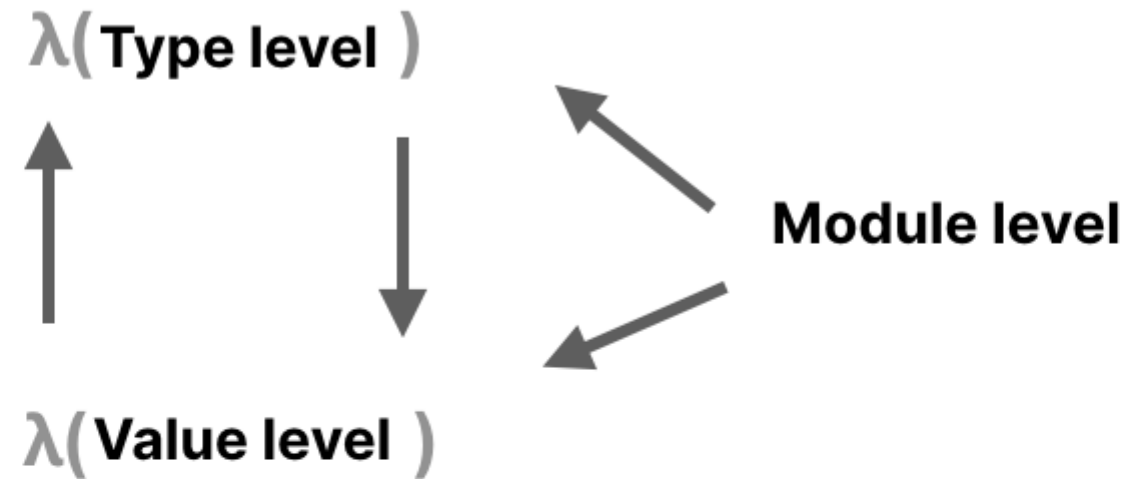
```
append :  
  (xs : Vect lengthA elem)  $\rightarrow$   
  (ys : Vect lengthB elem)  $\rightarrow$   
  Vect (lengthA + lengthB) elem
```


λ (Type level)



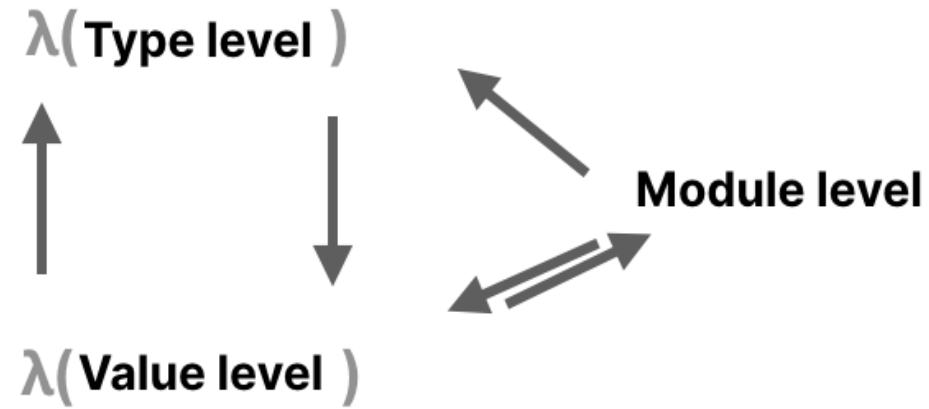
λ (Value level)

Module level



```
let print (type a) (module Showable : Show with type t = a) (x : a) =  
  print_endline (Showable.to_string x)
```

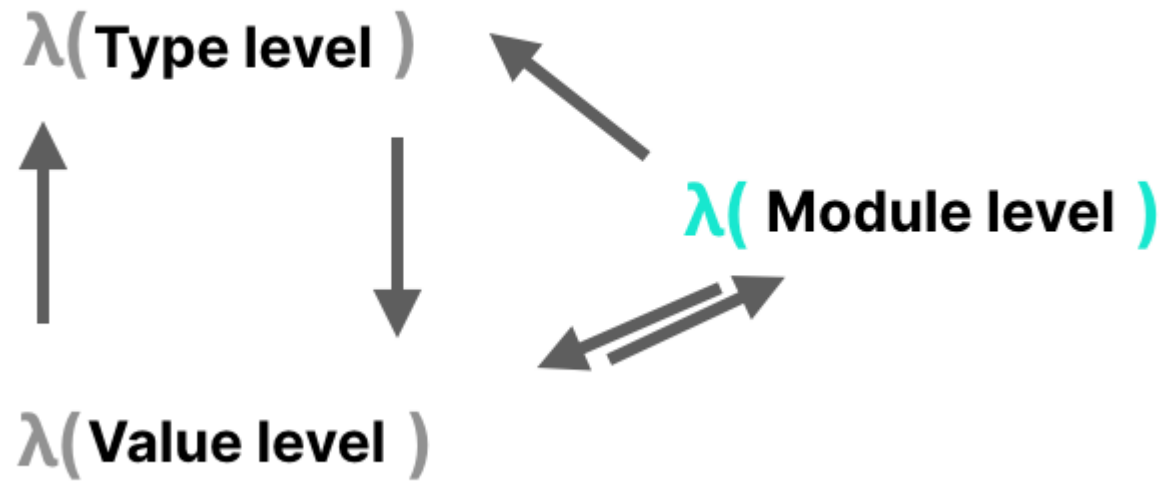
```
let () = print (module Gender) Female
```



```
module type Succ = sig  
  val succ : int → int  
end
```

```
let module_from_value_level () =  
  ( module struct  
    let succ x = x + 1  
  end : Succ )
```

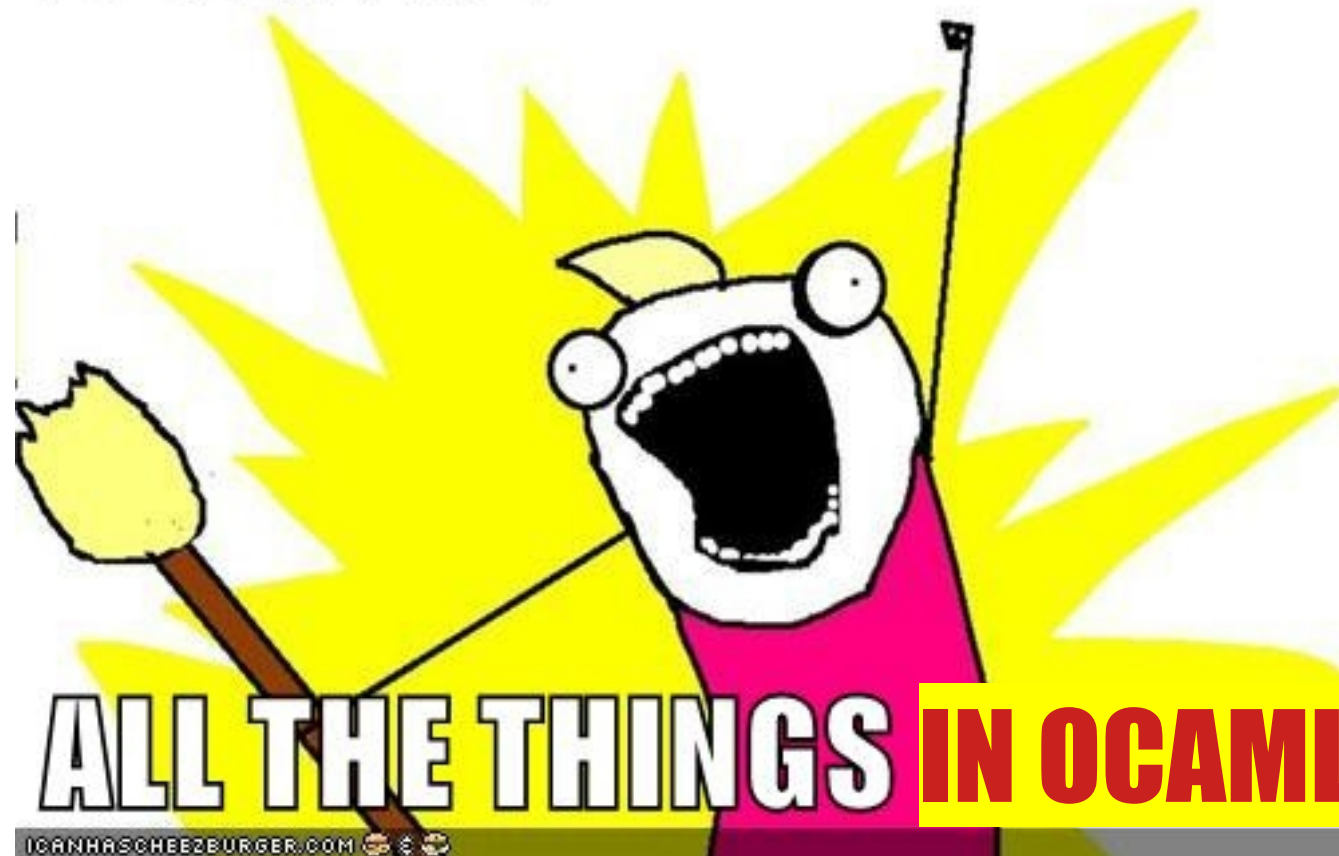
```
let flat_map (type ???) (module M : Monad with type 'a t = ???) f x =  
  M.flat_map f x
```



C'est en partie grâce à la richesse de ce système de modules que OCaml était un choix pertinent pour construire une *toolchain* destiné à la création d'Unikernel.

Faiblesses de MirageOS

REWRITE



Mais...

- Ce qui amène une stack plus safe (pas de manipulation de mémoire à la main, *:heartbleed:*)
- Beaucoup de contexte d'exécutions différents
- C'est cool d'écrire du OCaml

En conclusion



- Adapté aux services, ils ne sont pas approprié à l'informatique généraliste (mais bon...)
- Pas de support (pour le moment) du **Hot Code Swapping**. Un Unikernel compilé n'est pas modifiable. Il faut recompiler et redéployer une image altérée
- Ils demandent la construction d'un écosystème robuste et riche pour être réellement utilisable

- Les Unikernels sont, comme la programmation fonctionnelle pour la programmation impérative, le passé et le futur « potentiel » des approches *mainstream*
- Ils servent parfaitement les micro-services et les attentes du Cloud. Ils sont aussi des candidats adéquats pour l'IOT
- Ils améliorent la sécurité, diminuent la taille des images, optimisent le système, ont des temps de démarrage faibles
- Le cas de MirageOS pousse la composabilité et la réutilisabilité à un niveau extrême. Bénéficiant des garanties offertes par le langage.

Merci !





#Euratech10

CONTACT



Van de Woestyne Xavier



Data engineer, margo.com



xaviervdw@gmail.com



06 73 38 72 84

