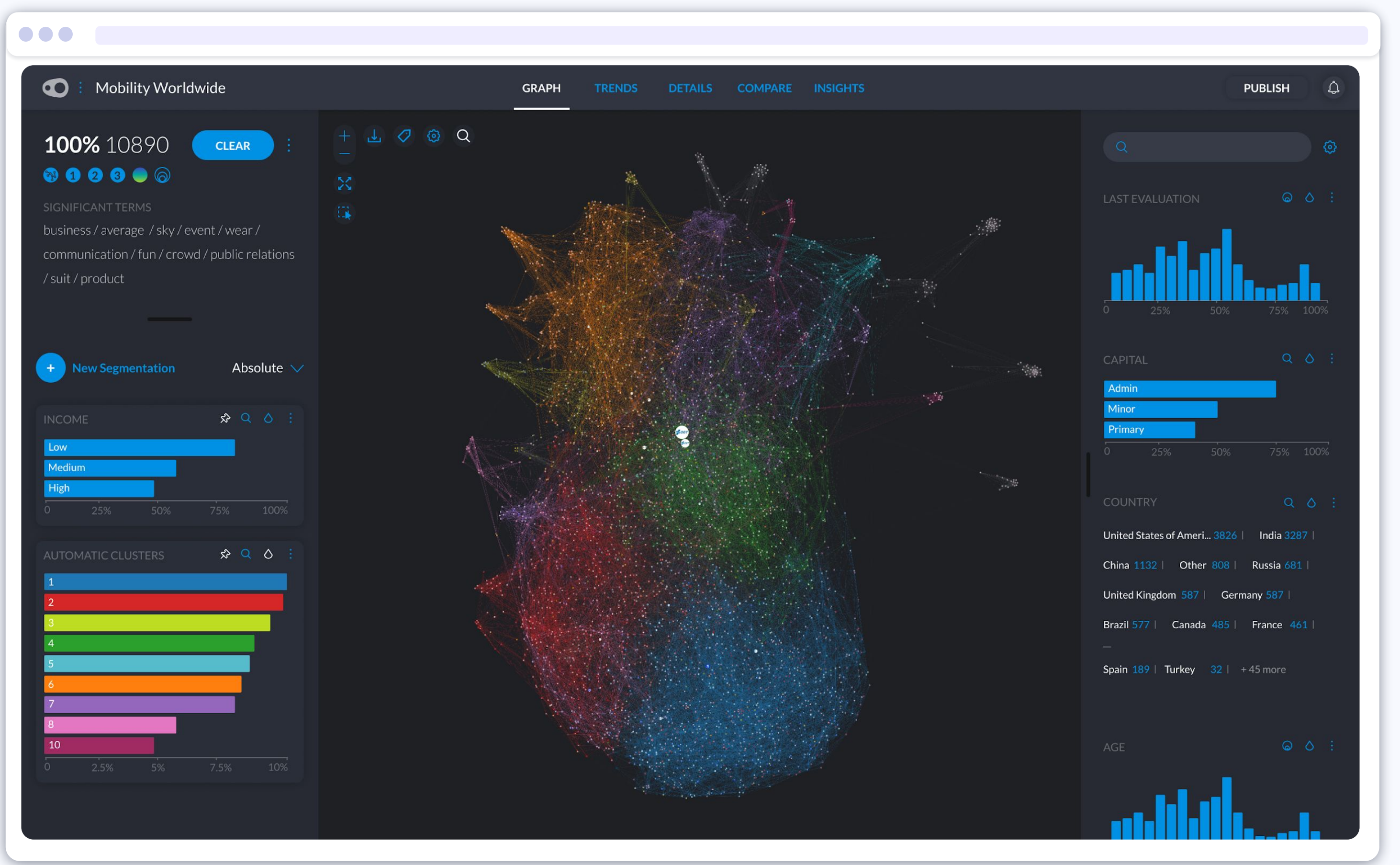
Cómo acelerar la web con C++ y WebAssembly

Hola

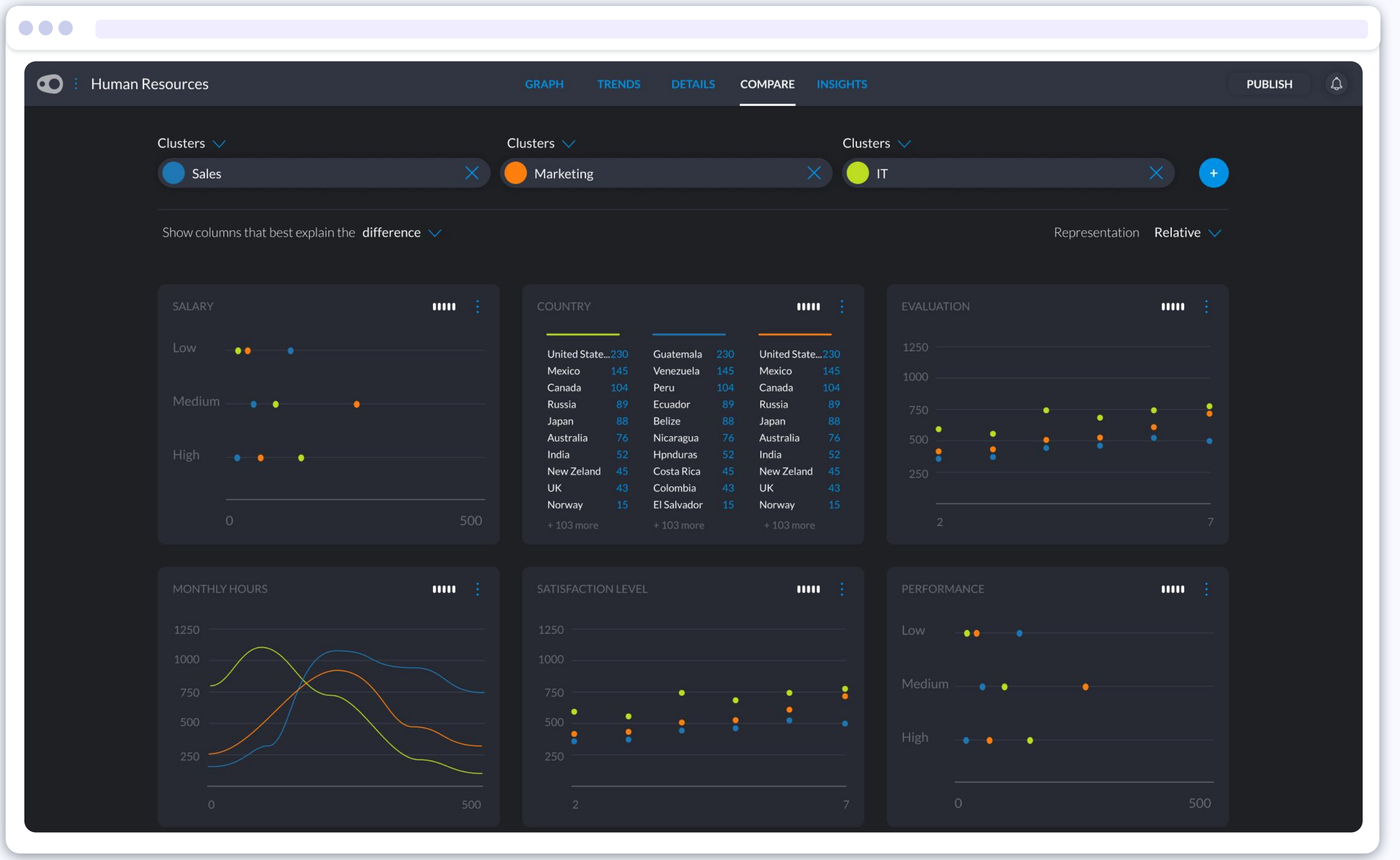
Miguel Cantón Cortés - CTO
miguel@graphext.com

Juan Morales del Olmo - Tech & Design
juan@graphext.com

¿Qué necesitamos acelerar con WebAssembly? Graphext



¿Qué necesitamos acelerar con WebAssembly? Graphext



¿Qué es WebAssembly?

- JavaScript es **EL** lenguaje de la Web
- Aunque tiene compiladores muy eficientes es **muy dinámico**
- Para algunas tareas **es demasiado lento**

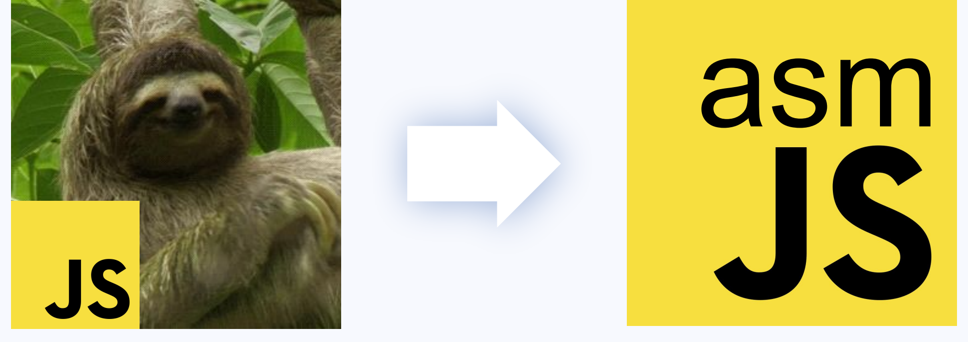
¿Qué es WebAssembly?

- JavaScript es **EL** lenguaje de la Web
- Aunque tiene compiladores muy eficientes es **muy dinámico**
- Para algunas tareas **es demasiado lento**

¿Qué es WebAssembly?

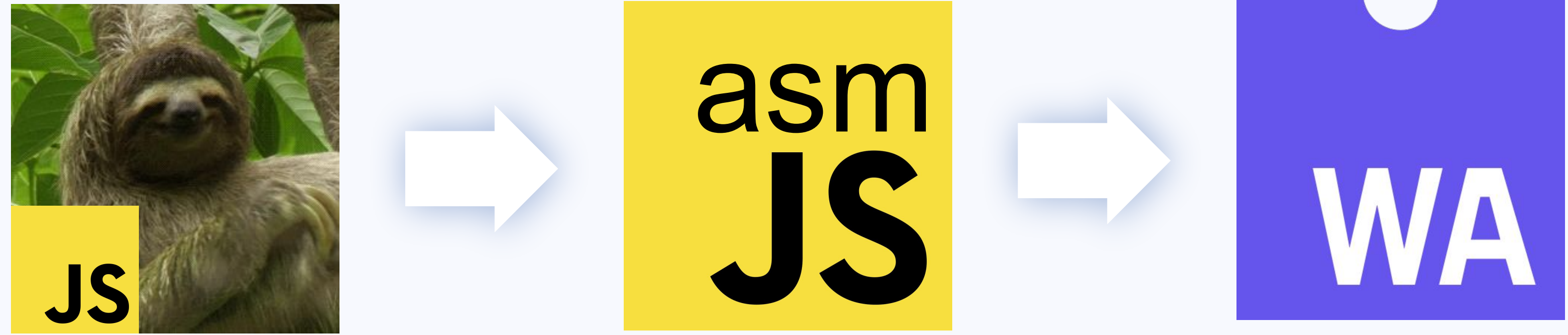
- JavaScript es **EL** lenguaje de la Web
- Aunque tiene compiladores muy eficientes es **muy dinámico**
- Para algunas tareas **es demasiado lento**
- Hay mucho código escrito en C/C++ no reutilizable

¿Qué es WebAssembly?



- Subconjunto de JS Optimizado
- Destino de compilación C/C++
- Castings para poder optimizar
  Ahead-Of-Time

¿Qué es WebAssembly?



- WebAssembly busca lo mismo que asm.js pero sin “hacks”
- Es un formato **binario**, **portable** y eficiente en **tamaño** y **tiempo de carga**
- Ejecuta a velocidad nativa ಠ_ಠ

WebAssembly

MVP Already well supported

WebAssembly

MVP Already well supported

- Seguro (sandboxed)
- 32 bits pointers (wasm32)
- Formato binario (.wasm) y texto (.wat)
- Linear Memory (solo crecer por ahora)

WebAssembly

MVP Already well supported

- Seguro (sandboxed)
- 32 bits pointers (wasm32)
- Formato binario (.wasm) y texto (.wat)
- Linear Memory (solo crecer por ahora)

- 4 types supported (i32, i64, f32, f64)
- OJO: Hay i64 pero Javascript no tiene
- OJO2: char* → String (utf16)

WebAssembly

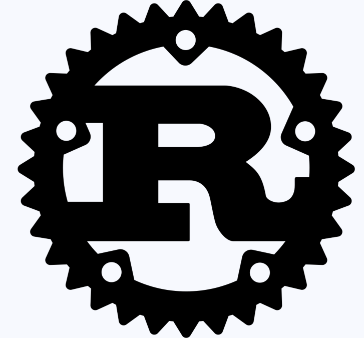
- **¡No sólo para la web!** (lambdas, edge CDN, IoT, ...)
- WebAssembly System Interface
- Estandarizar una interfaz para que WebAssembly corra fuera de la web
- Piensa en WASI como un POSIX pero restringido

WebAssembly

Wasmtime

WebAssembly (target)

Ahora mismo:



¡ En un futuro GC languages !

Emscripten

- Toolchain, empieza en Mozilla (2010), original para generar asm.js
  - **Fastcomp** era un clang parcheado → asm.js
  - **Binaryen** toolchain que hace asm.js → wasm
- Ahora usa directamente **LLVM (clang 10)**, tiene un **backend oficial** para wasm

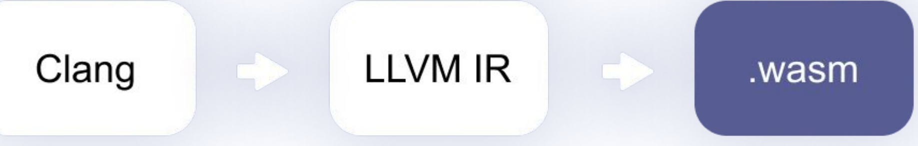
Emscripten

- Aporta una libc (musl) ← necesario para **portabilidad** de código existente
  - Pthread, File System, OpenGl ES 2.0/3.0, ...
  - Emscripten-ports (SDL2, bullet, sqlite)
  - Si tienes los fuentes lo puedes compilar a WASM
- ¿Y GCC? → Meeh, DragonEgg fué un intento de usar LLVM como backend para GCC

Emscripten (Flujo)

Antes:

Ahora:



Emscripten (Instalación)

- Emsdk:
  - $> git clone https://github.com/emscripten-core/emsdk.git
  - $> ./emsdk install 1.39.6
- Esto instala **emcc**, **em++**, **emar**, **emcmake**, **emconfigure**...
- En nuestra experiencia, se puede acelerar parte del desarrollo probando build nativos porque cambia “poco”. Salvo algunos puntos...
  - **time_t** es de 32 bits en emscripten y lo normal es que sea de 64 bits en otras libc

Bindings

Bindings C++ ↔ Js

- Embind: En C++ se describen las clases y los métodos
- WebIDL-binder: Usa **WebIDL** con pequeñas modificaciones (constructores, enumerados...)
  - Es declarativo
  - Genera glue.cpp
  - glue.js con getters/setters y prototypes. Puedes trabajar con objetos desde JS pero por debajo son offsets de memoria
  - WebIDL ⟷ TS
- Javascript inline: EM_ASM (macro)

Usando contenedores de la STL en JS con WebIDL binder

C++

```cpp
class VectorString : public std::vector<std::string> {
    using stdVector = std::vector<std::string>;
    using stdVector::stdVector;

    public:
        const char* at(uint32_t i) const {
            return stdVector::at(i).c_str();
        }
        void set(uint32_t i, const char* str) {
            (*this)[i] = str;
        }
};
```

WebIDL

```
interface VectorString {
    void VectorString(unsigned long size);
    [Const] DOMString at(unsigned long i);
    void set(unsigned long i, DOMString str);
    unsigned long size();
};
```

JavaScript

```javascript
const v = new wasm.VectorString(3);
v.set(0, "Hola ");
v.set(1, "Mundo");
v.set(2, "!");
for(let i = 0; i < v.size(); i++) {
    console.log(v.at(i));
}
wasm.destroy(v);
```

WebIDL binder

- WebIDL binder **copia** automáticamente a la memoria de WASM los arrays pasados como argumento
- También se encarga de realizar el cambio de encoding de **utf8**⟷ **utf16** (**char*** ⟷ **DOMString**)
- Estrategias para **evitar** copias innecesarias:
  - TypedArray views sobre la memoria de WASM
  - Wrap de contenedor
    - ej: std::vector<std::string>

WebIDL binder

C++	WebIDL
bool	boolean
float	float
double	double
char*	byte[]
char*	DOMString
int	long

C++	WebIDL
type&	**[Ref]** *type*
const *return-type*	**[Const]** *return-type*
const *member-type*	**readonly attribute** *member-type*

Usando contenedores de la STL en JS con WebIDL binder

C++

```
class VectorString : public std::vector<std::string> {
    using stdVector = std::vector<std::string>;
    using stdVector::stdVector;

    public:
        const char* at(uint32_t i) const {
            return stdVector::at(i).c_str();
        }
        void set(uint32_t i, const char* str) {
            (*this)[i] = str;
        }
};
```

WebIDL

```
interface VectorString {
    void VectorString(unsigned long size);
    [Const] DOMString at(unsigned long i);
    void set(unsigned long i, DOMString str);
    unsigned long size();
};
```

JavaScript

```
const v = new wasm.VectorString(3);
v.set(0, "Hola ");
v.set(1, "Mundo");
v.set(2, "!");
for(let i = 0; i < v.size(); i++) {
    console.log(v.at(i));
}
wasm.destroy(v);
```

Emscripten (APIs)

- Threads → WebWorkers (activar flag del navegador de SharedArrayBuffer)
- File System:
  - Sistema de archivos del host (solo en nodejs, **NODEFS**)
  - En memoria, sin persistencia (**MEMFS**)
  - Con persistencia, usando IndexedDB (**IDBS**)
  - Los assets se pueden empaquetar durante la compilación

Emscripten (APIs)

- SDL2 y 3 permite pintar en Canvas
- OpenGL ES 2.0, 3.0 → WebGL 1/2
- Sockets → Websockets + Proxy
- El código no se debe bloquear puesto que si no se devuelve el control de ejecución, la pestaña del navegador queda bloqueada.

Optimización

- Emscripten ofrece muchos flags de optimización
- Emscripten Memory Profile (flag --memoryprofiler)
  - Muestra Pila y Heap
  - Ojo fragmentación!!
    - WebAssembly no permite hacer ***shrink*** de la memoria reservada
  - Nº de allocations y deallocations. Se pueden pillar memory leaks.

Memory Profiler

Track all allocation sites larger than 16777216 bytes, and all allocation sites with more than 10000 outstanding allocations. (visit this page via URL query params foo.html?trackbytes=1000&trackcount=100 to apply custom thresholds starting from page load)
Total HEAP size: 512 MB.
STATIC memory area size: 1.03 MB. STATIC_BASE: 0x000000400
STACK memory area size: -1048576 B. STACK_BASE: 0x0001088f0. STACKTOP: 0x0001088f0. STACK_MAX: 0x000088f0.
STACK memory area used now (should be zero): 0 B. STACK watermark highest seen usage (approximate lower-bound!): 0 B
DYNAMIC memory area size: 295.15 MB. DYNAMIC_BASE: 0x0001088f0. DYNAMICTOP: 0x01282e000.
DYNAMIC memory area used: 258 B (0.00% of all dynamic memory and unallocated heap)
Free memory: DYNAMIC: 295.15 MB, Unallocated HEAP: 215.82 MB (100.00% of all dynamic memory and unallocated heap)
Preloaded memory used, most likely memory reserved by files in the virtual filesystem : 0 B
OpenAL audio data: 0 B (outside HEAP)
# of total malloc()s/free()s performed in app lifetime: 28/25 (currently alive pointers: 3)

Clear alloc stats

Sort allocations by: Bytes

Optimización (tamaño)

- Aligerar runtime mediante flags
  - Cambiar **allocator**: pequeño (emmalloc) vs rápido (jemalloc)
  - Desactivar el **catch** para excepciones
  - Desactivar el soporte para **RTTI**
  - Desactivar el **exit runtime**
  - Desactivar soporte para **sistema de archivos**
- Usar profilers de tamaño como Twiggy

DEMO

https://github.com/graphext/ejemplo_charla_webassembly

We are hiring !

miguel@graphext.com
juan@graphext.com

graphext.com/jobs